OBJECT ORIENTED MODELING AND DESING

Subject Code: 06CS71	**I.A. Marks : 25**
Hours/Week : 04	**Exam Hours: 03**
Total Hours : 52	**Exam Marks: 100**

PART - A

UNIT - 1 **7 Hours**

INTRODUCTION, MODELING CONCEPTS, CLASS MODELING: What is Object Orientation? What is OO development? OO themes; Evidence for usefulness of OO development; OO modeling history.Modeling as Design Technique: Modeling; abstraction; The three models.Class Modeling: Object and class concepts; Link and associations concepts; Generalization and inheritance; A sample class model; Navigation of class models; Practical tips.

UNIT - 2 **6 Hours**

ADVANCED CLASS MODELING, STATE MODELING: Advanced object and class concepts; Association ends; N-ary associations; Aggregation; Abstract classes; Multiple inheritance; Metadata; Reification; Constraints; Derived data; Packages; Practical tips.State Modeling: Events, States, Transitions and Conditions; State diagrams; State diagram behavior; Practical tips.

UNIT - 3 **6 Hours**

ADVANCED STATE MODELING, INTERACTION MODELING: Advanced State Modeling: Nested state diagrams; Nested states; Signal generalization; Concurrency; A sample state model; Relation of class and state models; Practical tips. Interaction Modeling: Use case models; Sequence models; Activity models. Use case relationships; Procedural sequence models; Special constructs for activity models.

UNIT - 4 7 Hours

PROCESS OVERVIEW, SYSTEM CONCEPTION, DOMAIN ANALYSIS: 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.

PART - B

UNIT - 5 **7 Hours**

APPLICATION ANALYSIS, SYSTEM DESIGN: 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.

UNIT - 6 **7 Hours**

CLASS DESIGN, IMPLEMENTATION MODELING, LEGACY SYSTEMS: 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.

UNIT - 7 **6 Hours**

DESIGN PATTERNS – 1: What is a pattern and what makes a pattern? Pattern categories; Relationships between patterns; Pattern description.

Communication Patterns: Forwarder-Receiver; Client-Dispatcher-Server; Publisher-Subscriber.

UNIT - 8 **6 Hours**

DESIGN PATTERNS – 2, IDIOMS: Management Patterns: Command processor; View handler. Idioms: Introduction; What can idioms provide? Idioms and style; Where to find idioms; Counted Pointer example.

TEXT BOOKS:

1. **Object-Oriented Modeling and Design with UML** – Michael Blaha, James Rumbaugh, 2nd Edition, Pearson Education, 2005.
2. **Pattern-Oriented Software Architecture: A System of Patterns - Volume 1**– Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, John Wiley and Sons, 2006.

REFERENCE BOOKS:

1. **Object-Oriented Analysis and Design with Applications** – Grady Booch et al, 3rd Edition, Pearson Education, 2007.
2. **Practical Object-Oriented Design with UML** – Mark Priestley, 2nd Edition, Tata McGraw-Hill, 2003.
3. **Object-Oriented Design with UML and JAVA** – K. Barclay, J. Savage, Elsevier, 2008.
4. **The Unified Modeling Language User Guide** – Booch, G., Rumbaugh, J., and Jacobson I, 2nd Edition, Pearson, 2005.
5. **Design Patterns: Elements of Reusable Object-Oriented Software** – E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1995.
6. **Object-Oriented Systems Analysis and Design Using UML** – Simon Bennett, Steve McRobb and Ray Farmer, 2nd Edition, Tata McGraw-Hill, 2002.

INDEX SHEET

Unit1: 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.

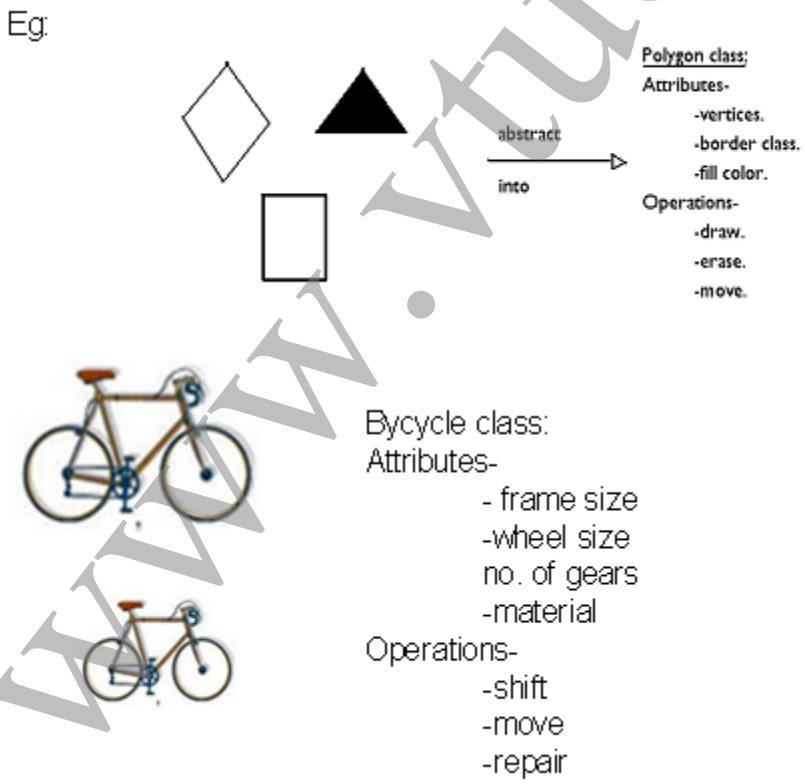
- Classification.
- Inheritance.
- Polymorphism.

□□**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?

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**

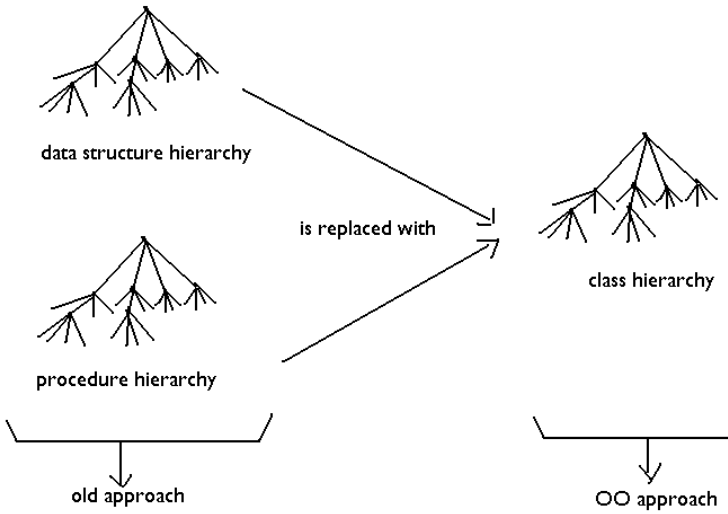
- Abstraction lets you focus on essential aspects of an application while ignoring details i.e focusing on what an object is and does, before deciding how to implement it.
- It's the most important skill required for OO development.

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 places 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 to describe 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.

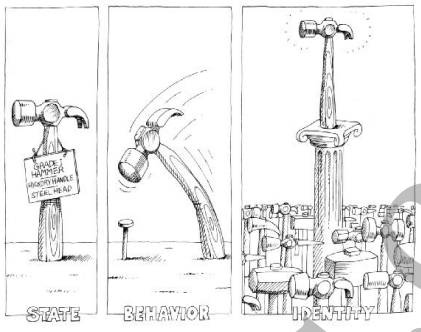
A **class** describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships and semantics.

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.

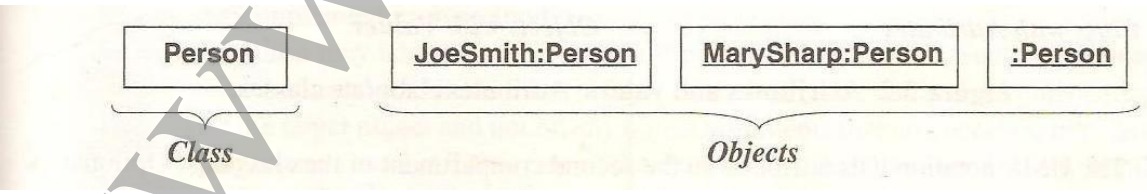
Note: An object diagram shows individual objects and their relationships.

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 boldface 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.

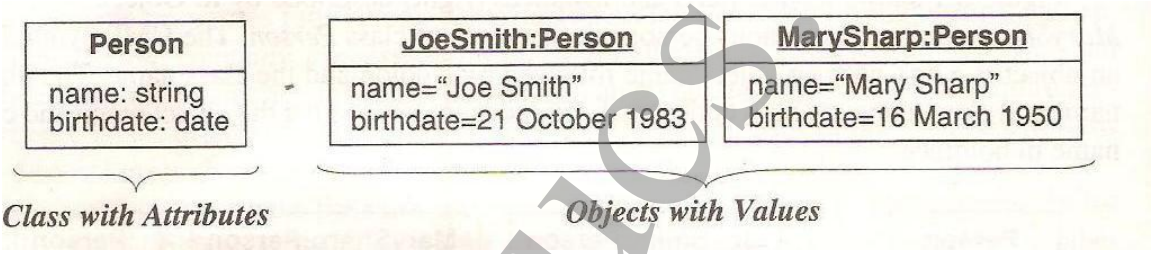
□□Following analogy holds:

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



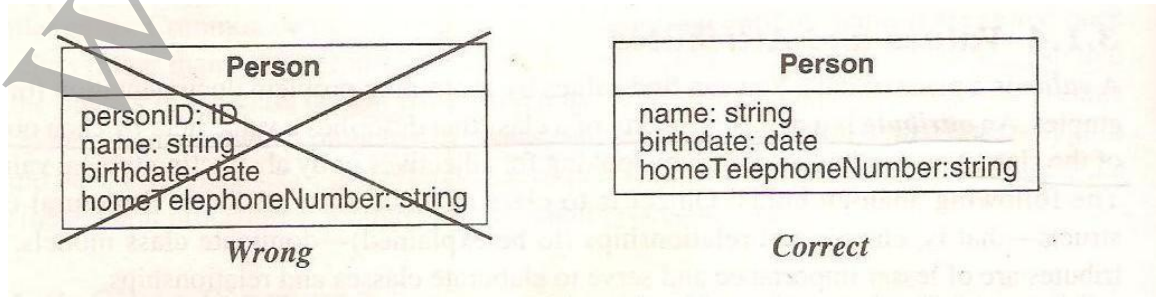
□□**Conventions used (UML):**

- 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 face, 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.



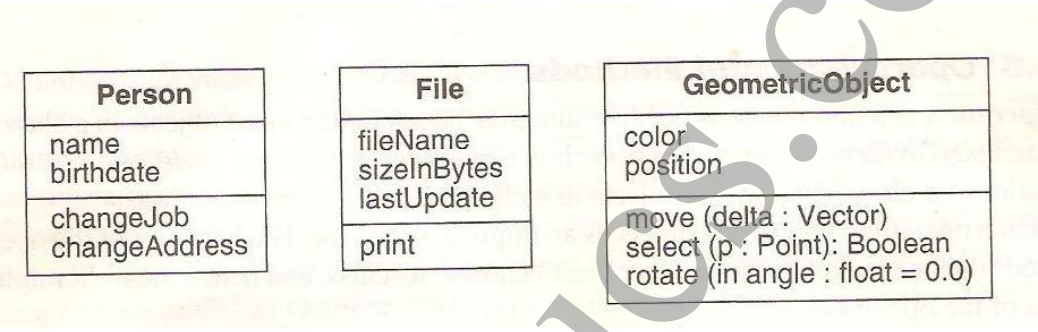
An **operation** is a function or procedure that maybe applied to or by objects in a class. E.g. Hire, fire and pay dividend are operations on Class Company. Open, close, hide and redisplay are operations on class window.

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 face, 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

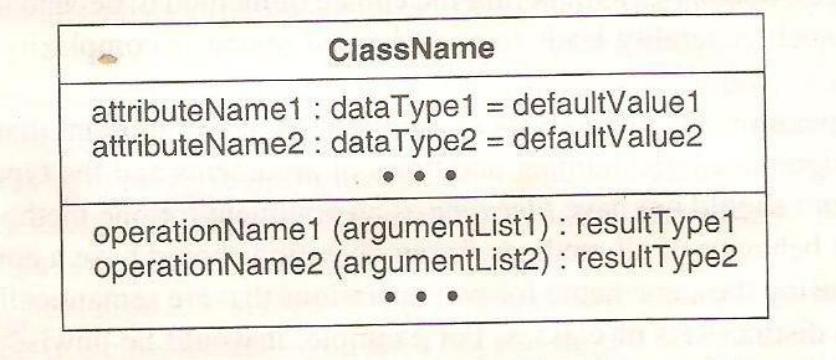


Fig: Summary of modeling notation for classes

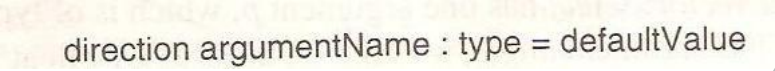
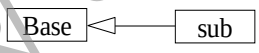


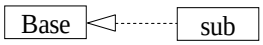
Fig: Notation for an argument of an operation

Class Digarms: Relationships

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

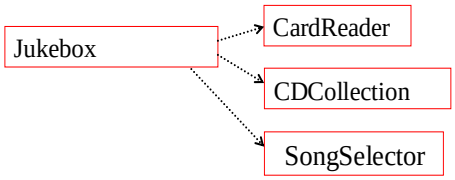
  - Association (delegation) 

  - Generalization (inheritance) 

  - Realization (interfaces) 

1) Dependency: A Uses Relationship

- Dependencies
  - occurs when one object depends on another
  - if you change one object's interface, 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

association

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.

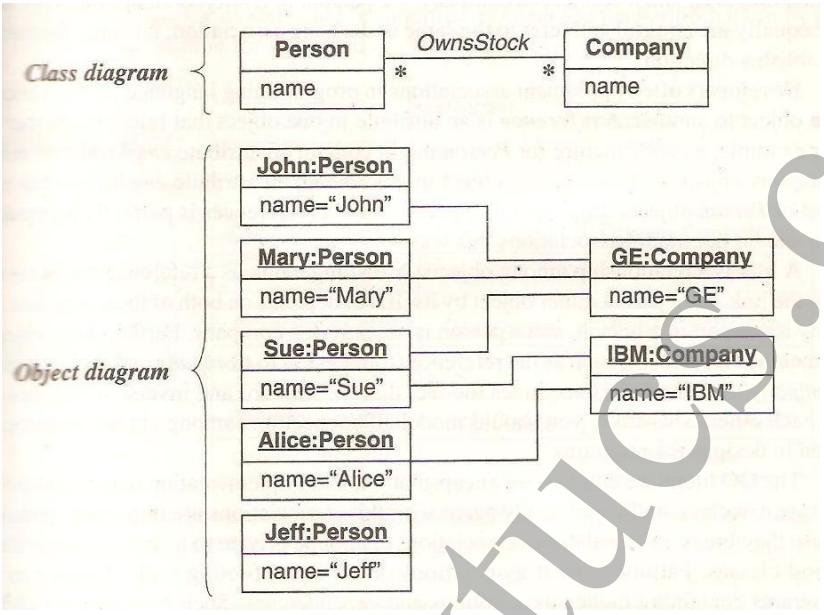
A link is an instance of an **association**.

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):

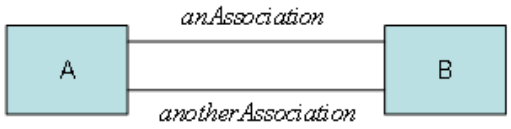
- Link is a line between objects; a line may consist of several line segments.
- If the link has the name, it is underlined.
- Association connects related classes and is also denoted by a line.
- Show link and association names in italics.

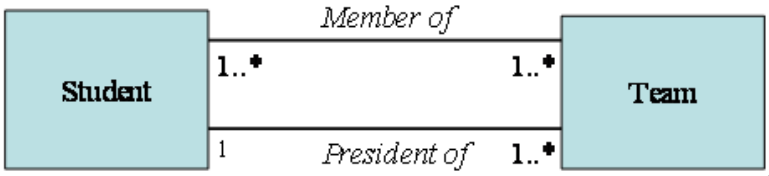
Note:

- 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.



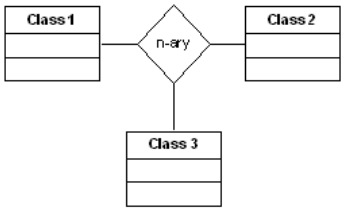


Class Diagrams (cont)

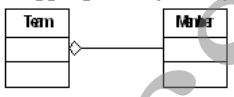
- Types of associations

Binary

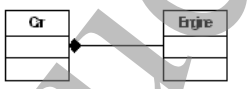
n-ary



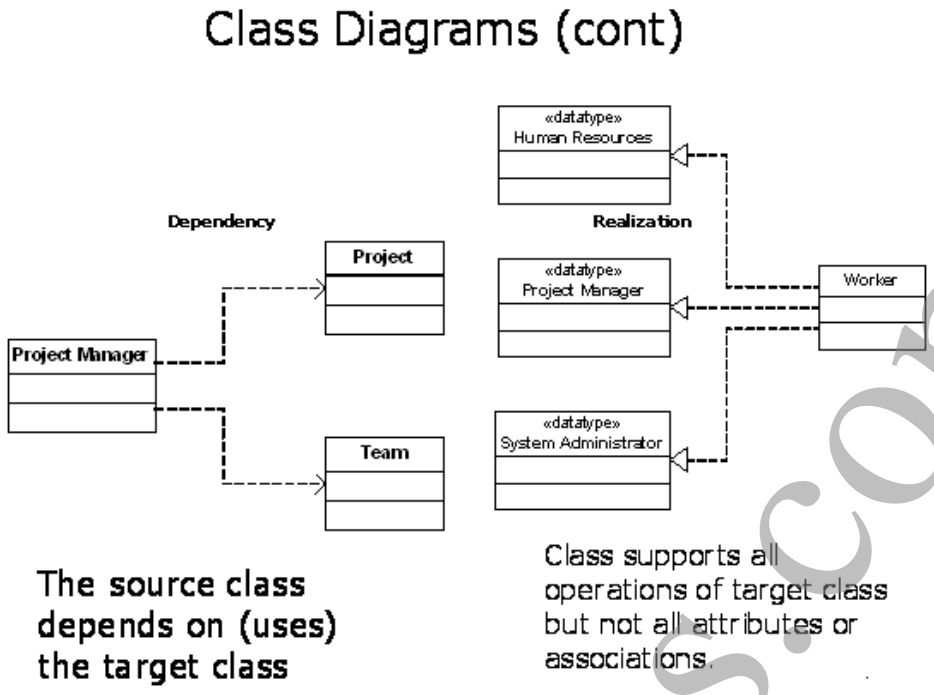
Aggregation (has-a)



Composition (is-composed-of)



Generalization (is-a-kind-of)



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

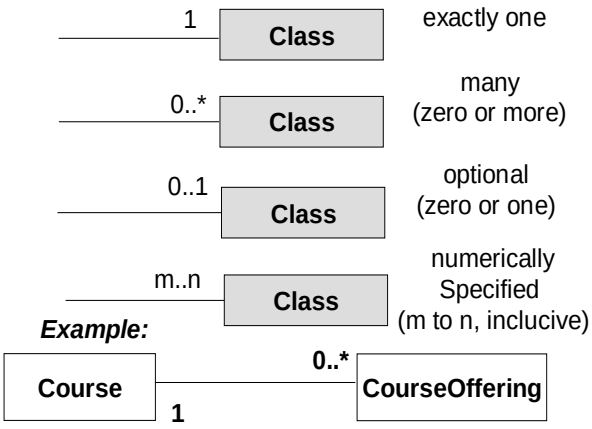
"1" (exactly one).

"1.."(one or more).

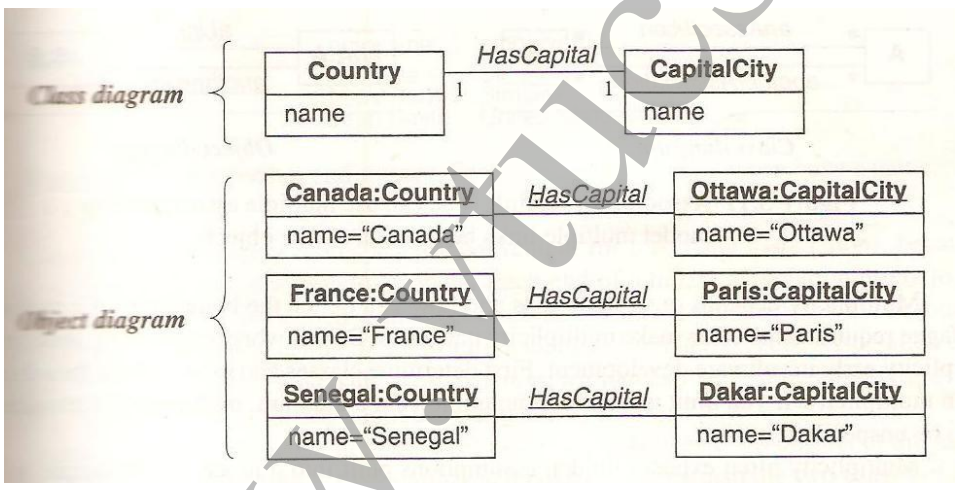
"3..5"(three to five, inclusive).

" * " (many, i.e zero or more).

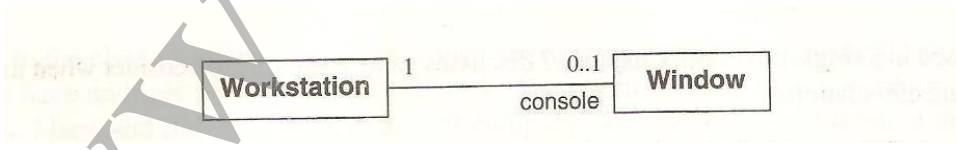
- notations



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.

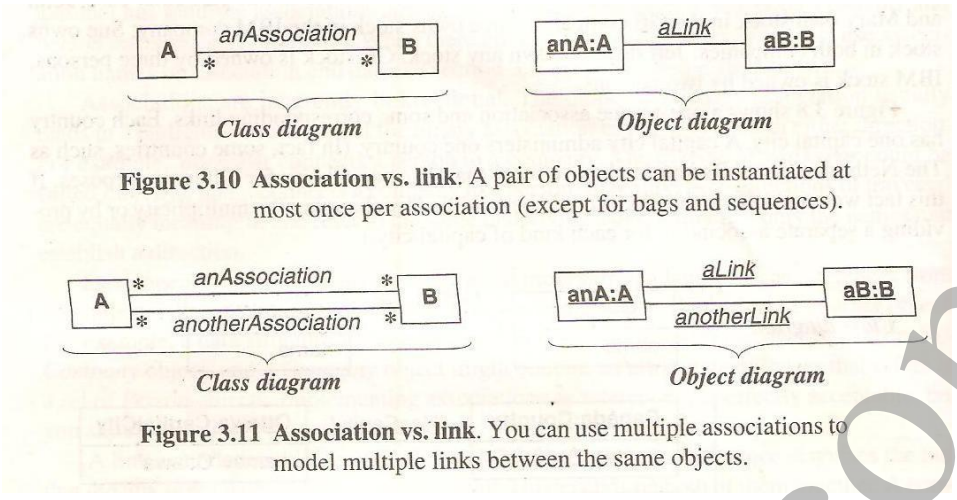
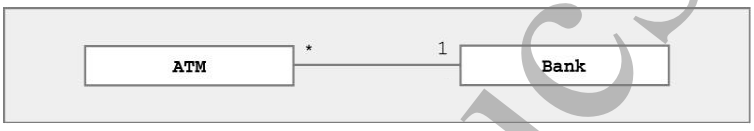


Figure 3.10 Association vs. link. A pair of objects can be instantiated at most once per association (except for bags and sequences).

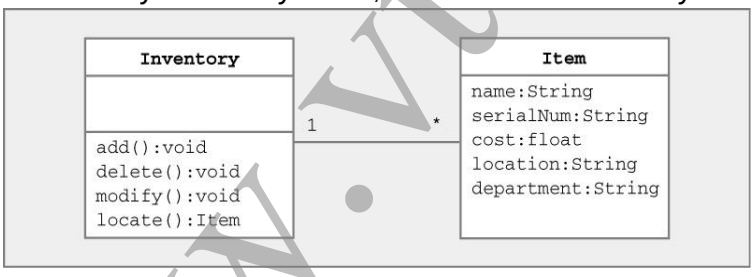
Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

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

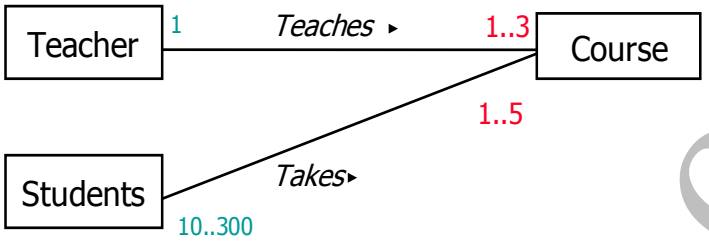


Association - Multiplicity

- 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.

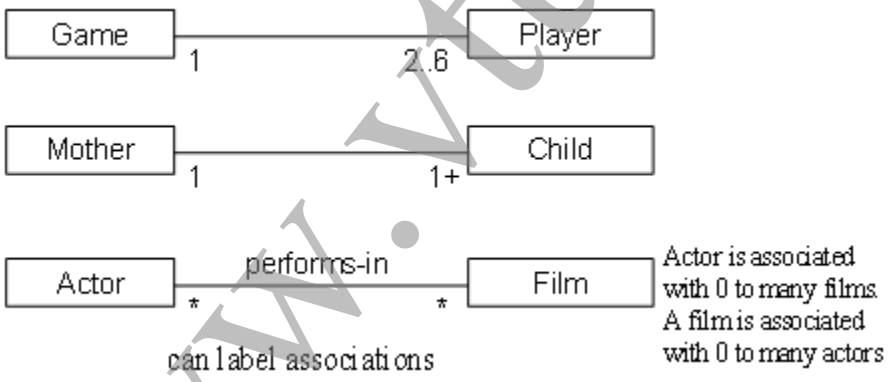
Association – Multiplicity

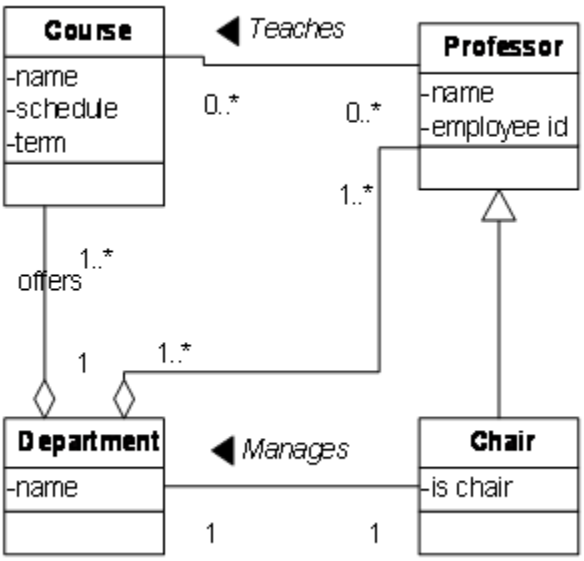
- A teacher teaches 1 to 3 courses (subjects)
- Each course is taught by only one teacher.
- A student can take between 1 to 5 courses.
- A course can have 10 to 300 students.



Multiplicity

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





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.

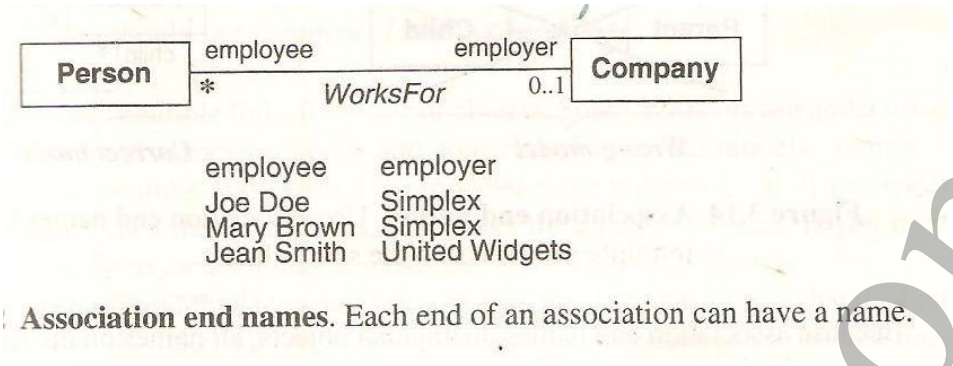
Note 3: The literature often describes multiplicity as being "one" or "many", but more generally it is a subset of the non negative numbers.

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.



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

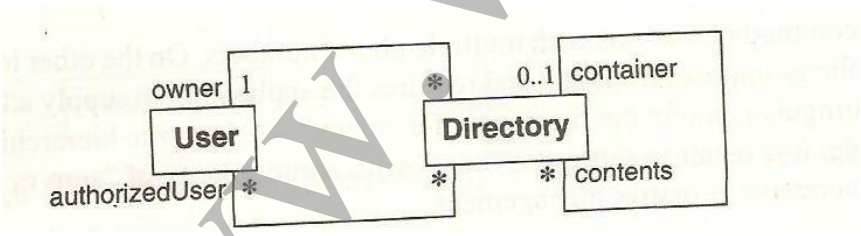
A person is an employee with respect to company.

A company is an employer with respect to a person.

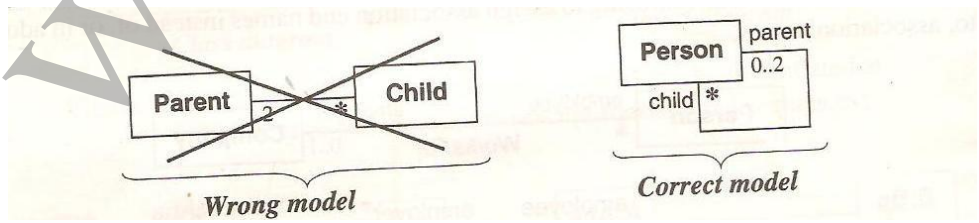
Note 1: Association end names are optional.

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.



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.

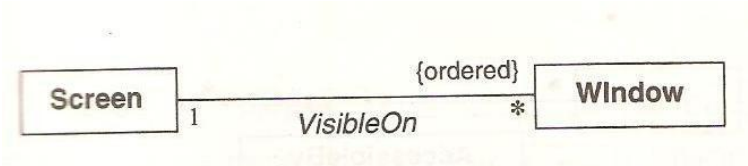


Fig: ordering sometimes occurs for "many" multiplicity

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:

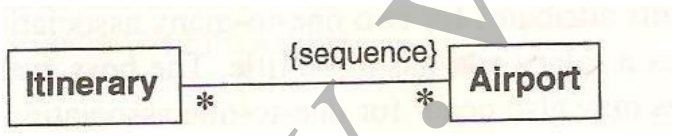


fig: an itinerary may visit multiple airports, so you should use {sequence} and not {ordered}

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

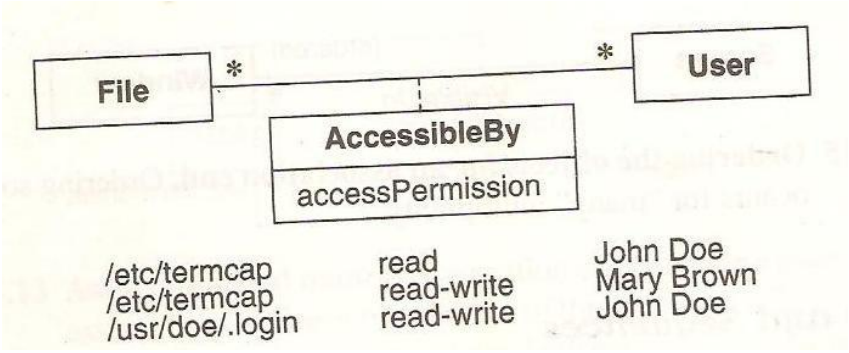
Association classes

□□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.

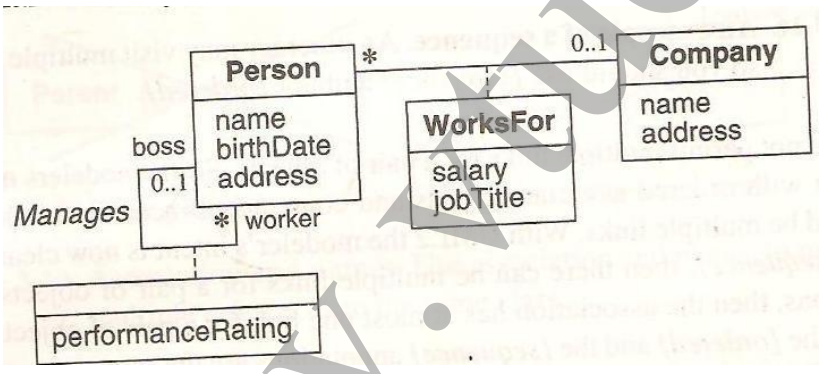
□□Ex:



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

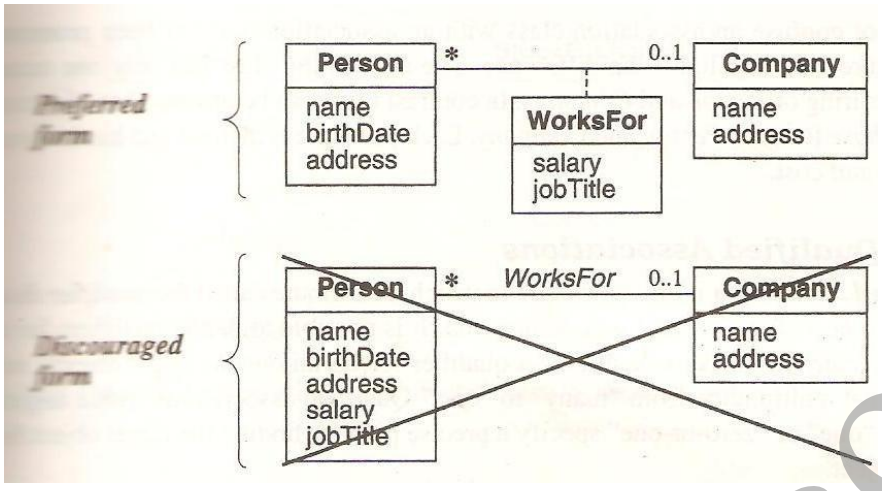
Note: Attributes for association class unmistakably belong to the link and cannot be ascribed to either object. In the above figure, accessPermission is a joint property of File and user cannot be attached to either file or user alone without losing information.

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.

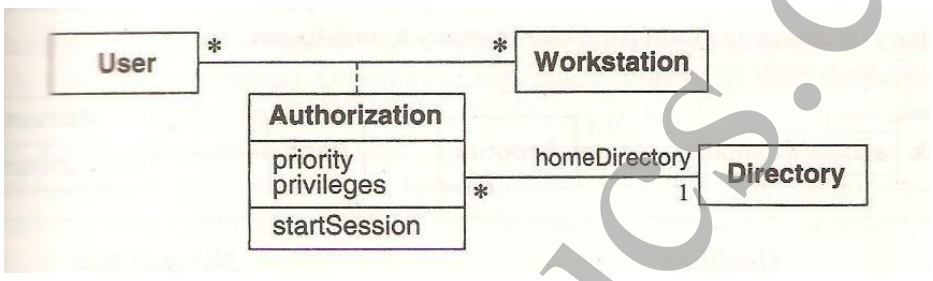


Note 1: Figure shows how it's possible to fold attributes for one-to-one and one-to-many associations into the class opposite a "one" end. This is not possible for many-to-many associations.

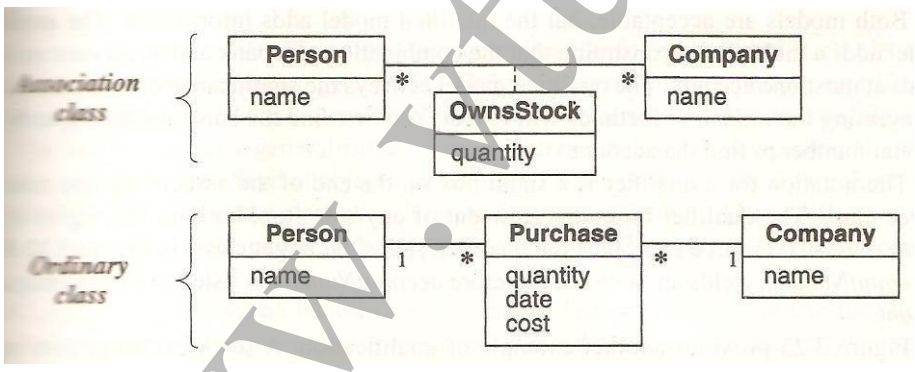
As a rule, you should not fold such attributes into a class because the multiplicity of the association may change.



Note 2: An association class participating in an association.



