UNIT 1 INTRODUCTION, MODELING CONCEPTS, CLASS MODELING

Dec.09/Jan.10

1.a. Explain the models in OO development. Bring out the relationships among the models. (10 marks)

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

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

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

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

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

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

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

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

Relationship among the models

Each model describes one aspect of the system but contains reference to the other models.

The class model describes the data structure on which the state and interaction model operate. The operations in the class model correspond to event and actions. The state model describes the control structure of objects. It shows decisions that depend on object values and causes actions that change object values and state. The interaction model focuses on the exchanges between objects and provides a holistic overview of the operation of a system.

b. With the help of a sample class model explain the following: (10 marks)
i. attributes and operations

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

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

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

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

A method is the implementation of an operation for a class.
ii. qualified association: **Qualified Associations:** is an association in which an attribute called the qualifier disambiguates the objects for a “many” association end. It is possible to define qualifiers for one – to – many and many –to – many associations.

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

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

![Diagram of Bank and Account with qualified association](image)

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

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

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

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

iv. association end names:

Multiplicity implicitly referred to the ends of associations.

Example: one – to – many associations has 2 ends – an end with a multiplicity “one” and an end with a multiplicity “many”.
Association end names often appear as nouns in problem descriptions. A name appears next to the association end.

Person employee employer Company

\[ \text{WorksFor} \quad 0..1 \]

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

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

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

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

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

A large hollow arrowhead denotes generalization. The arrowhead points to the super class.

**Dec 10/Jan**

1 a) **What is Object-Orientation? Explain four aspect of OO approach.**

Object orientation means we organize software as a collection of discrete object that incorporate both data structure and behaviour.

- The object-oriented way is to say “do the thing", and leaves it up to the object to carry out the request
- The procedural way is to specify each and every detail.

**Four aspect:**

**Identity:** Data is quantized into discrete, distinguishable entities called object.

E.g.: car, my workstation etc.

**Classification:** means object with same data structure(attributes) and behavior(operations) are grouped into a class. E.g. ChessPiece.
Each class is described possibly infinite set of individual objects. Each class is said to be an **instance** of a class.

**Inheritance:** Sharing of attributes and operations (features) among classes based on a hierarchical relationship. E.g.- Windows scrollbar

**Polymorphism:** Same operation may behave differently for different classes.
E.g. Move operation of chess.

b) **What is UML? What is the importance of UML?**

Unified Modeling Language, which is mainly a collection of graphical notation that methods use to express the designs.

The UML is language for visualizing, specifying, constructing and documenting the artifacts of software system.

UML is not a method or process but is the means to express the same

**Importance:**
- Captures business processes
- Enhance communication and ensures the right communication
- Capability to capture the logical software architecture independent of the implementation language
- Manages the complexity
- Enables reuse of design
- UML is highly successful and replace other notations available
- UML provides a very robust set of notation which grows from analysis to design.

b) **List the three kinds of models used in OOMD to describe a system. Also explain the relationship among them.**

**Class** model describe the static structure of an object of a system

- a. Class diagrams nodes are classes and arc are relationship between the classes
- **State** model describes the aspects of an object that change over time (Activity, State chart)

- b. State diagram is a graph whose nodes are state and arcs are transmission between states caused by event
- **Interaction** model describes how the object in a system cooperate to achieve the result
  - Use case diagram
  - Sequence diagram
  - Activity diagram

**Relationship:**

Each model describes one aspect of the system but contains references to other model.

- The class model describes data structure on which the state and interaction model operate. The operations in the class model correspond to event and actions.
- The state model describes the control structure of objects. It shows decisions that depend on object values and causes action that change
object values and state.

- The interaction model focuses on the exchange between objects and provides a holistic overview of operation of a system.

May/June 2010

1 a) Explain briefly 3 models used to describe a System? (10 marks)

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

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

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

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

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

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

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

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

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

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

b) Explain with diagram, how Association class Participated in another association? (6 Marks)

Multiplicity implicitly referred to the ends of associations.

Example: one – to – many associations has 2 ends – an end with a multiplicity “one” and an end with a multiplicity “many”.
Association end names often appear as nouns in problem descriptions. A name appears next to the association end.

```
Person                                      employee                      employer
    
WorksFor                  0..1                      Company
```

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

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

c) Explain Models and its Purposes?  4 marks

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

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

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

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

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

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

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

**Dec 08**

1. a. Explain with a diagram, how an association class participates in another association. **(10 marks)**

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

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

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

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

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

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

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

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

1.b. Explain the models in OO development. Bring out the relationships among the models. (10 marks)

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

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

Class diagrams express the class model. Classes define the attribute values carried by each object and the operations that each object performs or undergoes.
State Model: describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that context for events and the organization of events and states.

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

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

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

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

Relationship among the models

Each model describes one aspect of the system but contains reference to the other models.

The class model describes the data structure on which the state and interaction model operate. The operations in the class model correspond to event and actions. The state model describes the control structure of objects. It shows decisions that depend on object values and causes actions that change object values and state. The interaction model focuses on the exchanges between objects and provides a holistic

June 2012

1. a. Explain how systems are modeled from different viewpoints. (10 Marks)

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

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

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

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

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

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

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

b Elaborate on the major themes that are well supported in object oriented technology (10 Marks)

- Abstraction means focusing on what an object is and does, before deciding how to implement it.
- Encapsulation separates the external aspects of an object that are accessible to other objects, from the internal implementation details that are hidden from other objects.
- Combining data and behavior: The caller of an operation need not consider how many implementations exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy.
- Sharing: OO technologies promote sharing at different levels. Inheritance of both data structure and behavior lets subclasses share common code. This sharing via inheritance is one of the main advantages of OO languages.
  
  OO development not only lets you share information within an application but also offers the prospect of reusing designs and code on future projects.

- Emphasis on the essence of object:
  
  OO technology stresses what an object is, rather than how it is used. The uses of an object depend on the details of the application and often change during development.

- Synergy: Identity, classification, polymorphism and inheritance characterize OO languages. Each of these concepts can be used in isolation but together they complement each other synergistically.

UNIT2 ADVANCED CLASS MODELING, STATE MODELING

Dec.09/Jan.10

2. a. Explain the properties of association ends. (10 marks)

Association end is an end of an association. It has the following properties:

Association end names: An association end may have a name. Meaningful names often arise and it is useful to place the names within proper context.

Multiplicity: can be specified for each association end. “1” exactly one, “0..1” at most one, “*” many.

Ordering: The objects for “many” association end are usually just a set. But sometimes they have explicit order.
Bags and Sequences: The objects for a “many” association end can also be a bag or sequence.

Qualification: One or more qualifier attributes can disambiguate the objects for a “many” association end.

Association ends have some additional properties.

Aggregation: The association end may be an aggregate or constituent part.

Only a binary association can be an aggregation, one end must be an aggregate and the other
must be a constituent.

Changeability: specifies the update status of an association end. Possibilities are changeable
(able to update) and read-only (can only be initialized).

Navigability: An association may be traversed in either direction. But some implementation
may support only one direction. The UML shows navigability with an arrowhead on the
association end attached to the target class.

Visibility: Similar to the attributes and operations, association ends may be public, protected,
private or package.

b. Define an event in state modeling. Explain the kinds of events. (10 marks)

An event is an occurrence at a point in time. Events often correspond to verbs in the past tense.
An event happens instantaneously with regard to the time scale of an application.

There are several kinds of events. The most common are the signal event, the change event and
the time event.

A signal is an explicit one way transmission of information from one object to another . A
signal event is the event of sending or receiving a signal.

A change event is an event that is caused by the satisfaction of a Boolean expression. The intent
of a change event is that the expression is continually tested whenever the expression changes
from false to true, the event happens.

A time event is an event caused by the occurrence of an absolute time or the elapse of a time
interval.

Dec 10/Jan 11

2 a) List the three kinds of models used in OOMD to describe a system. Also explain the
relationship among them. (10 marks)
Class model describe the static structure of an object of a system
  Class diagrams nodes are classes and arc are relationship between the classes
State model describes the aspects of an object that change over time (Activity, State chart)
  State diagram is a graph whose nodes are state and arcs are transmission between states caused by event
Interaction model describes how the object in a system cooperate to achieve the result
  Use case diagram
  Sequence diagram
  Activity diagram

Relationship:

Each model describes one aspect of the system but contains references to other model.

- The class model describes data structure on which the state and interaction model operate. The operations in the class model correspond to event and actions.
- The state model describes the control structure of objects. It shows decisions that depend on object values and causes action that change object values and state.
- The interaction model focuses on the exchange between objects and provides a holistic overview of operation of a system.

b) What is the association? Write a brief note on qualified association? (10 marks)

Ans: Link is a physical or conceptual connection among objects.
Most links relate two or three or more objects
A link is an instance of an association
Association is a description of a group of links with common structure and common semantics. Links of associations connect objects from the same classes.

Qualified associations:

- A qualified associations is an association in which an attribute called the qualifier disambiguates the objects for a “many” association end.
- It is possible to define qualifiers for one-to-many and many-to-many association.
- Qualified associations with a target multiplicity of “one” or “zero-or-one” specify a precise path for finding the target object from the source object.

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

The source class plus the qualifier create the target class.

May/June 2010

2a) Explain an event and different types of events? (10 marks)

An event is an occurrence at a point in time, such as user depresses left button to know various options. Another example Rail departs from Mysore to Bangalore.
There are several kinds of events
- Signal event
- Change event
- Time event

A signal is an explicit one-way transmission of information from one object to another. A signal event is the event of sending or receiving a signal. Here we are more concerned about the receipt of a signal, because it causes effects in the receiving object. A signal is a message between objects while a signal event is an occurrence in time. Every signal transmission is a unique occurrence, but we group them into signal classes and give each signal class a name to indicate common structure and behavior. Let us consider an example: Kingfisher flight 123 departs from Mysore on March 23, 2010 is an instance of signal class Flight-Departure. Some signals are simple occurrences, but most signal classes have attributes indicating the values they convey. The UML notation is the keyword `signal` in guillemets `(<< >>)` above the signal class in the top section of a box. The second section lists the signal attributes.