Note 3: Association class vs ordinary class.



eg:

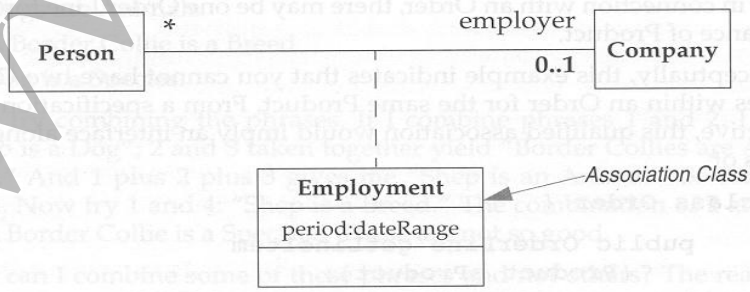
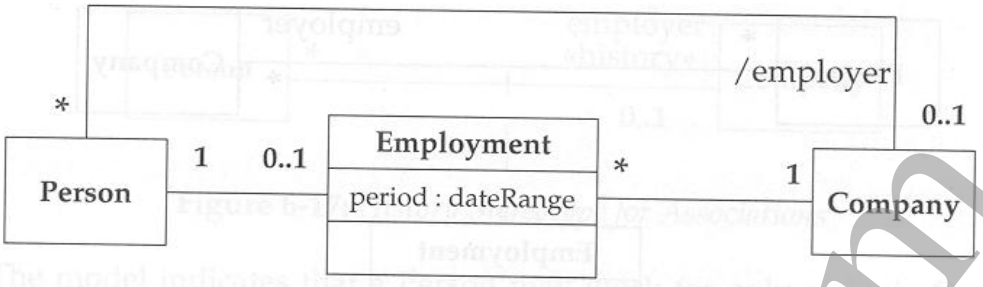


Figure 6-14: *Association Class*



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.

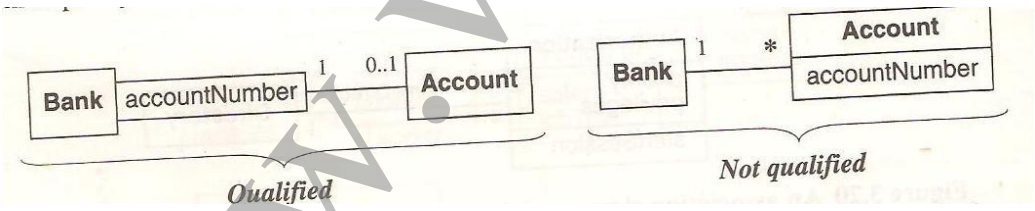
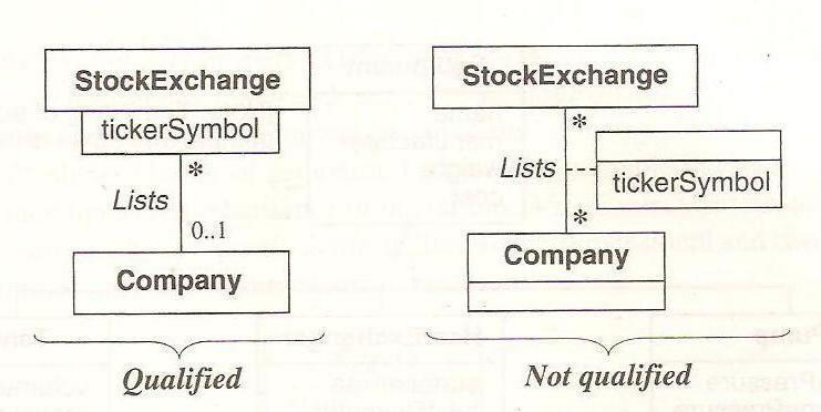
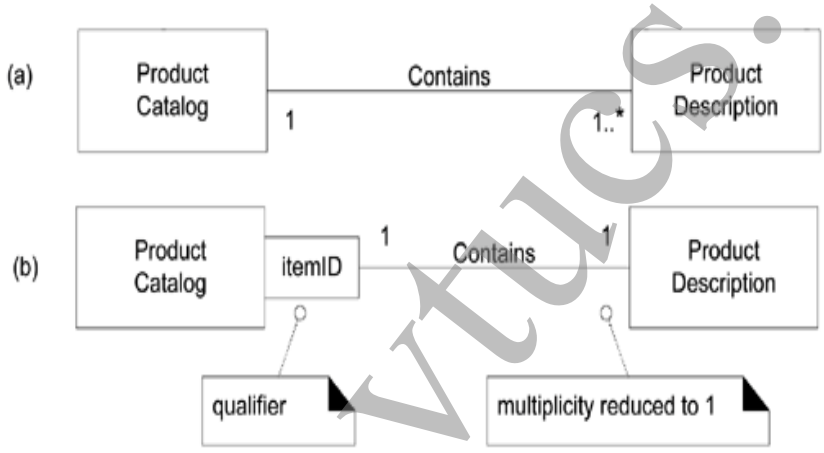


Fig: qualification increases the precision of a model. (note: however, both are acceptable)

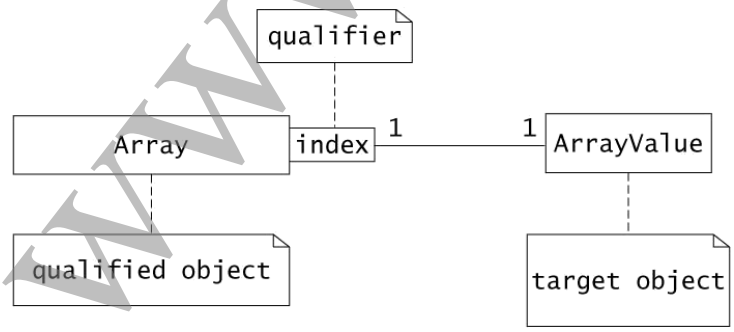
□□**Ex 2:** a stock exchange lists many companies. However, it lists only one company with a given ticker symbol. A company maybe listed on many stock exchanges, possibly under different symbols.



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.

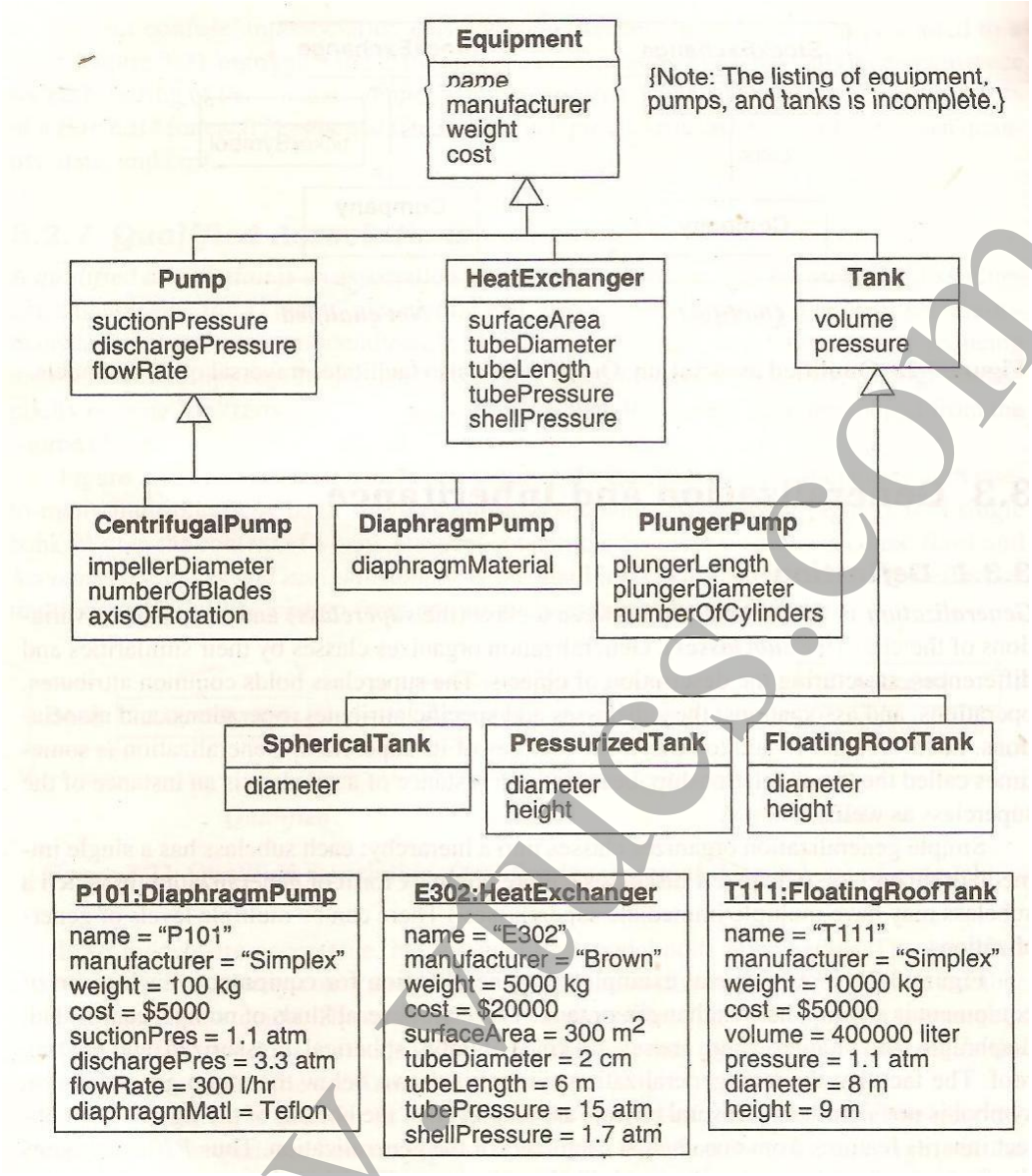
Each object inherits features from one class at each level of generalization.

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.



Fig(a)

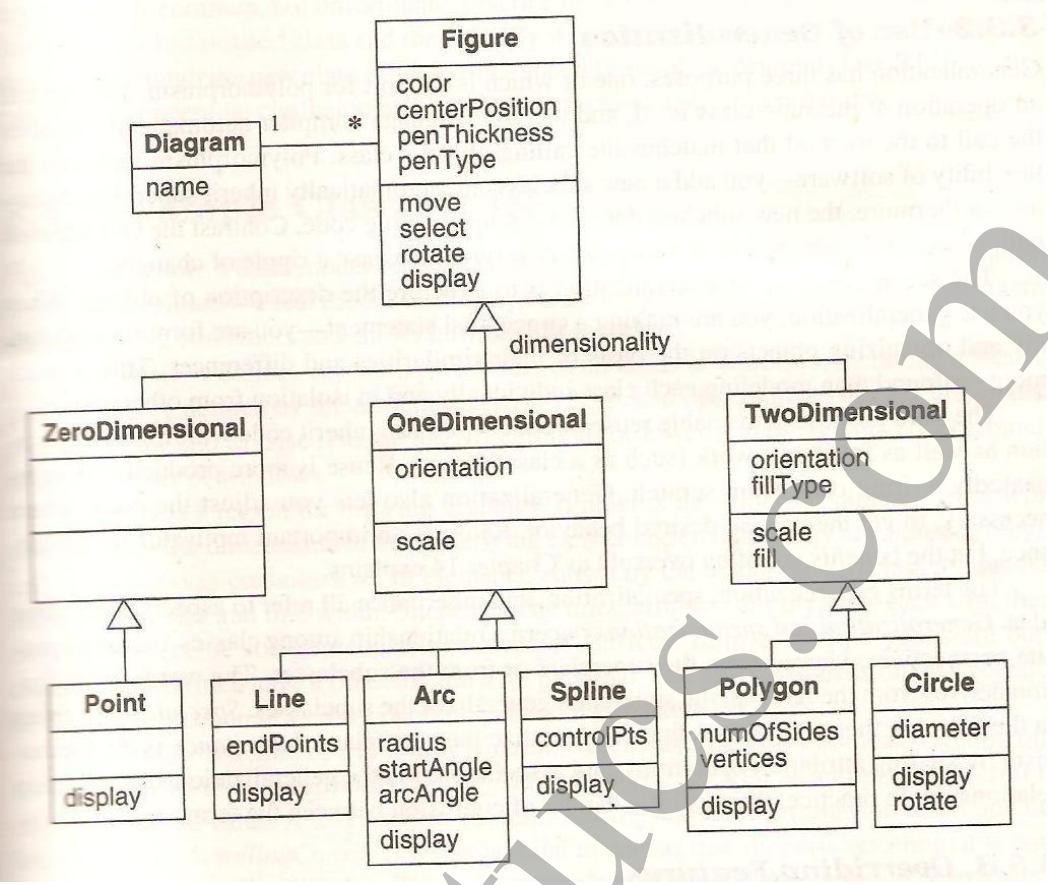


Fig (b)

'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 replaces the overridden feature (superclass feature) .

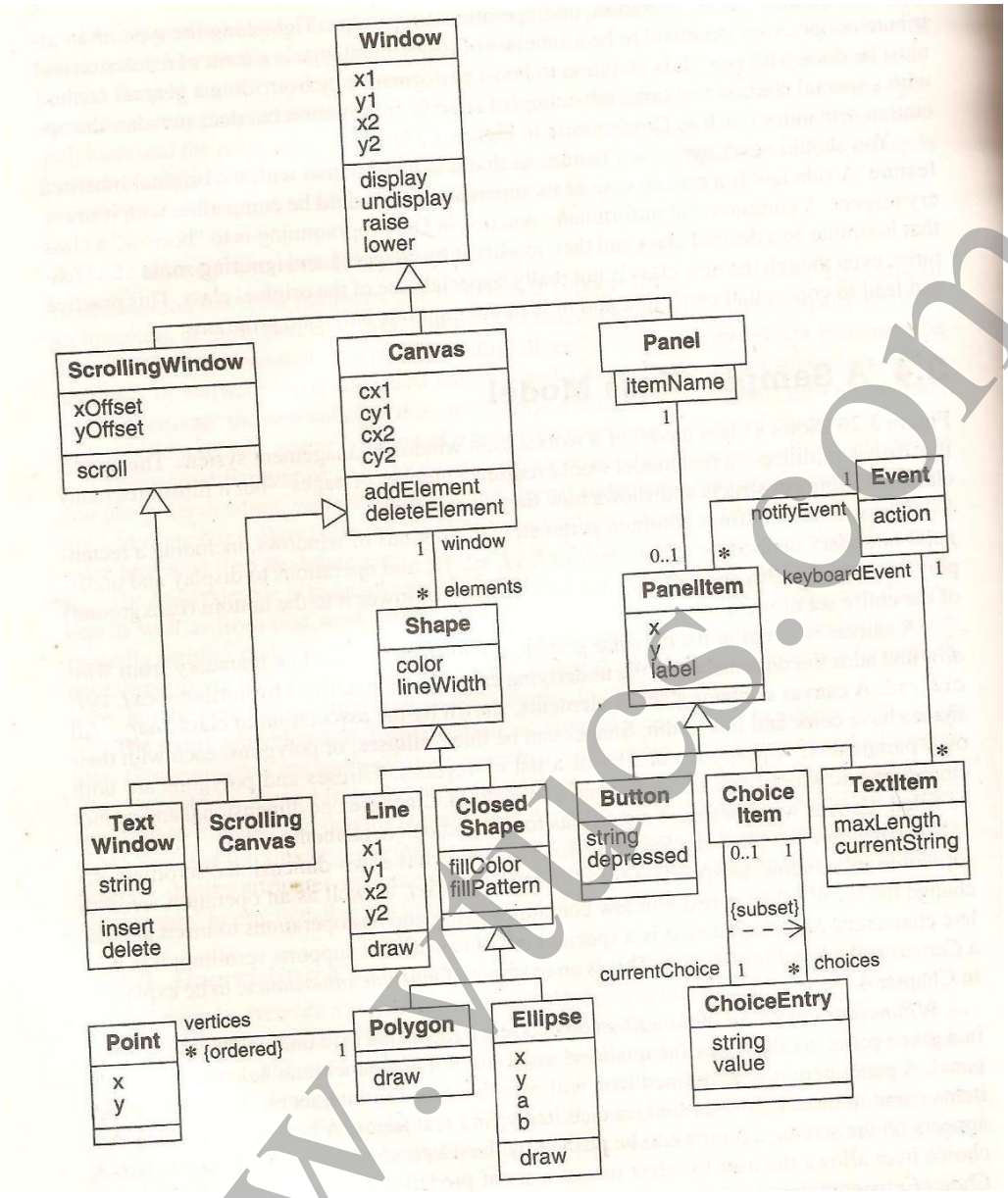
Why override feature?

- 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.

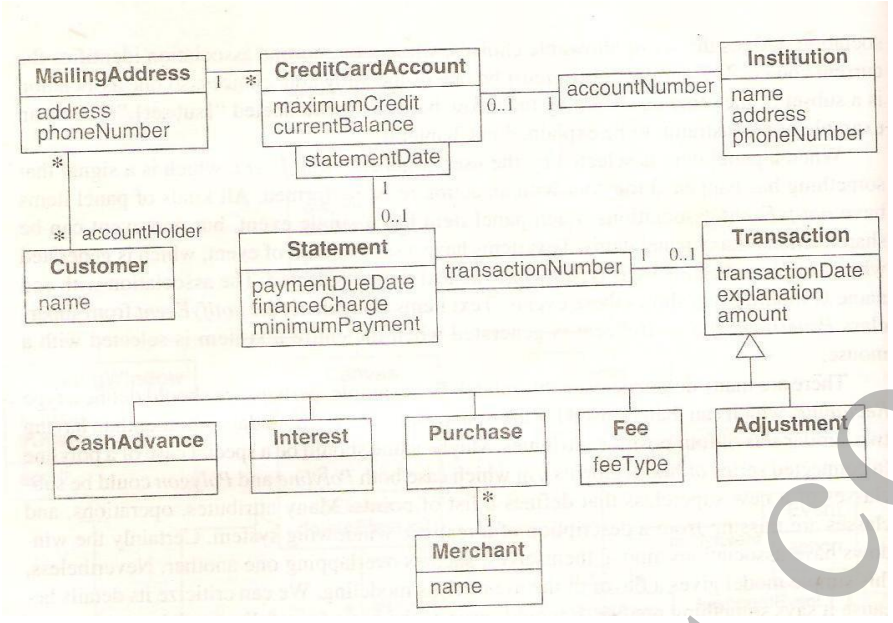
5. **Associations classes:** Given a link of an association class, you can find the constituent objects and vice versa.

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

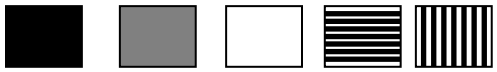
2.1.1 Enumerations

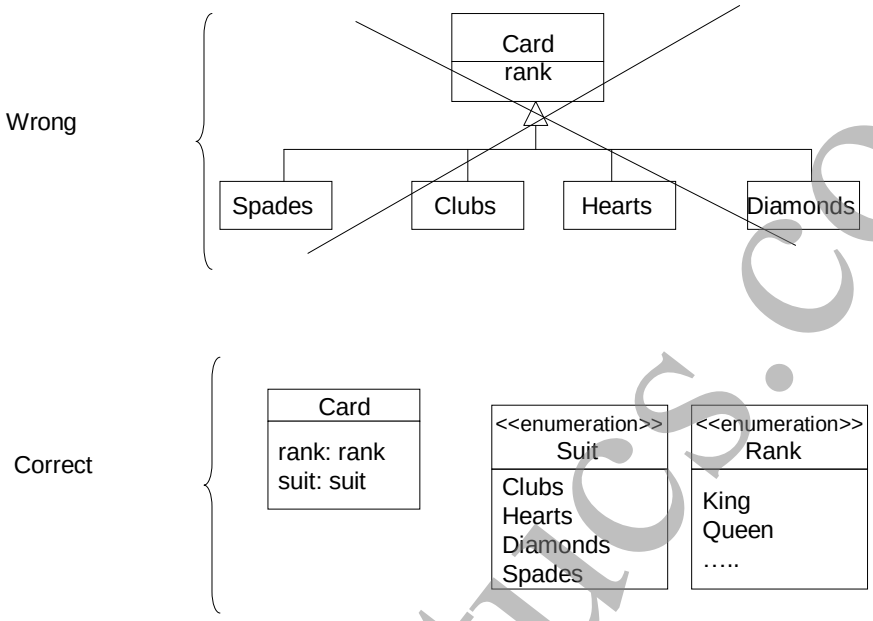
A data type is a description of values, includes numbers, strings, enumerations

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

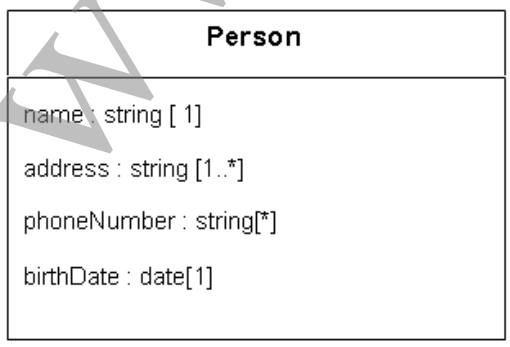




Modeling enumerations. Do not use a generalization to capture the values of an enumerated attribute

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.

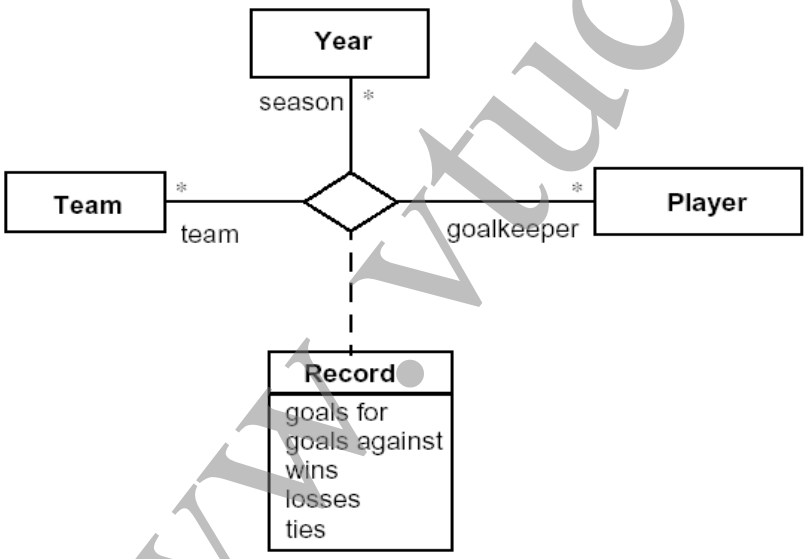
- **Extensibility**: many classes can depend on public methods, so it can be highly disruptive to change their signature. Since fewer classes depend on private, protected, and package methods, there is more latitude to change them.
- **Context**: private, protected, and package methods may rely on preconditions 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.

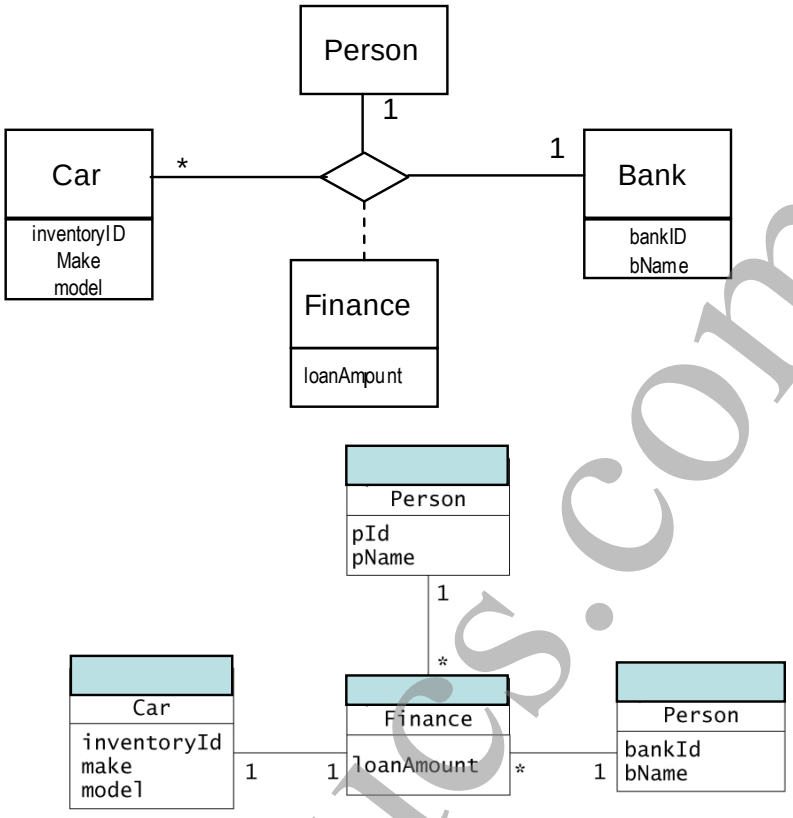
2.2 Associations ends

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

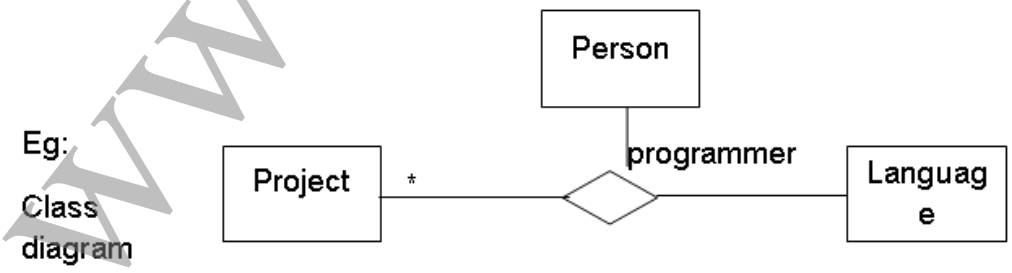
2.3 N-ary Association

- 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.





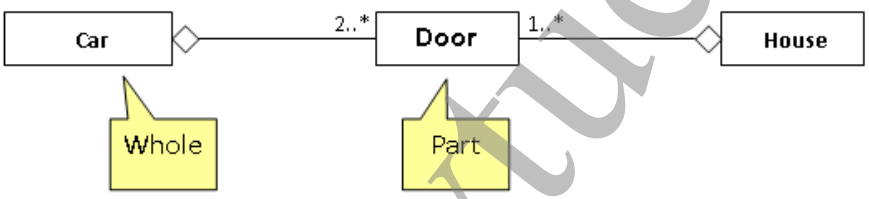
-
- The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.
- The OCL does not define notation for traversing n-ary associations.
- A typical programming language cannot express n-ary associations. 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 *see prescribed text book page no. 65 and fing no. 4.6*

2.4 Aggregation

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



2.4.1 Aggregation versus Association

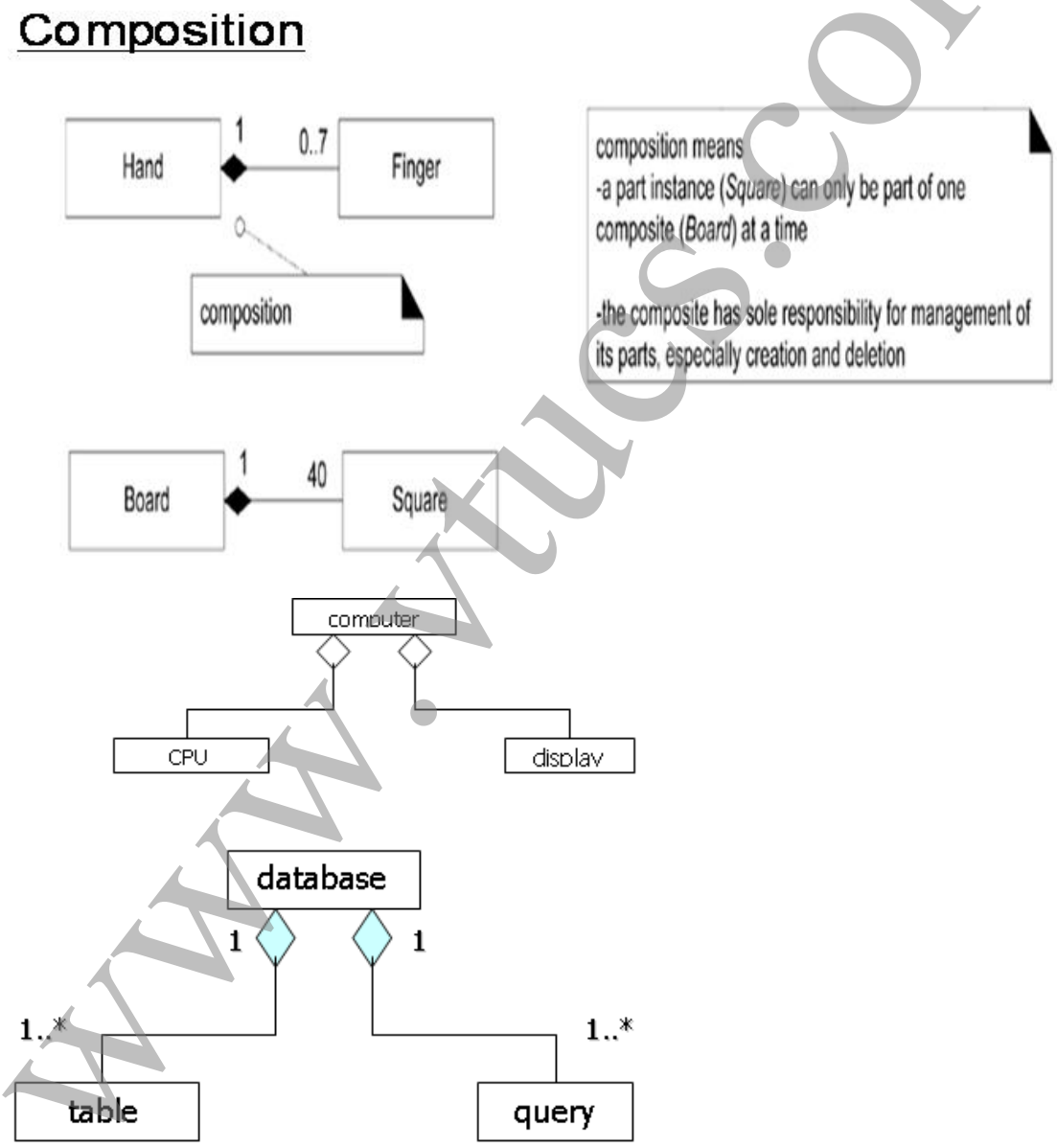
- Aggregation is a special form of association, not an independent concept.
- Aggregation adds semantic connotations.
- If two objects are tightly bound by a part-whole relationship, it is an aggregation. If the two objects are usually considered as independent, even though they may often be linked,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

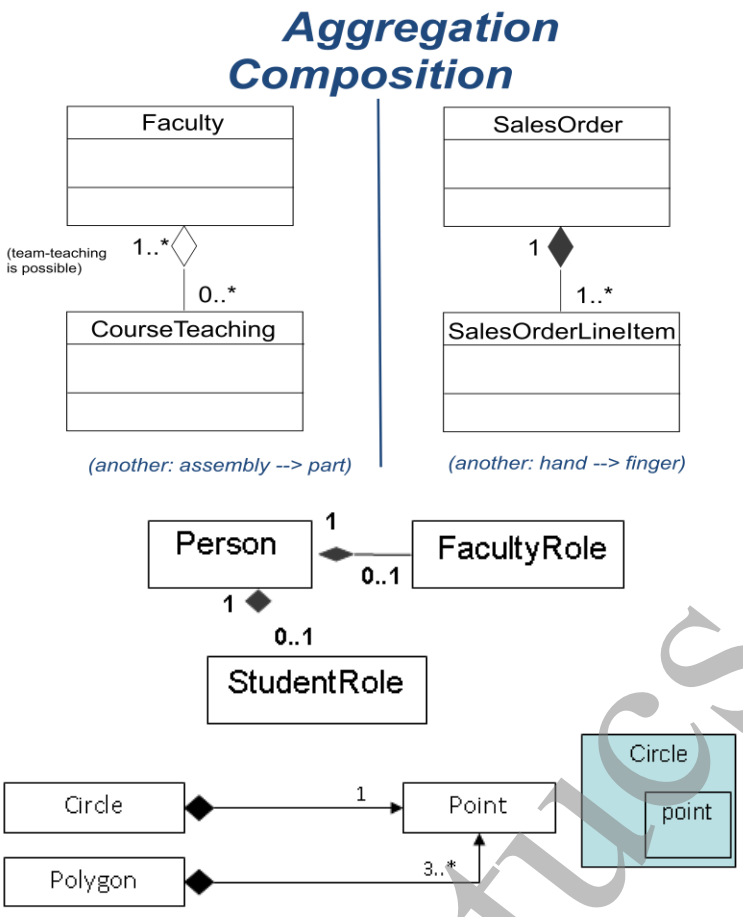
2.4.2 Aggregation versus Composition

- The UML has 2 forms of part-whole relationships: a general form called Aggregation and a more restrictive form called composition.
- Composition is a form of aggregation with two additional constraints.

- A constitute part can belong to at most one assembly.
- Once a constitute part has been assigned an assembly, it has a coincident lifetime with the assembly. Thus composition implies ownership of the parts by the whole.
- This can be convenient for programming: Deletion of an assembly object triggers deletion of all constituent objects via composition.
- Notation for composition is a small solid diamond next to the assembly class.

  Eg: see text book examples also





2.4.3 Propagation of Operations

- Propagation (triggering) is the automatic application of an operation to a network of objects when the operation is applied to some starting object.
- For example, moving an aggregate moves its parts; the move operation propagates to the parts.
- Provides concise and powerful way of specifying a continuum behavior.
- Propagation is possible for other operations including save/restore, destroy, print, lock, display.
- Notation (not an UML notation): a small arrow indicating the direction and operation name next to the affected association.

  Eg: see page no: 68 fig: 4.11

2.5 Abstract Classes

- Abstract class is a class that has no direct instances but whose descendant classes have direct instances.

- A concert class is a class that is insatiable; that is, it can have direct instances.
- A concrete class may have abstract class.
- Only concrete classes may be leaf classes in an inheritance tree.

  Eg: see text book page no: 69, 70 fig: 4.12, 4.13,4.14
- In UML notation an abstract class name is listed in an italic (or place the keyword {abstract} below or after the name).
- We can use abstract classes to define the methods that can be inherited by subclasses.
- Alternatively, an abstract class can define the signature for an operation with out supplying a corresponding method. We call this an abstract operation.
- Abstract operation defines the signature of an operation for which each concrete subclass must provid4 its own implementation.
- A concrete class may not contain abstract operations, because objects of the concrete class would have undefined operations.

2.6 Multiple Inheritance

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

2.6.1 Kinds of Multiple Inheritance

- The most common form of multiple inheritance is from sets of disjoint classes. Each subclass inherits from one class in each set.
- The appropriate combinations depend on the needs of an application.
- Each generalization should cover a single aspect.

- We should use multiple generalizations if a class can be refined on several distinct and independent aspects.
- A subclass inherits a feature from the same ancestor class found along more than one path only once; it is the same feature.
- Conflicts among parallel definitions create ambiguities that implementations must resolve. In practice, avoid such conflicts in models or explicitly resolve them, even if a particular language provides a priority rule for resolving conflicts.
- The UML uses a constraint to indicate an overlapping generalization set; the notation is a dotted line cutting across the affected generalization with keywords in braces.

  Eg: see text book page no: 71,72 fig: 4.15,4.16

2.6.2 Multiple Classification

- An instance of a class is inherently an instance of all ancestors of the class.
- For example, an instructor could be both faculty and student. But what about a Harvard Professor taking classes at MIT? There is no class to describe the combination. This is an example of multiple classification, in which one instance happens to participate in two overlapping classes.

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

2.6.3 Workarounds

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

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

   In this approach, we need not create the various combinations as explicit classes. All combinations of subclasses from the different generalization are possible.

2. **Inherit the most important class and delegate the rest:**
   Fig 4.19 preserves identity and inheritance across the most important generalization. We degrade the remaining generalization to composition and delegate their operations as in previous alternative.
3. **Nested generalization:** this approach multiplies out all possible combinations. This preserves inheritance but duplicates declarations and code and violets the spirit of OO programming.
4. **Superclasses of equal importance:** if a subclass has several superclasses, all of equal importance, it may be best to use delegation and preserve symmetry in the model.
5. **Dominant superclass:** if one superclass clearly dominates and the others are less important, preserve inheritance through this path.
6. **Few subclasses:** if the number of combinations is small, consider nested generalization. If the number of combinations is large, avoid it.
7. **Sequencing generalization sets:** if we use generalization, factor on the most important criterion first, the next most important second, and so forth.
8. **Large quantities of code:** try to avoid nested generalization if we must duplicate large quantities of code.
9. **Identity:** consider the importance of maintaining strict identity. Only nested generalization preserves this.

2.7 Metadata

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

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

2.8 Reification

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

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

2.9 Constraints

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

2.9.1 Constraints on objects

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

2.9.2 Constraints on generalization sets

- Class models capture many Constraints through their very structure. For example, the semantics of generalization imply certain structural constraints.
- With single inheritance the subclasses are mutually exclusive. Furthermore, each instance of an abstract superclass corresponds to exactly one subclass instance. Each instance of a concrete superclass corresponds to at most one subclass instance.
- The UML defines the following keyword s for generalization.

- **Disjoint:** The subclasses are mutually exclusive. Each object belongs to exactly one of the subclasses.
- **Overlapping:** The subclasses can share some objects. An object may belong to more than one subclass.
- **Complete:** The generalization lists all the possible subclasses.
- **Incomplete:** The generalization may be missing some subclasses.

2.9.3 Constraints on Links

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

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

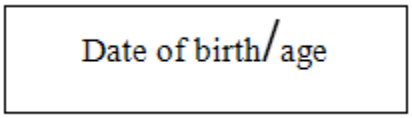
2.9.4 Use of constraints

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

2.10. Derived Data

- A derived element is a function of one or more elements, which in turn may be derived. A derived element is redundant, because the other elements completely

determine it. Ultimately, the derivation tree terminates with base elements. Classes, associations, and attributes may be derived. The notation for a derived element is a slash in front of the element name along with constraint that determines the derivation.



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

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

2.11 Packages

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

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

State Modeling

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

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

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

Events represent external stimuli and states represent values objects.

Events

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

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

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

Events include error conditions as well as normal conditions.

Three types of events:

- signal event,
- change event,
- time event.

Signal Event

- A signal is an explicit one-way transmission of information from one object to another.
- It is different form a subroutine call that returns a value.

- An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.
- A signal event is the event of sending or receiving a signal (concern about receipt of a signal).

The difference between signal and signal event

a signal is a message between objects

a signal event is an occurrence in time.

Eg:

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

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.

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

States

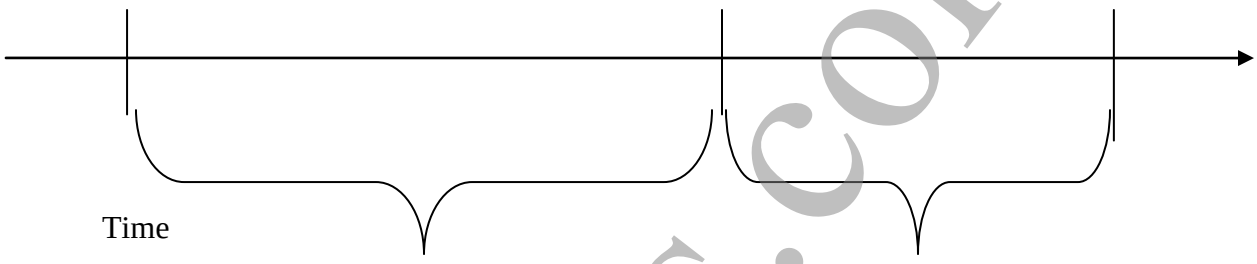
- A state is an abstraction of the values and links of an object.
- Sets of values and links are grouped together into a state according to the gross behavior of objects
- UML notation for state- a rounded box Containing an optional state name, list the state name in boldface, center the name near the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

- A state specifies the response of an object to input events.
- All events are ignored in a state, except those for which behavior is explicitly prescribed.

Event vs. States

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

- Eg: **power turned on** **power turned off** **power turned on**



- **Powered** **Not powered**

A state corresponds to the interval between two events received by an object.

The state of an object depends on past events.

Both events and states depend on the level of abstraction.

State Alarm ringing on a watch

- **State** : *Alarm Ringing*
- **Description**: alarm on watch is ringing to indicate target time
- **Event sequence that produces the state**:
  setAlarm (targetTIme)
  any sequence not including *clearAlarm*
  when (*currentTime = targetTime*)
- **Condition that characterrizes the state:**
  alarm = on, alarm set to *targetTime*, *targetTime<=currentTime<=targetTime+20 sec*, and no button has been pushed since *targetTime*
- **Events accepted in the state:**

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

42

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

a guard condition is checked only once	a change event is checked continuously
UML notation for a transition is a line	may include event label in italics

followed by guard condition in square brackets from the origin state to the target state

an arrowhead points to the target state.

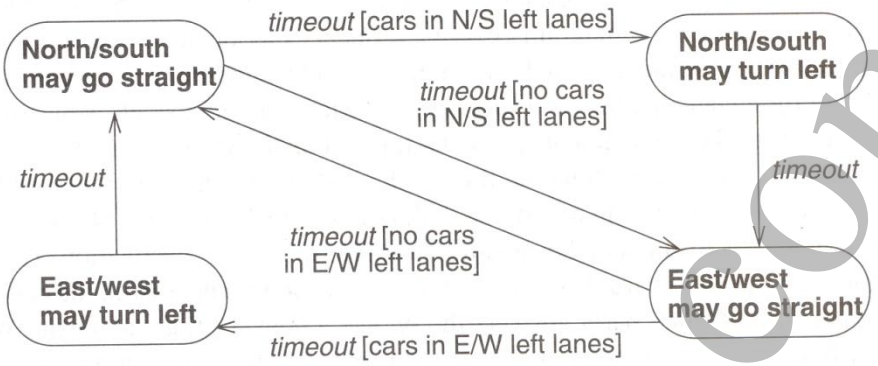


Figure 5.7 Guarded transitions. A transition is an instantaneous change from one state to another. A guard condition is a boolean expression that must be true in order for a transition to occur.

State Diagram

- A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states.
- A state diagram specifies the state sequence caused by event sequences.
- State names must be unique within the scope of a state diagram.
- All objects in a class execute the state diagram for that class, which models their common behavior.
- A state model consists of multiple state diagrams one state diagram for each class with important temporal behavior.
- State diagrams interact by passing events and through the side effects of guard conditions.
- UML notation for a state diagram is a rectangle with its name in small pentagonal tag in the upper left corner.
- The constituent states and transitions lie within the rectangle.
- States do not totally define all values of an object.
- If more than one transition leaves a state, then the first event to occur causes the corresponding transition to fire.

- If an event occurs and no transition matches it, then the event is ignored.
- If more than one transition matches an event, only one transition will fire, but the choice is nondeterministic.

Eg: Sample state diagram

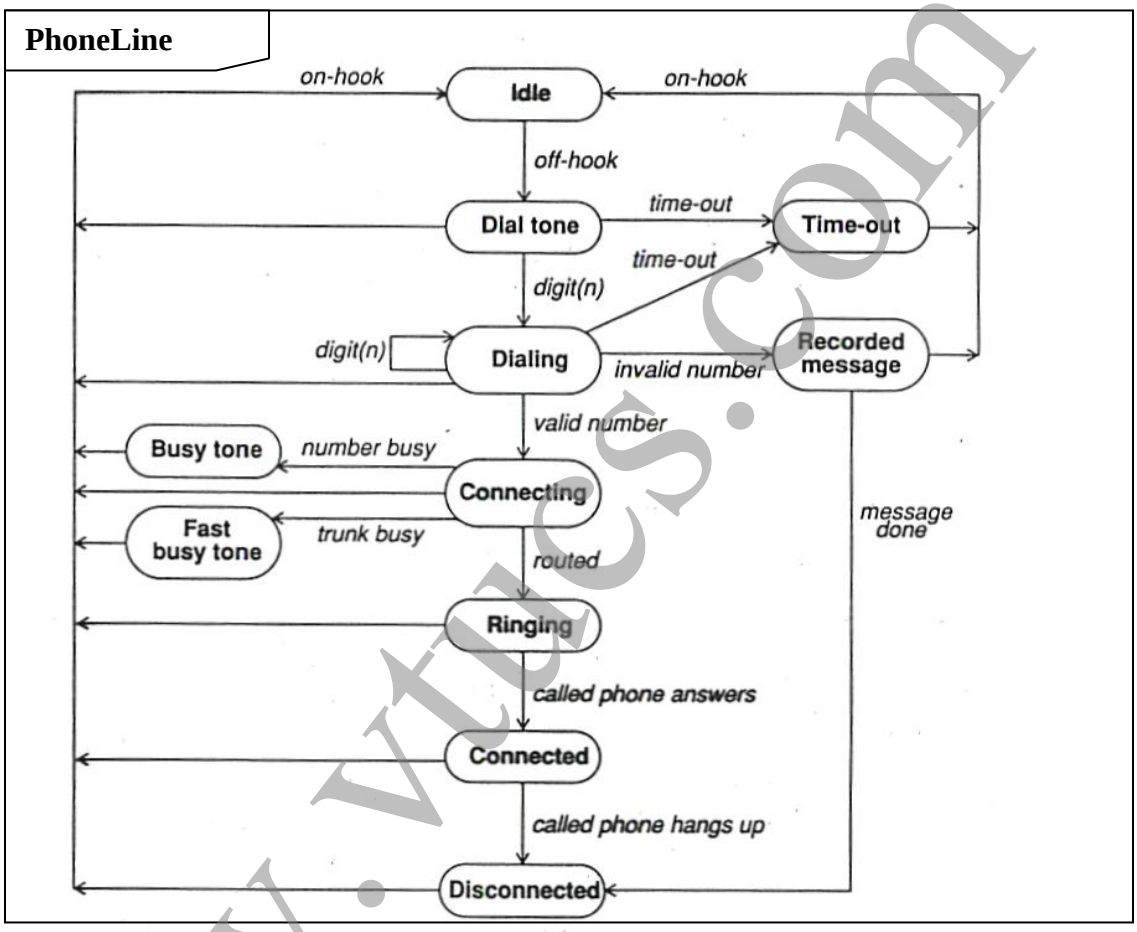


Figure 5.5 State diagram for phone line

One shot state diagrams

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

Diagram for the [hone line is a continuous loop

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

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

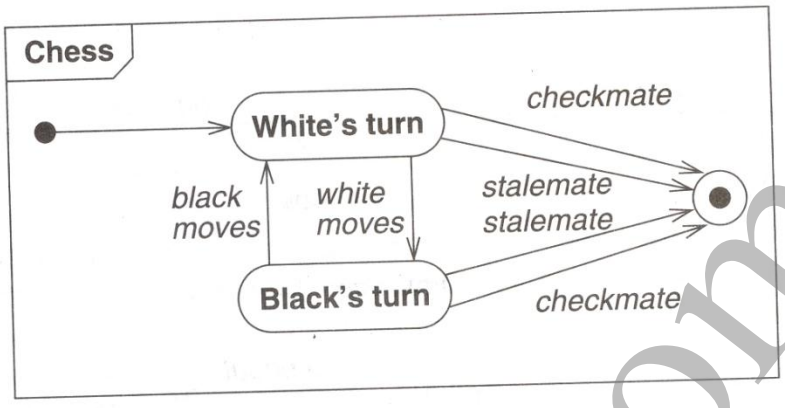


Figure 5.9 State diagram for chess game. One-shot diagrams represent objects with finite lives.

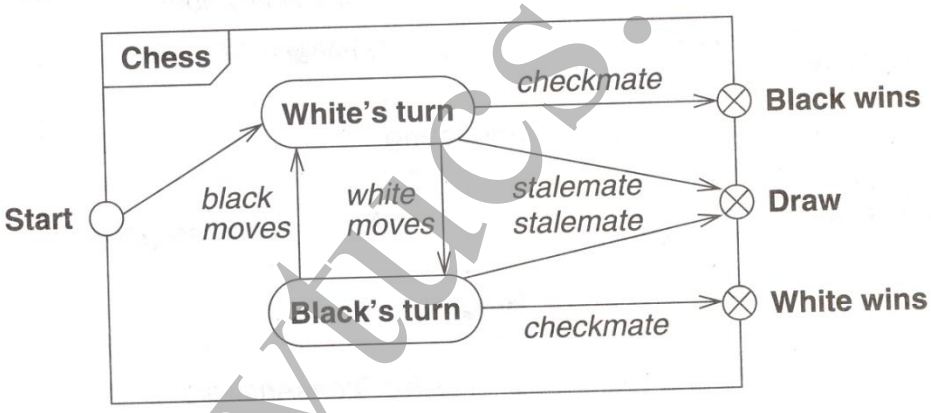


Figure 5.10 State diagram for chess game. You can also show one-shot diagrams by using entry and exit points.

5.4.3 Summary of Basic State Diagram Notation

Figure 5.11 summarizes the basic UML syntax for state diagrams.

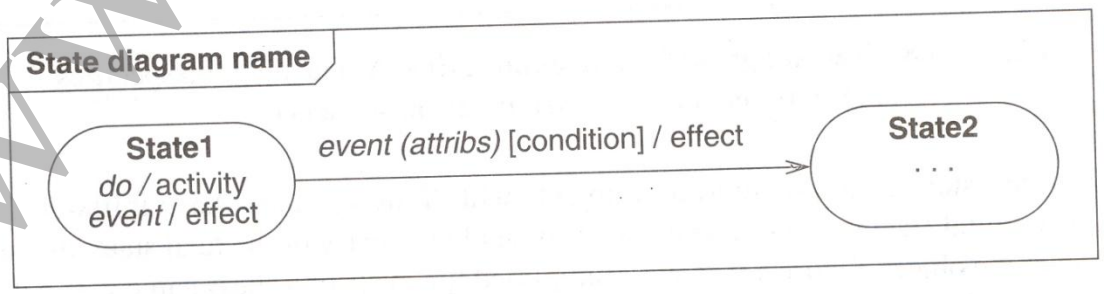


Figure 5.11 Summary of basic notation for state diagrams.

optional name. A special notation is ava

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
- Object ◊ n independent boolean atribute◊2n states

Expanding states

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

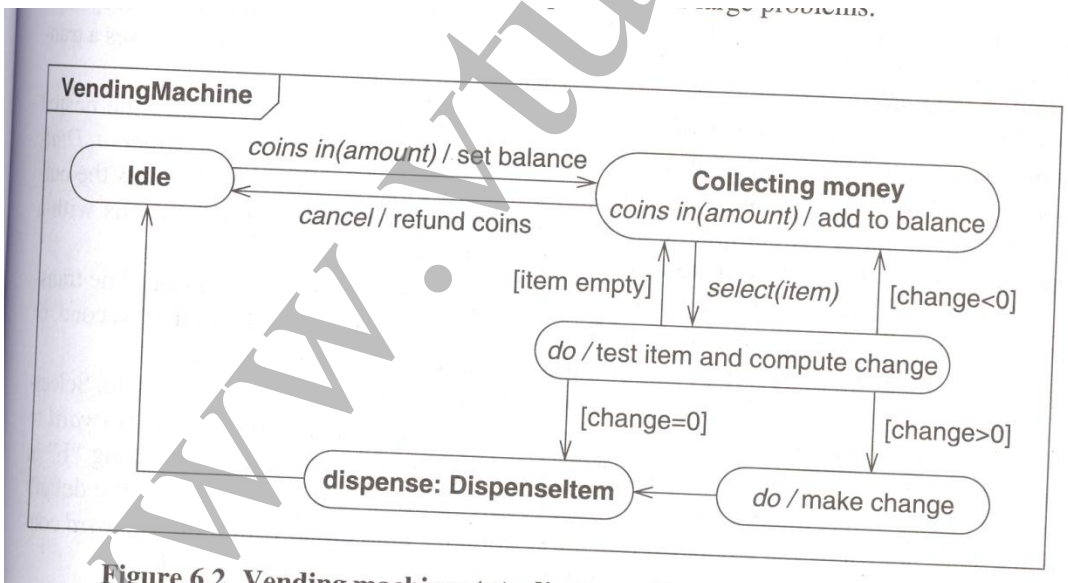


Figure 6.2 Vending machine state diagram. You can simplify state diagrams by using subdiagrams.

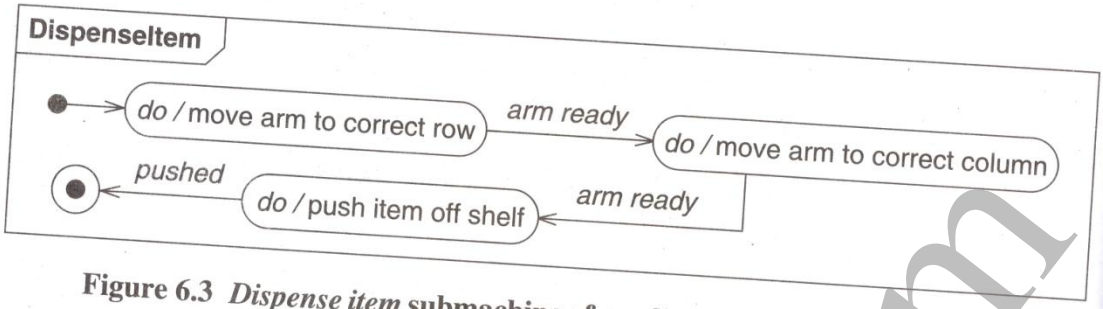


Figure 6.3 ***Dispense item*** **submachine of vending machine**. A lower-level state diagram can elaborate a state.

6.2 Nested States

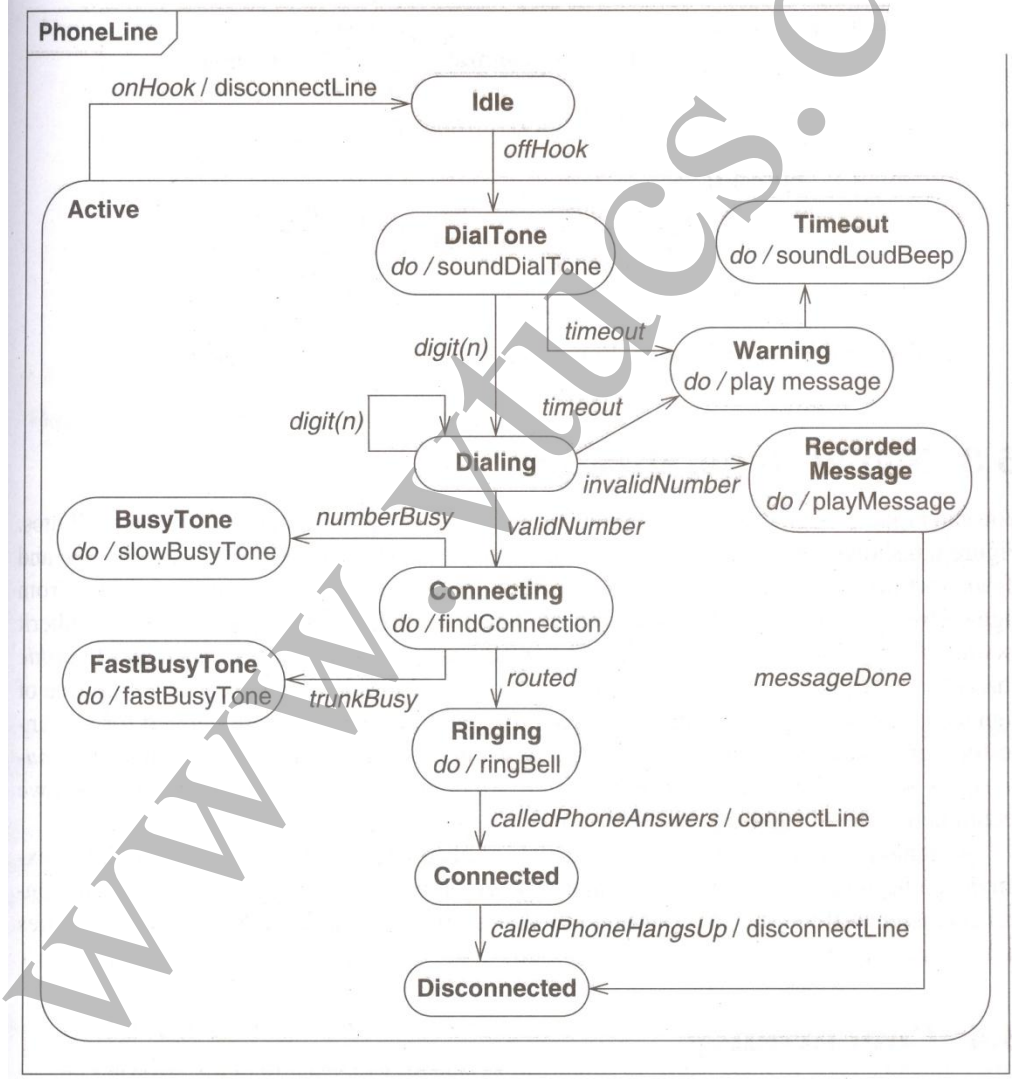


Figure 6.4 **Nested states for a phone line**. A nested state receives the outgoing transitions of its enclosing state.

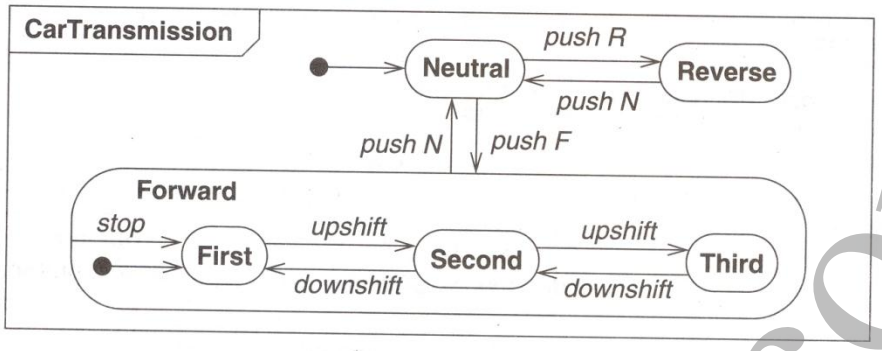


Figure 6.5 Nested states. You can nest states to an arbitrary depth.

Signal generalization

You can organize signals into generalization hierarchy with inheritan ce of signal attributes

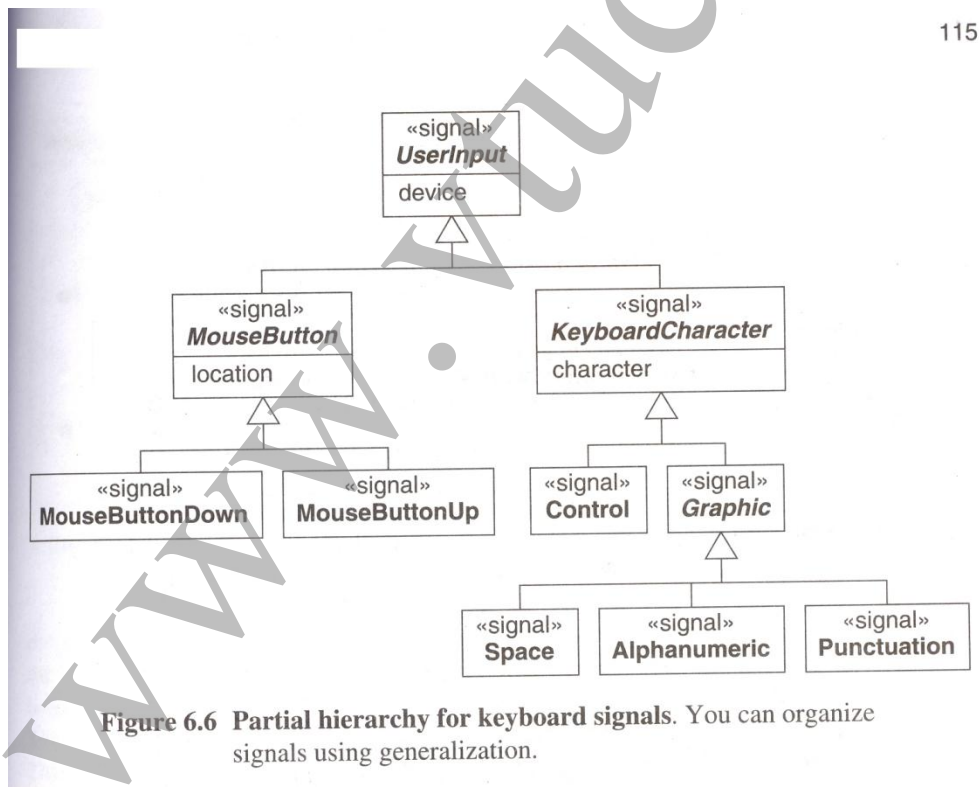
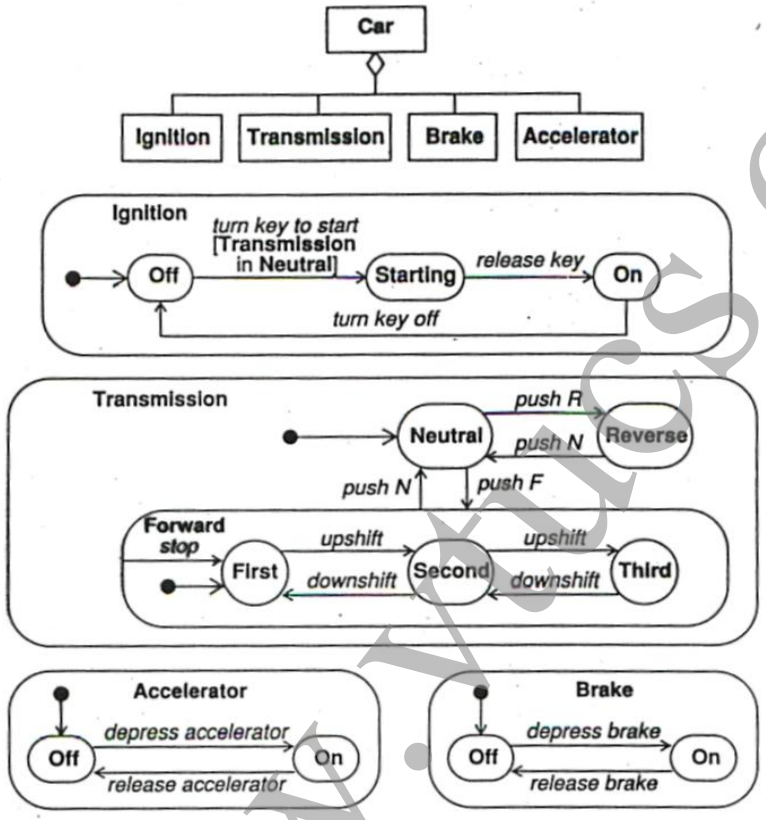


Figure 6.6 Partial hierarchy for keyboard signals. You can organize signals using generalization.

- 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

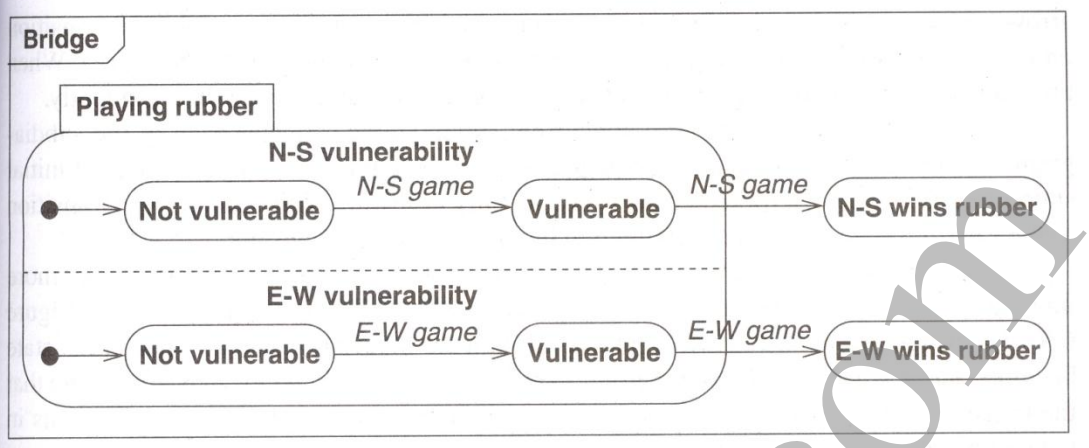


Figure 6.8 Bridge game with concurrent states. You can partition some objects into subsets of attributes or links, each of which has its own subdiagram.

3 synchronization of concurrent activities

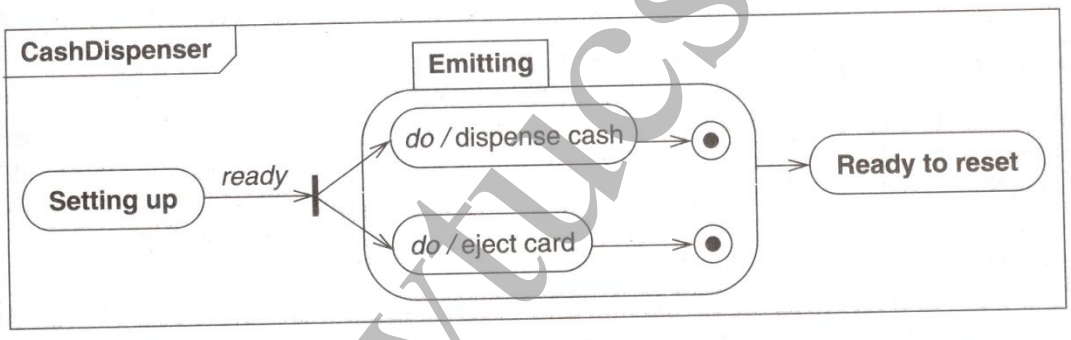


Figure 6.9 Synchronization of control. Control can split into concurrent activities that subsequently merge.

Additional diagrams:

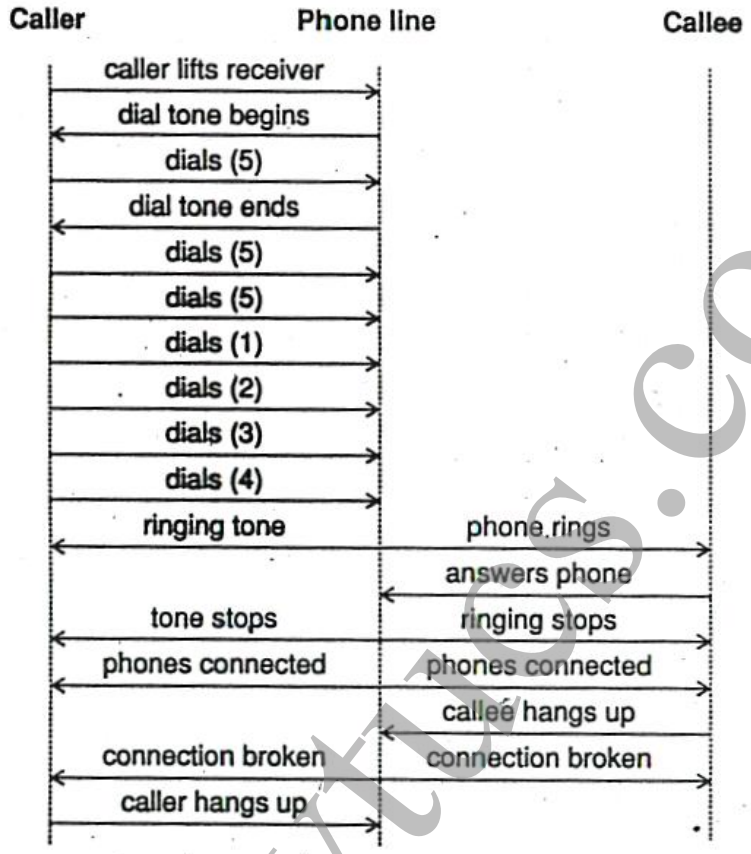
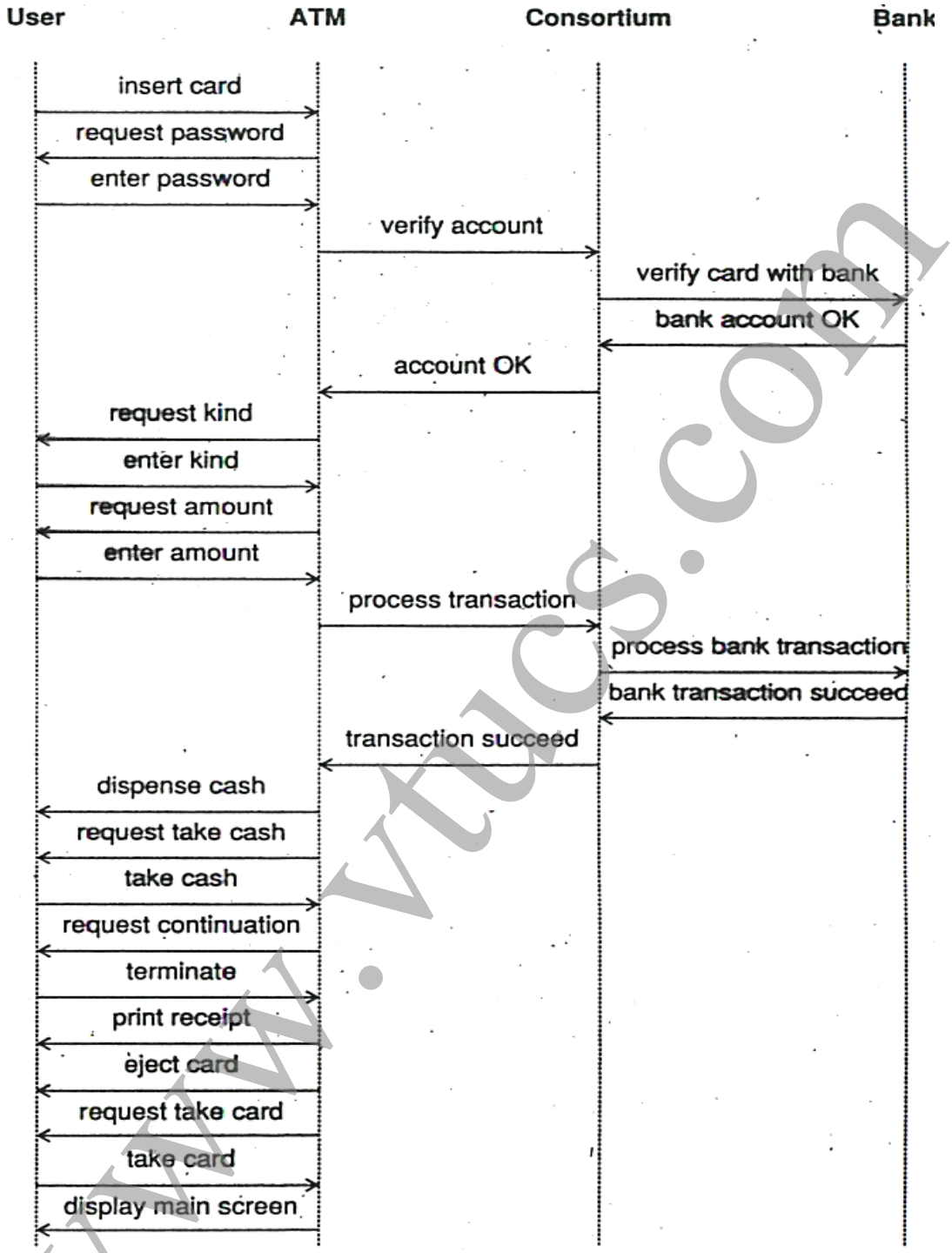


Figure 5.3 Event trace for phone call



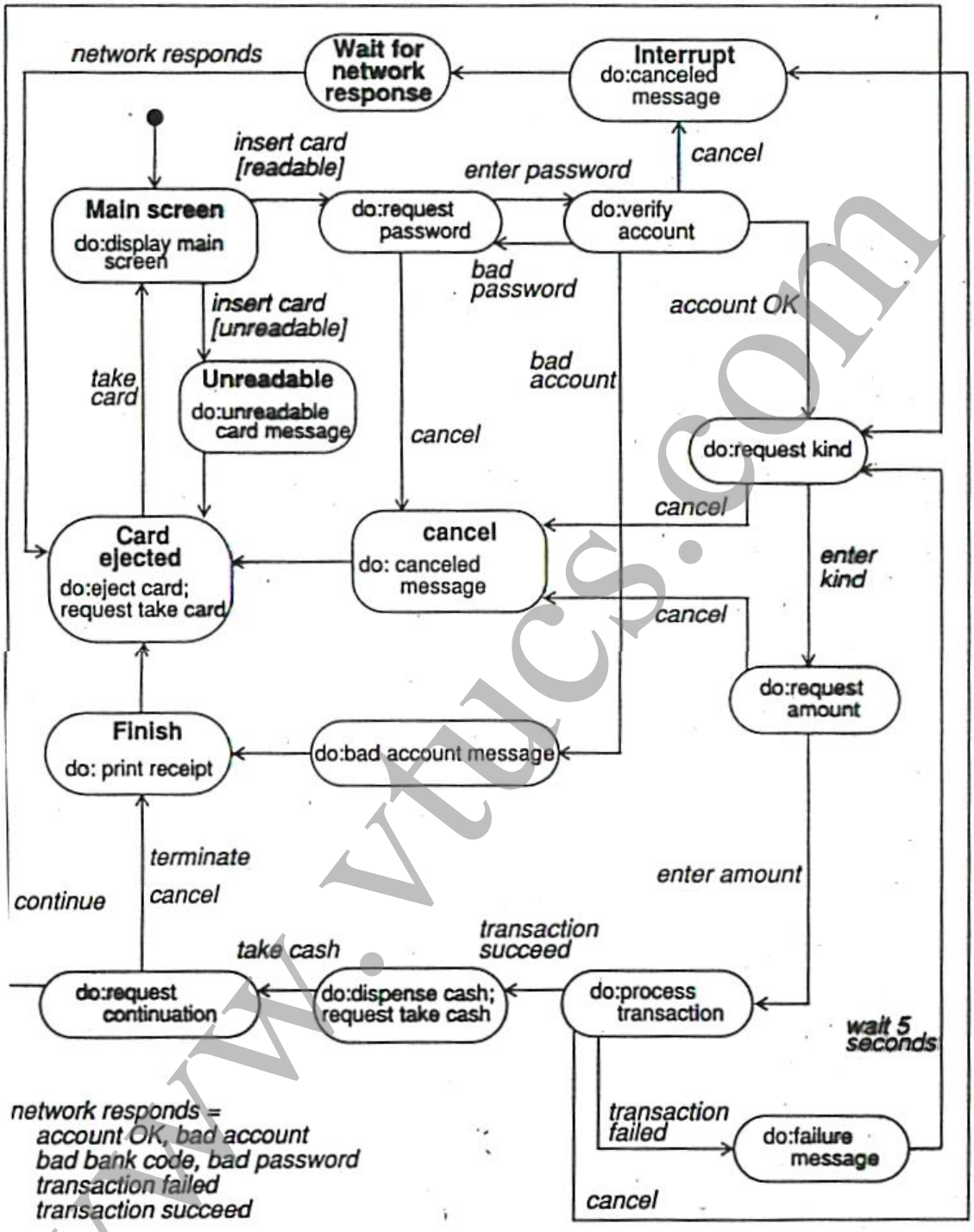


Figure 8.20 State diagram for class *ATM*

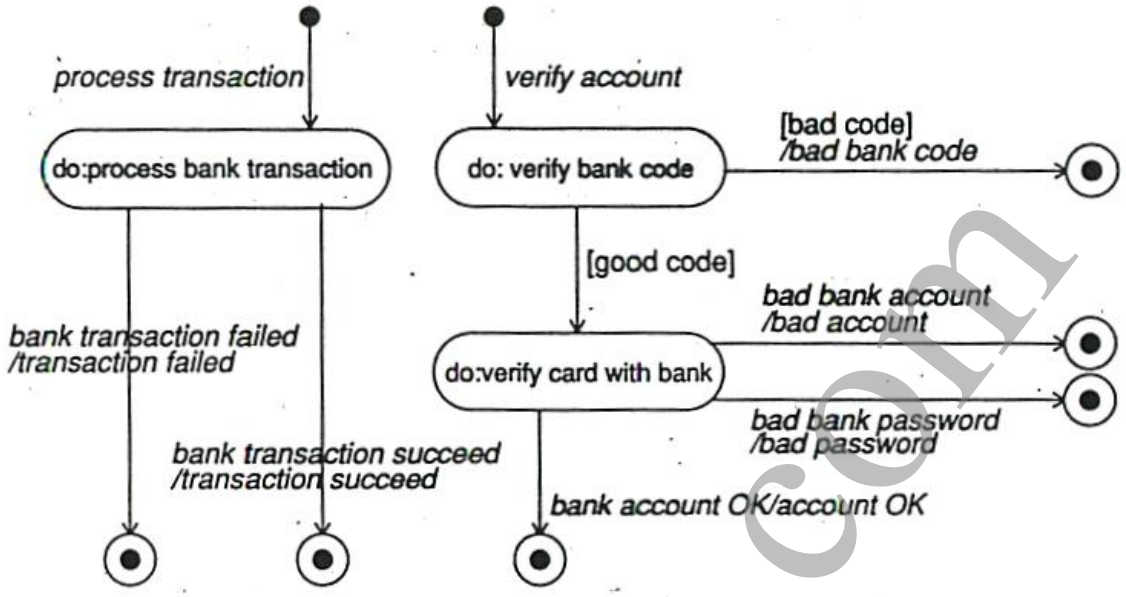


Figure 8.21 State diagram for class *Consortium*

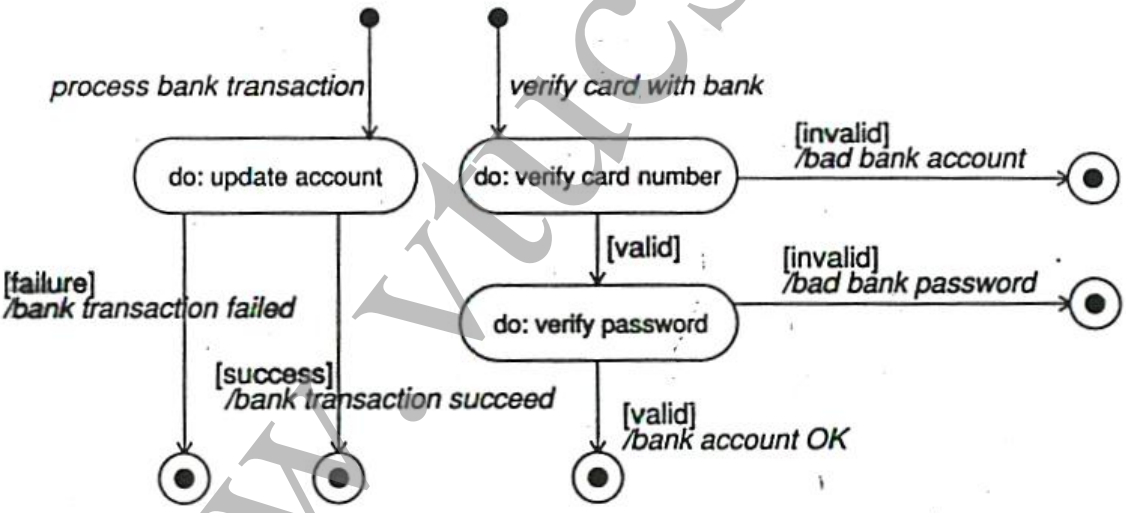


Figure 8.22 State diagram for class *Bank*

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.