A change event is an event that is caused by the satisfaction of a Boolean expression. Event of the expression is continually tested – whenever the expression changes from false to true, the event occurs. The UML notation for change event is the keyword `when` followed by a parenthesized Boolean expression.

A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval. The 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.

b) Define Reification? Explain it with diagram? (4 marks)
• IT is the promotion of something that is not an object into an object, It lets you shift the level of abstraction. Promote attributes, methods, constraints, and control information into objects A developer could write code for each application so that it can read and write from files. Reify the notion of data services and use a database manager.

c) **Explain with the diagram basic UML syntax for State Diagrams?** (6 marks)

State: Drawn as Rounded box containing an Optional name. Special name is available for initial states solid circle and final state Bulls EYE.

Transition: Drawn as a line from origin state to the Target State. An arrow head points to target state.

Event: A signal event is shown as label on a transition and may be followed by Parenthesized attributes.

State diagram: Enclosed in a rectangular frame with diagram name in a small pentagonal
tag in the upper left corner.

Guard Condition: Optionally listed in Square brackets after an event.

June 2012

2 a. List and explain the various restructuring techniques used with respect to workarounds (10 marks)

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

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

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

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

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

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

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

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

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

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

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

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

Use cases, sequence diagrams and activity diagrams document the interaction model. Use cases document major themes for interaction between the system and outside actors. Sequence diagrams show the objects that interact and the time sequence of their interactions. Activity diagrams show the flow of control among the processing steps of a computation.
b. What is a constraint with respect to a class modeling? Explain.
i) Constraints on generalization sets
ii) Constraints on links.  

(10 Marks)

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

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

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

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

The terms ancestor and descendent refer to generalization of classes across multiple levels. Use of Generalization: has 3 purposes, one of which is support for polymorphism. Polymorphism increases the flexibility of software you add a new sub class and automatically inherit super class behavior. The second purpose of generalization is to structure the description of objects. When generalization is used, you are making a conceptual statement you are forming a taxonomy and organizing objects on the basis of their similarities and differences. The third purpose is to enable reuse of code inherit code within the application as well as from part work (class library). The terms generalization, specialization and inheritance all refer to aspects of the same idea.

ii) A link is a physical or conceptual connection among objects.

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

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

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

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

Example: Model for a financial application:

Stock brokerage firms need to perform tasks such as recording ownership of various stocks, tracking dividends, alerting customers to changes in the market and computing margin requirements.
2 a Explain Models and its Purposes?
(10 marks)

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

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

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

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

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

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

2 b Explain with the diagram basic UML syntax for State Diagrams?
(4 marks)
State: Drawn as Rounded box containing an Optional name. special name is available for initial states solid circle and final state Bulls EYE.

Transition: Drawn as a line from origin state to the Target State. an arrow head points to target state.

Event: A signal event is shown as label on a transition and may be followed by Parenthesized attributes.

State diagram: Enclosed in a rectangular frame with diagram name in a small pentagonal tag in the upper left corner.

Guard Condition: Optionally listed in Square brackets after an event

2c. Explain the properties of association ends.

Association end is an end of an association. It has the following properties:

Association end names: An association end may have a name. Meaningful names often arise and it is useful to place the names within proper context.

Multiplicity: can be specified for each association end. “1” exactly one, “0..1” at most one, “*” many.

Ordering: The objects for “many” association end are usually just a set. But sometimes they have explicit order.

Bags and Sequences: The objects for a “many” association end can also be a bag or sequence.

Qualification: One or more qualifier attributes can disambiguate the objects for a “many” association end.

Association ends have some additional properties.

Aggregation: The association end may be an aggregate or constituent part.

Only a binary association can be an aggregation, one end must be an aggregate and the other must be a constituent.
**Changeability:** specifies the update status of an association end. Possibilities are changeable (able to update) and read-only (can only be initialized).

**Navigability:** An association may be traversed in either direction. But some implementation may support only one direction. The UML shows navigability with an arrowhead on the association end attached to the target class.

**Visibility:** Similar to the attributes and operations, association ends may be public, protected, private or package.

---

**UNIT 3 ADVANCED STATE MODELING, INTERACTION MODELING**

Dec.09/Jan.10

3. a. Explain the properties of association ends. (10 marks)

**Association end** is an end of an association. It has the following properties:

**Association end names:** An association end may have a name. Meaningful names often arise and it is useful to place the names within proper context.

**Multiplicity:** can be specified for each association end. “1” exactly one, “0..1” at most one, “*” many.

**Ordering:** The objects for “many” association end are usually just a set. But sometimes they have explicit order.

**Bags and Sequences:** The objects for a “many” association end can also be a bag or sequence.

**Qualification:** One or more qualifier attributes can disambiguate the objects for a “many” association end.

Association ends have some additional properties.

**Aggregation:** The association end may be an aggregate or constituent part.

Only a binary association can be an aggregation, one end must be an aggregate and the other must be a constituent.

**Changeability:** specifies the update status of an association end. Possibilities are changeable (able to update) and read-only (can only be initialized).
Navigability: An association may be traversed in either direction. But some implementation may support only one direction. The UML shows navigability with an arrowhead on the association end attached to the target class.

Visibility: Similar to the attributes and operations, association ends may be public, protected, private or package.

b. Define an event in state modeling. Explain the kinds of events. (10 marks)

An event is an occurrence at a point in time. Events often correspond to verbs in the past tense. An event happens instantaneously with regard to the time scale of an application.

There are several kinds of events. The most common are the signal event, the change event and the time event.

A signal is an explicit one way transmission of information from one object to another. A signal event is the event of sending or receiving a signal.

A change event is an event that is caused by the satisfaction of a Boolean expression. The intent of a change event is that the expression is continually tested whenever the expression changes from false to true, the event happens.

A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.

c. Give the general UML system for state diagram and explain.

A state diagram is a graph whose nodes are states and whose directed arcs are transition between states. A state diagram specifies the state sequence caused by event sequences.
State: drawn as a rounded box containing an optional name. A special notation is available for initial states (a solid circle) and final states (a bull’s eye or encircled “x”).

Transition: drawn as a line from the original state to the target state. An arrowhead points to the target state.

Event: a signal event is shown as a label on a transition and may be followed by parenthesized attributes.

State diagram: enclosed in a rectangular frame with the diagram name in a small pentagonal tag in the upper left corner.

Guard condition: optionally listed in square brackets after a new event.

Effects: can be attached to a transition or state and are listed after a slash ‘/’.

---

**June 2012**

3 a. Describe the two kinds of sequence models. What are the guidelines for sequence models?

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

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

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

Sequence diagrams are typically used to model:

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

3. **The logic of services.** A service is effectively a high-level method, often one that can be invoked by a wide variety of clients. This includes web-services as well as business transactions implemented by a variety of technologies such as CICS/COBOL or CORBA-compliant object request.

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

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

The reason why they’re called sequence diagrams should be obvious: the sequential nature of the logic is shown via the ordering of the messages (the horizontal arrows). The first message starts in the top left corner, the next message appears just below that one, and so on.
b. How an activity diagram differs from a traditional flowchart? Give the activity diagram for stock trade processing.

(08 Marks)

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

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

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

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

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

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

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

![Include Diagram]

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

![Extend Diagram]

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

**Dec 08**

3 a What is the association? Write a brief note on qualified association. (10 marks)

- Link is a physical or conceptual connection among objects.
- Most links relate two or three or more objects
- A link is an instance of an association

Association is a description of a group of links with common structure and common semantics. Links of associations connect objects from the same classes.

**Qualified associations:**
• A qualified associations is an association in which an attribute called the qualifier disambiguates the objects for a “many” association end.

• It is possible to define qualifiers for one-to-many and many-to-many association.

• Qualified associations with a target multiplicity of “one” or “zero-or-one” specify a precise path for finding the target object from the source object.

• The notation for a qualifier is a small box on the end of association line near the source class. The qualifier box may grow out any side (top, bottom, left, right) of the source class. The source class plus the qualifier create the target class.
3b What is an event? Explain its types with example (10 marks)

An event is an occurrence at a point in time. An event happens instantaneously with regard to time scale of an application.

E.g. User depresses left button or Air Deccan flight departs from Bombay.

- 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.

The difference between signal and signal event

- A signal is a message between objects
- A signal event is an occurrence in time.

**Change Event**

- A change event is an event that is caused by the satisfaction of a Boolean expression.
- Whenever the expression changes false to true the event happens
- UML notation for a change event is keyword when followed by a parenthesized Boolean expression.
  - when(room temperature < heating set point)
  - when(room temperature > cooling set point)

**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.
  - when(date = August 16, 2010)
- The notation for a time interval is the keyword after followed by a parenthesized expression that evaluates to a time duration.
  - after(10 seconds)
3 a) **What is the association? Write a brief note on qualified association.**

- Link is a physical or conceptual connection among objects.
- Most links relate two or three or more objects.
- A link is an instance of an association.

Association is a description of a group of links with common structure and common semantics. Links of associations connect objects from the same classes.

**Qualified associations:**

- A qualified associations is an association in which an attribute called the qualifier disambiguates the objects for a “many” association end.
- It is possible to define qualifiers for one-to-many and many-to-many association.
- Qualified associations with a target multiplicity of “one” or “zero-or-one” specify a precise path for finding the target object from the source object.
- The notation for a qualifier is a small box on the end of association line near the source class. The qualifier box may grow out any side (top, bottom, left, right) of the source class. The source class plus the qualifier create the target class.

b) **What is an event? Explain its types with example**

An event is an occurrence at a point in time. An event happens instantaneously with regard to time scale of an application.