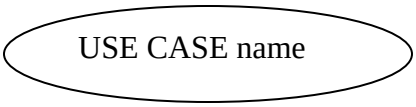
- A use case involves a sequence of messages among the system and its actors.
- Error conditions are also part of a use case.
- A use case brings together all of the behavior relevant to a slice of system functionality.

Use Case Description (see text book fug 7.2)

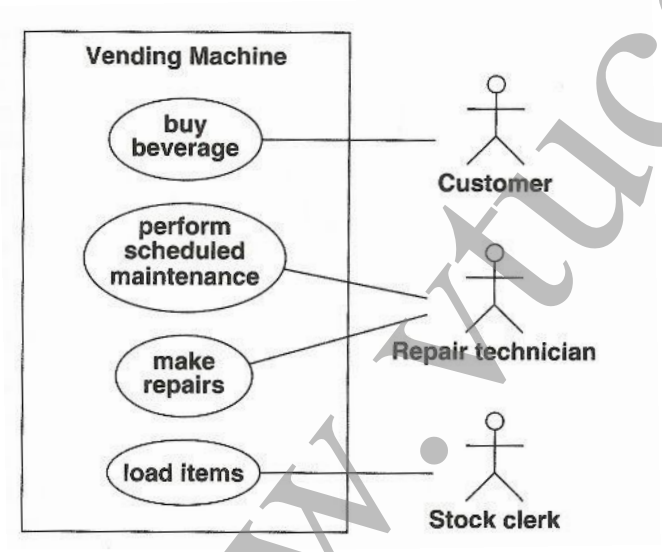
- Use Case Name
- Summary
- Actors
- Preconditions
- Description
- Exception
- Postcondition

- **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

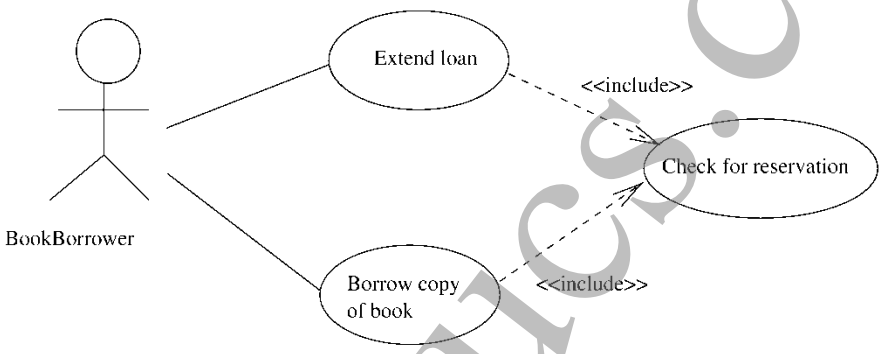
- Include Relationship
  - Incorporate one use case within the behavior sequence of another use case.
- Extend Relationship
  - Add incremental behavior to a use case.

- 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)

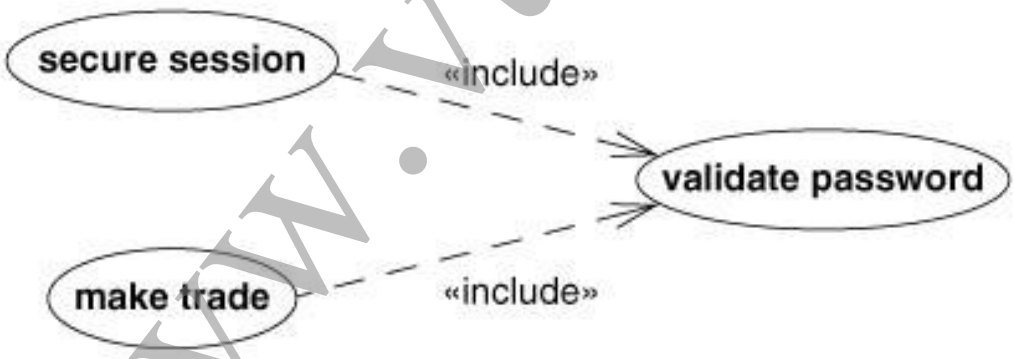
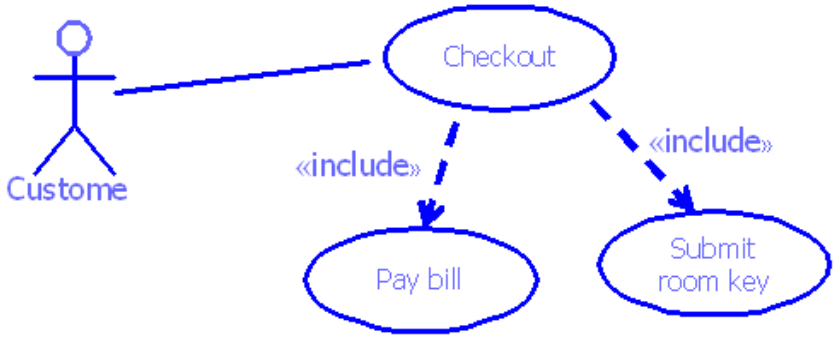
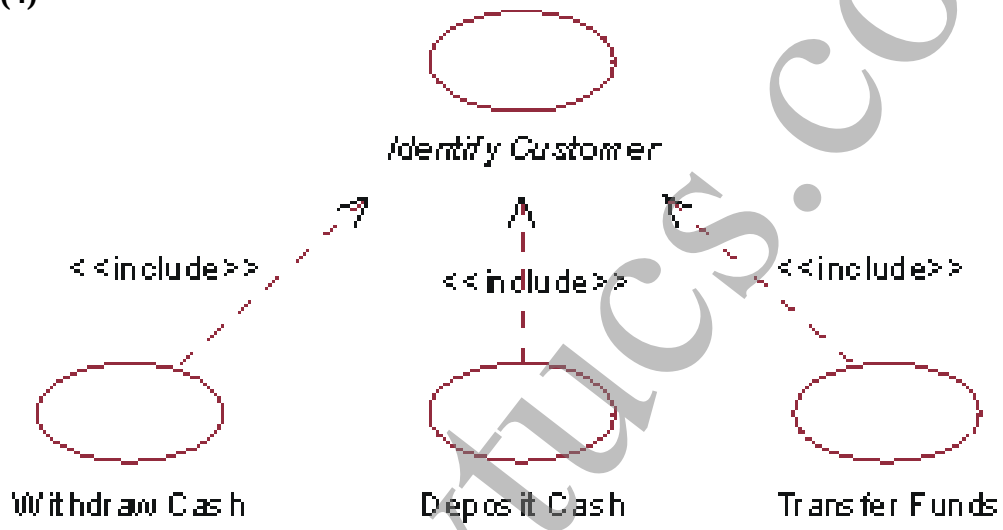


Figure 8.1 Use case inclusion. The *include* relationship lets a base use case incorporate behavior from another use case.

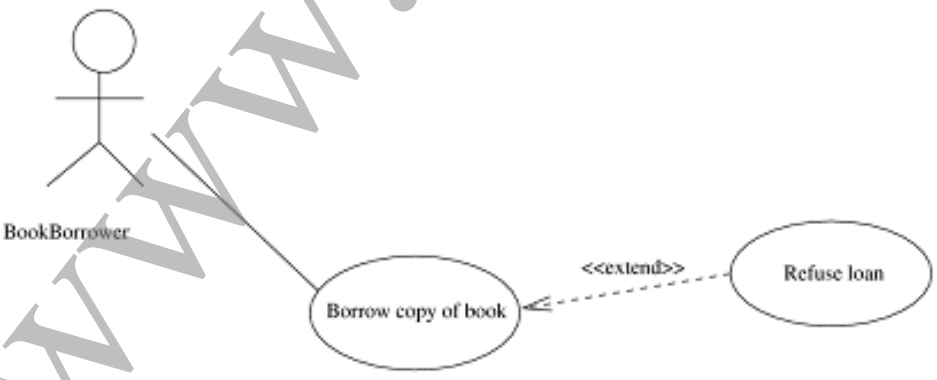
(3)



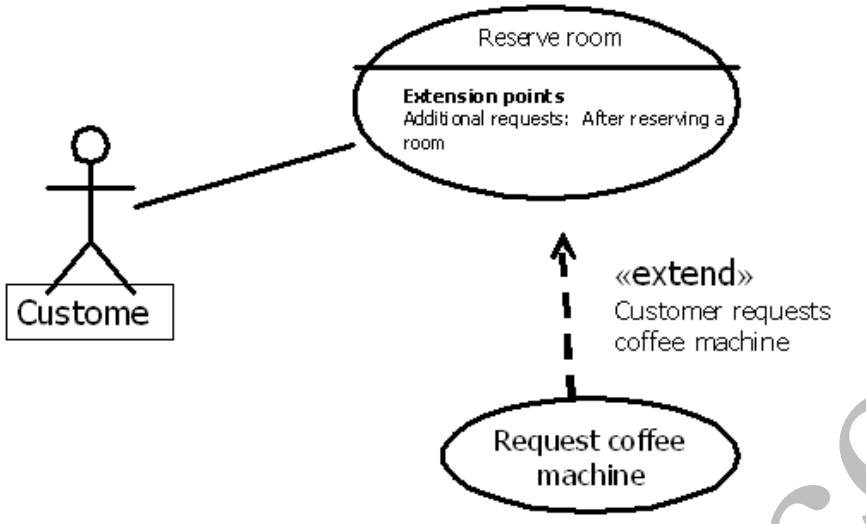
(4)



Extend Relationship examples:
<<extend>> for special cases:
(1)



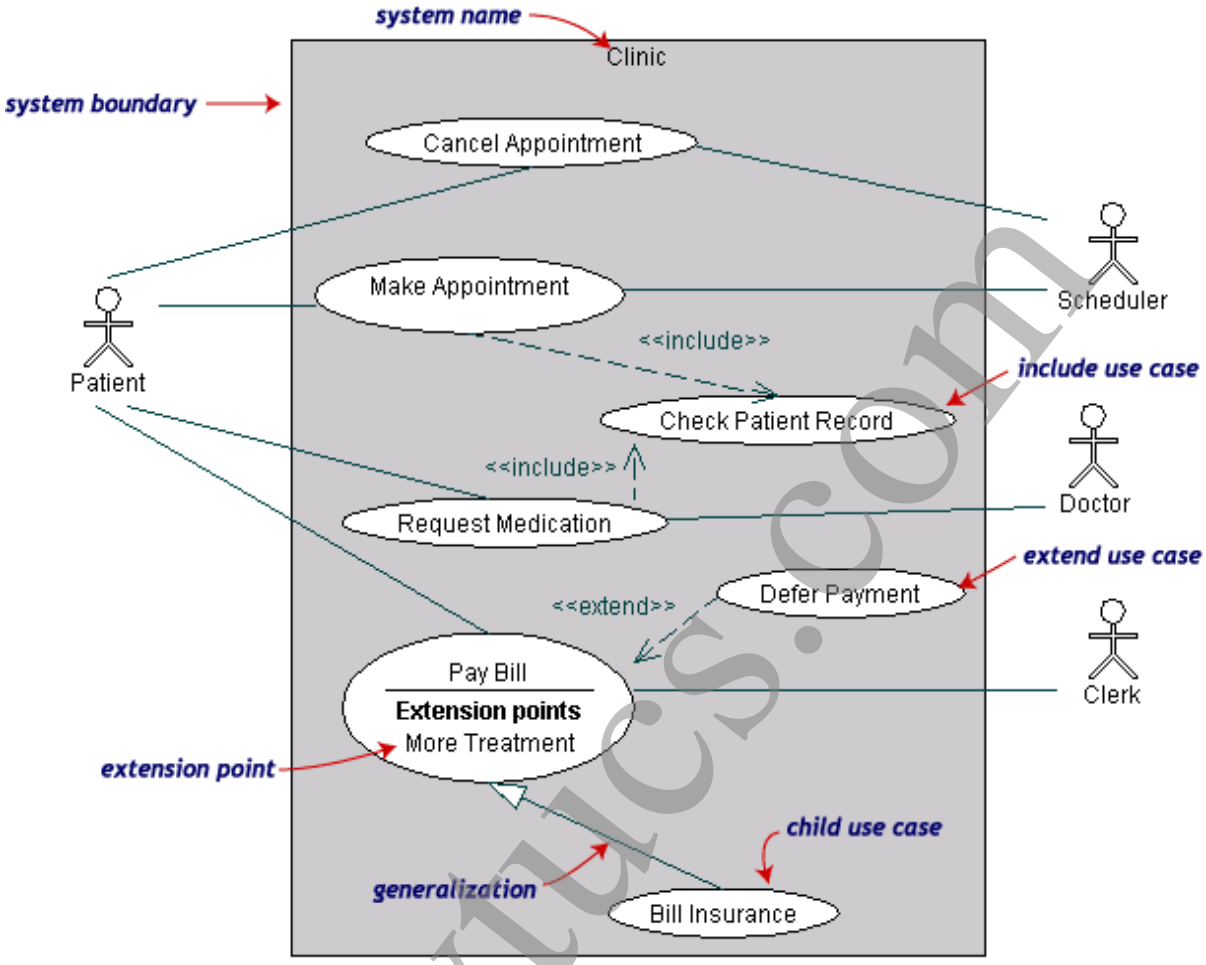
(2)



(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.

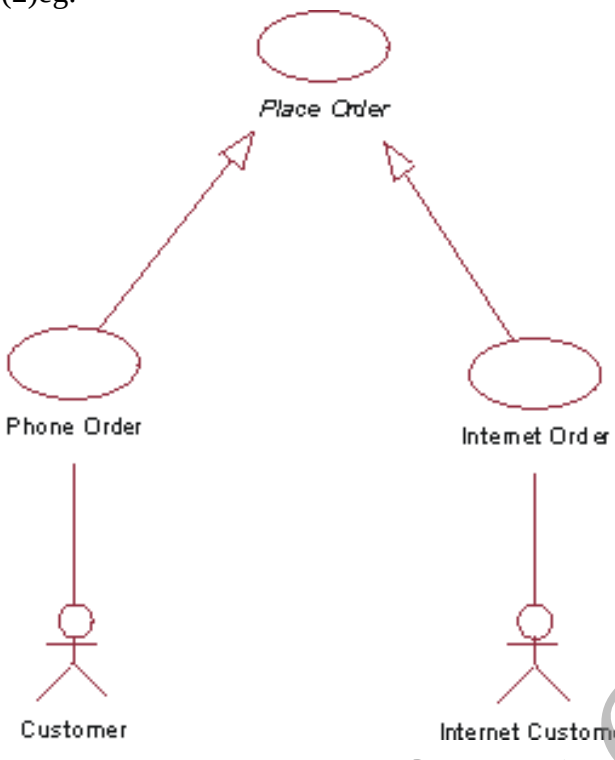
Medical Clinic: «include» and «extend»



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

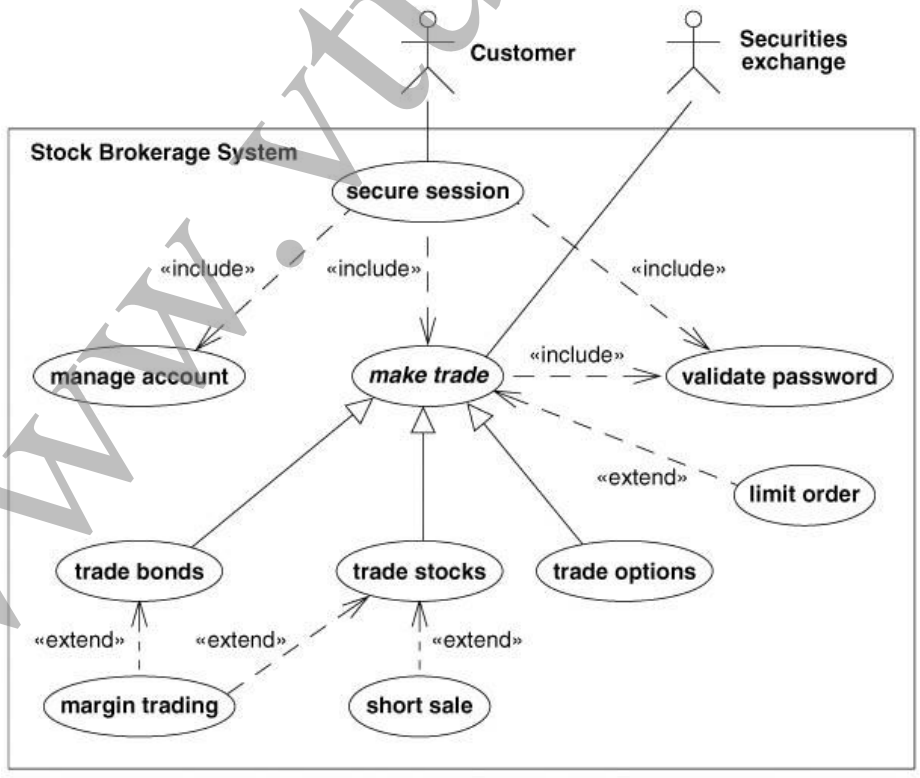


Figure 8.4 Use case relationships. A single use case diagram may combine several kinds of relationships.

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.

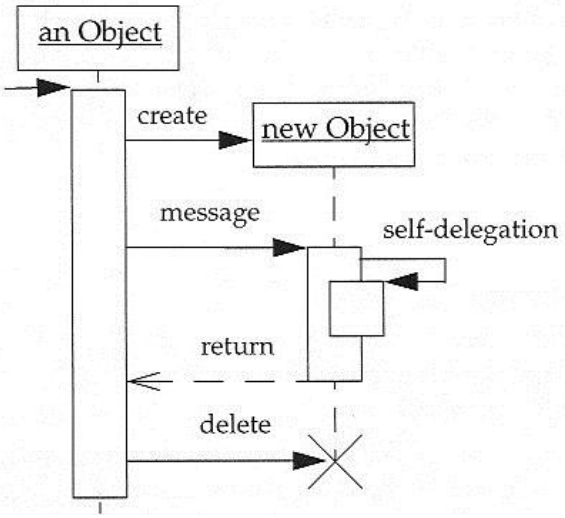
Figure 7.4 Scenario for a session with an online stock broker. A scenario is a sequence of events that occurs during one particular execution of a system.

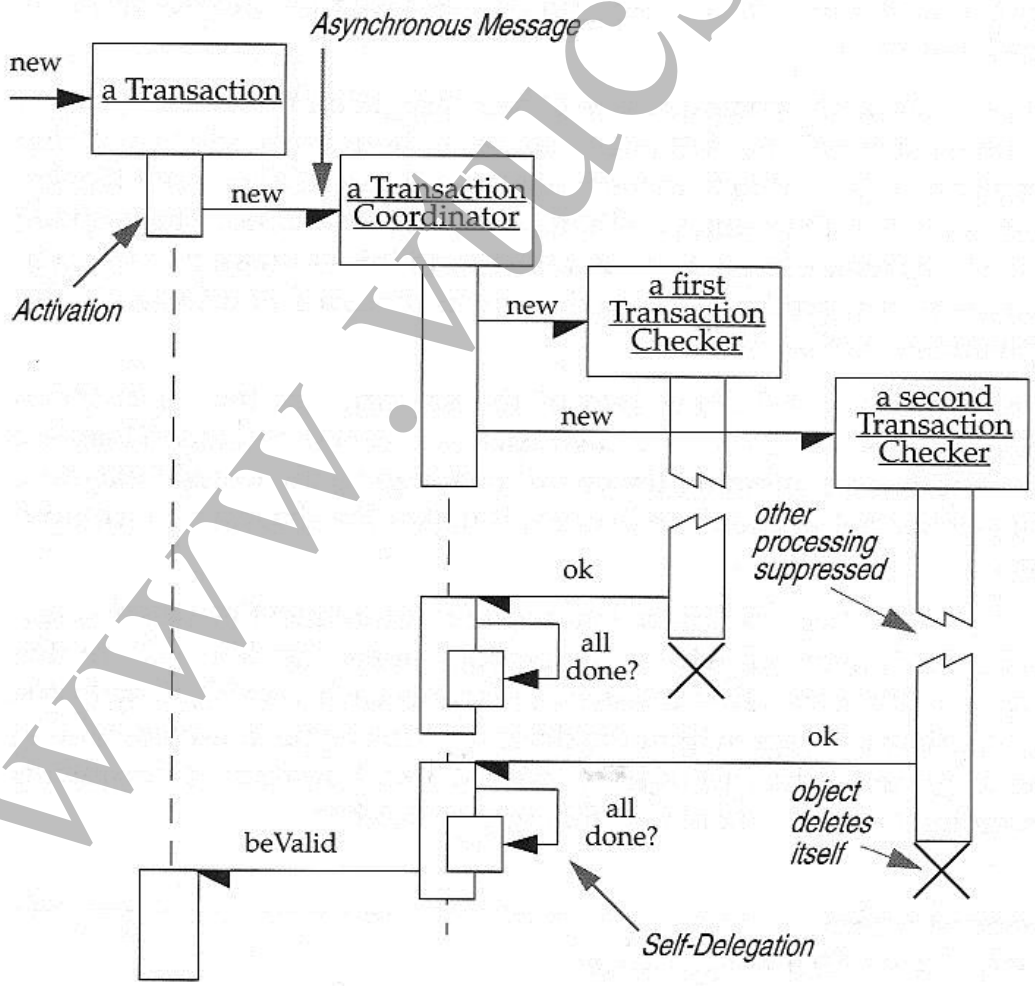
Interaction Modeling 22

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.

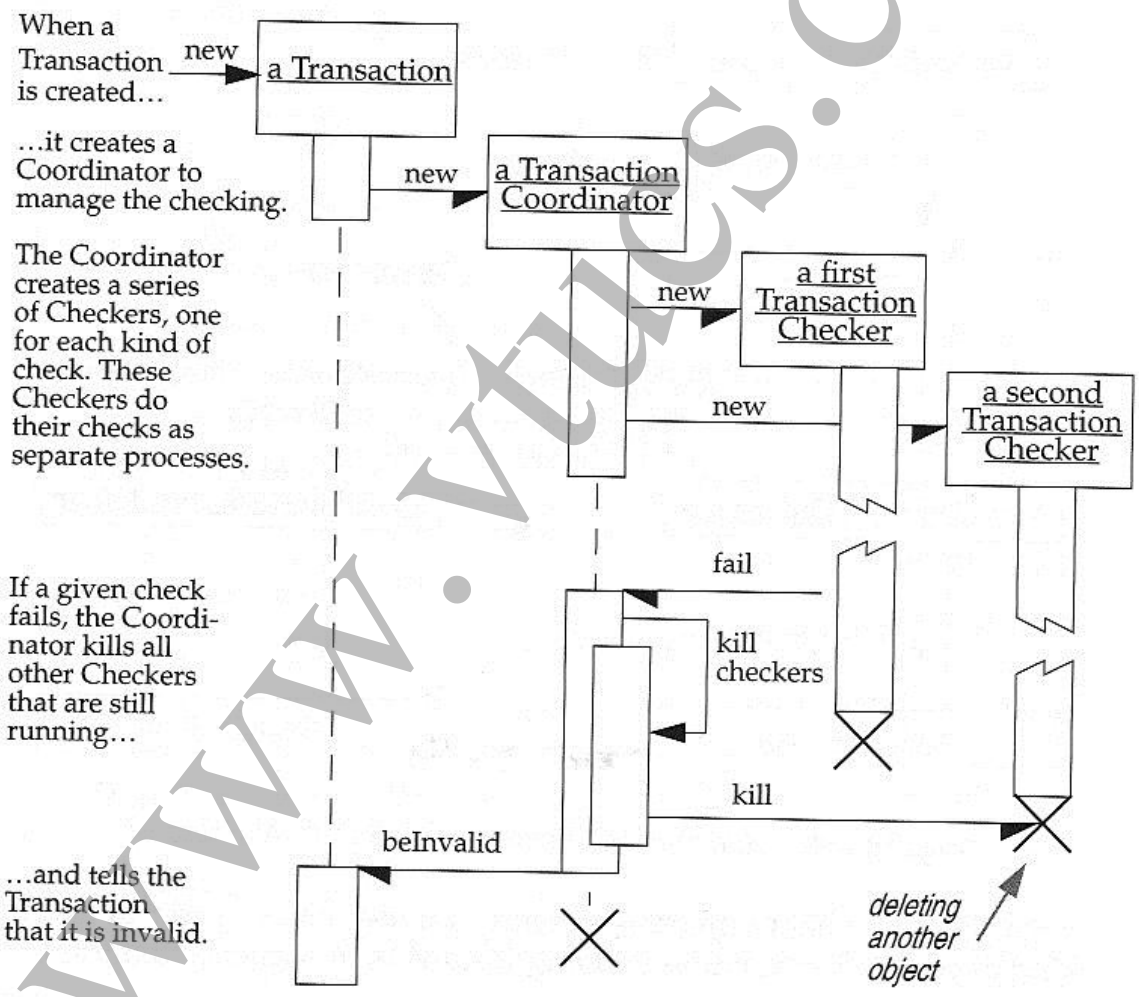
Sequence Diagram





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



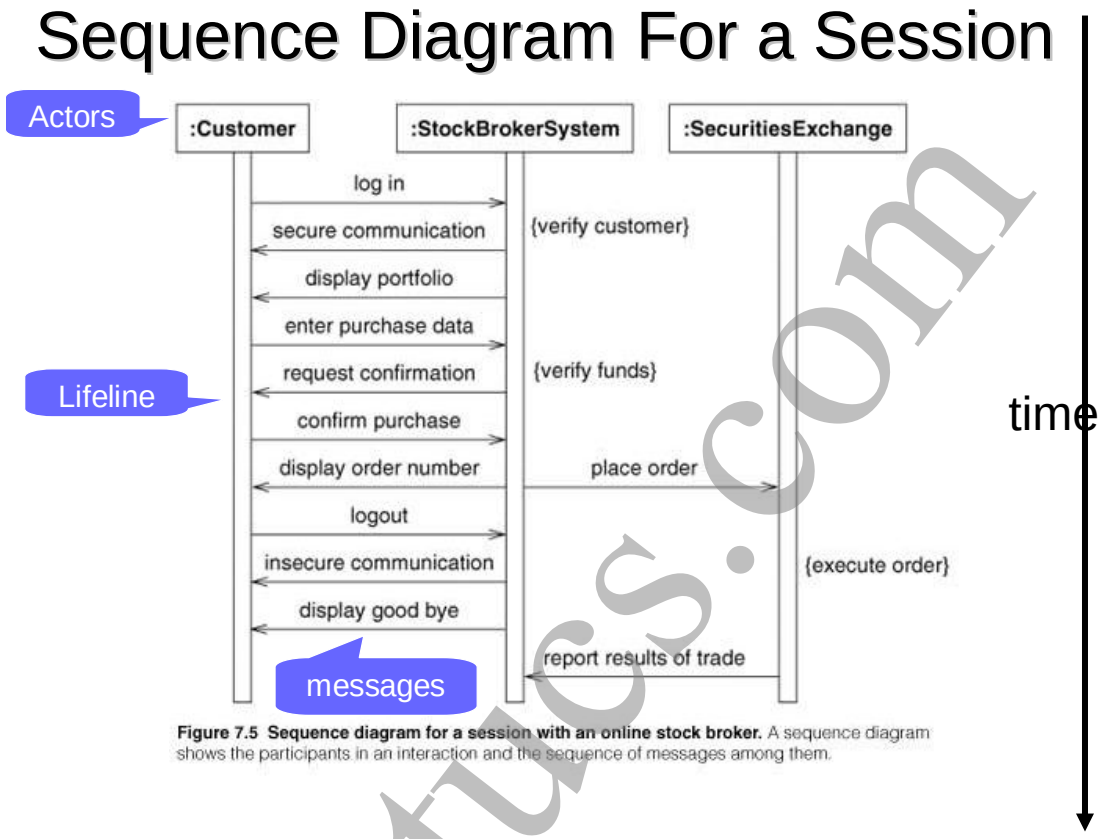


Figure 7.5 Sequence diagram for a session with an online stock broker. A sequence diagram shows the participants in an interaction and the sequence of messages among them.

A stock purchase

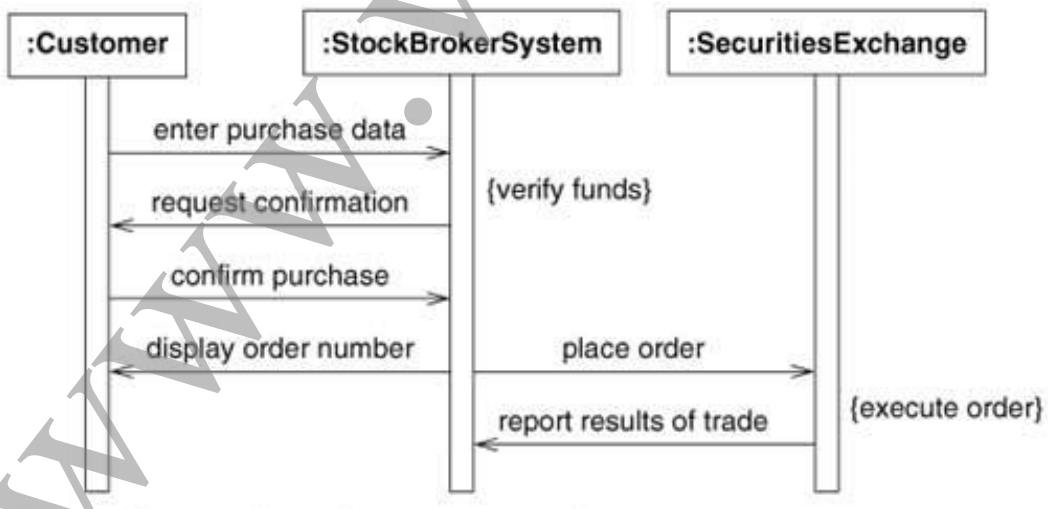


Figure 7.6 Sequence diagram for a stock purchase. Sequence diagrams can show large-scale interactions as well as smaller, constituent tasks.

A stock quote

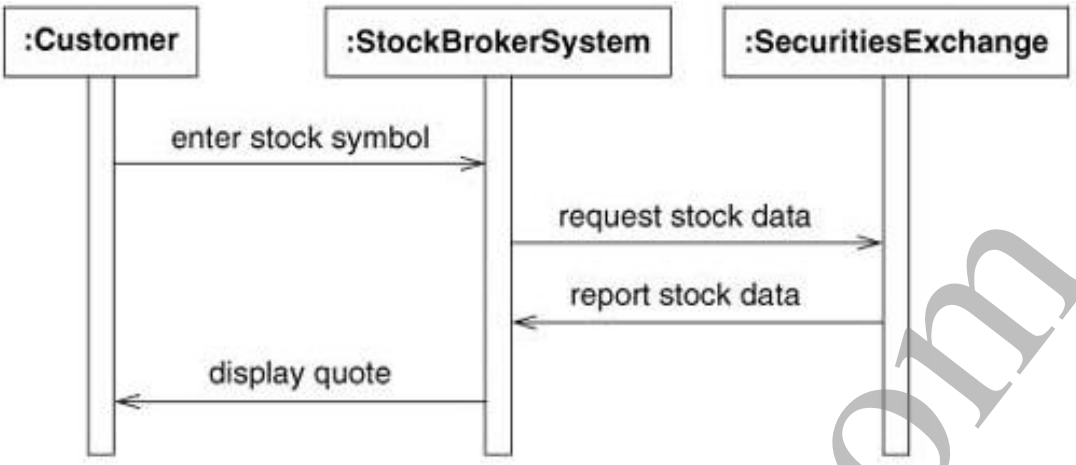


Figure 7.7 Sequence diagram for a stock quote.

A exception case

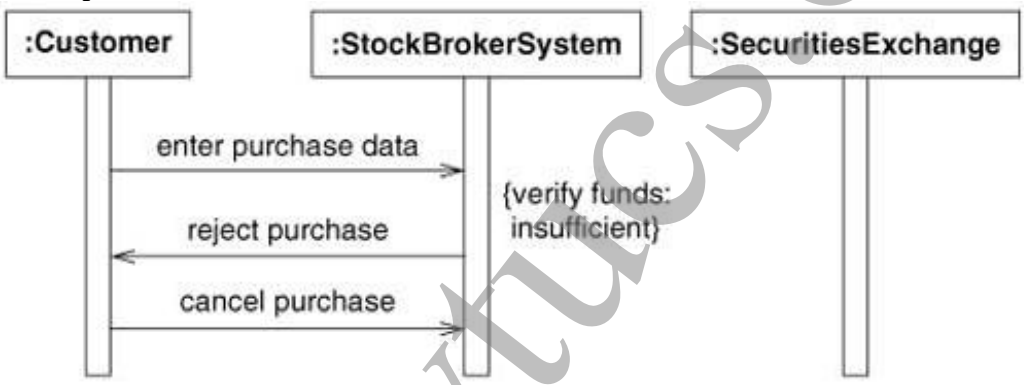


Figure 7.8 Sequence diagram for a stock purchase that fails.

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.