E.g. User depresses left button or Air Deccan flight departs from Bombay.

- 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.

The difference between signal and signal event

- A signal is a message between objects
- A signal event is an occurrence in time.

Signal is an explicit one way transmission of information from one object to another.
Change Event

- A change event is an event that is caused by the satisfaction of a Boolean expression.
- Whenever the expression changes false to true the event happens.
- UML notation for a change event is keyword when followed by a parenthized Boolean expression.
  - when(room temperature < heating set point)
  - when(room temperature > cooling set point)

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.
  - when(date = August 16, 2010)
- The notation for a time interval is the keyword after followed by a parenthesized expression that evaluates to a time duration.
  - after(10 seconds)

c) Explain briefly the concept of class diagram with its notation.

Class diagram: provide graphic notation for modeling classes and their relationships, thereby describing possible objects. It is useful for abstract modeling and for designing actual programs. Class is concise and easy to understand.

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

<table>
<thead>
<tr>
<th>ClassName</th>
<th>GeometricObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeName1: dataType1 = defaultValue1</td>
<td></td>
</tr>
<tr>
<td>attributeName2: dataType2 = defaultValue2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GeometricObject</td>
</tr>
<tr>
<td></td>
<td>position</td>
</tr>
<tr>
<td></td>
<td>Color</td>
</tr>
<tr>
<td></td>
<td>operate on()</td>
</tr>
<tr>
<td></td>
<td>Walk()</td>
</tr>
<tr>
<td></td>
<td>move(delta:Vector)</td>
</tr>
<tr>
<td></td>
<td>rotate(in angle:float==0.0)</td>
</tr>
<tr>
<td></td>
<td>select(p:Point):Boolean</td>
</tr>
</tbody>
</table>
3  a) Explain with the diagram, Nested state for a Phone line?

The most important innovation of UML state machines over the traditional FSMs is the introduction of hierarchically nested states (that is why state charts are also called hierarchical state machines, or HSMs). [http://en.wikipedia.org/wiki/UML_state_machine - cite_note-6#cite_note-6](http://en.wikipedia.org/wiki/UML_state_machine - cite_note-6) The semantics associated with state nesting are as follows (see Figure 3): If a system is in the nested state, for example “result” (called the substate), it also (implicitly) is in the surrounding state “on” (called the superstate). This state machine will attempt to handle any event in the context of the substate, which conceptually is at the lower level of the hierarchy. However, if the substate “result” does not prescribe how to handle the event, the event is not quietly discarded as in a traditional “flat” state machine; rather, it is automatically handled at the higher level context of the superstate “on”. This is what is meant by the system being in state “result” as well as “on”. Of course, state nesting is not limited to one level only, and the simple rule of event processing applies recursively to any level of nesting.

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

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

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

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

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

b) What is a Use Case? Explain the Guidelines for Use Case Models?

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

• First determine the system boundary

If system boundaries is unclear it's difficult to identify use cases or actors.

• Ensure that actors are focused

Each actor should have a single, coherent purpose

• Each use case must provide value to users

Complete transaction

• Relate use cases and actors
  one-many, many-one, many-many

• Remember that use cases are informal

  - Bit loose at a first

• Use cases can be structured

c) What do you mean by Swim Lane? Explain briefly activity diagram with swim lanes for servicing an aeroplane?

Swim Lane is to know which human organization is responsible for an activity
Here flight attends must clean the trash, the ground crew must add the fuel and catering must load the food and drinks before plane is serviced and ready for the next flight.

Dec.09/Jan.10

UNIT 4 PROCESS OVERVIEW, SYSTEM CONCEPTION, DOMAIN ANALYSIS

4.a. Explain the stages in software development process. Which life cycle would you prefer in the development? Why?

(10 marks)

Software development has a sequence of well-defined stages, each with a distinct purpose, input and output.

System conception: conceive an application and formulate tentative requirements.

Analysis: deeply understand the requirements by constructing models. The goal of analysis is to specify what needs to be done, not how it is done.

System design: devise a high-level strategy – the architecture- for solving the application problem.

Class design: augment and adjust the real-world models from analysis so that they are amenable to computer implementation.

Implementation: translate the design into programming code and database structures.

Testing: ensure that the application is suitable for actual use and that it truly satisfies the requirements.

Deployment: place the application in the field and gracefully cut over from legacy applications.
Maintenance: preserve the long-term viability of the application.

There are 2 kinds of development life cycle, waterfall development and iterative development.

Iterative development is more flexible. First develop the nucleus of the system – analyzing, designing, implementing, and delivering working code. Then grow the scope of the system, adding properties and behavior to existing objects, as well as adding new kinds of objects. There are multiple iterations as the system evolves to the final deliverable.

b. Identify the classes of an ATM for a bank. What criteria would you take into consideration to select the right classes? Explain. (10 marks)

The first step in constructing a class model is to find the relevant classes for objects from the application domain. Objects include physical entities, such as houses, persons, as well as concepts such as trajectories, seating assignments. ATM classes are:

Software, banking network, cashier, ATM, consortium, bank, bank computer, account, transaction, cashier station, account data, transaction data, central computer, cash card, user, cash, receipt, system, recordkeeping provision, security provision, access, cost, customer.

Discard the unnecessary and incorrect classes according to the following criteria.

Redundant classes: if 2 classes express the same concept, keep the most descriptive name.

Irrelevant classes: if a class has little or nothing to do with the problem eliminate it.

Vague classes: a class should be specific. Some tentative classes may have ill-defined boundaries or be too broad in scope.

Attributes: names that describe objects should be restated as attributes.

Operations: if a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.

Roles: the name of the class should reflect its intrinsic nature and not a role that it plays in an association.

Implementation constructs: eliminate constructs from the analysis model that are extraneous to the real world.

Derived classes: as a general rule, omit classes that can be derived from other classes.

June 2012
4 a. What is system conception? List and explain questions that must be answered by a good system concept.  

(10 Marks)

**Development stages**

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

**System Conception**

**Analysis**

To specify what must be done.

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

**System Design**

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

**Class Design**

- To emphasize from application concepts toward computer concepts.
- To choose algorithms to implement major system functions.
DEVELOPMENT LIFE CYCLE

Waterfall Development

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

Iterative Development

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

SYSTEM CONCEPTION

- System conception deals with the genesis of an application.

b. Describe the steps performed in constructing a domain state model. (10 Marks)

The Domain Class Model captures the concepts in the domain of the problem, and the relationships between them. It establishes the vocabulary of the problem domain. The Class Model notation is similar to the entity relationship model notation.

Analysis Object

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

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

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

<table>
<thead>
<tr>
<th>object name</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>paul</td>
<td>Person</td>
</tr>
<tr>
<td>person</td>
<td></td>
</tr>
</tbody>
</table>

Analysis Class

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

An analysis class has a name and attributes.

<table>
<thead>
<tr>
<th>class name</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>firstname</td>
</tr>
<tr>
<td></td>
<td>lastname</td>
</tr>
<tr>
<td></td>
<td>birthdate</td>
</tr>
</tbody>
</table>

Association

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

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

Association: Multiplicity

The multiplicity (or cardinality) defines the number of objects which are allowed to be associated with each other in an association. Multiplicity is shown by annotating the line end, called the association end, connecting the class. The full form is a range (or even a set of ranges), e.g. 0..1, 1..1, 1..4, 0..*, 1..*, an asterisk * meaning that the upper
limit is unlimited. Sometimes short forms are used: 1 for 1..1, 2 for 2..2, etc., and * for 0..*.
Multiplicity in UML is written on the opposite branch of the association compared with other (especially French) entity relationship model notations.

**Association: Multiplicity**

![Association Diagram]

**Association: Roles**

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

![Association Diagram]

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

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

As a child, a person has exactly two parents.

As a parent, a person may have several children.

Association Class

An association class is an association that is also a class. An association class has both association and class properties: it connects two or more classes, and it also has attributes and sometimes operations. Like for all associations, the identity of an occurrence stems from the connected objects. Like all classes, an association class can have attributes, and sometimes operations. In the UML, an association class can participate in an association. We do not recommend this practice.

Association Class

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

We want to record the mark a student has for a particular test.
The attribute does not belong to the student, because a student may take many tests.
Neither does it belong to a test, because many students take the same test.

Dec 08

4 a Explain sequence of Software Development Stages? (10 marks)

1  System Conception Conceive an application and formulate tentative requirements
2  Analysis Deeply understand the requirements by constructing models
3  System design Devise the architecture
4. Class design Determine the algorithms for realizing the operations
5. Implementation Translate the design into programming code and database structures
6. Testing Ensure that the application is suitable for actual use and actually satisfies requirements
7. Training Help users master the new application
8. Deployment Place the application in the field and gracefully cut over from legacy application
9. Maintenance Preserve the long term viability of the application

4.b. Explain the stages in software development process. Which life cycle would you prefer in the development? Why?
   (10 marks)

Software development has a sequence of well-defined stages, each with a distinct purpose, input and output. System conception: conceive an application and formulate tentative requirements.

Analysis: deeply understand the requirements by constructing models. The goal of analysis is to specify what needs to be done, not how it is done.

System design: devise a high-level strategy – the architecture- for solving the application problem.

Class design: augment and adjust the real-world models from analysis so that they are amenable to computer implementation.

Implementation: translate the design into programming code and database structures.

Testing: ensure that the application is suitable for actual use and that it truly satisfies the requirements.

Deployment: place the application in the field and gracefully cut over from legacy applications.

Maintenance: preserve the long-term viability of the application.

There are 2 kinds of development life cycle, waterfall development and iterative development.

Iterative development is more flexible. First develop the nucleus of the system – analyzing, designing, implementing, and delivering working code. Then grow the scope of the system, adding properties and behavior to existing objects, as well as adding new kinds of objects. There are multiple iterations as the system evolves to the final deliverable.
Change Event

- A change event is an event that is caused by the satisfaction of a Boolean expression.
- Whenever the expression changes false to true the event happens.
- UML notation for a change event is keyword when followed by a parenthesized Boolean expression.
  - when(room temperature < heating set point)
  - when(room temperature > cooling set point)

**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.
  - when(date = August 16, 2010)
- The notation for a time interval is the keyword after followed by a parenthesized expression that evaluates to a time duration.
  - after(10 seconds)

a) Explain briefly the concept of class diagram with its notation.

**Class diagram:** provide graphic notation for modeling classes and their relationships, thereby describing possible objects. It is useful for abstract modeling and for designing actual programs. Class is concise and easy to understand.

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

b) What is the difference between sequence diagrams and class diagrams?

**Sequence diagram** shows the participants in an interaction and sequence message among them. It also shows interaction with actors in use cases.

Sequence diagram capture behaviors of the single scenario. It show object interaction arranged in time sequence.

These diagrams capture behaviors of the single scenario.

**Components of sequence diagram:**

-objects
- object lifeline
- message
- pre/post conditions

Class diagram captures static structure of a system by characterizing the object in the system. The relationship between objects, attributes, operations for each class of objects is represented. An object is an instance or occurrence of a class.

c) What is the purpose of the activity model? Give one example.

Activity diagrams are to represent the parallel behavior of an operation as a set of actions. An activity is some task that needs to be done, whether by a human or a computer.

Activity arrangement

Sequential – one activity is followed by another

Parallel – two or more sets of activities are performed concurrently, and order is irrelevant permitted – we can jump between the parallel flows

- Interleaving is
4 a Explain sequence of Software Development Stages? (10 marks)

1. System Conception Conceive an application and formulate tentative requirements
10. Analysis Deeply understand the requirements by constructing models
11. System design Devise the architecture
12. Class design Determine the algorithms for realizing the operations
13. Implementation Translate the design into programming code and database structures
14. Testing Ensure that the application is suitable for actual use and actually satisfies requirements
15. Training Help users master the new application
16. Deployment Place the application in the field and gracefully cut over from legacy application
17. Maintenance Preserve the long term viability of the application

4.b. Explain the stages in software development process. Which life cycle would you prefer in the development? Why? (10 marks)

Software development has a sequence of well-defined stages, each with a distinct purpose, input and output. System conception: conceive an application and formulate tentative requirements.

Analysis: deeply understand the requirements by constructing models. The goal of analysis is to specify what needs to be done, not how it is done.

System design: devise a high-level strategy – the architecture- for solving the application problem.

Class design: augment and adjust the real-world models from analysis so that they are amenable to computer implementation.

Implementation: translate the design into programming code and database structures.

Testing: ensure that the application is suitable for actual use and that it truly satisfies the requirements.

Deployment: place the application in the field and gracefully cut over from legacy applications.

Maintenance: preserve the long-term viability of the application.

There are 2 kinds of development life cycle, waterfall development and iterative development. Iterative development is more flexible. First develop the nucleus of the system – analyzing,
designing, implementing, and delivering working code. Then grow the scope of the system, adding properties and behavior to existing objects, as well as adding new kinds of objects. There are multiple iterations as the system evolves to the final deliverable.

**Dec.09/Jan.10**

**UNIT 5 APPLICATION ANALYSIS, SYSTEM DESIGN**

5 a. **What is an activity diagram? Explain the special construct for activity models. (10 marks)**

An activity diagram shows the sequence of steps that make up a complex process such as an algorithm or workflow. An activity diagram shows flow of control, similar to a sequence diagram, but focuses on operations rather than on objects. Activity diagrams are most useful during the early stages of designing algorithms and workflows.

Guidelines for activity models:

Don’t misuse activity diagrams: Activity diagrams are intended to elaborate use case and sequence models so that a developer can study algorithm and workflow.

Level diagrams: Activities on a diagram should be at a consistent level of detail. Place additional detail for an activity in a separate diagram.

Be careful with branches and conditions: if there are conditions, at least one must be satisfied when an activity completes.

Be careful with concurrent activities: concurrency means that the activities can complete in any order and still yield an acceptable result.

Consider executable activity diagrams: executable activity diagrams can help developers understand their systems better.

b. **What are use case models? Give the guidelines for constructing a use case model. (6 marks)**

A use case is a coherent piece of functionality that a system can provide by interacting with actors.

The main purpose of a system is almost always found in the use cases, with requirements lists supplying additional implementation constraints. The guidelines for constructing use case models are:

First determine the system boundary: it is impossible to identify use cases or actors if the system boundary is unclear.
Ensure that actors are focused: each actor should have a single, coherent purpose. If a real world object embodies multiple purposes, capture them with separate actors.

Each use case must provide values to users: a use case should represent a complete transaction that provides value to users and should not be defined too narrowly.

Relate use cases and actors: every use case should have at least one actor and every actor should participate in at least one use case.

Remember that the use cases are informal: it is important not to obsessed by formalism in specifying use cases.

Use cases can be structured.

5.c. What are the steps involved in constructing an application state model. (4 marks)

The application state model focuses on application classes and augments the domain state model. Application classes are more likely to have important temporal behavior than domain classes.

Steps:

Determine application classes with states.

Find events.

Build state diagrams.

Check against other state diagrams.

Check against the class model.

Check against the interaction model.

1. Determine application classes with states: the application class model adds computer-oriented classes that are prominent to users and important to the operation of an application. Consider each application class and determine which one have multiple states.

2. Finding events: for the application interaction model number of scenarios is prepared. Those scenarios are studied and events are extracted. Even though the scenarios may not cover every contingency, they ensure that common interactions are not overlooked and they highlight the major events.

3. Building state diagrams; the next step is to build the state diagram for each application class with temporal behavior. Choose one of these classes and consider a sequence diagram. Arrange the events involving the class into a path whose arcs are labeled by the events.

4. Checking against other state diagrams: check the state diagrams of each class for
completeness and consistency. Every event should have a sender and a receiver, occasionally the same object. States without predecessors or successors are suspicious, make sure they represent starting or termination points of the interaction sequence.

5. Checking against class model: make sure that the state diagrams are consistent with domain and application class models.

**June 2012**

5 a. Describe the steps involved to allocate each concurrent subsystem to a hardware unit, either a general purpose processor or a fractional unit.  

(12 Marks)

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.

![Figure 6.2](image)

*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
b. What is an interactive interface? Explain in brief steps in designing an interactive interface. 

(08 Marks)

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 are 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:
5 a. What is an activity diagram? Explain the special construct for activity models. (10 marks)

An activity diagram shows the sequence of steps that make up a complex process such as an algorithm or workflow. An activity diagram shows flow of control, similar to a sequence diagram, but focuses on operations rather than on objects. Activity diagrams are most useful during the early stages of designing algorithms and workflows.

Guidelines for activity models:

Don’t misuse activity diagrams: Activity diagrams are intended to elaborate use case and sequence models so that a developer can study algorithm and workflow.

Level diagrams: Activities on a diagram should be at a consistent level of detail. Place additional detail for an activity in a separate diagram.

Be careful with branches and conditions: if there are conditions, at least one must be satisfied when an activity completes.

Be careful with concurrent activities: concurrency means that the activities can complete in any order and still yield an acceptable result.

Consider executable activity diagrams: executable activity diagrams can help developers understand their systems better.

b Draw a State and Sequence diagram for telephone talk process. State Diagram
Sequence Diagram 1
A Explain different aspects of re-usability? explain reusable things?(10 marks)

Two aspects of reuse:

Using existing things

Creating reusable new things

Reusable things include:

a. Models
b. Libraries
c. Frameworks
d. Patterns

Reusable Libraries:

A library is a collection of classes that are useful in many contexts.

• Qualities of “Good” class libraries:
  – Coherence – well focused themes
  – Completeness – provide complete behavior
– **Consistency** - polymorphic operations should have consistent names and signatures across classes
– **Efficiency** – provide alternative implementations of algorithms
– **Extensibility** – define subclasses for library classes
– **Genericity** – parameterized class definitions

• Problems limit the reuse ability:
  – Argument validation
    • Validate arguments by collection or by individual
  – Error Handling
    • Error codes or errors
  – Control paradigms
    • Event-driven or procedure-driven control
  – Group operations
  – Garbage collection
  – Name collisions

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

Reusable Patterns:-
A pattern is a proven solution to a general problem, There are patterns for analysis, architecture, design, and implementation. A pattern is more likely to be correct and robust than an untested, custom solution, Patterns are prototypical model fragments that distill some of the knowledge of experts.

**B What are the Steps in designing pipeline for continuous transformation?**

In Continuous transformation The output actively depend on changing inputs, Continuously updates the outputs Example : Signal processing, Windowing systems.

Sequence of steps for a graphics application
Step1: Break overall transformation into stages, with each stage performing one kind of transformation.

Step2: Define Input, Output and intermediate models between each pair of successive stages.

Step3: Differentiate each operation to obtain incremental changes to each stage.

Step4: Add additional intermediate objects for optimization.
Q.5. a) Draw a State and Sequence diagram for telephone talk process.
(10 MARKS)

State Diagram

B) Sequence Diagram 1

Sequence Diagram 2: (In Case of Phone Busy)
6.a. How would you improve the organization of a class design? (10 marks)

Programs consist of discrete physical units that can be edited, compiled, imported or otherwise manipulated. The organization of a class design can be improved by the following steps:

- Hide internal information from outside view.
- Maintain coherence of entities.
- Fine-tune definition of packages.