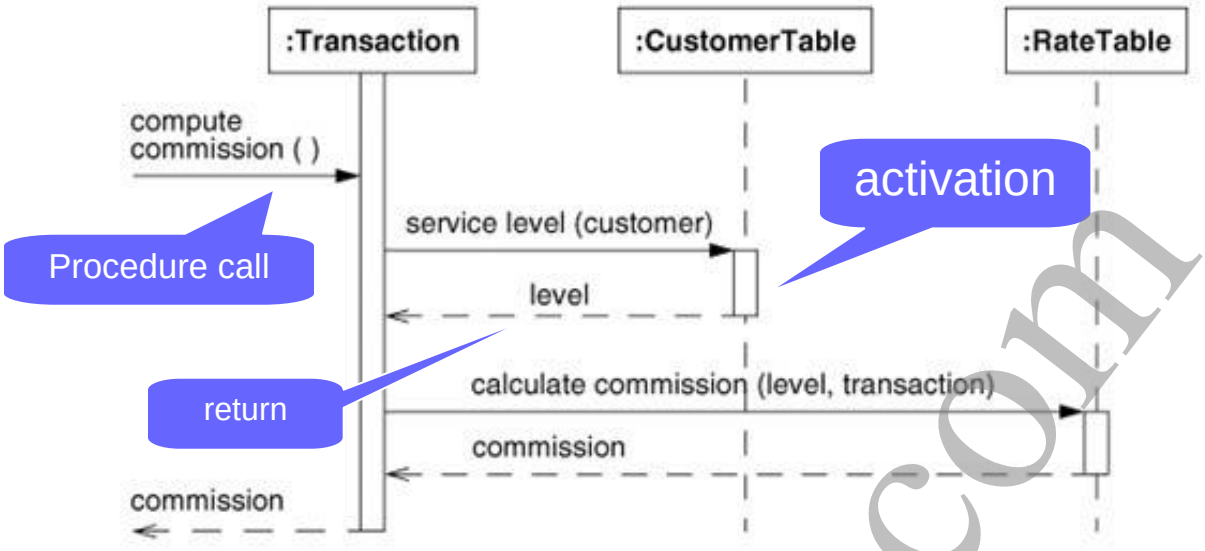


Figure 8.5 Sequence diagram with passive objects. Sequence diagrams can show the implementation of operations.

Sequence Diagrams with Transient Objects

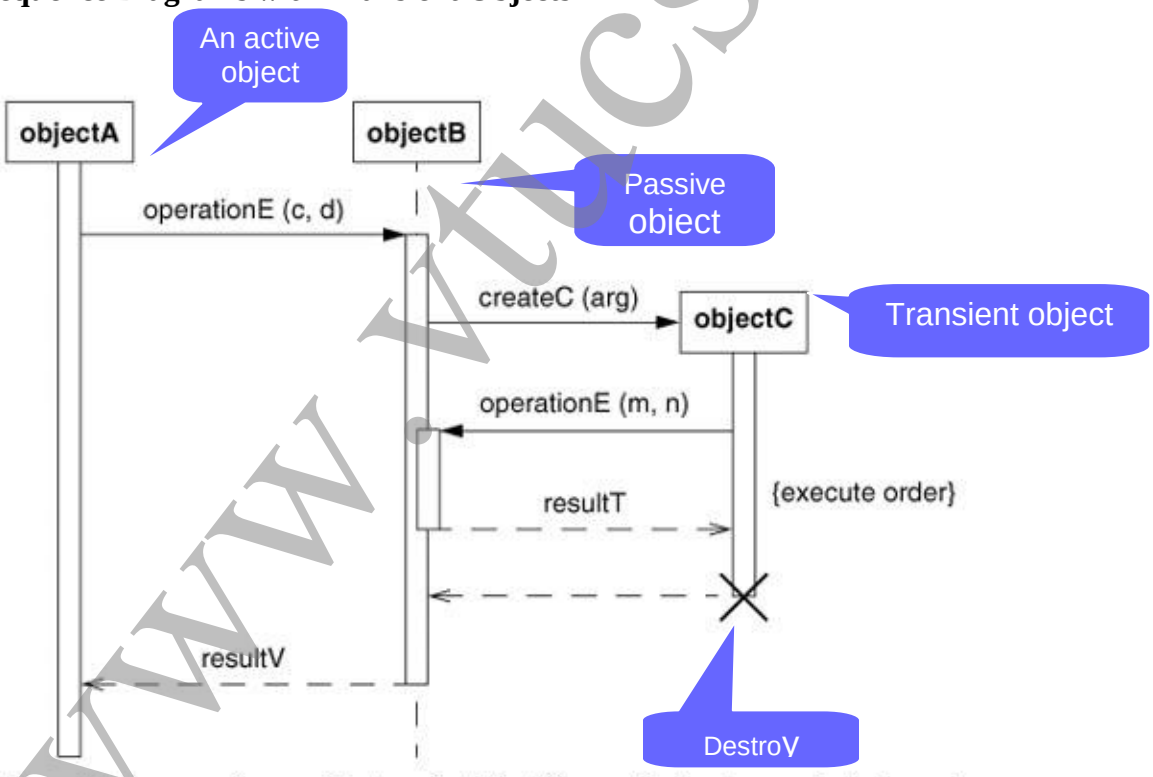


Figure 8.6 Sequence diagram with a transient object. Many applications have a mix of active and passive objects. They create and destroy objects.

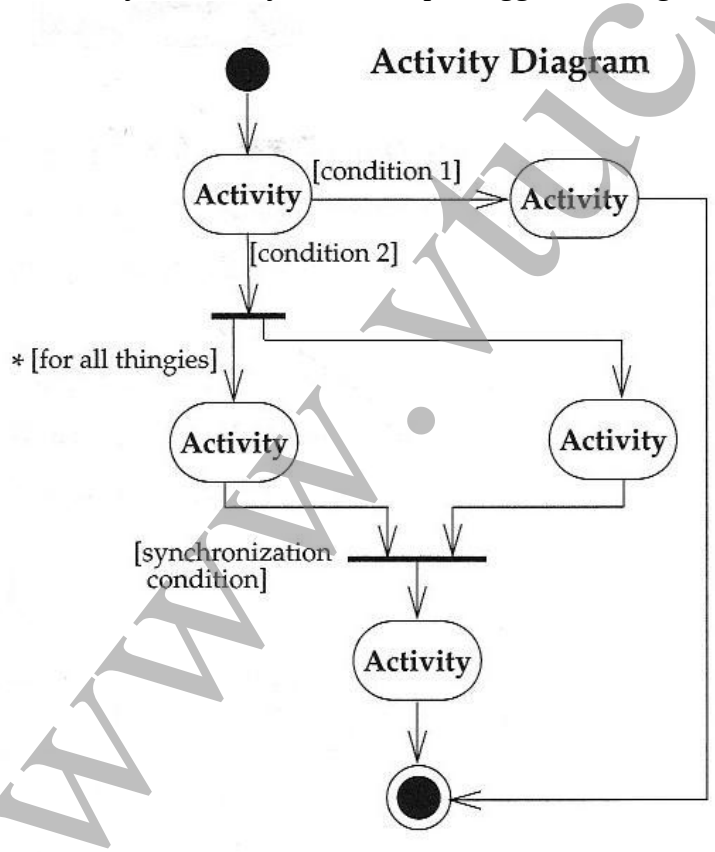
Activity Models

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

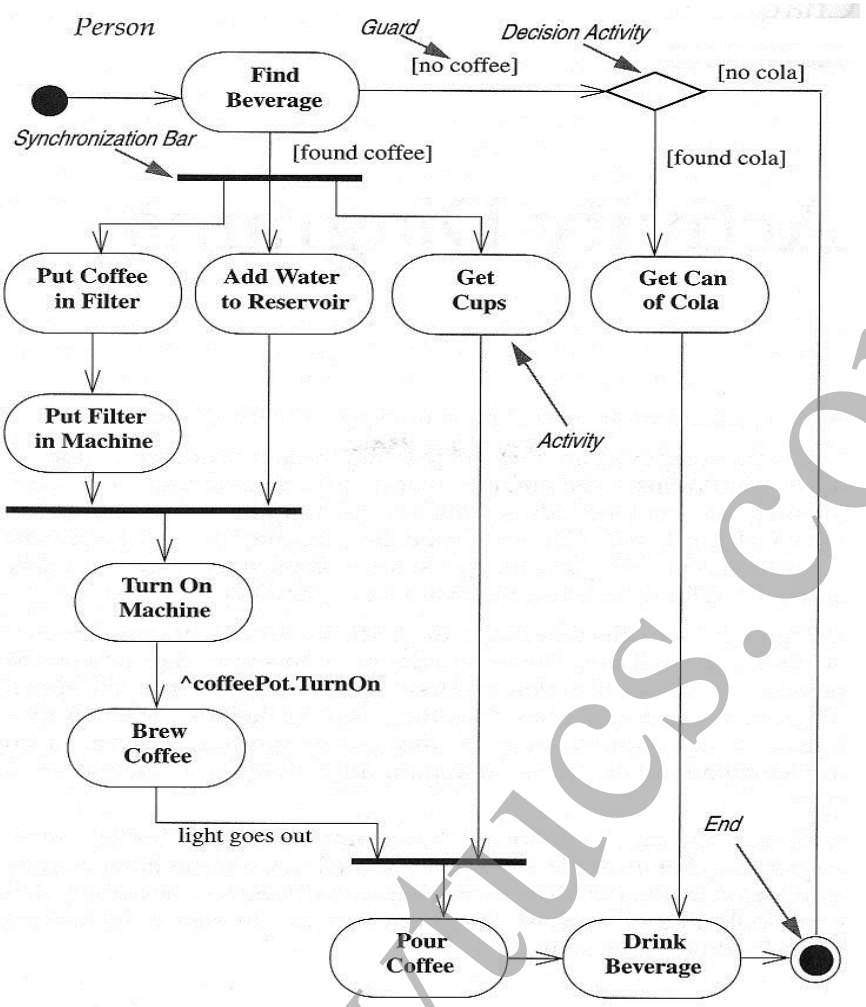
- Activity diagrams are most useful during the early stages of designing algorithms and workflows.
- Activity diagram is like a traditional flowchart in that it shows the flow of control from step to step

Activity diagram Notation

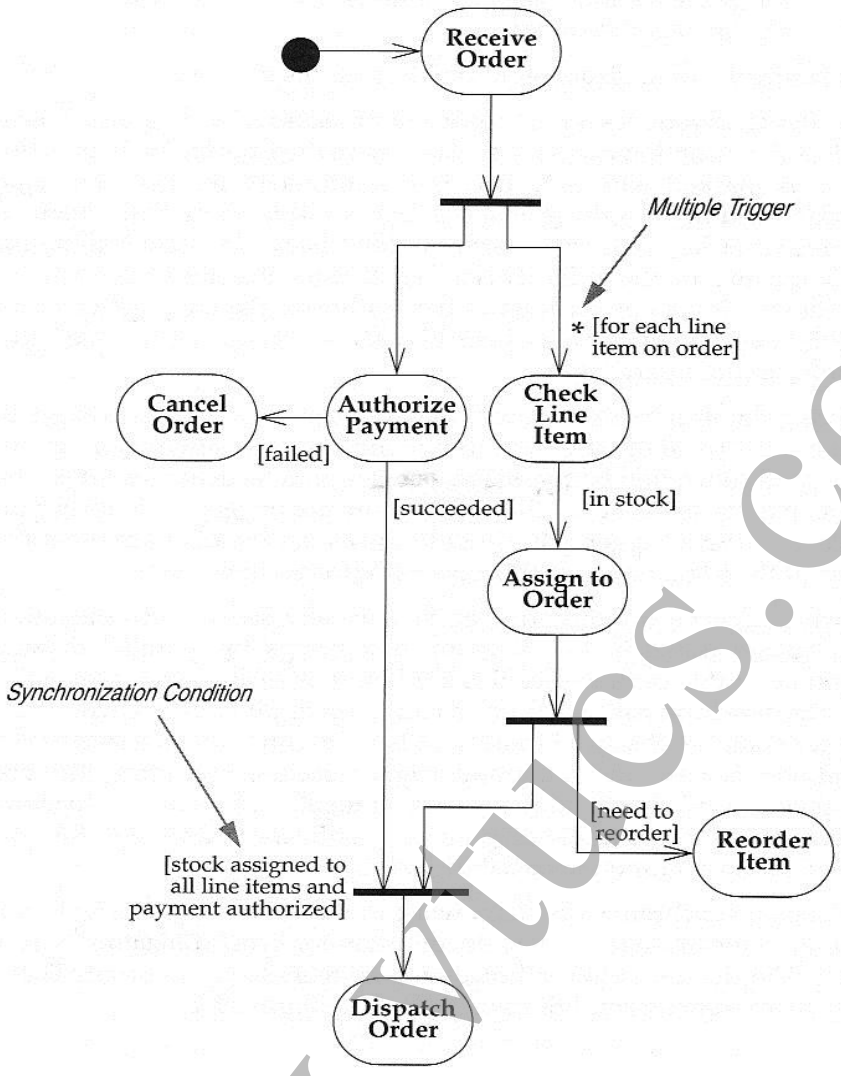
- Start at the top black circle
- If condition 1 is TRUE, go right; if condition 2 is TRUE, go down
- At first bar (a synchronization bar), break apart to follow 2 parallel paths
- At second bar, come together to proceed only when both parallel activities are done
- Activity – an oval
- Trigger – path 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)



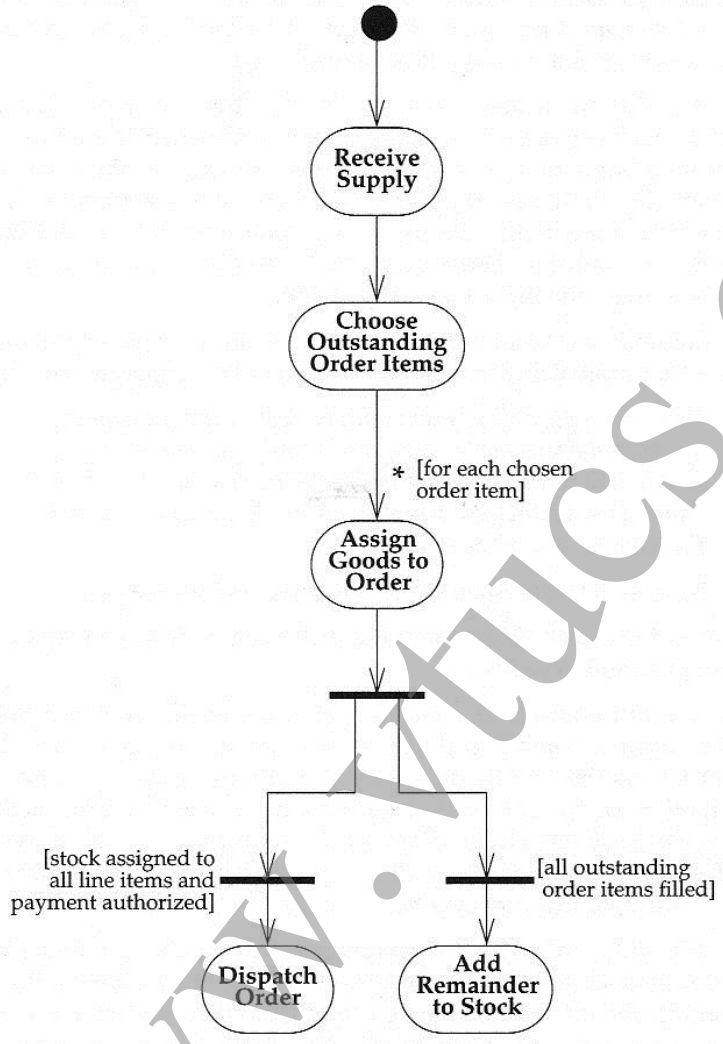
Eg:



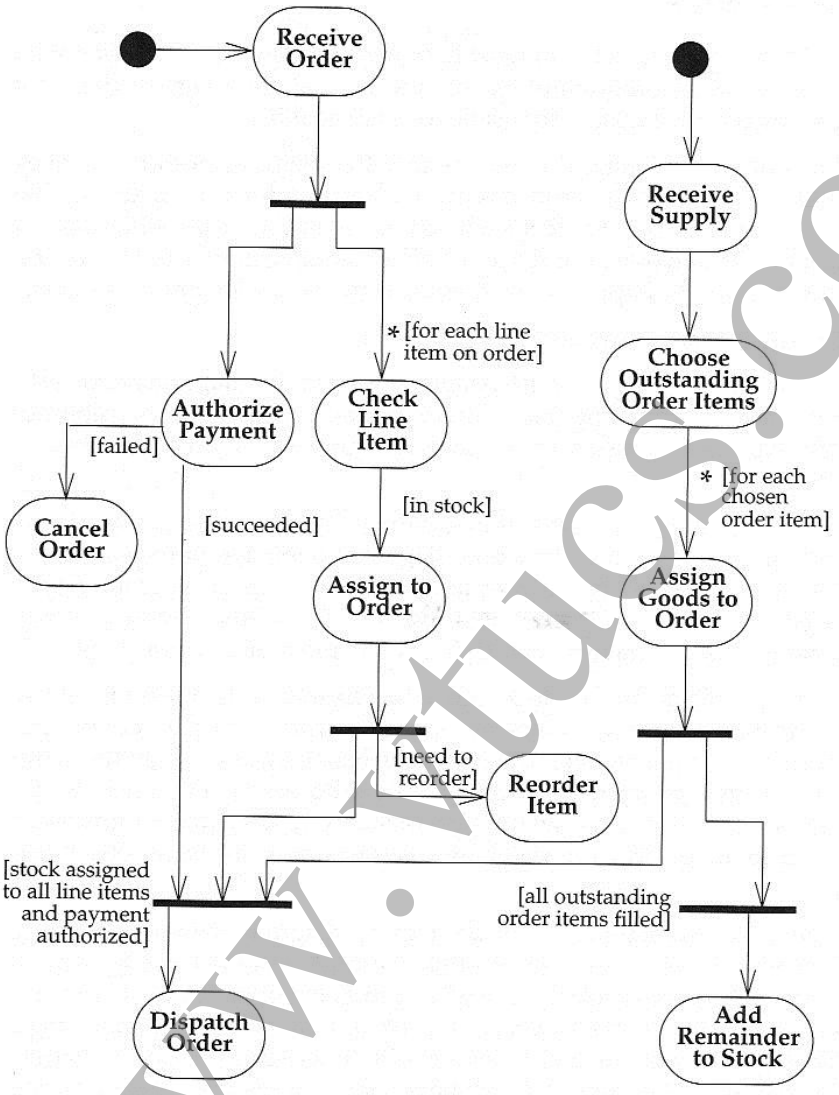
Eg: activity diagram for Use Case: Receiving an Order



Activity diagram for Use Case: Receiving a Supply



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



Activity diagram for stock trade processing

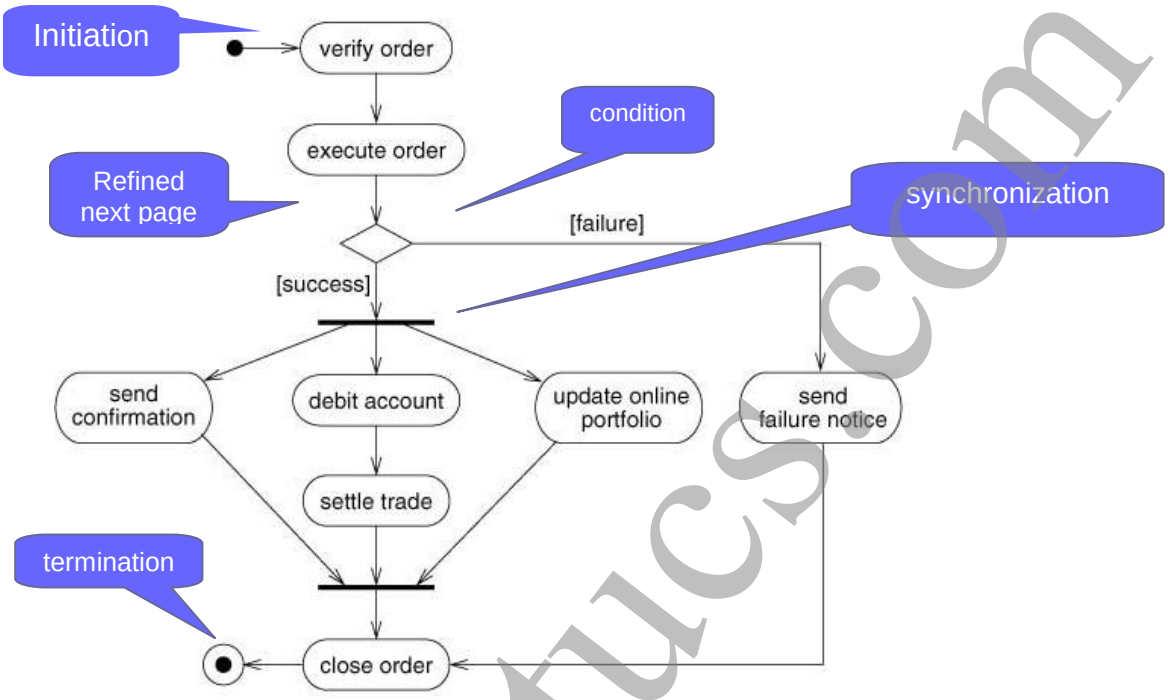


Figure 7.9 Activity diagram for stock trade processing. An activity diagram shows the sequence of steps that make up a complex process.

A Finer Activity for *execute order*

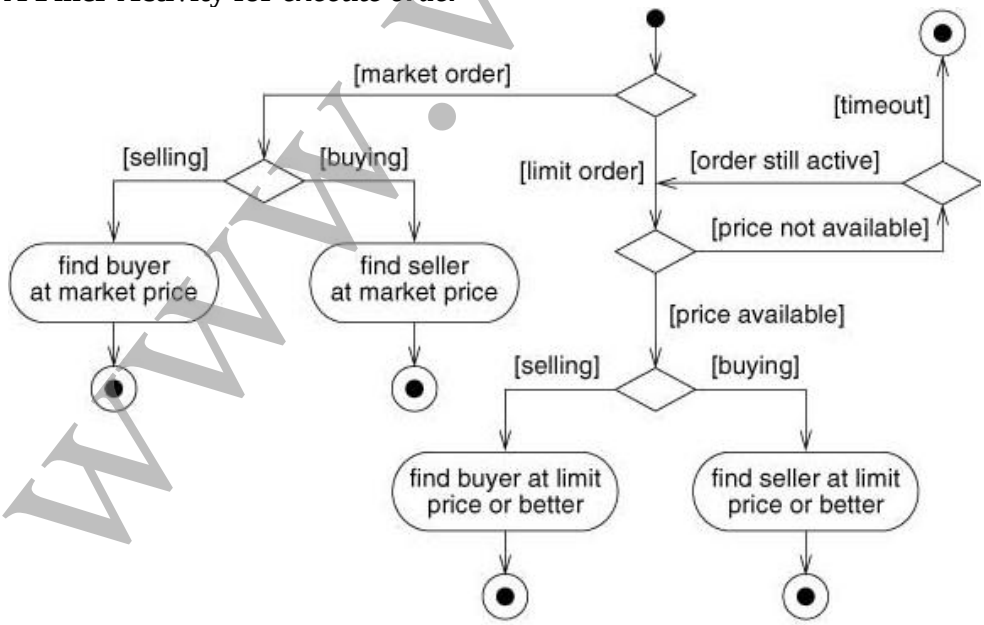


Figure 7.10 Activity diagram for *execute order*. An activity may be decomposed into finer activities.

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

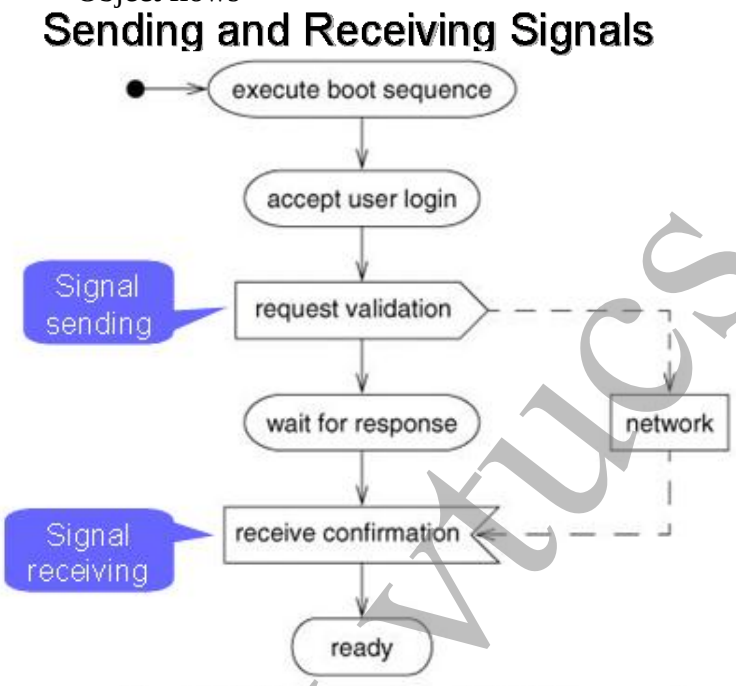


Figure 8.7 Activity diagram with signals. Activity diagrams can show fine control via sending and receiving events.

Swimlanes

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

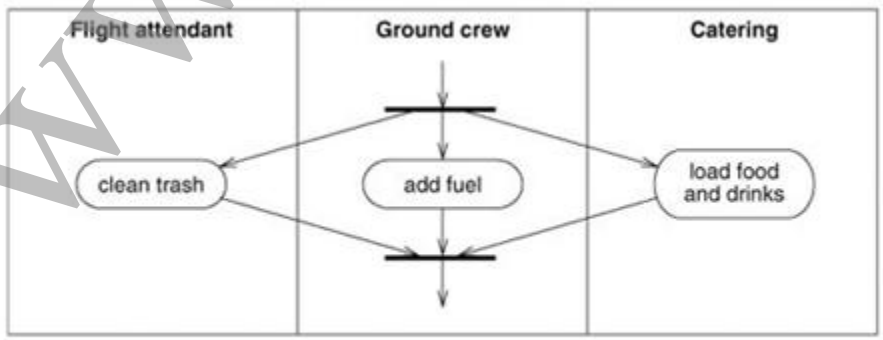
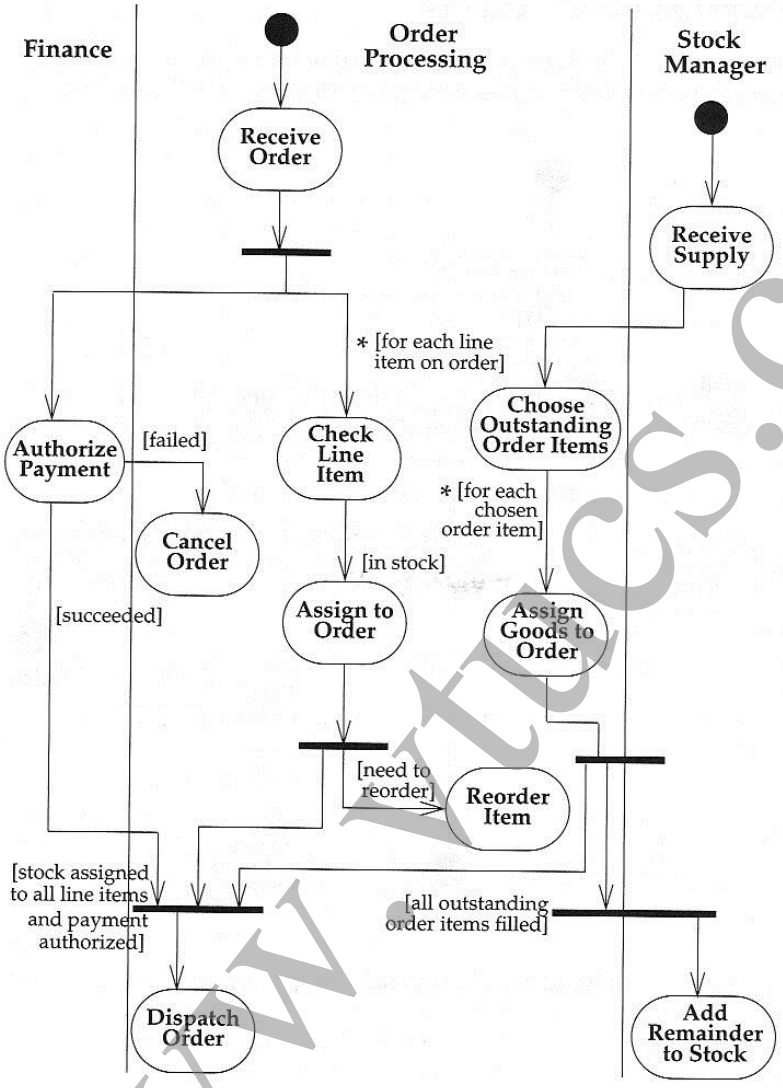


Figure 8.8 Activity diagram with swimlanes. Swimlanes can show organizational responsibility for activities.

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.

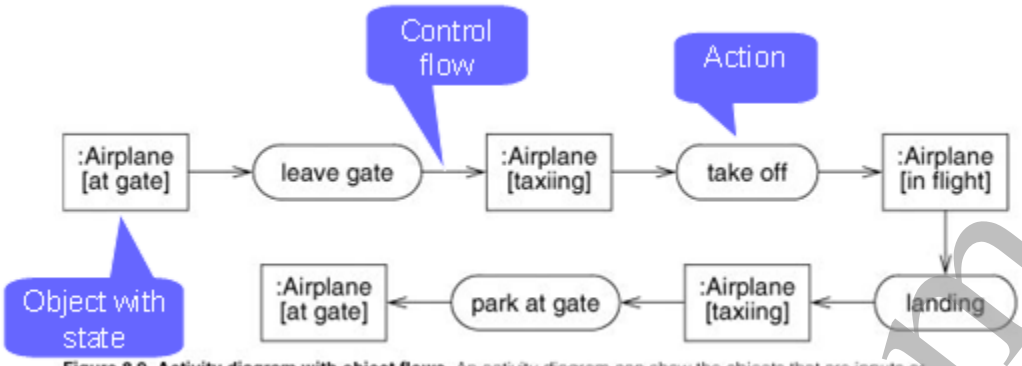


Figure 8.9 Activity diagram with object flows. An activity diagram can show the objects that are inputs or outputs of activities.

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.**

A **software development process** is a structure imposed on the development of a software product. Similar terms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. Some people consider a lifecycle model a more general term and a software development process a more specific term. For example, there are many specific software development processes that 'fit' the spiral lifecycle model.

Software development activities

The activities of the software development process represented in the waterfall model. There are several other models to represent this process.

1. Analysis

Ask yourself: What input your project needs as input? Does it need names, numbers, or values? What is the output that the project should give? How should the output be displayed? Who should use the program? These basic questions will form the guidelines in which you'll be referring to throughout the stages of development.

2. **Systems design** is the process or art of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. One could see it as the application of systems theory to product development. There is some overlap with the disciplines of systems analysis, systems architecture and systems engineering.[1][2]

3. Planning

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete,ambiguous, or even contradictory requirements are recognized by skilled and experienced software

engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Once the general requirements are gathered from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document.

Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.

4. Implementation, testing and documenting

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important part of the software development process. This part of the process ensures that defects are recognized as early as possible.

Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the writing of an API, be it external or internal. It is very important to document everything in the project.

5. Deployment and maintenance

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

Software Training and Support is important and a lot of developers fail to realize that. It would not matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality of at least one prior phase is poor.[*citation needed*] In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

Bug Tracking System tools are often deployed at this stage of the process to allow development teams to interface with customer/field teams testing the software to identify any real or perceived issues. These software tools, both open source and commercially licensed, provide a customizable process to acquire, review, acknowledge, and respond to reported issues.

Software Development Models

Several models exist to streamline the development process. Each one has its pros and cons, and it's up to the development team to adopt the most appropriate one for the project. Sometimes a combination of the models may be more suitable.

Waterfall Model

Main article: waterfall model
The waterfall model shows a process, where developers are to follow these phases in order:

Iterative

Iterative development prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes are preferred[*citation needed*] by commercial developers because it allows a potential of reaching the design goals of a customer who does not know how to define what they want

ATM

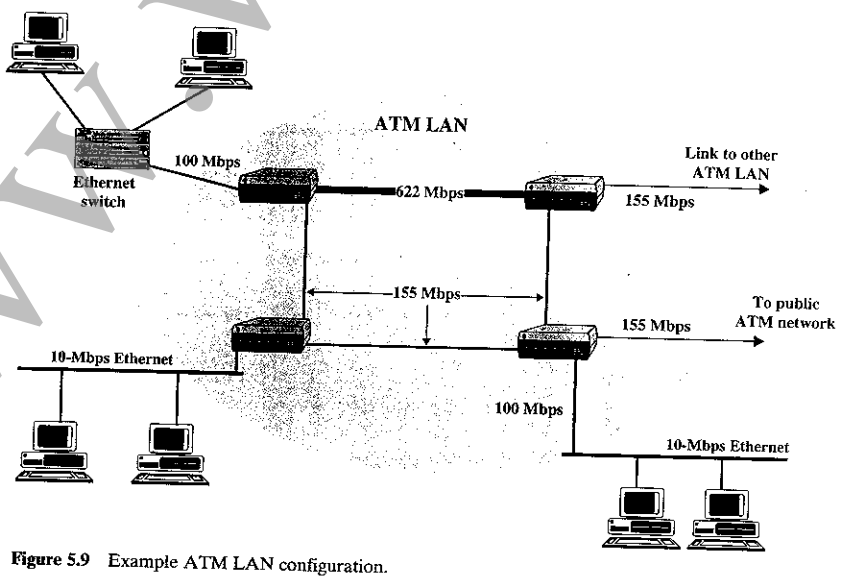


Figure 5.9 Example ATM LAN configuration.

ATM

- ATM standard (defined by CCITT) is widely accepted by common carriers as mode of operation for communication – particularly BISDN.
- ATM is a form of cell switching using small fixed-sized packets.

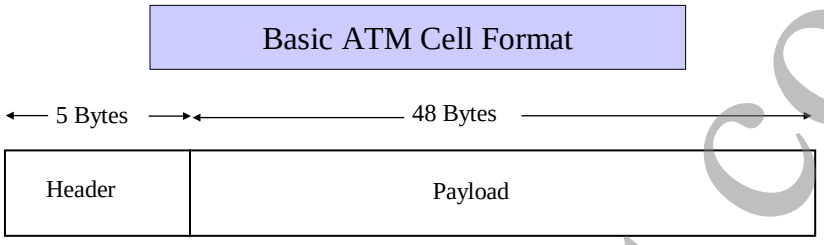


Figure 9.1

Domain class model

Perform the following steps to construct a domain class model

- Find classes
- Prepare a data dictionary
- Find assopciations
- Find attributers of objects and links
- Organize and simplify classes using inheritance
- Verfy that access paths exist for likely queries.
- Iterate and refine the model
- Reconsider the level of abstraction
- Group classes into packages.

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

1. Development Stages: The development stages can be summarized as follows:

- System Conception
- Analysis
- System design
- Class design

- Implementation
- Testing
- Training
- Deployment
- Maintenance

System Conception:

Conceive an application and formulate tentative requirements.

Analysis:

- Understand the requirements by constructing models…focus on what…rather then how?
- Two sub stages of analysis: Domain analysis and application analysis.
- Domain analysis focuses on real-world things whose semantics the application captures.

Eg: Airplane flight is a real world object that a flight reservation system must represent.

- Domain : Generally passive information captured in class diagrams
- Domain analysis is then followed by application analysis.
- Application analysis addresses the computer aspects of the application that are visible to users. Eg : flight reservation screen is a part of Flight Reservation System.
- Application Objects are meaningful only in the context of an application.
- Not the implementation aspect (black box view)

System Design:

- Devise a high level strategy-the architecture.
- Formulate an architecture and choose global strategies and policies.
- High Level plan or strategy.
- Depends on the present requirement and past experiences.
- The architecture must also support future modifications to the application
- For simple systems. architecture follows analysis.
- For large and complex systems: there is interplay between the construction of a model and the model's architecture and they must be built together

Class design:

- Augment and adjust the real-world models from analysis so that they

are amenable to implementation.

- Developers choose algorithms to implement major system functions.

Implementation:

- Translate the design into code and database structure.
- Often tools can generate some of the code from the design model.

Testing:

- Ensure that the application is suitable for actual use and that it truly satisfies the requirements.
- Unit tests discover local problems and often require that extra instrumentation be built into the code.
- System test exercise q major subsystem or the entire application. This can discover broad failures to meet specifications.
- Both unit and system tests are necessary.
- Testing should be planned from the beginning and many tests can be performed during implementation.

Training:

- Help users master the new application.
- Organization must train users so that they can fully benefit from an application.

Deployment:

- Place the application in the field.
- The eventual system must work on various platforms and in various configurations.
- Developers must tune the system under various loads and write scripts and install procedures.
- Localize the product to different languages.

Maintenance:

- Preserve the long-term viability of the application.
- Bugs that remain in the original system will gradually appear during use and must be fixed.
- A successful application will also lead to enhancement requests and long lived application will occasionally have to be restructured.
- Models ease maintenance and transition across staff changes.

- A model expresses the business intent for an application that has been driven into the programming code, user interface and data base structure

2. Development Life Cycle.

(a) Waterfall Development

- Rigid linear sequence with no backtracking.
- Suitable for well understood applications with predictable outputs from analysis and design, such systems seldom occur.
- A waterfall approach also does not deliver a useful system until completion.
- This makes it difficult to assess progress and correct a project that has gone awry.

(b) Iterative Development

- More flexible.
- There are multiple iterations as the system evolves to final deliverable.
- Each iteration includes a full complement of stages: analysis, design, Implementation and testing.
- This is the best choice for most applications because it gracefully responds to changes and minimizes risk of failure.
- Management and business users get early feedback about progress.

3. Chapter Summary

- A software Engineering Process provides a basis for the organized production of software .
- There is a sequence of well defined stages that can apply to each piece of a system.
- Parallel development teams might develop a database design, key algorithms , and an user interface.
- An iterative development of software is flexible and responsive to evolving requirements. First you prepare a nucleus of a system, and then you successively grow its scope until you realize the final desired software.

Exercises

1. It seems there is enough time to do a job right the first time, but there is always time to do it over. Discuss how the approach presented in this chapter overcomes this tendency of human behavior. What kinds of errors do you make if you rush into the implementation phase of a software project? Compare the effort required to prevent errors with that needed to detect and correct them.