Information hiding: the goals of design are different. During design we adjust the analysis model so that it is practical to implement and maintain. One way to improve the viability of a design is by carefully separating external specification from internal implementation. This is called information hiding.

Coherence of entities: coherence is another important design principle. An entity, such as a class, an operation, or a package, is coherent if it is organized on a consistent plan and all its part fit together toward a common goal. An entity should have a single major theme, it should not be a collection of unrelated parts.

Fine tuning packages: during analysis you partitioned the class model into packages. This initial organization may not be suitable or optimal for implementation. Define packages so that their interfaces are minimal and well defined. The interface between two packages consists of the associations that relate classes in one package to classes in the other and operations that access classes across package boundaries.
b. How would you choose association traversal? Explain the following (10 marks)

i) one way association  ii) two way association

Associations are inherently bidirectional, which is certainly true in an abstract sense. But if application has some associations that are traversed in only one direction, their implementation can be simplified.

One-way associations:

If an association is traversed only in one direction, it can be implemented as a pointer- an attribute that contains an object reference. If the multiplicity is one then it is a simple pointer, if the multiplicity is many then it is a set of pointers.

Class model:

Two-way associations: many associations are traversed in both directions, although not usually with equal frequency. There are 3 approaches to their implementation.

Implement one-way. Implement as a pointer in one direction only and perform a search when backward traversal is required. This approach is useful only if there is a great disparity in traversal frequency in the 2 directions and minimizing both the storage and update costs is important.

Implement two way: implement with pointers in both directions. This approach permits fast access, but if either direction is updated, then the other must also be updated to keep the link consistent.
6 a. Elaborate three tasks involved in design optimization. 

(10 Marks)

Good way to design a system: first get the logic correct and then optimize it

The design model builds on the analysis model.
Analysis model – captures the logic of a system
Design model – adds development details

Design Optimization involves the following tasks:

- Provide efficient access paths
- Rearrange the computation for greater efficiency
- Save intermediate results to avoid recomputation

Adding redundant associations for efficient access

Design has different motivations and focuses on the viability of a model for implementation

Some improvements for the above analysis class model

- use hashed set for HasSkills
- ex: japanese speaking
- indexing

In cases where the number of hits from a query is low because few objects satisfy the test, an index can improve access to frequently retrieved objects.

Indexes incur a cost:

- Require additional memory
- Require to be updated whenever the base associations are updated examine each operation and see what associations it must
traverse to obtain its information
  o For each operation
    • Frequency of access
    • Fan-out
    • Selectivity

Provide indexes for frequent operations with a low hit ratio, because such operations are inefficient when using nested loops to traverse a path in the network laborate the class model

ATM Example:

postTransaction() operation
  o Relate Receipt to CashCard (Assigning operations to classes)
  o Tracing from CashCard to CardAuthorization to Customer has no fan-out
  o Derived association from Bank to Update speed the process

b. What are the outputs from reverse engineering? In brief, discuss reverse engineering tips

(10 marks)

What is Reverse Engineering?
General definition: A systematic methodology for analyzing the design of an existing device or system, either as an approach to study the design or as a prerequisite for re-design.

Reverse Engineering helps you to:
  Develop a systematic approach to thinking about the engineering design of devices
  and systems
  Acquire a mental data bank of mechanical design solutions

Levels of Analysis in Reverse Engineering
System-Wide Analysis
  Subsystem Dissection Analysis
  Individual Component Analysis

System-Wide Analysis
  Customer Requirements
  Engineering Requirements
  Functional Specifications
  Prediction of Subsystems and Components

Subsystem Dissection Analysis
  Document Disassembly
  Define Subsystems
  Determine Subsystem Functional Specifications
Determine Subsystem Physical/Mathematical Principles

**Individual Component Analysis**
- Repeat Dissection Steps to Individual Component
- Define Component Material Selection and Fabrication Process
- Suggest Alternative Designs, Systems, Components, and Materials

**What is Reverse Engineering?**
You have an unexpected case:
- You finished one course project using Java
  - Your program runs OK
  - But, by accident, you delete the java file
  - How to hand in your project?

Reverse Engineering

**What is Reverse Engineering?**
- RE encompasses any activity that is done to determine how a product works, to learn
  - the ideas and technology that were used in developing that product.
- RE can be done at many levels
- RE generally belongs to Software Maintenance

**Waterfall Model of software development**

Reverse Engineering - SE context
- Trying to figure out the structure and behaviour of existing software by building
  - general-level static and dynamic models’

Links:
  - Compact information on reverse engineering
- [http://users.ece.gatech.edu/~linda/revegr/revrepos.html](http://users.ece.gatech.edu/~linda/revegr/revrepos.html)
  - Reengineering Resource Repository
  - Listings of tools, literature, …
The Early Days of RE
Law of Software Revolution (Lehman, 1980)
Fundamental strategies for program comprehension (Brooks, 1983)
Taxonomy of Reverse Engineering (Chikofsky&Cross, 1990)
WCRE (Working Conference on R.E., 1990)
  • IWPC (Int. Workshop on Program Comprehension)

Why do we need RE?
• Recovery of lost information
  providing proper system documentation
• Assisting with maintenance
  identification of side effects and anomalies
• Migration to another hw/sw platform
• Facilitating software reuse
• Benefits
  maintenance cost savings
  quality improvements
  competitive advantages
  software reuse facilitation

Difficulties of Reverse Engineering
• Gap between problem/solution domain
  • Gap between concrete and abstract
  • Gap between coherency/disintegration
  • Gap between hierarchical/associational

Scope and Task of Reverse Engineering
• program understanding
• Reverse Engineering is focused on the challenging task of understanding legacy
  program code without having suitable documentation
• However, the major effort in software engineering organizations is spent after
  development, on maintaining the systems to remove existing errors
  and to adapt them to changed requirements.

Reverse Engineering
• mature software systems often have incomplete, incorrect or even nonexistent
  design documentation.
• Ideally, the system documentation describes the product and the complete design,
  including its rationale.
• The major purpose of tools essentially is to aid maintainers
  understand the program
• Pure plan recognition is unlikely to be powerful enough to drive
reverse engineering for the following reasons: legacy code is written using rather tricky encodings to achieve better efficiency all the necessary plan patterns in all the variations must be supplied in advance for a particular application, different abstract concepts map to the same code within one application
• A huge amount of conventionally developed software exists and such systems will continue to be developed for a long time.
• These systems have errors and continual demand for the enhancement of their functional and performance requirements

Dec 08

6 a. Elaborate three tasks involved in design optimization. (10 Marks)

Good way to design a system: first get the logic correct and then optimize it
The design model builds on the analysis model.
Analysis model – captures the logic of a system
Design model – adds development details
Design Optimization involves the following tasks:
  o Provide efficient access paths
  o Rearrange the computation for greater efficiency
Save intermediate results to avoid recomputation

**Adding redundant associations for efficient access**

Design has different motivations and focuses on the viability of a model for implementation

```
Employs  Person  HasSkill  Skill
1  *  *  *
```

**Some improvements for the above analysis class model**

- use hashed set for HasSkills
- ex: japanese speaking
- indexing

**In cases where the number of hits from a query is low because few objects satisfy the test, an index can improve access to frequently retrieved objects.**

```
/ SpeaksLanguage

Company  Employs  Person  HasSkill  Skill
```

**Indexes incur a cost:**

- Require additional memory
- Require to be updated whenever the base associations are updated examine each operation and see what associations it must traverse to obtain its information
  - For each operation
    - Frequency of access
    - Fan-out
    - Selectivity

Provide indexes for frequent operations with a low hit ratio, because such operations are inefficient when using nested loops to traverse a path in the network laborate the class model

ATM Example:

- postTransaction() operation
  - Relate Receipt to CashCard (Assigning operations to classes)
6. Explain the considerations for choosing alternative algorithms? (10 marks)

- Considerations for choosing alternative algorithms
- Computational complexity
- Ease of implementation and understandability
- Flexibility

Simple but inefficient

Complex efficient

Interactions between the consortium computer and bank computers could be complex.

Considerations are Distributed computing, The scale of consortium computer (scalability), The inevitable conversions and compromises in coordinating the various data formats, All these issues make the choice of algorithms for coordinating the consortium and the banks important

May/June 2010
6) a) Explain the considerations for choosing alternative algorithms?

- Considerations for choosing alternative algorithms
- Computational complexity
- Ease of implementation and understandability
- Flexibility

Interactions between the consortium computer and bank computers could be complex. Considerations are Distributed computing, The scale of consortium computer (scalability), The inevitable conversions and compromises in coordinating the various data formats, All these issues make the choice of algorithms for coordinating the consortium and the banks important

b) List adjustment of Inheritance?

To increase inheritance perform the following steps are

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

b) Explain with an example Fine Tuning Classes?
Fine tune classes before writing code in order to simplify development or to improve performance, Partition a class, Merge classes, Partition / merge attributes, Promote an attribute / demote a class.

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

Explain Structure, Dynamics and Implementation of View Handler pattern with Diagrams?

6 a) Diagrams?

Structure:

- The view handler is the central component of this pattern.
- It is responsible for opening new views and clients can specify the view they want.
- If the requested view is open already, the view handler brings this open view to the foreground.
- If the requested view is open but iconized, the view handler tells the view to display itself full size.
- The view handler also offers functions for closing views, both individual ones and all currently open views.
- The main responsibility of the view handler is to offer view management services.
- Additional responsibility of the view handler is coordination.
**Dynamics:** Scenario I shows how the view handler creates a new view.

There are 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 handler adds the new view to display itself.