Answer:

We have learned this lesson more times than we would care to admit. Carpenters have a similar maxim: "Measure twice, cut once." This exercise is intended to get the student to think about the value of software engineering in general. There is no single correct answer. It is probably too early in the book for the student to answer in detail about how software engineering will help. Look for indications that the student appreciates the pitfalls of bypassing careful design.

The effort needed to detect and correct errors in the implementation phase of a software system is an order of magnitude greater than that required to prevent errors through careful design in the first place. Many programmers like to design as they code, probably because it gives them a sense of immediate progress. This leads to conceptual errors which are difficult to distinguish from simple coding mistakes. For example, it is easy to make conceptual errors in algorithms that are designed as they are coded. During testing, the algorithm may produce values that are difficult to understand. Analysis of the symptoms often produces misleading conclusions. It is difficult for the programmer to recognize a conceptual error, because the focus is at a low level. The programmer "cannot see the forest for the trees".

Exercise on requirement capturing using use cases:

2. Case Study: Online travel agent:

Prepare a use case diagram, using the generalization and include relationships.

Purchase a flight. Reserve a flight and provide payment and address information. Provide
payment Information: Provide a credit card to pay for the incurred charges. Provide
address: provide mailing and residence address. Purchase car rentals: Reserve a rental car and provide payment and address information.

Purchase a hotel stay: reserve a hotel room and provide payment and address info.

Make a purchase: Make a travel purchase and provide payment and address information.

8.2 Figure A8.2 shows a use case diagram for an online travel agent.

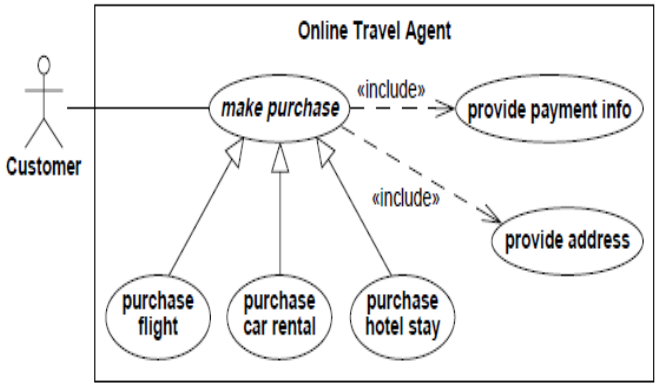


Figure A8.2 Use case diagram for an online travel agent

System Conception deals with the genesis of an application. Some people, who understand both business needs and technology, think of an idea for an application. Developers then explore the idea to understand the needs and devise possible solutions. Purpose of System Conception is to defer the details and understand the big picture…..what need does the proposed system meet, can it be developed at a reasonable cost, and will the demand for the result justify the cost of building it..?

1. Devising a System Concept.

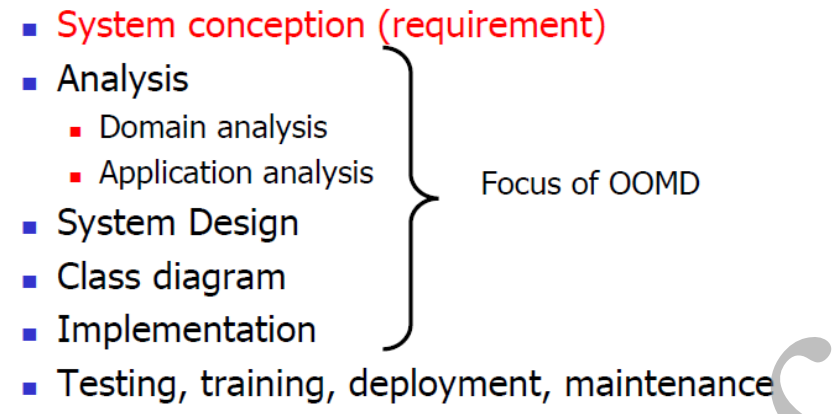
- Most ideas for a new system are extension of existing ideas.
- Eg: HR dept may have a DB of employee benefit choices and require that a clerk enter changes.
- An obvious extension is to allow employees to view and enter their own changes.
- Many issues :(Security, reliability, privacy ...)
- But.. New idea is a straightforward extension of an existing concept.
- Eg2: online auction system. which is the automation of the existing system which is running presently manually .

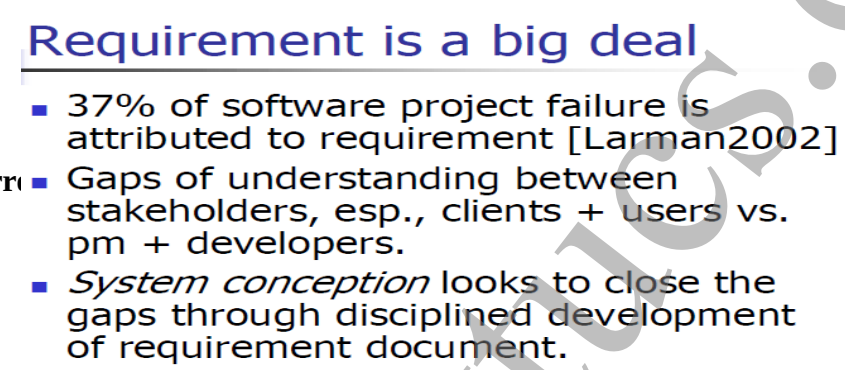
Some ways to find new system concepts:

- New Functionality- Add functionality to an existing system.
- Streamlining- remove restriction or generalize the way a system works.
- Simplification- Let ordinary persons perform tasks previously assigned to specialists.
- Automation- automate manual processes
- Integration- Combine functionality from different systems.
- Analogies- Look for analogies in other problem domains and see if they have useful ideas.
- Globalization-Travel to other countries and observe their cultural and business

practices.

Following is the summary of system conception highlighting some essential points:





2. Elaborate a concept

A good system concept must answer the following questions.

- Who is the application for ?
- What problems will it Solve?
- Where will it be used ?
- When is it needed?
- Why is it needed ?
- How will it work

Who is the application for?

- Understand the stakeholders of the system;
- Usually two important ones are: Financial sponsor (client) End Users.

What problems will it Solve?

- Bound the size of effort and scope of system
- Determine what feature's in and what's out.

When is it needed?

- Feasible time, *T_f*
  - The time in which the system can be developed within the constraints of cost and available resource
- Required time, *T_r*
  - The time that the system is needed to meet the business goals.
- If (*T_r* < *T_f*), work with technologists and business experts to trim the system

Why is it needed?

- Prepare a business case
  - Financial justification, including
    - cost,
    - tangible benefits,
    - intangible benefits,
    - risk, and
    - alternatives (why build when you can buy?)
  - For a commercial product, estimate the number of units you can sell and determine a reasonable price.
    - Revenue must cover the cost

How will it work?

- Investigate feasibility of the problem
- Build prototype, if it helps clarifying a concept or removing a technological risk.

CASE STUDY

ATM Case Study (Automated Teller machine)

System Concept for an automated teller machine.

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

- Who is the application for?

The Vendor building the software (assume its we)

- What problems will it Solve?

ATM is built to serve both the bank (automation) and the customer (ubiquitous)

- Where will it be used ?

ATM are available at many stores, sporting events and other locations throughout the world

- When is it needed?

From an economic perspective, it is desirable to minimize the investment, maximize the revenue, and realize revenue as soon as possible. (OOMD Helps)

- Why is it needed ?

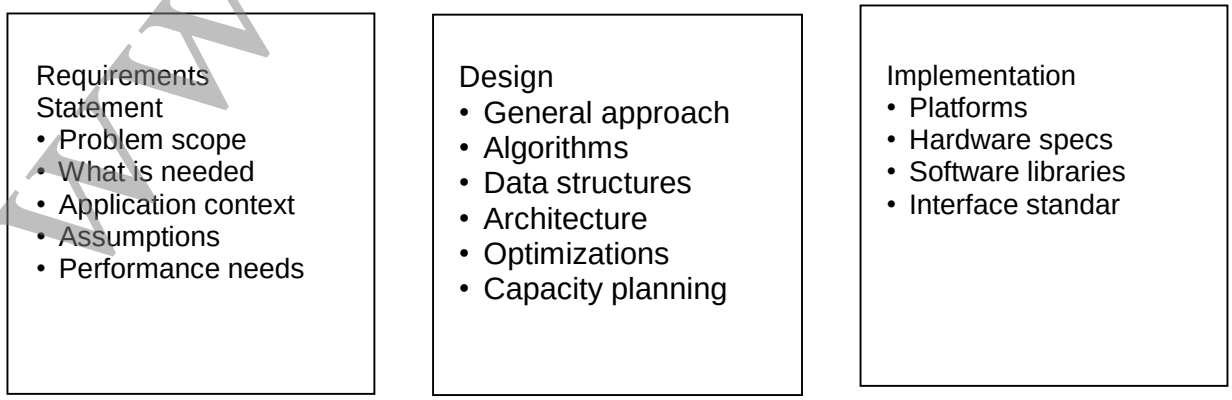
A novel product could outflank competitors and lead to premium pricing.Here the case study is taken for explanation.

- How will it work?

We adopt a three-tier architecture

3. Preparing a Problem Statement

- Throughout development, you should distinguish among requirements, design, and implementation.
- Do not Make early design and implementation decisions or you will compromise development?
- Most problem statements are ambiguous, incomplete, or even inconsistent.
- Some requirements are just plain wrong.
- …..The purpose of Analysis is to fully understand the problem and its implications



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

- 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.

System Boundaries

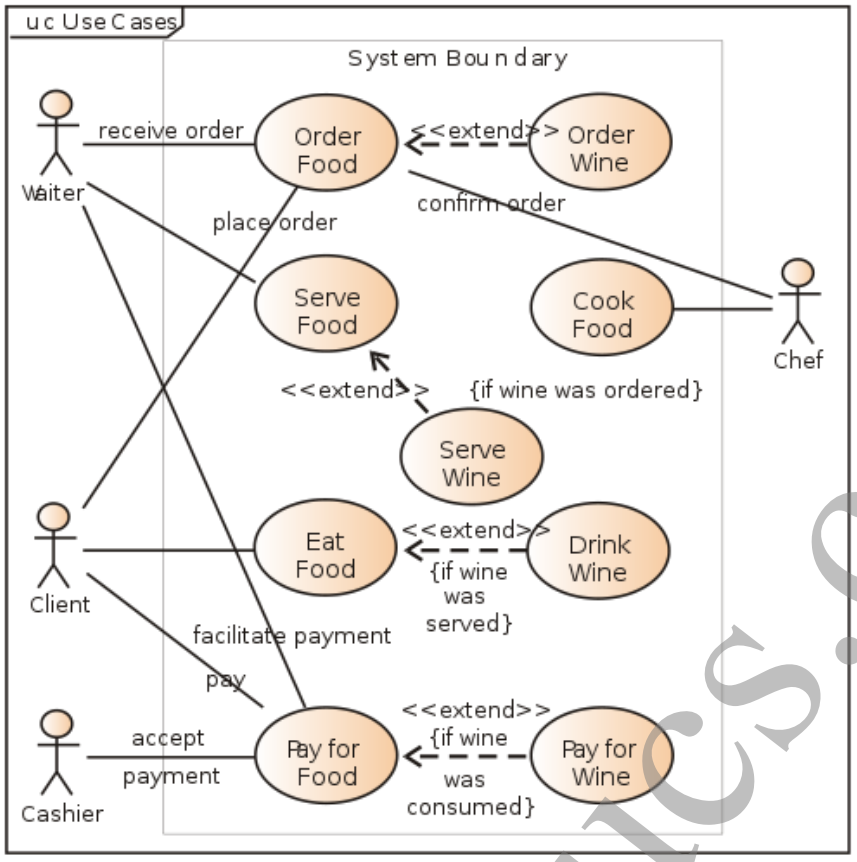
When comparing LECs for alternative systems, it is very important to define the boundaries of the 'system' and the costs that are included in it. For example, should transmissions lines and distribution systems be included in the cost? Typically only the costs of connecting the generating source into the transmission system is included as a cost of the generator. But in some cases wholesale upgrade of the Grid is needed. Careful thought has to be given to whether or not these costs should be included in the cost of power.

Should R&D, tax, and environmental impact studies be included? Should the costs of impacts on public health and environmental damage be included? Should the costs of government subsidies be included in the calculated LEC?

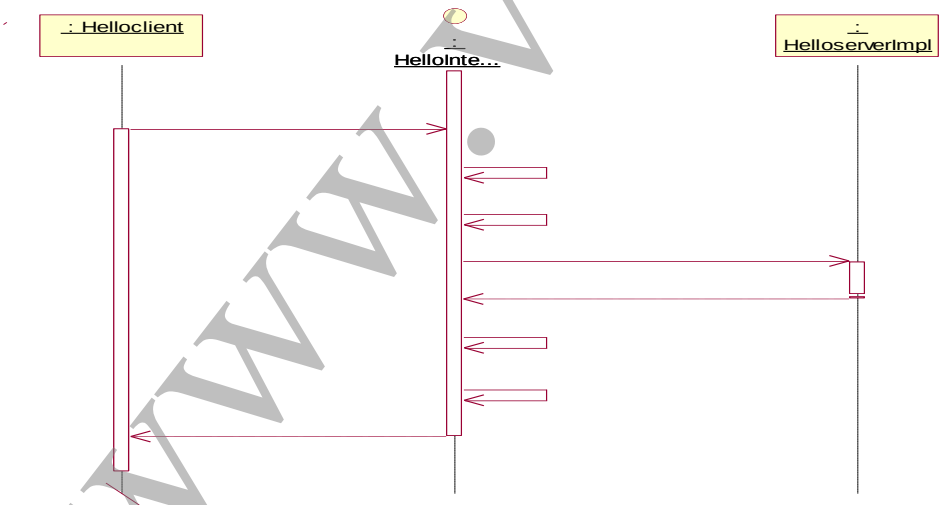
Find use case

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

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



Finding external events
Ex:



What is A Scenario Sequence Diagram?

The Scenario Sequence Diagram (SSD) is a graphic depiction of a behavior sequence (Use Case) layered over the system architecture, at the appropriate layer of abstraction. The key elements are:

- **Actors** – Actors are represented as boxes, except for human actors which are shown as stick figures. The actors of a scenario are shown across the top of the diagram.
- **Lifelines** – Lifelines are the vertical lines drawn down from each actor. Time proceeds down the page.
- **Use Case** – Use Cases are the behaviors of the actors. Use Cases are shown as "bubbles" on the lifeline of the actor executing the behavior.
- **Messages** – Messages are sent from a source to a destination actor. Messages typically trigger an activity of the destination actor. Messages contain data and/or control directives.
- **Transaction** – A transaction is a sequence of behavior consisting of a message from a source actor to the destination actor, initiating the Use Case of the destination actor.

Figure 1 illustrates these elements in a trivially simple system.

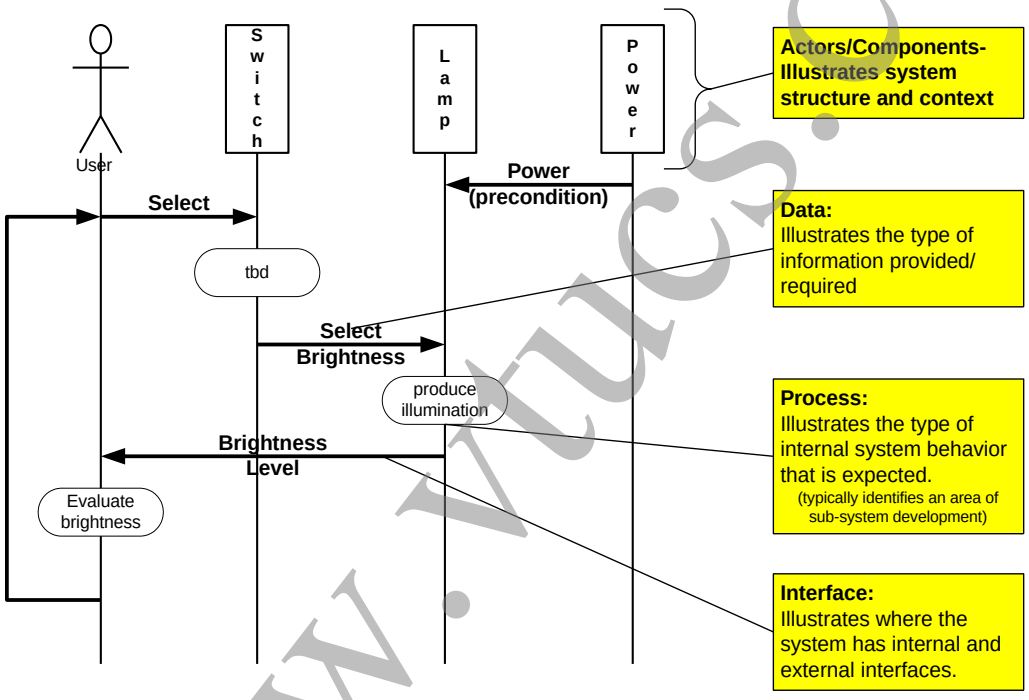
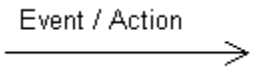
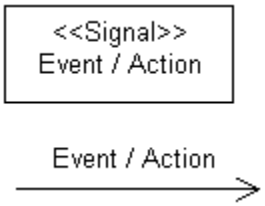


Figure 1 3-way lamp Scenario Sequence Diagram

State diagram

Element and its Description	**Symbol**
Initial State: This shows the starting point or first activity of the flow. Denoted by a solid circle. This is also called as a "**pseudo state**," where the state has no variables describing it further and no activities.	●

State: Represents the state of object at an instant of time. In a state diagram, there will be multiple of such symbols, one for each state of the Object we are discussing. Denoted by a rectangle with rounded corners and compartments (such as a class with rounded corners to denote an Object). We will describe this symbol in detail a little later.	
Transition: An arrow indicating the Object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow, separated by a slash. Transitions that occur because the state completed an activity are called "**triggerless**" transitions. If an event has to occur after the completion of some event or action, the event or action is called the guard condition. The transition takes place after the guard condition occurs. This guard condition/event/action is depicted by square brackets around the description of the event/action (in other words, in the form of a Boolean expression).	
History States: A flow may require that the object go into a trance, or wait state, and on the occurrence of a certain event, go back to the state it was in when it went into a wait state—its last active state. This is shown in a State diagram with the help of a letter H enclosed within a circle.	
Event and Action: A trigger that causes a transition to occur is called as an event or action. Every transition need not occur due to the occurrence of an event or action directly related to the state that transitioned from one state to another. As described above, an event/action is written above a transition that it causes.	
Signal: When an event causes a message/trigger to be sent to a state, that causes the transition; then, that message sent by the event is called a signal. Represented as a class with the <<Signal>> icon above the action/event.	
Final State: The end of the state diagram is shown by a bull's eye symbol, also called a final state. A final state is another example of a pseudo state because it does not have any variable or action described.	

Note: Changes in the system that occur, such as a background thread while the main process is running, are called "**sub states**." Even though it affects the main state, a sub state is not shown as a part of the main state. Hence, it is depicted as contained within the main state flow.

As you saw above, a state is represented by a rectangle with rounded edges. Within a state, its Name, variables, and Activities can be listed as shown in Figure 6.1.

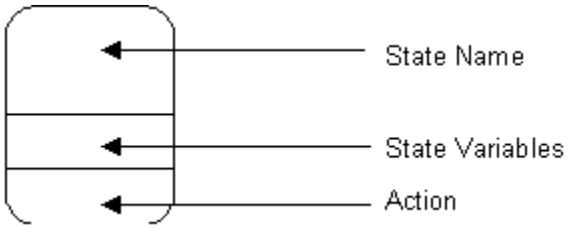
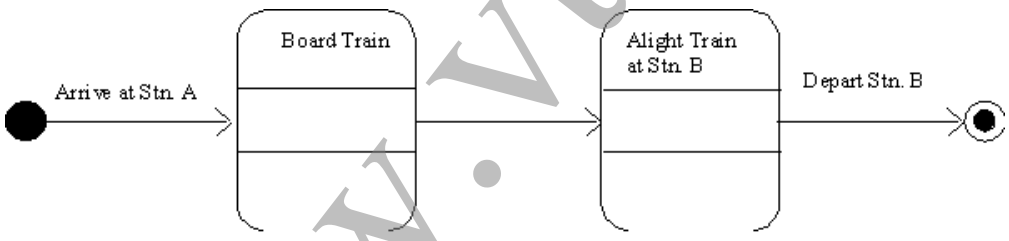


Figure 6.1: the structure of the state element

Creating a State Diagram

Let us consider the scenario of traveling from station A to station B by the subway. Figure 6.2 would be a state diagram for such a scenario. It represents the normal flow. It does not show the sub states for this scenario.



Click here for a larger image.

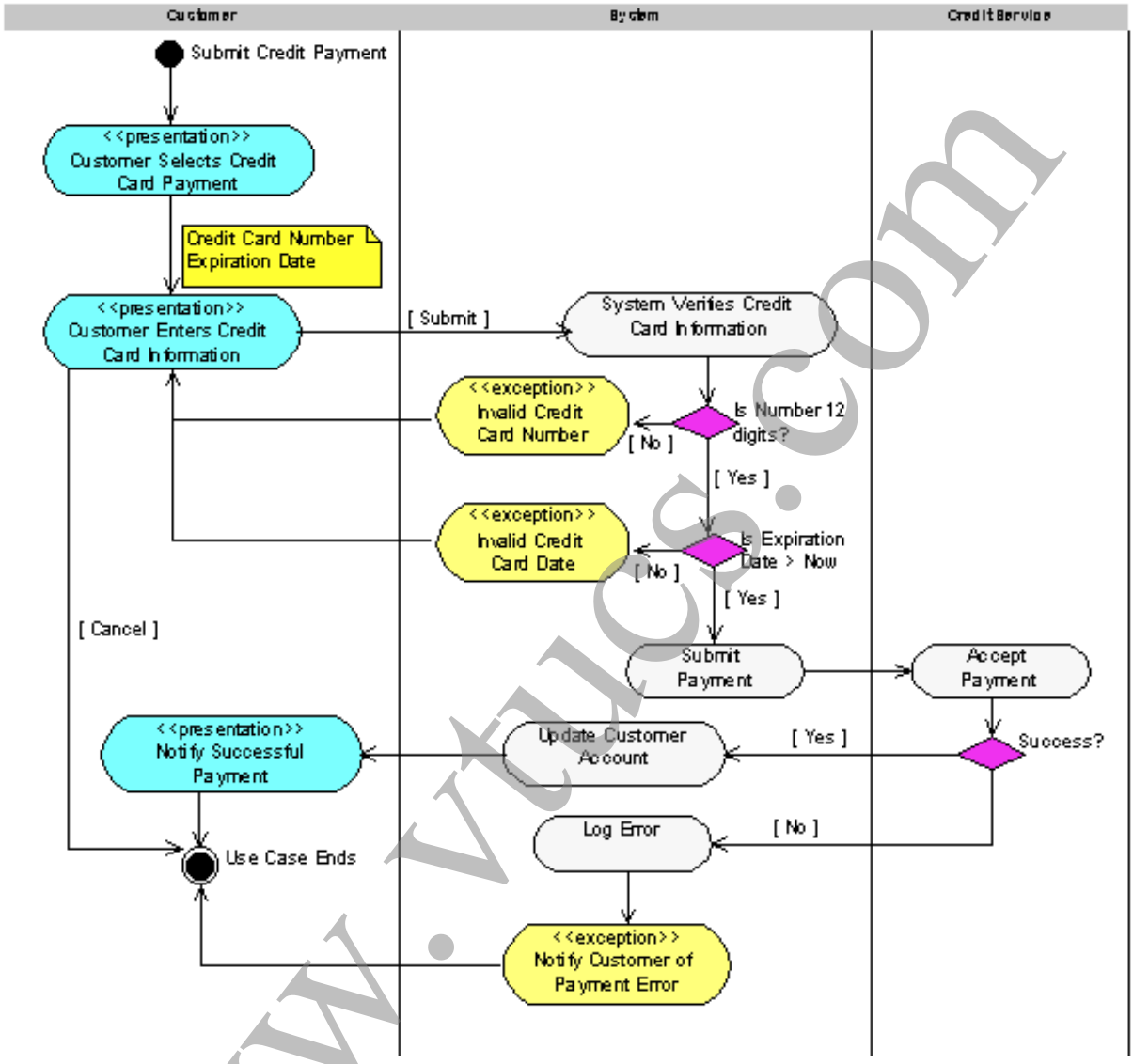
Figure 6.2: an example flow in a state diagram

Things to Remember

Create state diagrams when the business logic for a particular flow is very complex, or needs greater understanding by the developer to be coded perfectly.

Arrange the states and transitions to ensure that the lines that cross each other are minimal. Criss-crossing lines are potential sources of confusion in conveying the diagram's meaning.

Activity diagram for card verification



Unit 6: CLASS DESIGN, IMPLEMENTATION MODELING, LEGACY SYSTEMS 7hrs

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.**

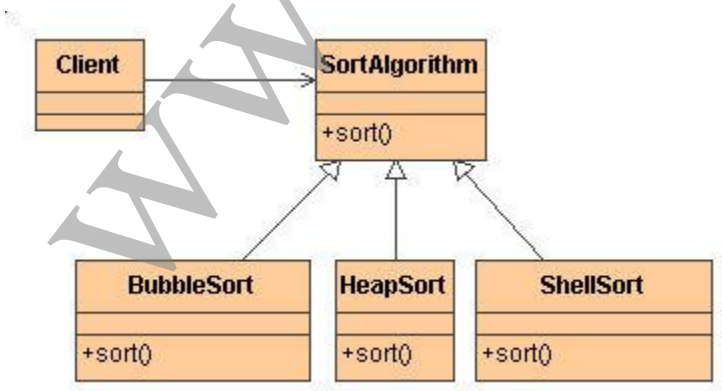
Design Class Diagrams

Strategy

"Strategy, State, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems." [Coplien, *Advanced C++*, p58]

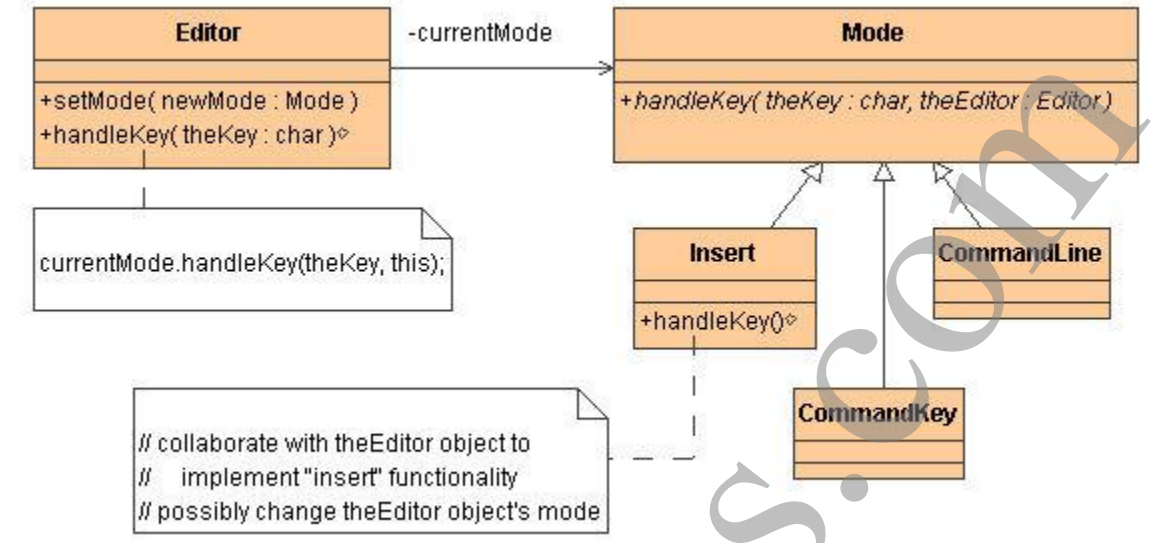
Most of the GoF patterns exercise the two levels of indirection demonstrated here.

1. Promote the "interface" of a method to an abstract base class or interface, and bury the many possible implementation choices in concrete derived classes.
2. Hide the implementation hierarchy behind a "wrapper" class that can perform responsibilities like: choosing the best implementation, caching, state management, remote access.



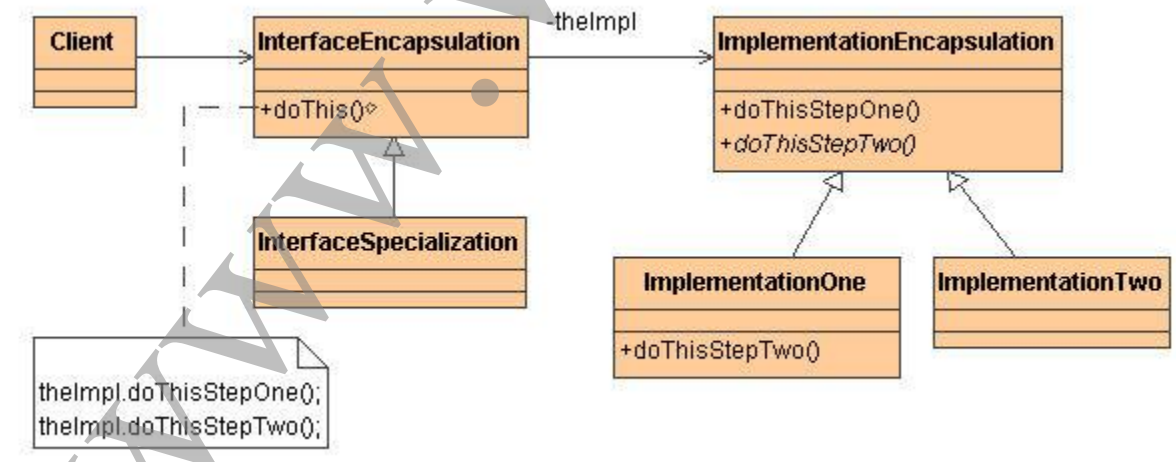
State

"Strategy is a bind-once pattern, whereas State is more dynamic." [Coplien, *C++ Report*, Mar96, p88]



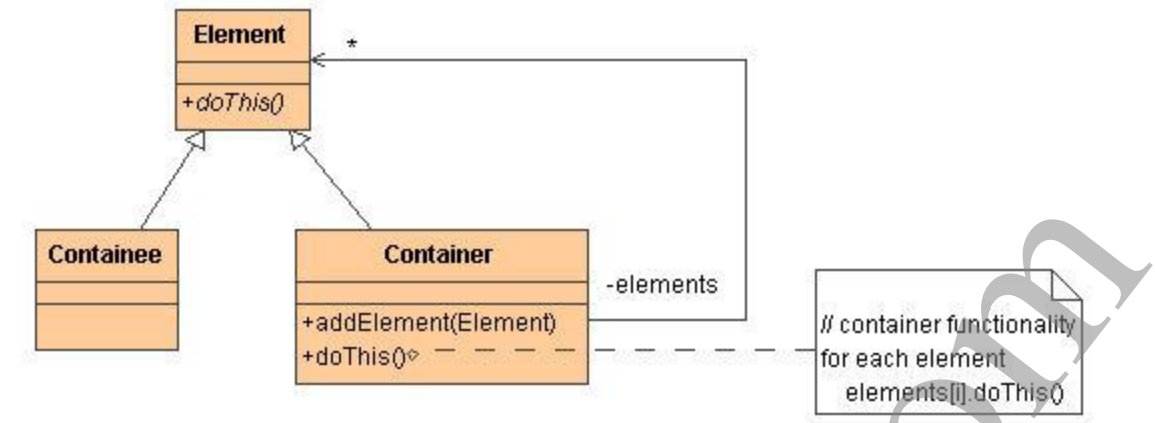
Bridge

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, *C++ Report*, May 95, p58]



Composite

Three GoF patterns rely on recursive composition: Composite, Decorator, and Chain of Responsibility.

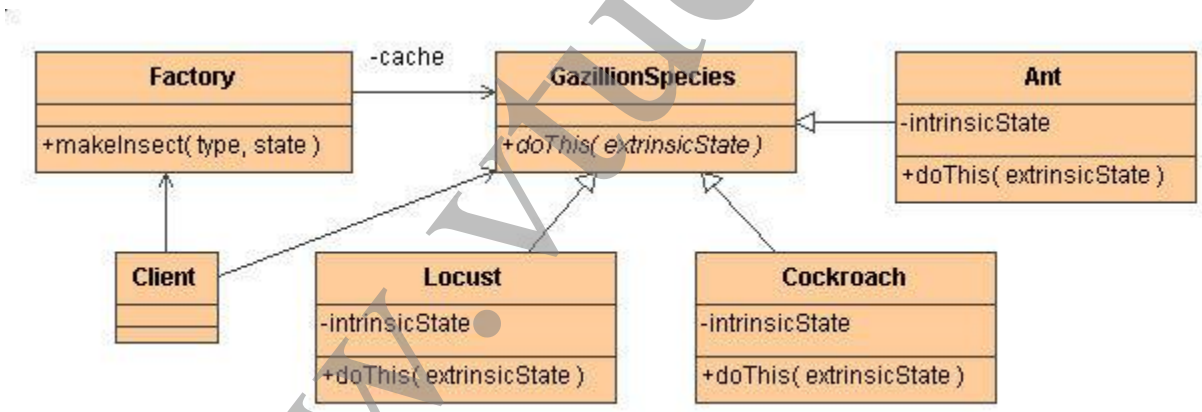


Flyweight

Flyweight is often combined with Composite to implement shared leaf nodes. [GoF, p206]

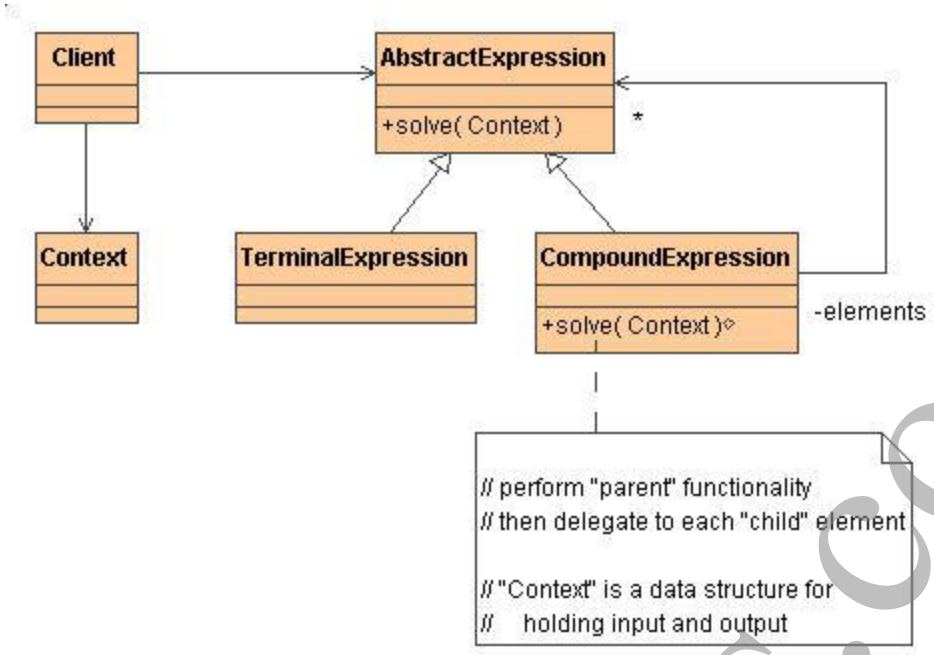
Flyweight shows how to make lots of little objects. Facade shows how to make a single object represent an entire subsystem. [GoF, p138]

This diagram is perhaps a better example of Composite than the Composite diagram.



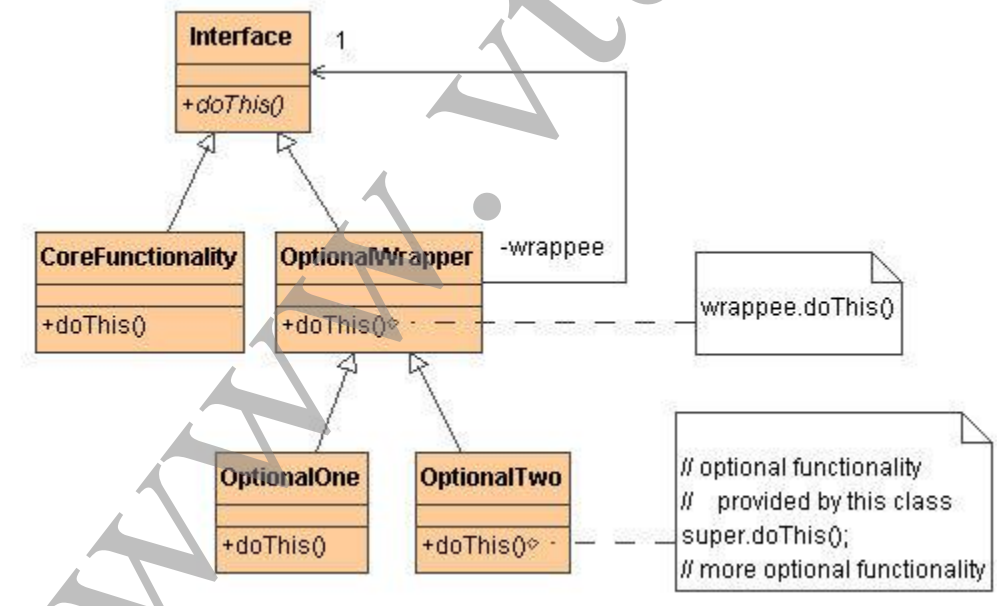
Interpreter

Interpreter is really an application of Composite.