- The view handler calls the view to display itself.
Scenario II illustrates how the view handler organizes the tiling of views. This is divided into three phases:

- The user invokes the command to tile all open windows.
- 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.
Implementation:

The implementation of a view Handler structure can be divided into four steps.

1) 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.

2) Implement the views.
   a. Derive a separate class from the AbstractView class for each specific type of view identified in Step1.
   b. Implement the view specific part of the interface.

3) Define the view handler.
   a. Implement functions for creating views as Factory Methods.
   b. Client can specify the view they want.

b) Explain Dynamics and Implementation for Forward-Receiver Pattern with diagrams.

Forwarder-Receiver

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.

Implementation:

• Specify a name to address mapping
  ▪ Since peers reference other peers by name, you need to introduce an appropriate **name space**
  ▪ A name space defines rules and constraints to which names must conform in a given context.
  ▪ A name does not necessarily refer to a single address—it may refer to a group of addresses
• Specify the message protocols to be used between peers and forwarders.
  ▪ This protocol defines the detailed structure of message data a forwarder receives from its peer.
  ▪ Perform the same task for the message protocol to be used between receivers and their peers.
  ▪ class message consists of sender and data
  ▪ **class Message** {
public String sender;

public String data;

public Message(String thesender, String rawData) {
    sender = thesender;
    data = rawData;
}

• Choose a communication mechanism
  ▪ This decision is driven mainly by the communication mechanisms available in the operating system you use.
  ▪ When specifying an IPC facility you need to consider the following aspects:
    • If efficiency is important, a low-level mechanism such as TCP/IP may be the first choice.
    • Low-level mechanisms such as TCP/IP require substantial programming effort, and are dependent on the platform you use, restricting portability.

• Implement the forwarder
  ▪ Encapsulate all functionality for sending messages across process boundaries in the forwarder.
  ▪ The forwarder provides its functionality through a public interface and encapsulates the details of a particular IPC mechanism.
  ▪ Define a repository that maps names to physical addresses.

Implement the receiver:
  ▪ blocking and non blocking recvmsg(), unmarshal(the msg)
  ▪ Encapsulate all functionality for receiving IPC messages in the receiver.
  ▪ Provide the receiver with a general interface that abstracts from details of a particular IPC mechanism.
  ▪ The receiver needs to include functionality for receiving and unmarshaling IPC messages.

• Implement the peers of the application – partitioning into client and servers.
  ▪ Partition the peers into two sets, clients and servers. The intersection of these sets does not need to be empty.
  ▪ If a peer acts as a client, it sends a message to a remote peer and waits for the response.
  ▪ After receiving the response, it continues with its task.
  ▪ Peers acting as servers continuously wait for incoming messages.
  ▪ When such a message arrives, they execute a service that depends on the message they received, and send a response back to the originator of the
request.

• Implement a start up configuration- initialize F-R with valid name to address mapping
  ▪ When your system starts up forwards and receivers must be initialized with a valid name- \- to address mapping.
  ▪ Introduce a separate start-up routine that creates a repository and enters all name/address pairs.
  ▪ Such a configuration routine could read these pairs from an external file, removing the need to touch the source code when changing the mapping.

Dec.09/Jan.10

UNIT 7 DESIGN PATTERN 1

7.a. What is a pattern? Explain the model view controller design pattern for software architecture, with OMT diagram. (10 marks)

When experts work on a particular problem, it is unusual for them to tackle it by inventing a new solution that is completely distinct from existing ones. They often recall a similar problem they have already solved and reuse the essence of its solution to solve the new problem.

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

The model view controller architectural pattern divides an interactive application into 3 components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change propagation mechanism ensures consistency between the user interface and the model.

b. List and explain the different pattern categories. Give the differences between patterns and methods. (10 marks)

Pattern categories: Patterns are grouped into 3 categories:

- Architectural patterns

- Design patterns

- Idioms

Architectural Patterns: are templates for concrete software architectures. They specify the system wide structural properties of an application and have an impact on the architecture of its sub systems.
Design Patterns: are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language.

Idioms: represent the lowest-level patterns. They address aspects of both design and implementation.

Pattern description: Patterns must be presented in an appropriate form if we are to understand and discuss them. It should also be described uniformly. Describing a pattern based exclusively on a context-problem-solution schema is not enough. A pattern must be named preferably with an intuitive name.

c. Explain client-dispatcher-server design pattern.

**Client-Dispatcher-Server**

Imagine we are developing a software system ACHILLES for the retrieval of new scientific information. The information providers are both on our local network and distributed over the world. To access an individual information provider, it is necessary to specify its location and the service to be executed. When an information provider receives a request from a client application, it runs the appropriate service and returns the requested information to the client.

**Context:** A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.

**Problem:** The Client-Dispatcher-Server pattern is useful when you need to balance the following forces:

- A component should be able to use a service independent of the location of the service provider.
- The code implementing the functional core of a service consumer should be
separate from the code used to establish a connection with service providers.

**Solution:** Provide a *dispatcher* component to act as an intermediate layer between clients and *servers*. The dispatcher implements a name service that allows clients to refer to servers by names instead of physical locations, thus providing location transparency. In addition, the dispatcher is responsible for establishing the communication channel between a client and a server.

**Structure:**

- The task of a client is to perform domain-specific tasks.
- A *server* provides a set of operations to clients.

![Diagram showing class and collaborators](image)

The *dispatcher* offers functionality for establishing communication channels between clients and servers.

![Diagram showing class and collaborators](image)
Dynamics:

A typical scenario for the Client-Dispatcher-Server design pattern includes the following phases:

- A server registers itself with the dispatcher component.
- At a later time, a client asks the dispatcher for a communication channel to a specified server.
- The dispatcher looks up the server that is associated with the name specified by the client in its registry.
- The dispatcher establishes a communication link to the server. If it is able to initiate the connection successfully, it returns the communication channel to the client. If not, it sends the client an error message.
- The client uses the communication channel to send a request directly to the server.
- After recognizing the incoming request, the server executes the appropriate service.
- When the service execution is completed, the server sends the results back to the client.
7 a. What is a pattern? Explain the model view controller design pattern for software architecture with OMT class diagram.

(10 Marks)

Approach - 1
☐ Each pattern is a three-part rule.
☐ A relation between a certain context.
☐ A problem.
☐ A solution.

Approach - 2
☐ Each pattern is a relationship.
☐ A context.
☐ A system of forces which occurs repeatedly in that context.

Force
denote any aspect of the problem that should be considered when
solving it.
☐ A spatial configuration which allows these forces to resolve themselves.

Approach - 3
☐ Each pattern is an instruction which helps for re-usability.
☐ A relevant context.
☐ To resolve system of forces.

Approach - 4
☐ Each pattern is both a process and thing.
☐ A description of thing.
☐ A description of the process which will generate the thing.

Let consider software with a human-computer user interface. The interface is prone to change. To provide the functionality extension will be challenging. Usually user interface will be adapted for specific customers. The interface will look different when posted to another platform. When upgrading an operating system the parameters to provide user interface may change. Keeping track of the above points, if the flexibility is
provided, it will increase the cost of development and copies of different implementations have to be saved in memory. There exist certain scenarios where non-graphical user interface has to be re-modified for graphical interface or vice-versa. The user-interface can also be dependent on the input device. Therefore we can make the functional core to be independent from user interface.

The important aspects of user-interface development are:

1. Changes to user interface should be easy and possible at run time.
2. Adapting or porting the user interface should not impact code in the functional core of the application.

Display should be separately designable/evolvable.

Component

sic parts of any application:
1. Data being manipulated by the input events.
2. A user-interface through which data manipulation occurs.
3. The data is logically independent from how it is displayed to the user.

hey are often badly designed and have an incomplete, nonexistent, or, even worse, wrong documentation without any design information, this is a challenging task.

b. Describe three categories of patterns.

**Types of patterns:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td>Architectural patterns are fundamental structural organization</td>
</tr>
<tr>
<td></td>
<td>MVC</td>
</tr>
<tr>
<td></td>
<td>is expressed for software systems that provide a set of predefined subsystems. Relationships are</td>
</tr>
</tbody>
</table>
These patterns include the rules and guidelines for organizing the relationships.

**Design**

Capture the static - dynamic roles & relationships in solutions that occur repeatedly.

**Observer**

**Patterns**

Restricted to a particular language.

**Publisher-subscriber**

**Idioms**

Model-view-controller is a pattern used to isolate business logic from the user interface. The Model represents the information of the application and the business rules used to manipulate the data. The View corresponds to user interface such as text and checkbox items. The Controller manages the communication between the model and view. The controller handles user actions such as keystrokes and mouse movements. User actions are sent to the model or view as required. Example 1: Calculator

1) The Form view
2) Events controller
3) Controller will call methods from model ADD/Subtract/…

The model takes care of all the components and it maintains the current state of the calculator.

**DEC 08**

7 a. How would you choose association traversal? Explain the following (10 marks)
i) one way association       ii) two way association

Associations are inherently bidirectional, which is certainly true in an abstract sense. But if application has some associations that are traversed in only one direction, their implementation can be simplified.

One-way associations:

If an association is traversed only in one direction, it can be implemented as a pointer— an attribute that contains an object reference. If the multiplicity is one then it is a simple pointer, if the multiplicity is many then it is a set of pointers.

Class model:

```
Person                      Company
| employer                  | employees |
|                           |          |
```

Two-way associations: many associations are traversed in both directions, although not usually with equal frequency. There are 3 approaches to their implementation.

Implement one-way. Implement as a pointer in one direction only and perform a search when backward traversal is required. This approach is useful only if there is a great disparity in traversal frequency in the 2 directions and minimizing both the storage and update costs is important.