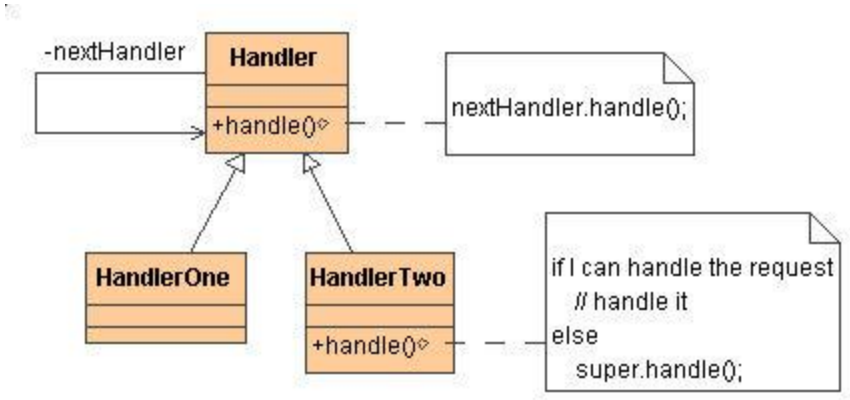
Decorator

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GoF, p220]



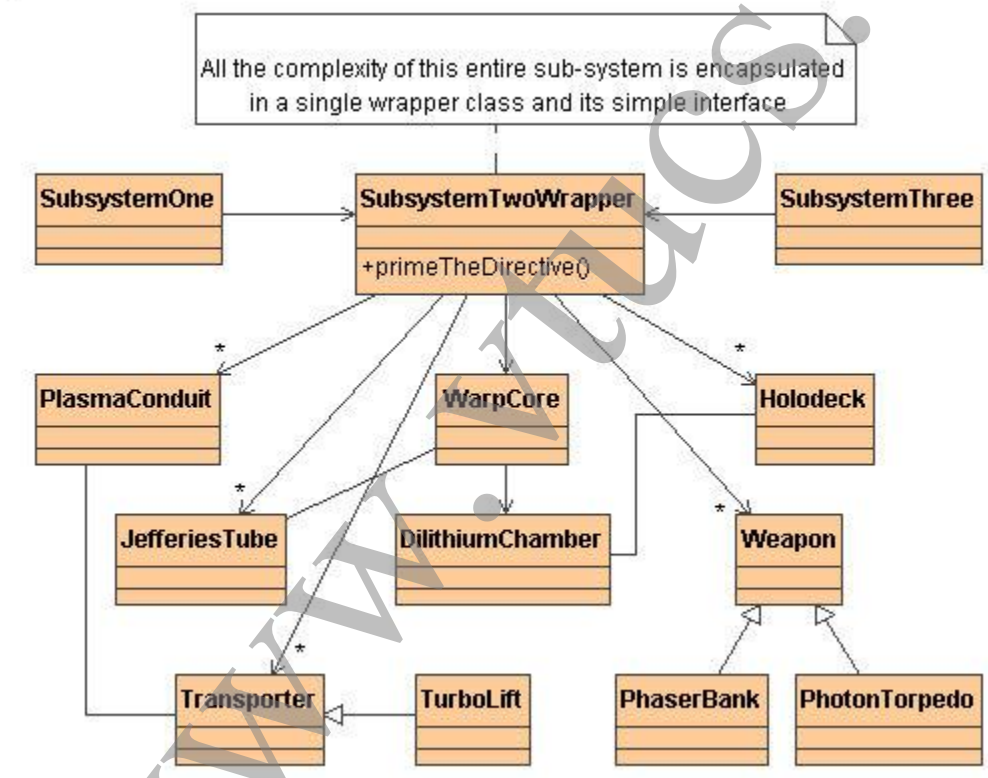
Chain of Responsibility

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.



Facade

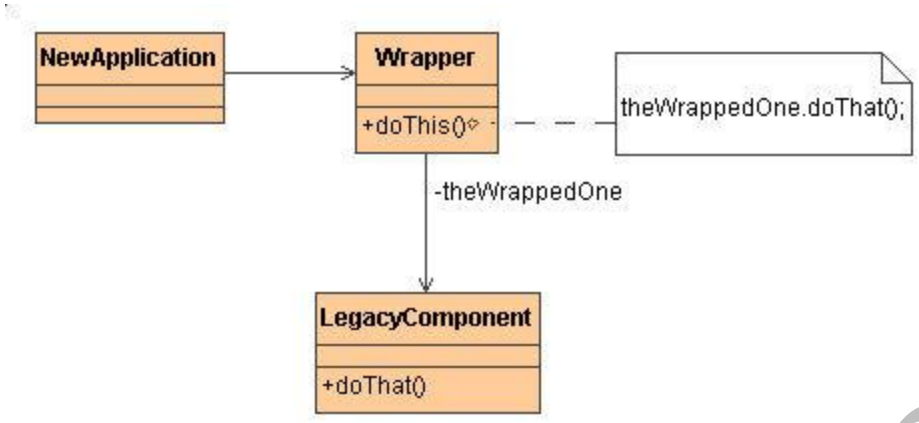
Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GoF, p219]



Adapter

Adapter makes things work after they're designed; Bridge makes them work before they are. [GoF, p219]

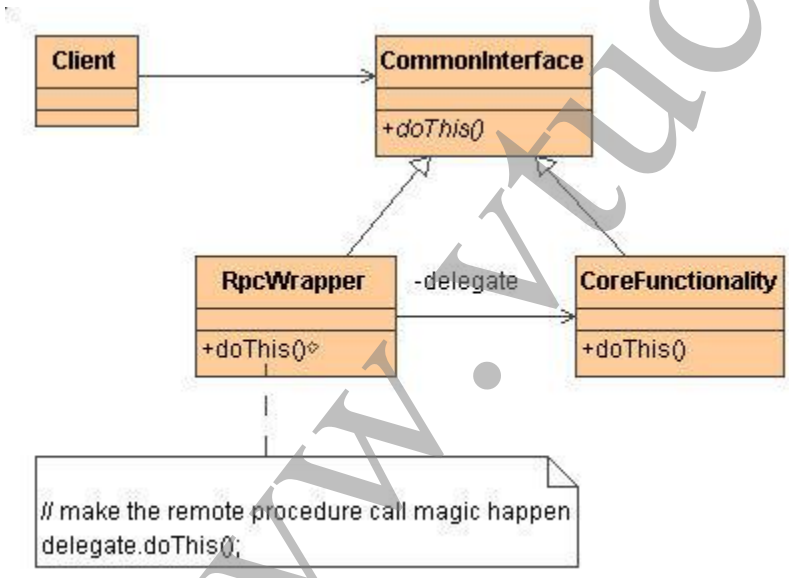
Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GoF, 216]



Proxy

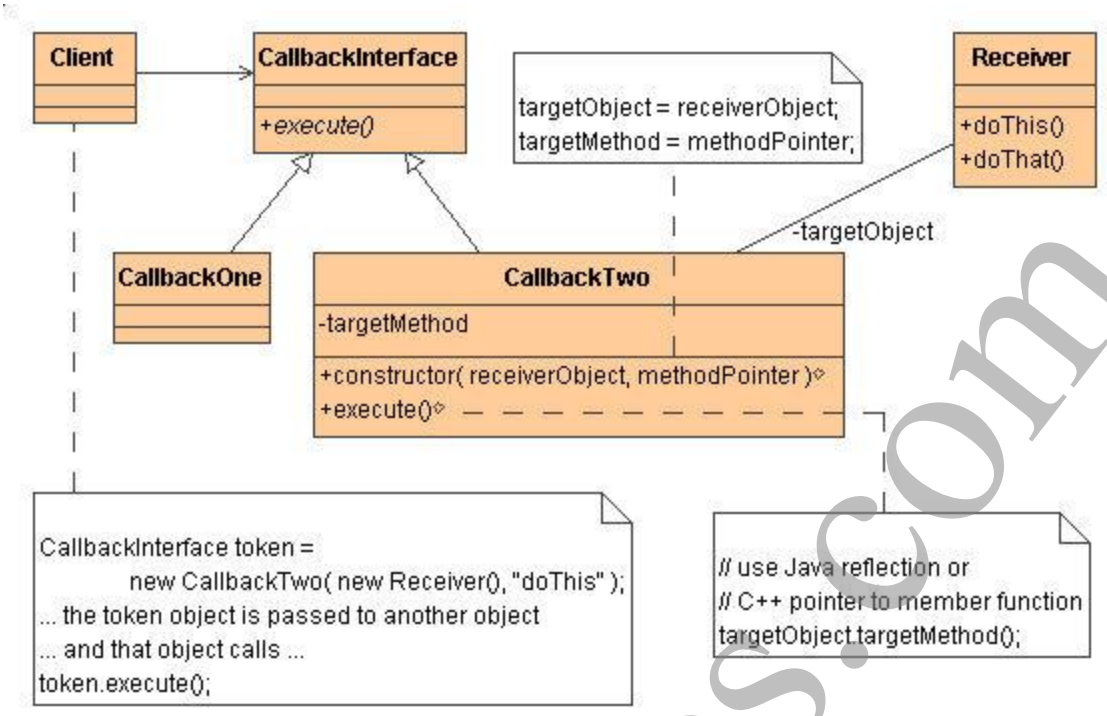
Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GoF, p220]

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GoF. p216]



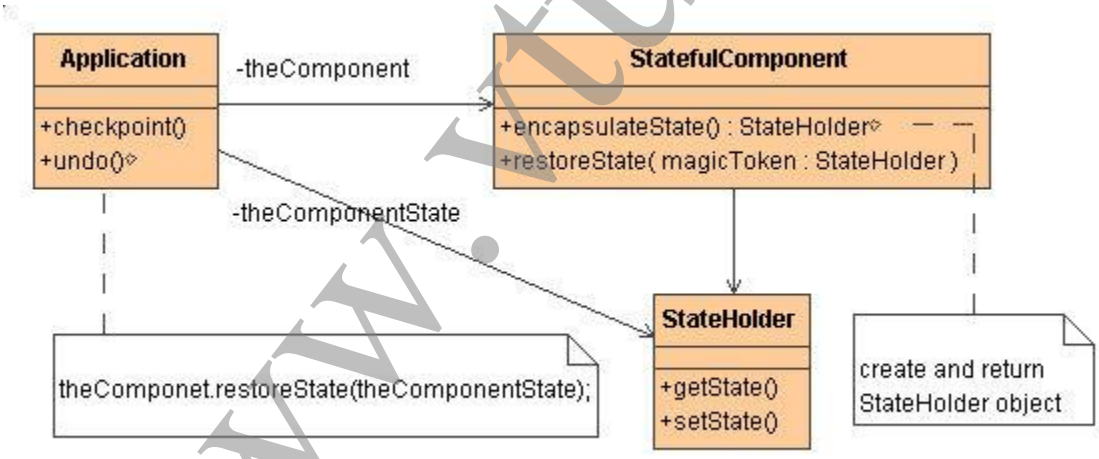
Command

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GoF, p346]



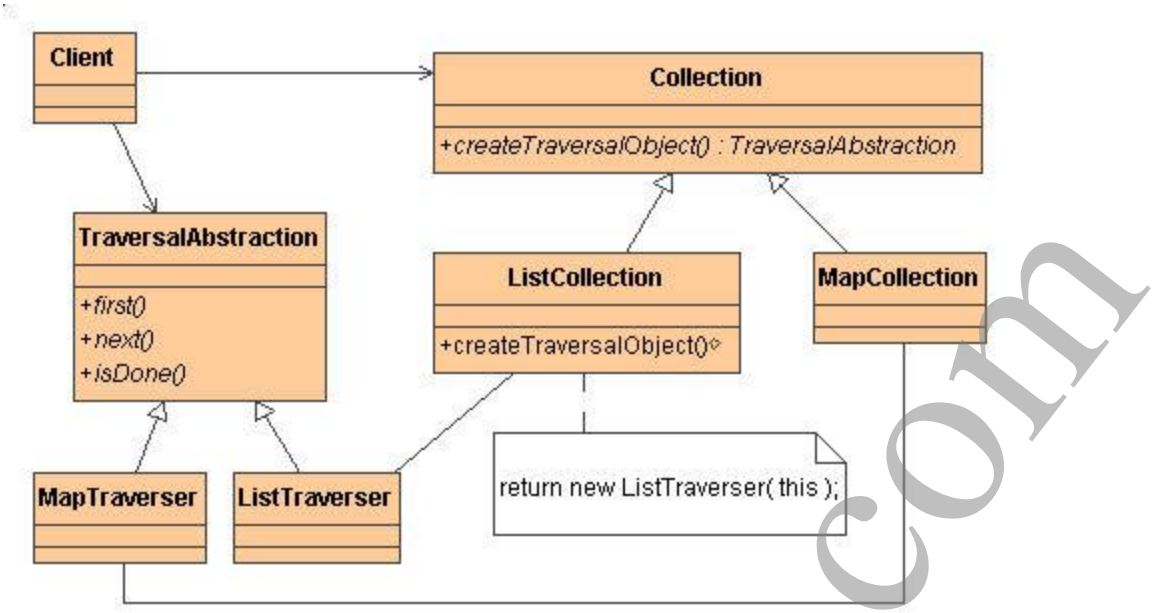
Memento

Command can use Memento to maintain the state required for an undo operation. [GoF, 242]



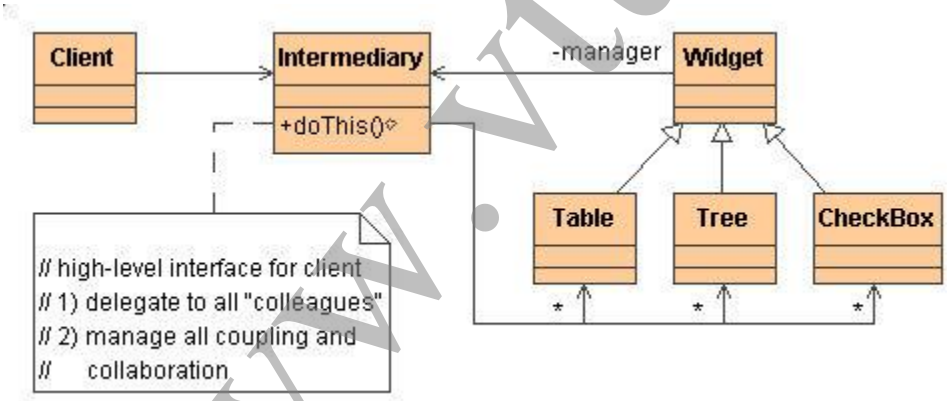
Iterator

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GoF, p271]



Mediator

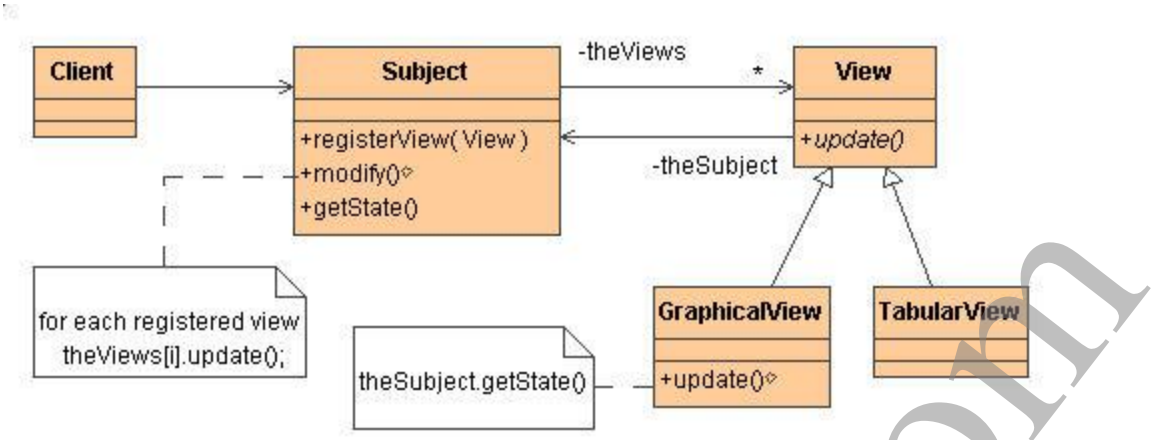
Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes. [GoF, p193]



Observer

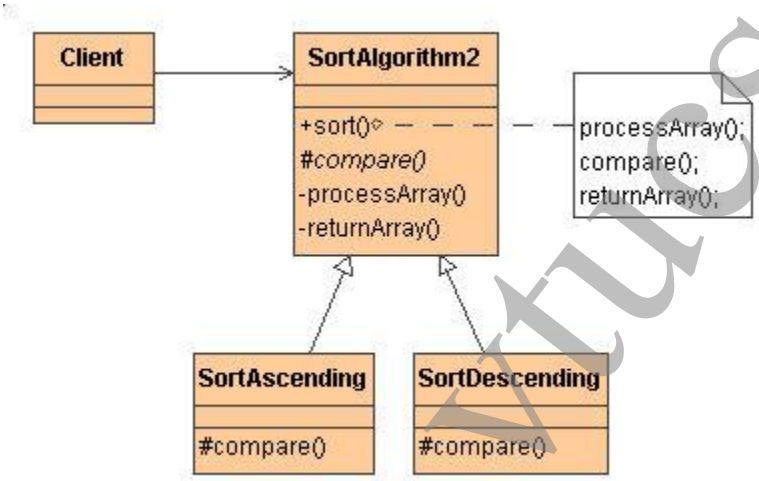
Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GoF, p346]

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GoF, p282]



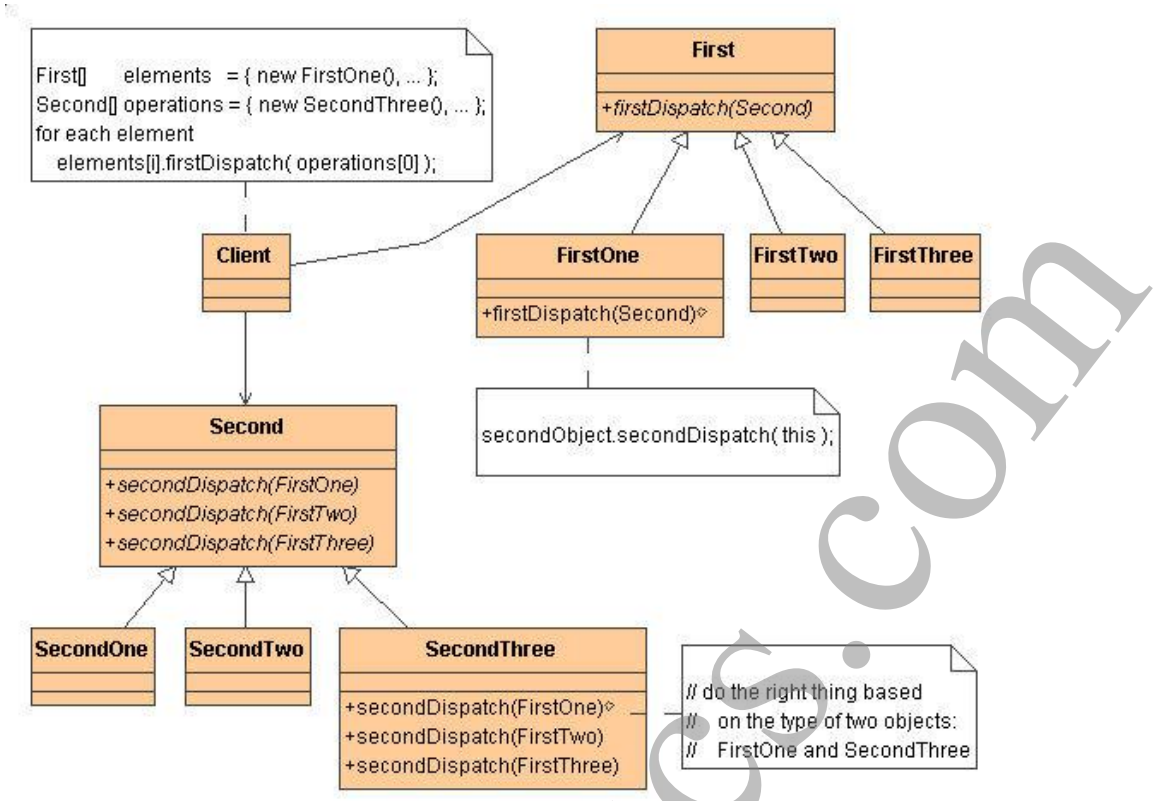
Template Method

Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GoF, p330]



Visitor

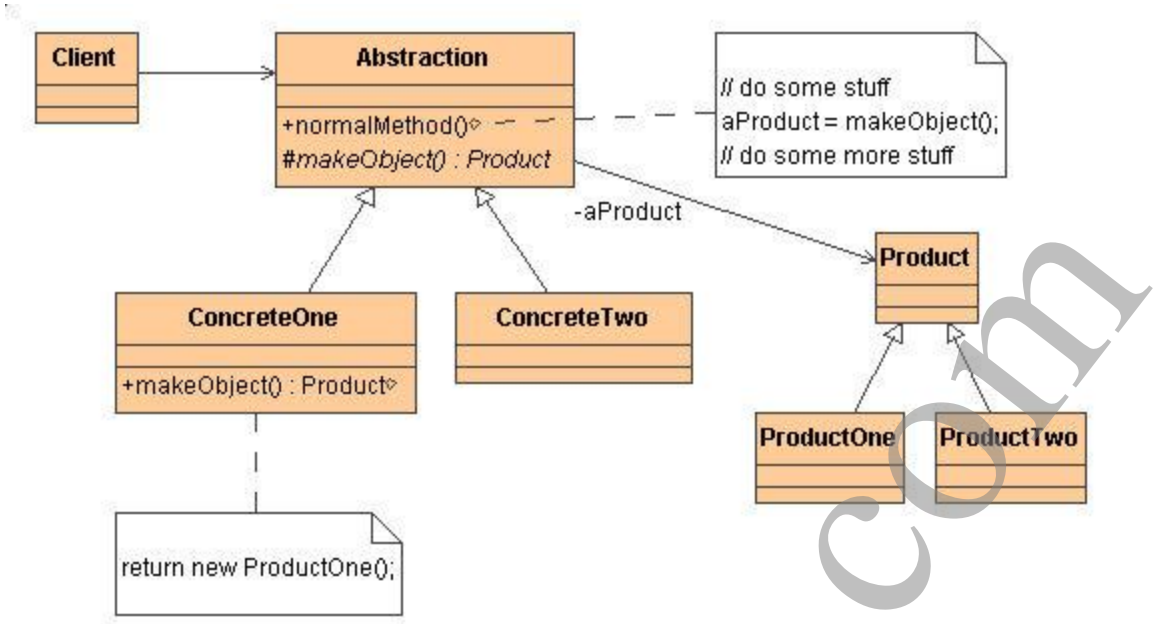
The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts. [Vlissides, "Type Laundering", *C++ Report*, Feb 97, p48]



Factory Method

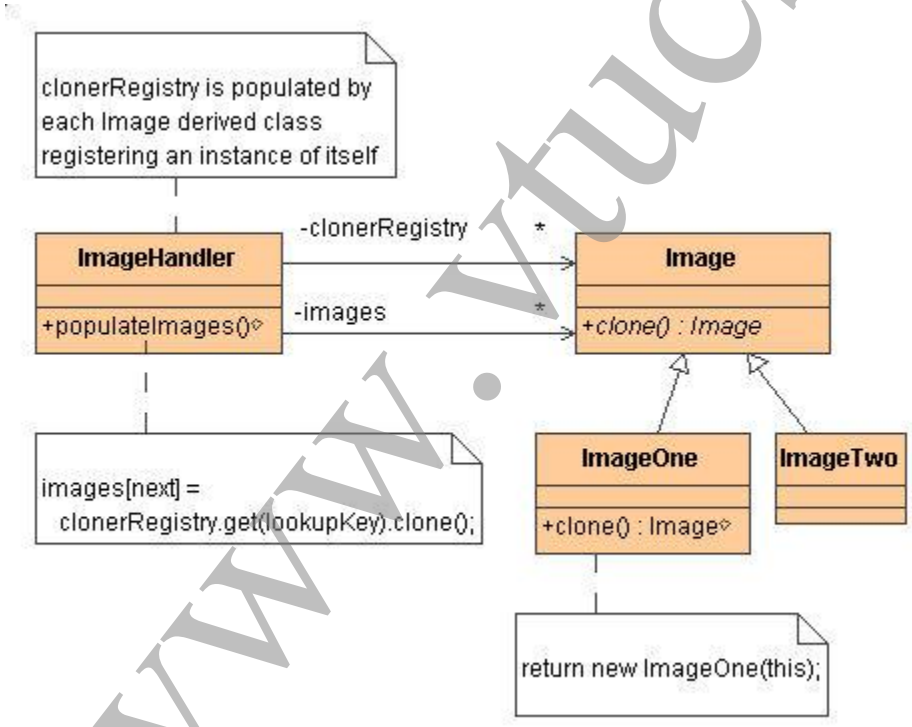
Factory Methods are usually called within Template Methods. [GoF, p116]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GoF, p136]



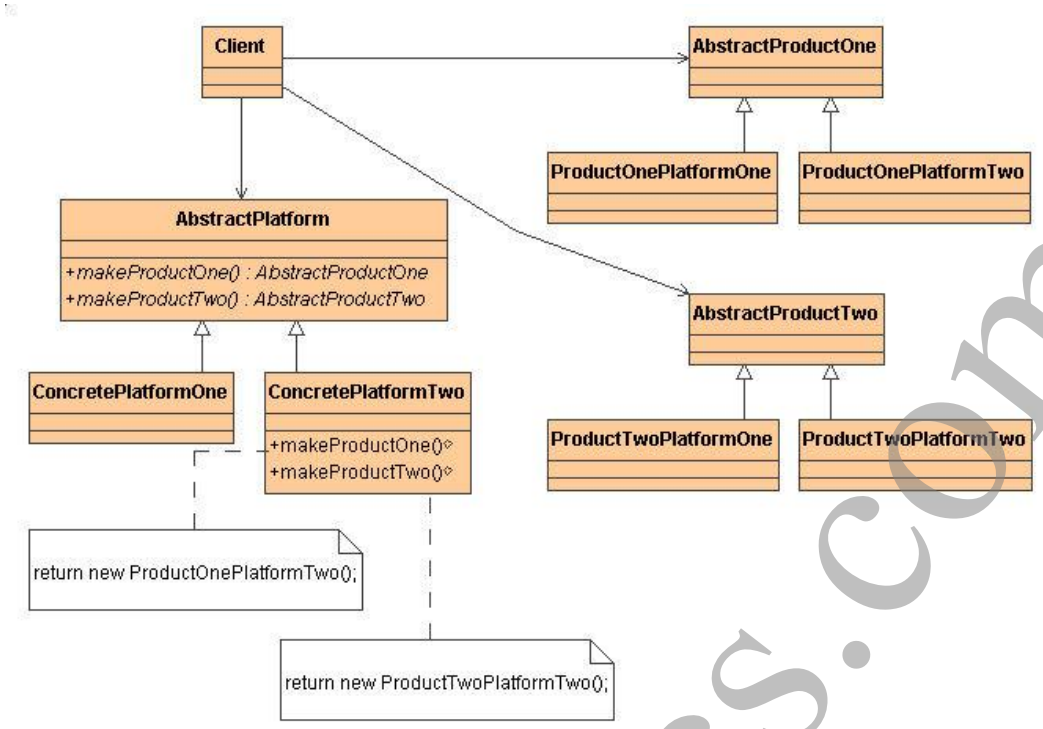
Prototype

Factory Method: creation through inheritance. Prototype: creation through delegation.



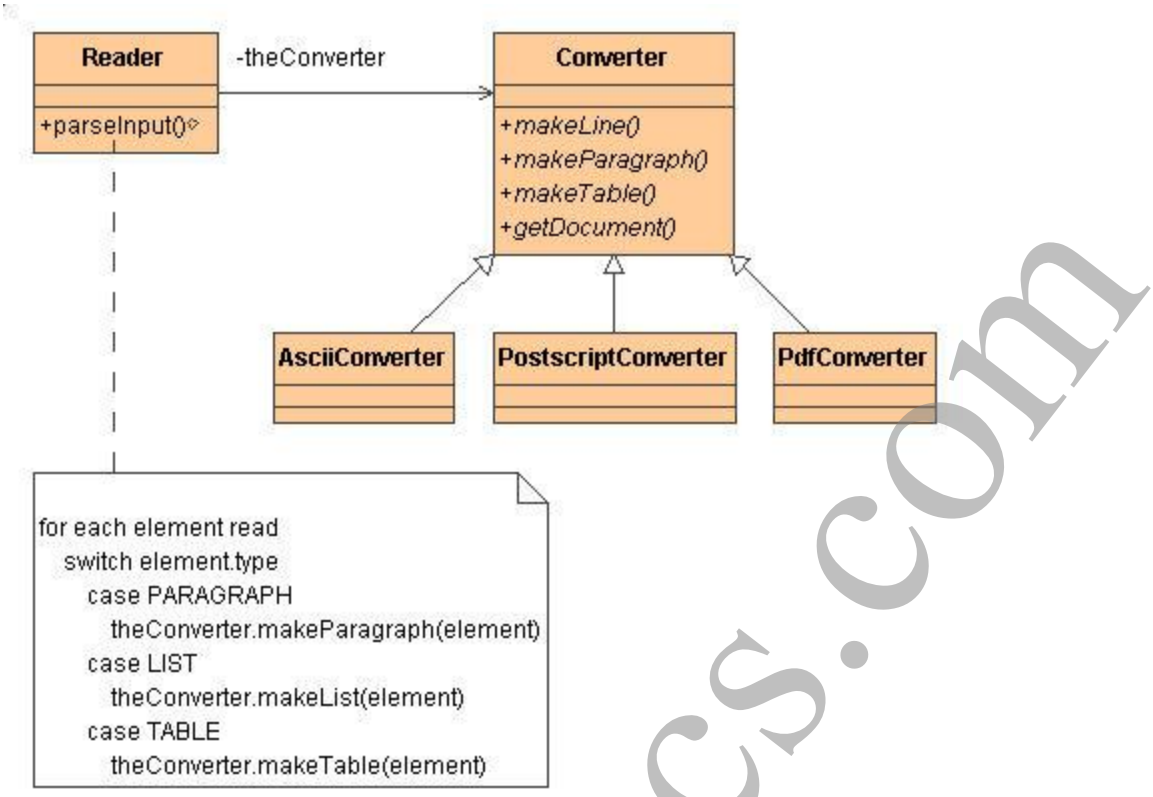
Abstract Factory

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GoF, p95]



Builder

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GoF, p105]



Singleton

Singleton should be considered only if all three of the following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

Designing The Algorithm

An algorithm is a general solution of a problem which can be written as a verbal description of a precise, logical sequence of actions. Cooking recipes, assembly instructions for appliances and toys, or precise directions to reach a friend's house, are all examples of algorithms. A computer program is an algorithm expressed in a specific programming language. An algorithm is the key to developing a successful program.

Suppose a business office requires a program for computing its payroll. There are several people employed. They work regular hours, and sometimes overtime. The task is to compute pay for each person as well as compute the total pay disbursed.

Given the problem, we may wish to express our recipe or *algorithm* for solving the payroll problem in terms of repeated computations of total pay for several people. The logical modules involved are easy to see.

Legacy Systems

- Software systems developed specifically for an organisation tend to have a long lifetime
- Many software systems that are still in use were developed using technologies now obsolete
- These systems are may be business critical - essential for the normal functioning of the business
- These are LEGACY SYSTEMS
-

Legacy system replacement

- The risk in simply scrapping a legacy system and replacing it with a new development is high.
- Legacy systems rarely have a complete specification. Major changes which may or may not have been documented will have happened in their lifetime
- Business processes are reliant on the legacy system
- The system may embed business rules that are not formally documented elsewhere – there is KNOWLEDGE in the systems
- New software development is risky and may not be successful

Legacy system change

- Systems must change in order to remain useful
- But changing legacy systems is expensive
- Different parts implemented by different teams - no consistent programming style
- The system may use an obsolete programming language
- The system documentation is either not there or out-of-date
- The system structure may be corrupted by many years of maintenance
- Techniques to save space or increase speed at the expense of understandability may have been used
- File structures used may be incompatible

The legacy dilemma

- It is expensive and risky to replace the legacy system
- It is expensive to maintain the legacy system

Businesses must weigh up the costs and risks and may choose to extend the system lifetime using techniques such as re-engineering.

Legacy system structures

Legacy systems can be considered to be socio-technical systems and not simply software systems

System hardware - may be mainframe hardware

Support software - operating systems and utilities

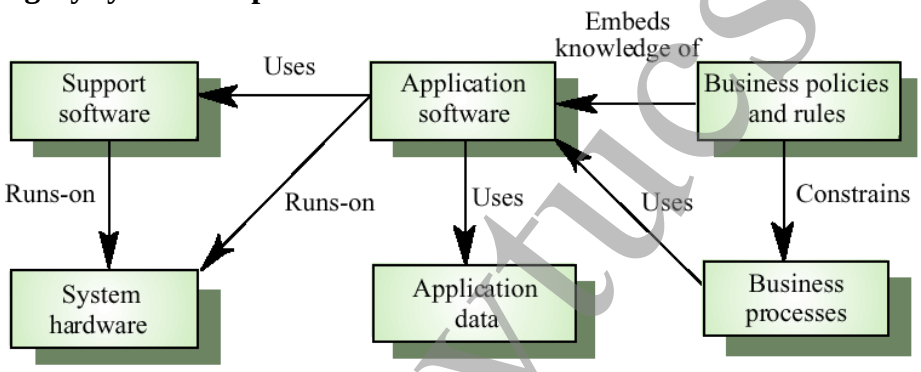
Application software - several different programs

Application data - data used by these programs that is often critical business information

Business processes - the processes that support a business objective and which rely on the legacy software and hardware

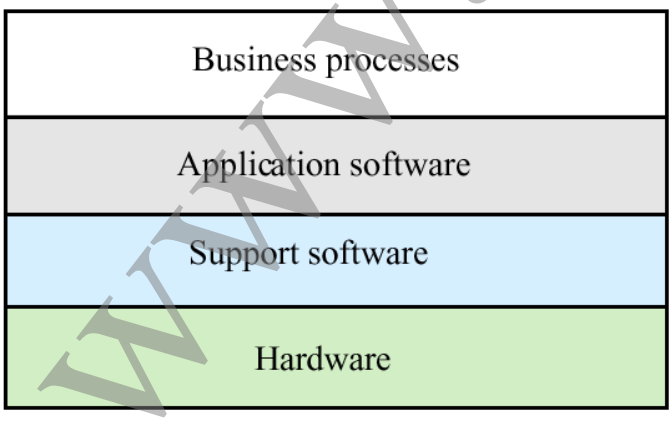
Business policies and rules - constraints on business operations

Legacy system components



Layered model

Socio-technical system



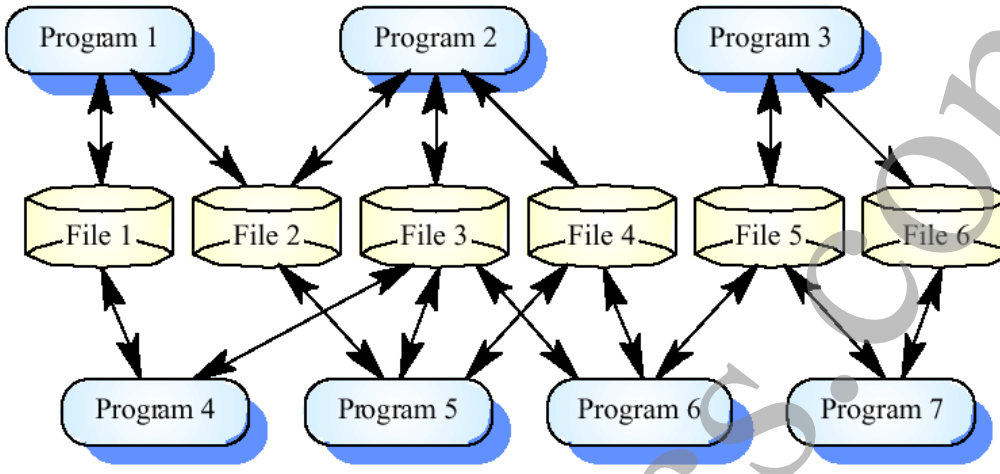
System change

In principle, it should be possible to replace a layer in the system leaving the other layers unchanged

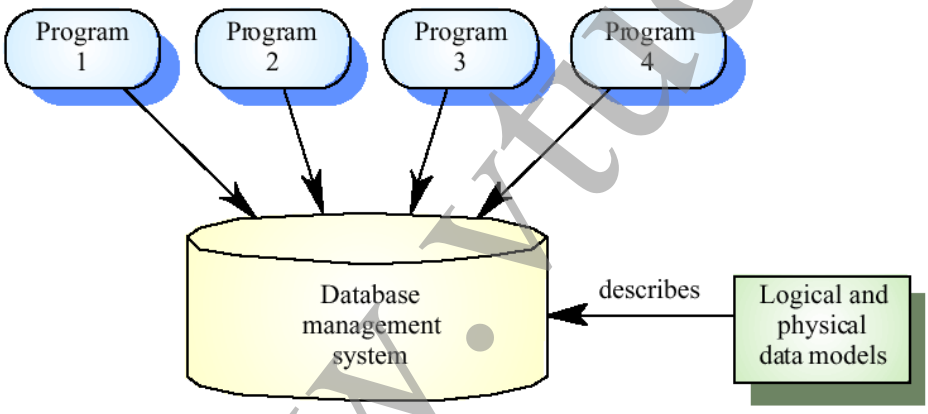
In practice, this is usually impossible

- Changing one layer introduces new facilities and higher level layers must then change to make use of these
- Changing the software may slow it down so hardware changes are then required
- It is often impossible to maintain hardware interfaces because of the wide gap between mainframes and client-server systems

Legacy application system



Database-centred system



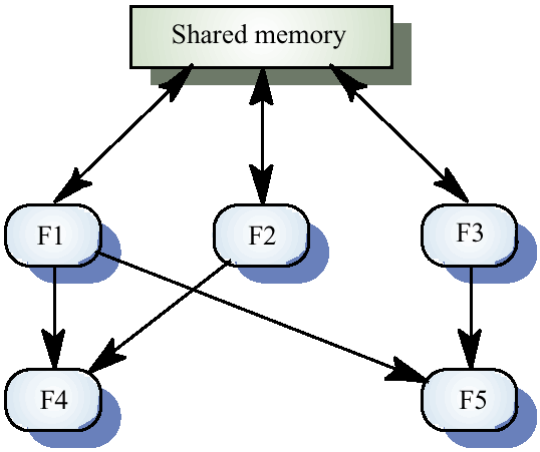
Legacy data

- The system may be file-based with incompatible files. The change required may be to move to a database-management system
- In legacy systems that use a DBMS the database management system may be obsolete and incompatible with other DBMSs used by the business

Legacy system design

- Most legacy systems were designed before object-oriented development was used
- Rather than being organised as a set of interacting objects, these systems have been designed using a function-oriented design strategy
- Several methods and CASE tools are available to support function-oriented design and the approach is still used for many business applications

A function-oriented view of design



Functional design process

- Data-flow design
- Model the data processing in the system using data-flow diagrams
- Structural decomposition
- Model how functions are decomposed to sub-functions using graphical structure charts
- Detailed design
- The entities in the design and their interfaces are described in detail. These may be recorded in a data dictionary.

Data flow diagrams

- Show how an input data item is functionally transformed by a system into an output data item
- Are an integral part of many design methods and are supported by many CASE systems
- May be translated into either a sequential or parallel design. In a sequential design, processing elements are functions or procedures; in a parallel design, processing elements are tasks or processes

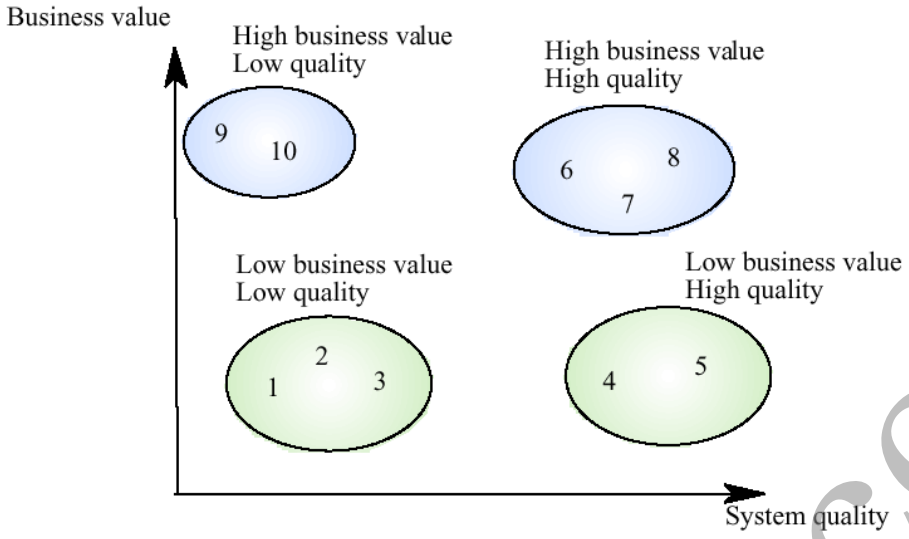
Using function-oriented design

- For some classes of system, such as some transaction processing systems, a function-oriented approach may be a better approach to design than an object-oriented approach
- Companies may have invested in CASE tools and methods for function-oriented design and may not wish to incur the costs and risks of moving to an object-oriented approach

Legacy system assessment

- Organisations that rely on legacy systems must choose a strategy for evolving these systems
- Scrap the system completely and modify business processes so that it is no longer required
- Continue maintaining the system
- Transform the system by re-engineering to improve its maintainability
- Replace the system with a new system
- The strategy chosen should depend on the system quality and its business value

System quality and business value



Legacy system categories

- Low quality, low business value
- These systems should be scrapped
- Low-quality, high-business value
- These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available
- High-quality, low-business value
- Replace with COTS, scrap completely or maintain
- High-quality, high business value
- Continue in operation using normal system maintenance

Business value assessment

- Assessment should take different viewpoints into account
- System end-users
- Business customers
- Line managers
- IT managers
- Senior managers
- Interview different stakeholders and collate results

System quality assessment

- Business process assessment
- How well does the business process support the current goals of the business?
- Environment assessment
- How effective is the system's environment and how expensive is it to maintain
- Application assessment
- What is the quality of the application software system

Business process assessment

- Use a viewpoint-oriented approach and seek answers from system stakeholders
- Is there a defined process model and is it followed?

- Do different parts of the organisation use different processes for the same function?
- How has the process been adapted?
- What are the relationships with other business processes and are these necessary?
- Is the process effectively supported by the legacy application software?

Environment assessment

Application assessment

System measurement

- You may collect quantitative data to make an assessment of the quality of the application system
- The number of system change requests
- The number of different user interfaces used by the system
- The volume of data used by the system

Key points

- A legacy system is an old system that still provides essential business services
- Legacy systems are not just application software but also include business processes, support software and hardware
- Most legacy systems are made up of several different programs and shared data
- A function-oriented approach has been used in the design of most legacy systems

Key points

- The structure of legacy business systems normally follows an input-process-output model
- The business value of a system and its quality should be used to choose an evolution strategy
- The business value reflects the system's effectiveness in supporting business goals
- System quality depends on business processes, the system's environment and the application software

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 interface". 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

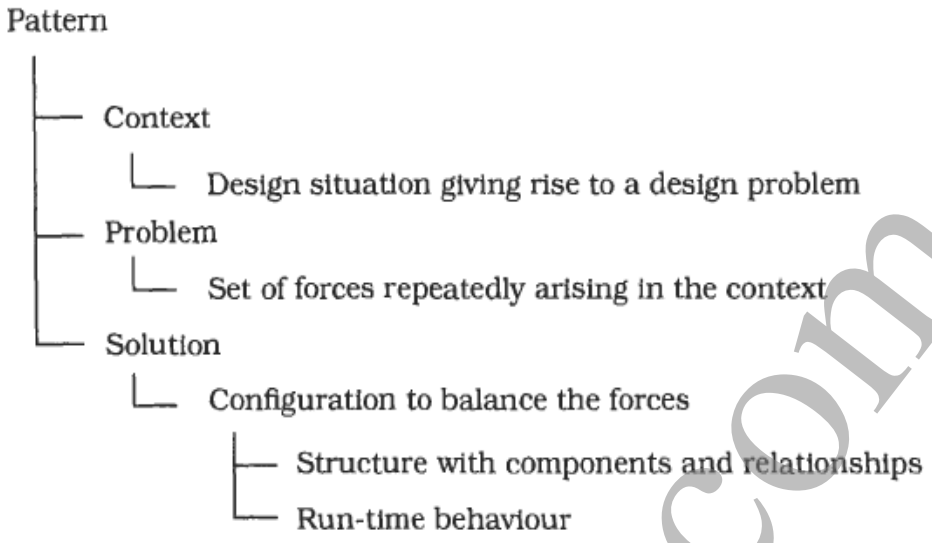
Solution:

- 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 Categories

we group patterns into three categories:

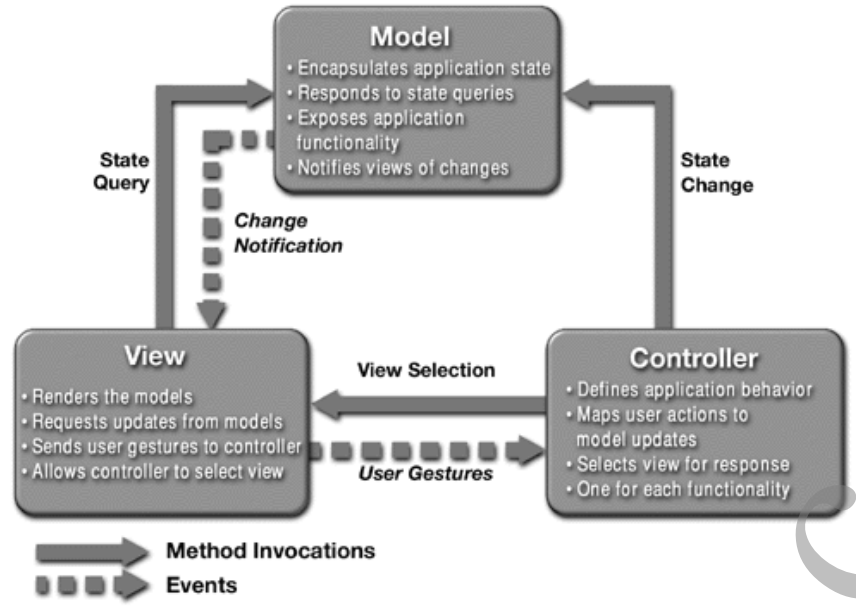
- Architectural patterns
- Design patterns
- Idioms

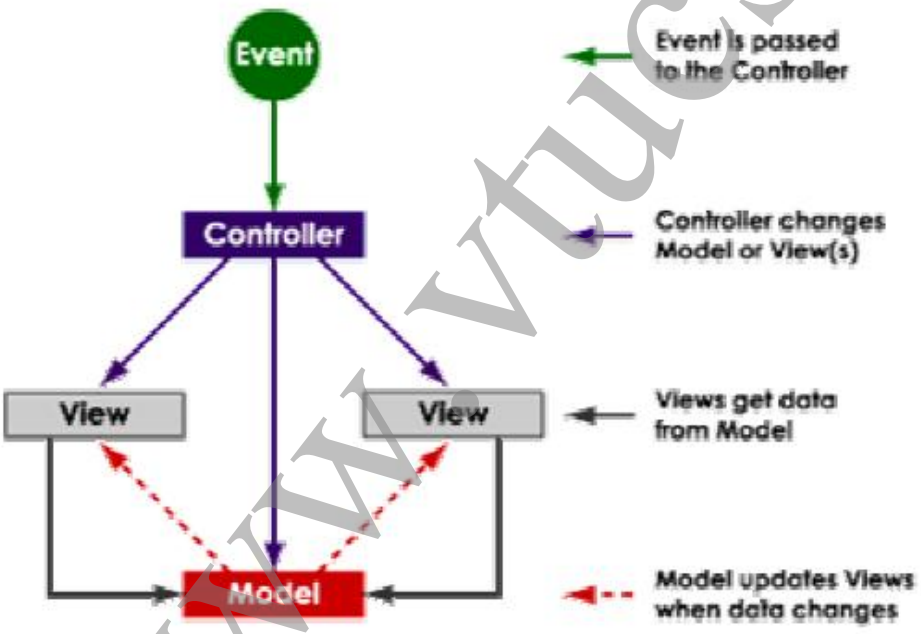
Each category consists of patterns having a similar range of scale or abstraction.

Architectural patterns

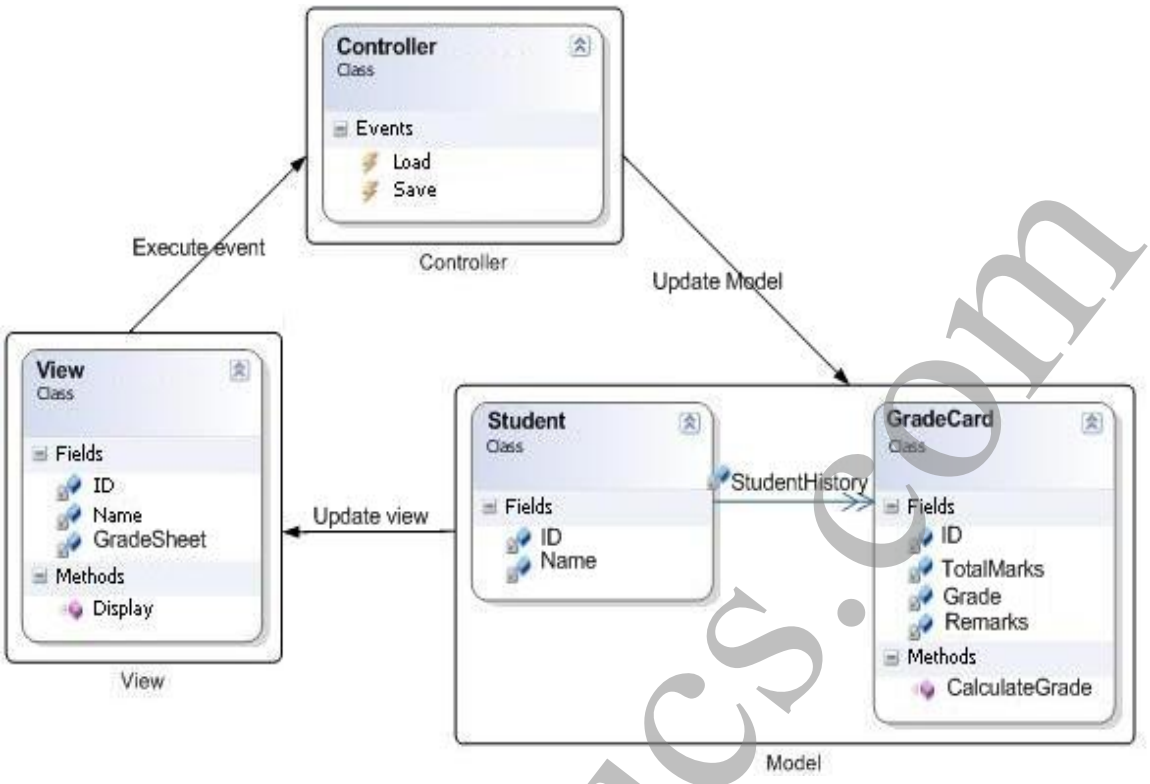
- Architectural patterns are used to describe viable software architectures that are built according to some overall structuring principle.
- Definition: An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems,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

Structure→





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.

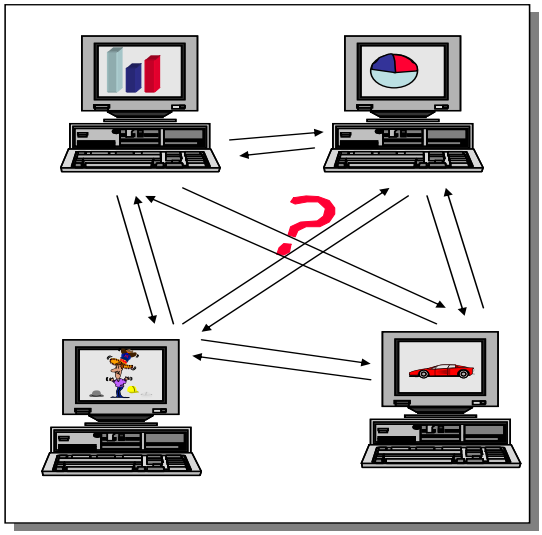
Idioms

- 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 (see text book for 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:



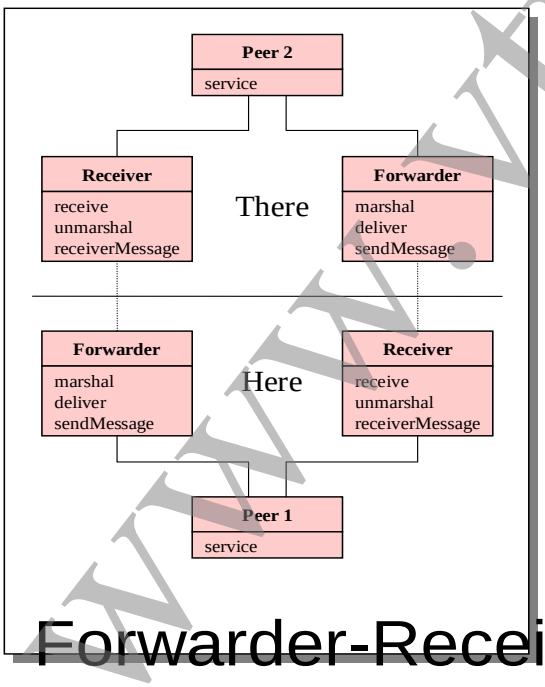
Forwarder-Receiver

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:

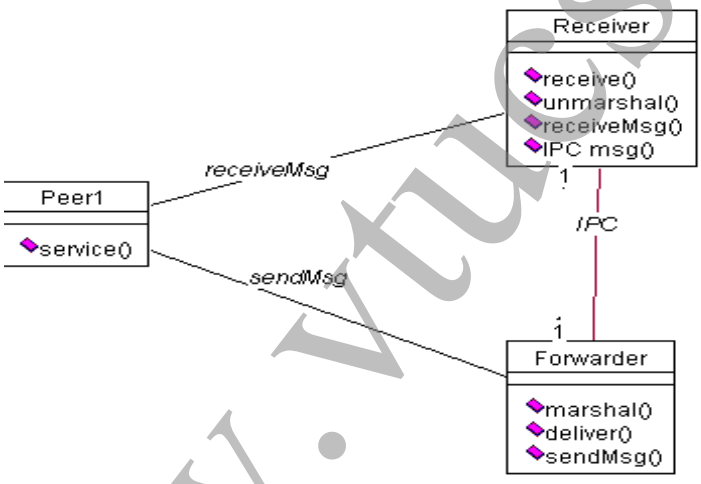
- *Peers* implement application services.
- *Forwarders* are responsible for sending requests or messages to remote peers using a specific IPC mechanism.
- *Receivers* are responsible for receiving IPC requests or messages sent by remote peers using a specific IPC mechanism and dispatching the appropriate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

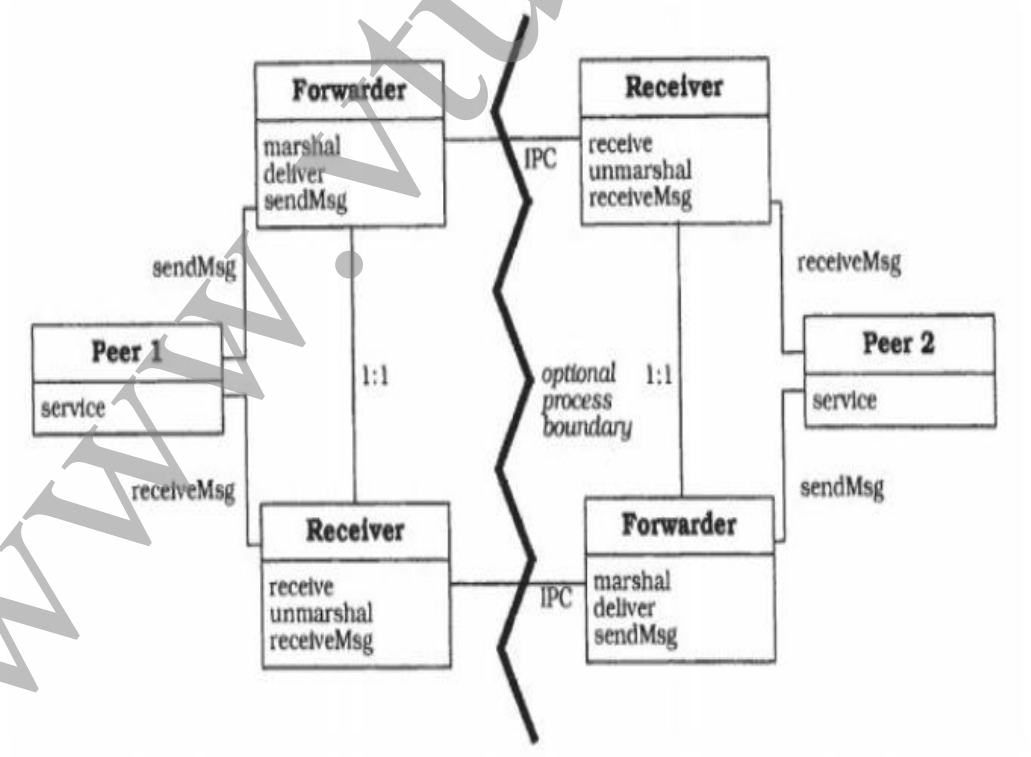
  recvmsg() unmarshal(the msg)
- Implement the peers of the application – partitioning into client and servers.

- 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.



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 interfaces 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 interface 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:
  - The abstract command component
  - A command component
  - The controller component
  - The command processor component
  - The supplier component

Components

- **Abstract command Component:**
- Defines a uniform interface 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 interface Interface to execute commands • Extends the interface 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 interface 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 interface of abstract command • Uses suppliers to perform requests	

- **The Controller Component:**
- Represents the interface 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	

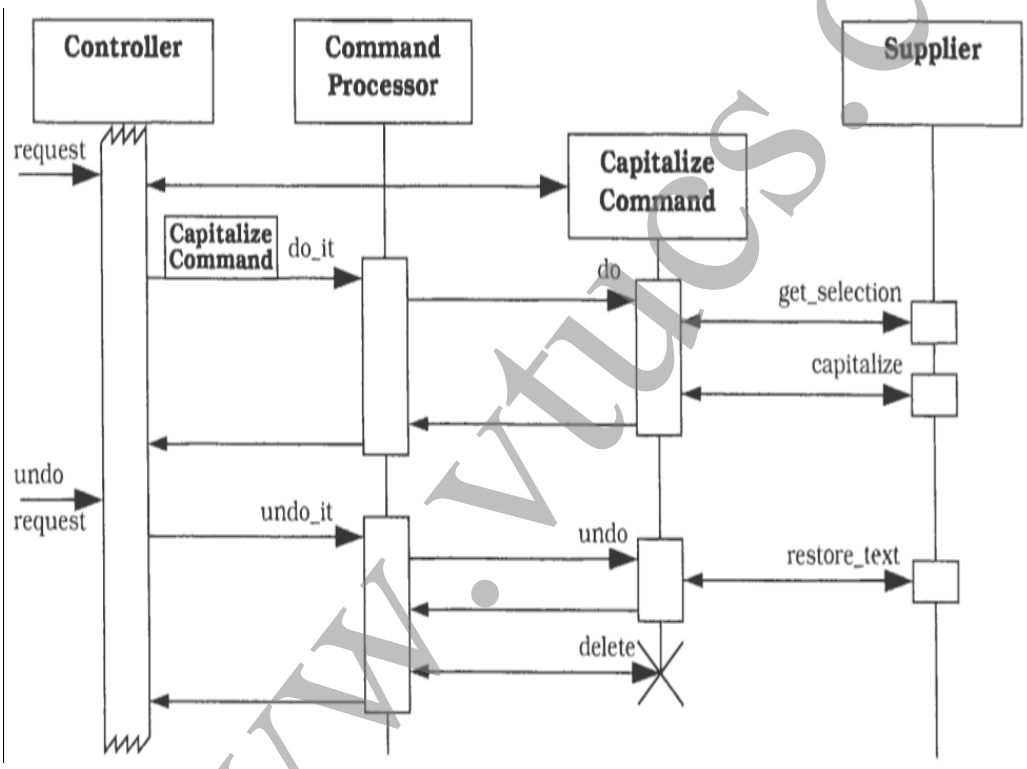
- **Command processor Component:**
- Manages command objects, schedule them and start their execution
- Key component that implements additional services (e.g. stores commands for later undo)
- Remains independent of specific commands (uses abstract command interface)

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.

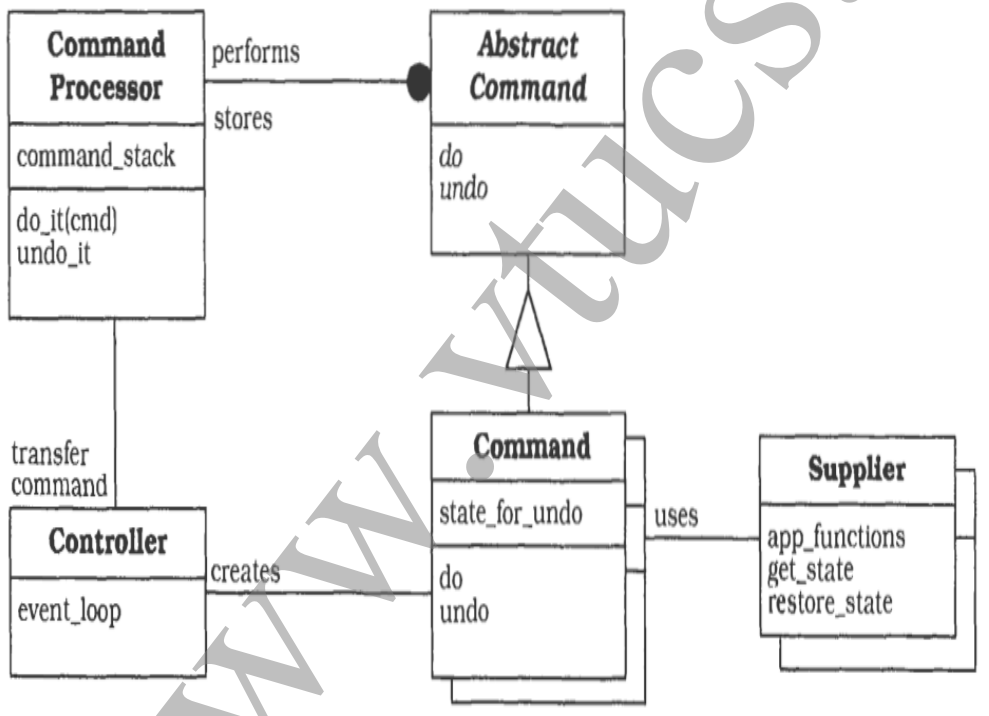
After accepting an undo request, the controller transfers this request to the command processor.

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

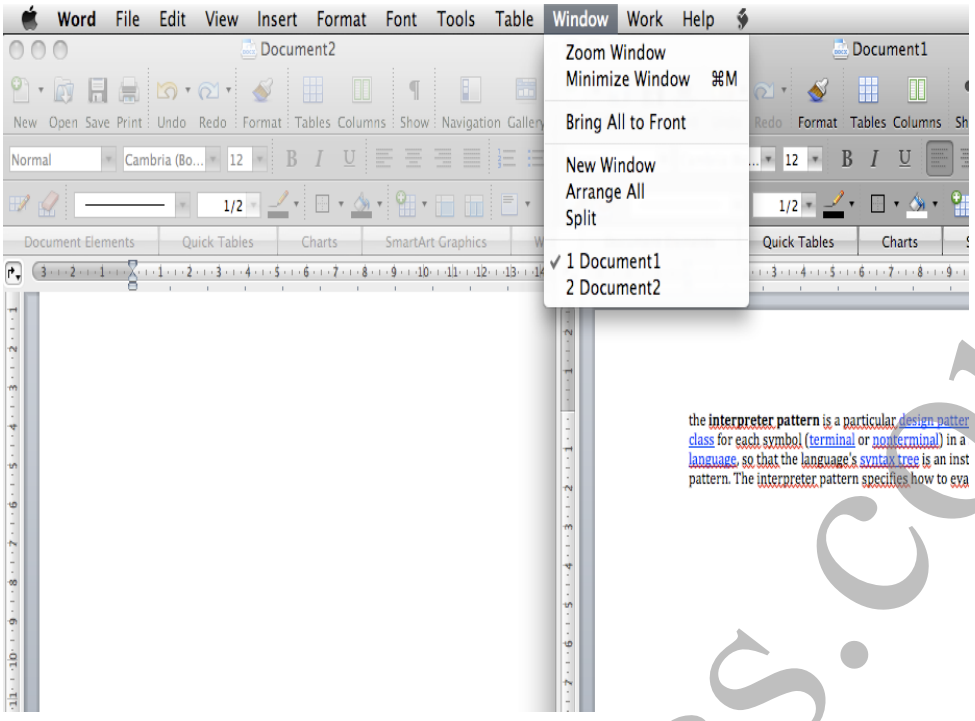
▫ Complexity in acquiring command parameters

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 interface
- 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



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 interface 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 interface

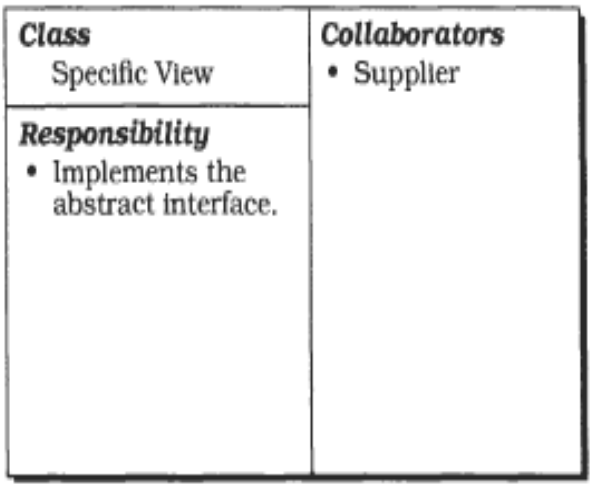
• Knows how to display itself

- Retrieves data from supplier(s) and change data

• Prepares data for display

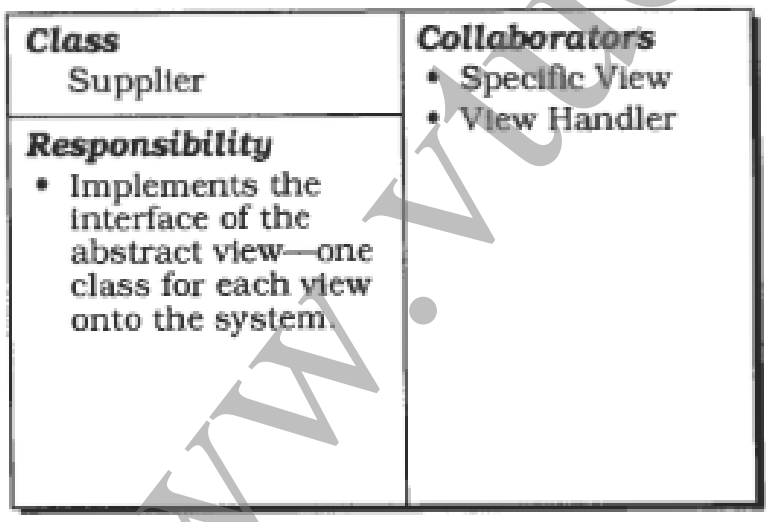
• Presents them to the user

• Display function called when opening or updating a view

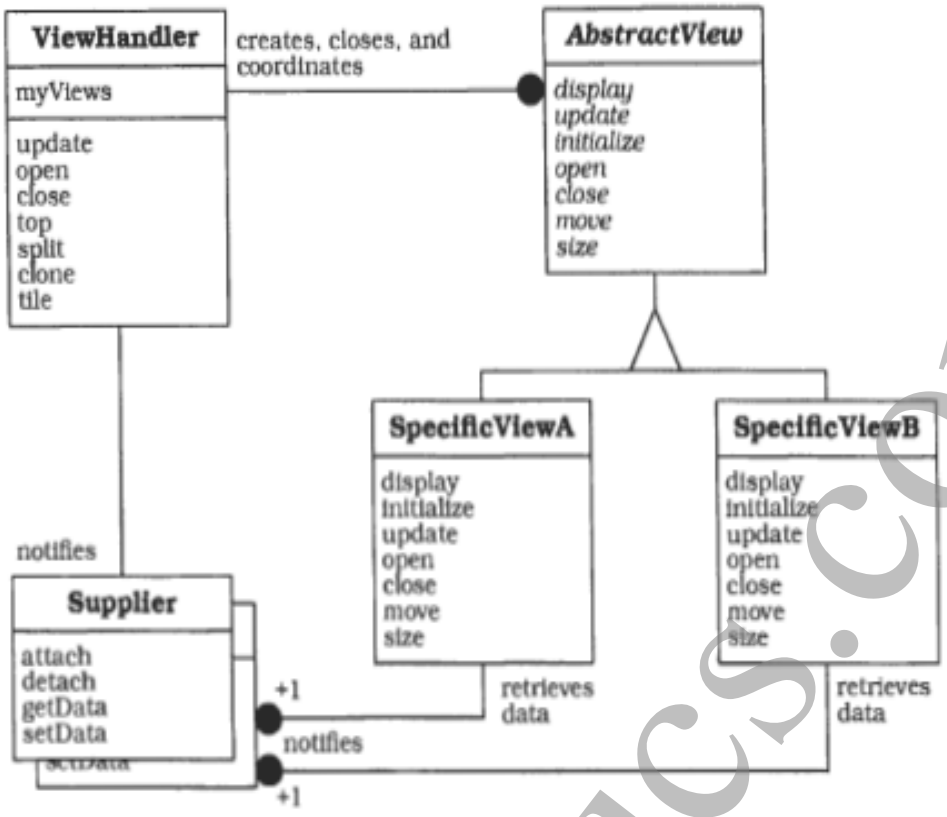


❖ **Components**

- **Supplier**
  - Provides the data that is displayed by the view components
  - Offers interface to retrieve or change data
  - Notifies dependent component about changes in data



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

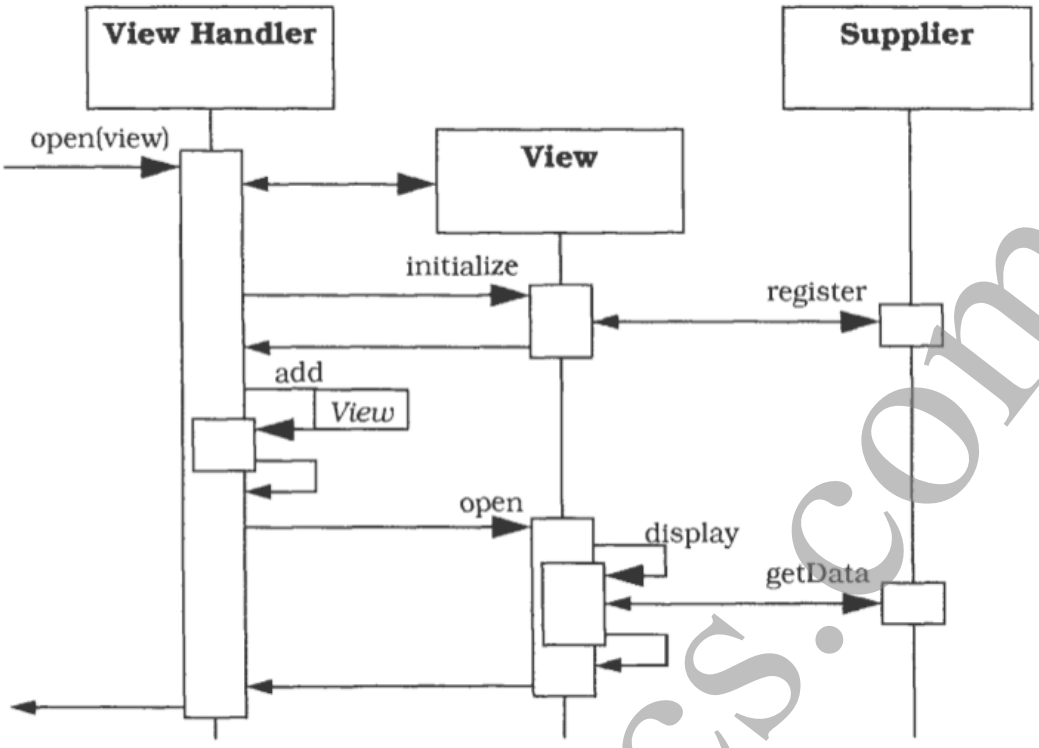
Both scenarios assume that each view is displayed in its own window.

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

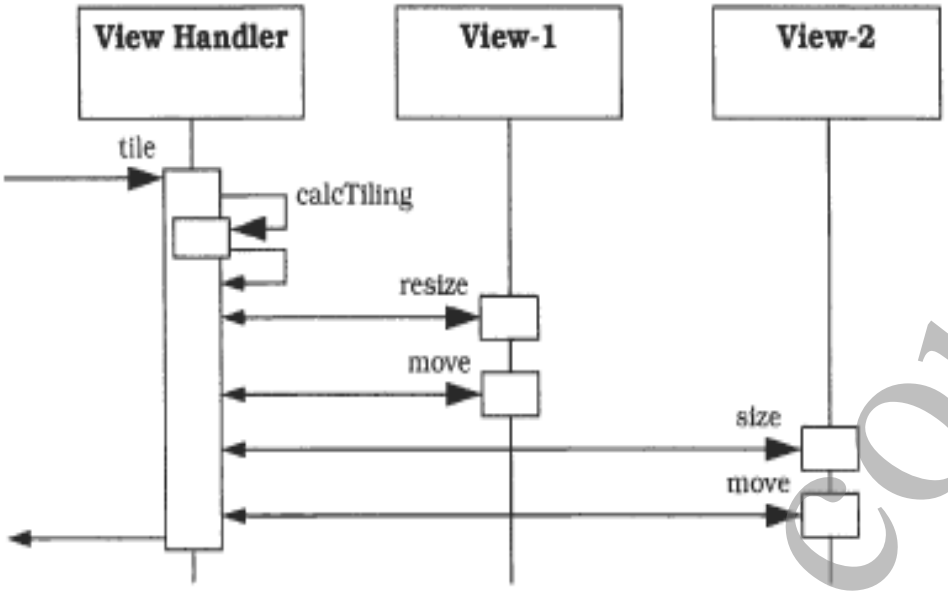
Illustrates how the view handler organizes the tiling of views. For simplicity, we assume that only two views are open. The scenario is divided into three phases:

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

The public interface 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 interface, 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 interface
- 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 interface
- 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.