Implement two way: implement with pointers in both directions. This approach permits fast access, but if either direction is updated, then the other must also be updated to keep the link consistent.
7 b. What is a pattern? Explain the model view controller design pattern for software architecture, with OMT diagram. (10 marks)

When experts work on a particular problem, it is unusual for them to tackle it by inventing a new solution that is completely distinct from existing ones. They often recall a similar problem they have already solved and reuse the essence of its solution to solve the new problem.

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

The model view controller architectural pattern divides an interactive application into 3 components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change propagation mechanism ensures consistency between the user interface and the model.

May/June 2010

7 a) List the Properties of pattern in SA?

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 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.

b) Explain Pattern Categories?

Patterns are grouped into 3 categories:

- Architectural patterns
- Design patterns
- Idioms

Architectural Patterns: are templates for concrete software architectures. They specify the system wide structural properties of an application and have an impact on the
architecture of its sub systems.

Design Patterns: are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language.

Idioms: represent the lowest-level patterns. They address aspects of both design and implementation.

Pattern description: Patterns must be presented in an appropriate form if we are to understand and discuss them. It should also be described uniformly. Describing a pattern based exclusively on a context-problem-solution schema is not enough. A pattern must be named preferably with an intuitive name.

Further diagrams and scenarios are used to illustrate the static and dynamic aspects of the solution.

c) What is Forwarder-Receiver Pappern? How its helpful?

The Forwarder-Receiver design pattern provides transparent inter-process communication for software systems with a peer-to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.

The forwarder-receiver pattern is useful when you need to balance the following forces:

- The system should allow the exchangeability of the communication mechanisms.
- The cooperation of components follows a peer-to-peer model, in which a sender only needs to know the names of its receivers.
- The communication between peers should not have a major impact on performance.

Solution: 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:
The *Forwarder-Receiver* design pattern consists of three kinds of components: *forwarders, receivers*, and *peers*:

- **Peer** components are responsible for application tasks. To carry out their tasks peers need to communicate with other peers.

### Peer Components

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer</td>
<td></td>
</tr>
</tbody>
</table>
| **Responsibility** | • Provides application services.  
|                  | • Communicates with other peers. |
| **Collaborators** | • Forwarder  
|                  | • Receiver |

### Forwarder Components

*Forwarder* components send messages across process boundaries. A forwarder provides a general interface that is an abstraction of a particular IPC mechanism, and includes functionality for marshaling and delivery of messages.

### Receiver Components

*Receiver* components are responsible for receiving messages. A receiver offers a general interface that is an abstraction of a particular IPC mechanism. It includes functionality for receiving and unmarshaling messages.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forwarder</td>
<td></td>
</tr>
</tbody>
</table>
| **Responsibility** | • Provides a general interface for sending messages.  
|                  | • Marshals and delivers messages to remote receivers.  
|                  | • Maps names to physical addresses. |
| **Collaborators** | • Receiver |

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver</td>
<td></td>
</tr>
</tbody>
</table>
| **Responsibility** | • Provides a general interface for receiving messages.  
|                  | • Receives and unmarshals messages from remote forwarders. |
| **Collaborators** | • Forwarder |
Dynamics:

Two peers P1 and P2 communicate with each other. For this purpose, P1 uses a forwarder Forwl and a receiver Recvl. P2 handles all message transfers with a forwarder Forw2 and a receiverRecv2:

- P1 requests a service from a remote peer P2. For this purpose, it sends the request to its forwarder Forwl and specifies the name of the recipient.
- Forwl determines the physical location of the remote peer and marshals the message.
- Forwl 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 Forwl.
- 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 pl to the forwarder Forw2. The forwarder marshals the result and delivers it Recvl.
- Recvl receives the response from P2, unmarshals it and delivers it to P1.
Implementation:

1. **Specify a name-to-address mapping.** Since peers reference other peers by name, you need to introduce an appropriate name space.

2. **Specify the message protocols** to be used between peers and forwarders. This protocol defines the detailed structure of message data a forwarder receives from its peer. Perform the same task for the message protocol to be used between receivers and their peers.

   ```java
   class Message {
       public String sender;
       public String data;
       public Message(String theSender, String rawData) {
           sender = theSender;
           data = rawData;
       }
   }
   ```

3. **Choose a communication mechanism.** This decision is driven mainly by the communlcation mechanisms available in the operating system you use. When specifying an IPC facility you need to consider the following aspects:

   *If efficiency is important, a low-level mechanism such as TCP/IP may be the fist choice.*
Low-level mechanisms such as TCP/IP require substantial programming effort, and are dependent on the platform you use, restricting portability. If your system must be portable between platforms, it is better to use IPC mechanisms such as sockets instead.

4. Implement the forwarder. Encapsulate all functionality for sending messages across process boundaries in the forwarder. The forwarder provides its functionality through a public interface and encapsulates the details of a particular IPC mechanism.

```java
class Entry {
    private String destinationId; // target machine
    private int portNr; // socket port
    public Entry(String theDest, int thePort) {
        destinationId = theDest;
        portNr = thePort;
    }
    public String dest() {
        return destinationId;
    }
    public int port() {
        return portNr;
    }
}

class Registry {
    private Hashtable hTable = new Hashtable();
    public void put(String theKey, Entry theEntry) {
        hTable.put(theKey, theEntry);
    }
    public Entry get(String aKey) {
        return (Entry) hTable.get(aKey);
    }
}

class Forwarder {
    private Socket s;
    private OutputStream oStr;
    private String myName;
    public Forwarder(String theName) { myName = theName; }
    private byte[] marshal(Message theMsg) { /* ... */ }
    private void deliver(String theDest, byte[] data) {
        try {
            Entry entry = fr.reg.get(theDest);
            s = new Socket(entry.dest(), entry.port());
            oStr = s.getOutputStream();
            oStr.write(data);
            oStr.flush();
            oStr.close();
            s.close();
        } catch (IOException e) { /* ... */ }
    }
    public void sendMsg(String theDest, Message theMsg) {
        deliver(theDest, marshal(theMsg));
    }
}
```

5. Implement the receiver. Encapsulate all functionality for receiving IPC messages in the receiver. Provide the receiver with a general interface that abstracts from details of a particular IPC mechanism.
Two other aspects need special consideration when you design the receivers. Since all peers run asynchronously, you need to decide whether the receivers should block until a message arrives:

* If so, the receiver waits for an incoming message. It only returns control back to its peer when it receives a message. In other words, the peer cannot continue until message reception is successful. This behavior is appropriate if the peer depends on the incoming message to continue its work.

* In all other cases, you should implement non-blocking receivers that allow peers to specify time-out values (see also step 2). If no message has arrived in the specified time period, the receiver returns an exception to its peer.

```java
class Receiver {
    private ServerSocket srvS;
    private Socket s;
    private InputStream iStr;
    private String myName;
    public Receiver(String theName) { myName = theName; }
    private Message unmarshal(byte[] anArray) { /* ... */ }
    private byte[] receive() { 
        int val;
        byte buffer[] = null;
        try {
            Entry entry = fr.reg.get(myName);
            srvS = new ServerSocket(entry.port(), 1000);
            s = srvS.accept(); iStr = s.getInputStream();
            val = iStr.read(); buffer = new byte[val];
            iStr.read(buffer);
            iStr.close(); s.close(); srvS.close();
        }
        catch (IOException e) { /* ... */ }
        return buffer;
    }
    public Message receiveMsg() {
        return unmarshal(receive());
    }
}
```

6. Implement the peers of your application

Here is an example of a peer acting as a server:

```java
class Server extends Thread {
    Receiver r;
    Forwarder f;
    public void run() {
        Message result = null;
        r = new Receiver("Server");
        result = r.receiveMsg();
        f = new Forwarder("Server");
        Message msg = new Message("Server","I am alive");
        f.sendMsg(result.sender, msg);
    }
}
```

7. Implement a start-up configuration.
In the DwarfWare example we introduce the following configuration class, allowing us to register a server and a client with the central repository:

```java
class Configuration {
    public Configuration() {
        Entry entry = new Entry("127.0.0.1", 1111);
        fr.reg.put("Client", entry);
        entry = new Entry("127.0.0.1", 2222);
        fr.reg.put("Server", entry);
    }
}
```

Variants:

**Forwarder-Receiver without name-to-address** mapping. Oftentimes performance issues are more important than being able to encapsulate all details of the underlying IPC mechanism. To achieve this, you can remove the mapping from names to physical locations within forwarders and receivers, for example. In such a configuration, peers need to tell their forwarder the physical location of the recipient. This variant, however, might significantly decrease the ability to change the IPC mechanism.

**Benefits:**

- *Eminent inter-process communication.*
- *Encapsulation of IPC facilities.*

**Liabilities:**

- *No support for flexible re-configuration of components.*

---

**Dec 10/Jan**

7 a) When a software system uses servers distributed over a network it must provide a means for communication between them. Clients should not need to know where servers are located. To solve this which pattern is used and explain the dynamics and implementation of the pattern. ?

The **Client-Dispatcher-Server** design pattern is used to solve this problem.

**Structure:**

- The task of a client is to perform domain-specific tasks. The client accesses operations offered by servers in order to carry out its processing tasks.
- Before sending a request to a server, the client asks the dispatcher for a communication channel. The client uses this channel to communicate with
the server.
- A server provides a set of operations to clients. It either registers itself or is registered with the dispatcher by its name and address.
- A server component may be located on the same computer as a client, or may be reachable via a network.
- The **dispatcher** offers functionality for establishing communication channels between clients and servers. To do this, it takes the name of a server component and maps this name to the physical location of the server component.

The static relationships between clients, servers and the dispatcher are as follows:

![Diagram](image)

**Dynamics:**

- A server registers itself with the dispatcher component.
- At a later time a client asks the dispatcher for a communication channel to a specified server.
- The dispatcher looks up the server that is associated with the name specified by the client in its registry.
- The dispatcher establishes a communication link to the server.
- If it is able to initiate the connection successfully.
- The client uses the communication channel to send a request directly to the server.
- After recognizing the incoming request, the server executes the appropriate service.
- When the service execution is completed, the server sends the results back to the client.
Implementation:

- Separate the application into server and clients.
  - Define which components should be implemented as servers, and identify the clients that will access these servers.
  - This partitioning may be predefined.
- Decide which communication facilities are required.
  - Select communication facilities for the interaction between clients and the dispatcher, between servers and the dispatcher and between clients and servers.
  - You can use a different communication mechanism for each connection.
  - Specify the interaction protocols between components.
  - A protocol specifies an ordered sequence of activities for initializing and maintaining a communication channel between two components, as well as the structure of messages or data being transmitted.
  - The Client-Dispatcher-Server pattern implies three different kinds of protocol.
- Decide how to name servers.
  - The four-byte Internet IP address scheme is not applicable, because it does not provide location transparency.
  - If IP addresses were used, a client would depend on the concrete location of the server. You need to introduce names that uniquely identify servers.
but do not carry any location information.

- Design and implement the dispatcher
  - Determine how the protocols you introduced in step 3 should be implemented using available communication facilities.
  - If, for example, the dispatcher is located within the address space of the client, local procedure calls should be used for CDprotocol.
  - For all other cases and protocols, you need to use facilities such as TCP ports or shared memory.
- Implement the client and server components
  - According to our desired solution and the decisions, you make about the dispatcher interface.
  - Configure the system and either register the servers with the dispatcher or let the servers dynamically register and unregister themselves.
  - Follow the same strategies for optimizing performance that are described in step 5.

7 b) Explain structure, dynamics and implementation of Command processor pattern.

**Structure:**

- The abstract command component defines the interface of all command objects.
- As a minimum this interface consists of a procedure to execute a command.
- The additional services implemented by the command processor require further interface procedures for all command objects.
- For each user function we derive a command component from the abstract command.
- A command component implements the interface of the abstract command by using zero or more supplier components.
**Dynamics:**

The following diagram shows a typical scenario of the Command Processor pattern implementing an undo mechanism.

- 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

---

**Implementation**
• Define the interface of the abstract command.
  ▪ The abstract command class hides the details of all specific commands.
  ▪ This class always specifies the abstract method required to execute a command.
  ▪ It also defines the methods necessary to implement the additional services offered by the command processor.
• Design the command components for each type of request that the application supports.
  ▪ There are several options for binding a command to its suppliers.
  ▪ The supplier component can be hard coded within the command. or the controller can provide the supplier within the command. or the controller can provide the supplier.
• Increase flexibility by providing macro commands that combine several successive commands
  ▪ The command type of a MacroCmd depends on the commands that are added to the macro.
  ▪ An appended command of type no-undo will prevent the undo of the complete macro command.
• Implement the controller component
  ▪ Command objects are created by the controller.
  ▪ A generic menu controller provides an example of the application of the Prototype pattern.
• Implement access to the additional services of the command processor
  ▪ A user-accessible additional service is normally implemented by a specific command class.
  ▪ The command processor supplies the functionality for the 'do' method. Directly calling the interface of the command processor is also an option.

• Implement the command processor component
  ▪ The command processor receives command objects from the controller and takes responsibility for them.
  ▪ For each command object, the command processor starts the execution by calling the do method.

Dec.09/Jan.10

UNIT 8 DESIGN PATTERN 2

8.a. Explain the command processor design pattern. (10 marks)

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.

Example: A text editor usually provides a way to deal with mistakes made by the user. A simple example is undoing the most recent change. A more attractive solution is to enable the undoing of multiple changes. We want to develop such an editor. For the purpose of discussion, let us call it TEDDI.

The design of TEDDI includes a multi-level undo mechanism and allows for future enhancements, such as the addition of new features or a batch mode of operation.

Context: Applications that need flexible and extensible user interfaces or applications that provide services related to the execution of user functions such as scheduling or undo.

Problem: An application that includes a large set of features benefits from a well structured solution for mapping its interface to its interval functionality.

The following forces shape the solution:

- Different users like to work with an application in different ways
- Enhancements of the application should not break existing code.
- Additional services such as undo should be implemented consistently for all requests.

Solution: The command Processor pattern builds on the command design pattern in [GHJV95]. Both patterns follow the idea of encapsulating requests into objects. Whenever a user calls a specific function of the application, the request is turned into command object.

Structure: The abstract command component defines the interface of all command objects. As a minimum this interface consists of a procedure to execute a command.

The command processor manages command objects, schedules them and starts their execution. It is the key component that implements additional services related to the execution of commands. The command processor remains independent of specific commands because it only uses the abstract command interface.

The supplier components provide most of the functionality required to execute concrete commands. Related commands often share supplier components. When an undo mechanism is required, a supplier usually provides a means to save and restore its internal state.
Dynamics:

- 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.

**b. Explain publisher-subscriber design pattern. (6 marks)**

The Publisher-Subscribe pattern is useful when you need to balance the following **forces:**
- One or more components must be notified about state changes in a particular component.

- The number and identities of dependent components is not known a priori, or may even change over time.

- Explicit polling by dependents for new information is not feasible.

- The information publisher and its dependents should not be tightly coupled when introducing a change-propagation mechanism.

Solution:

One dedicated component takes the role of the publisher (called subject in [GHJV95]). All components dependent on changes in the publisher are its subscribers (called observers in [GHJV95]). The publisher maintains a registry of currently-subscribed components. Whenever a component wants to become a subscriber, it uses the subscribe interface offered by the publisher. Analogously, it can unsubscribe. Whenever the publisher changes state, it sends a notification to all its subscribers. The subscribers in turn retrieve the changed data at their discretion.

The pattern offers the following degrees of freedom in its implementation:

* You can introduce abstract base classes to let different classes be publishers or subscribers, as described in [GHJV95].

* The publisher can decide which internal state changes it will notify its observers about. It may also queue several changes before calling notify( ).

* An object can be a subscriber to many publishers.

* An object can take both roles, that of a publisher as well as subscriber.

* Subscription and the ensuing notification can be differentiated according to event type. This allows subscribers to get messages only about events in which they are interested.

* The publisher can send selected details of the data change when it notifies its subscribers, or can just send a notification and give the subscribers the responsibility to find out what changed.

Variants:

**Gatekeeper.** The Publisher-Subscriber pattern can be also applied to distributed systems. In this variant a publisher instance in one process notifies remote subscribers. The publisher may alternatively be spread over two processes. In one process a component
sends out messages, while in the receiving process a singleton 'gatekeeper' demultiplexes them by surveying the entry points to the process. The gatekeeper notifies event-handling subscribers when events for which they registered occur. The Reactor pattern (Sch94) describes this scheme in detail.

The Event Channel variant was proposed by the OMG in its Event Service Specification [OMG95] and is targeted at distributed systems. This pattern strongly decouples publishers and subscribers.

C What are idioms and styles? Explain with an example, a style guide idiom.(6 marks)

Idioms represent low-level patterns. There are always many ways to solve a particular programming problem with a given language.

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

Eg: of such problems are memory management, object creation, naming of methods, source code formatting for readability, efficient use of specific library components and so on.

Idioms and style:

A good program written by a single programmer will contain many applications of his set of patterns. Knowing the patterns a programmer uses makes understanding their programs a lot easier.

Eg:

Consider the following section of c/c++ code which both implement a string copy
function for C-style strings:

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

```c
void strcopyPascal(char d[], const char s[])
{
    int i;
    for(i=0;s[i]!="\0";i++)
    {
        d[i]=s[i];
    }
    d[i]="\0";
}
```

Both functions achieve the same result—they copy characters from string s to string d until a character with the value zero is reached.

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

June 2012

8 a Briefly describe the management and Command processor.
(10 marks)

Management

Systems must often handle collections of objects of similar kinds, of services, or even of complex components. One example is incoming events from users or other systems, which must be interpreted and scheduled appropriately.

In well-structured software systems, separate ‘manager’ components are often used to handle such homogeneous collections of objects. Two design patterns are described.
• The command Processor 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.

• The view handler pattern helps to manage views in a software system. A view handler component allows clients to open, manipulate and dispose of views, coordinates, and dependencies between views and organizes their update.

**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.

Example: A text editor usually provides a way to deal with mistakes made by the user. A simple example is undoing the most recent change. A more attractive solution is to enable the undoing of multiple changes. We want to develop such an editor. For the purpose of discussion, let us call it TEDDI.

The design of TEDDI includes a multi-level undo mechanism and allows for future enhancements, such as the addition of new features or a batch mode of operation.

Context: Applications that need flexible and extensible user interfaces or applications that provide services related to the execution of user functions such as scheduling or undo.

Problem: An application that includes a large set of features benefits from a well structured solution for mapping its interface to its interval functionality.

The following forces shape the solution:

• Different users like to work with an application in different ways

• Enhancements of the application should not break existing code.

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

Solution: The command Processor pattern builds on the command design pattern in [GHJV95]. Both patterns follow the idea of encapsulating requests into objects. Whenever a user calls a specific function of the application, the request is turned into command object.

Structure: The abstract command component defines the interface of all command objects. As a minimum this interface consists of a procedure to execute a command.
The abstract command class of TEDDI for example, defines an additional undo method. For each user function we derive a command component from the abstract command.

The controller represents the interface of the application. It accepts requests, such as ‘paste text’, and creates the corresponding command objects.

The command processor manages command objects, schedules them and starts their execution. It is the key component that implements additional services related to the execution of commands. The command processor remains independent of specific commands because it only uses the abstract command interface.

The supplier components provide most of the functionality required to execute concrete commands. Related commands often share supplier components. When an undo mechanism is required, a supplier usually provides a means to save and restore its internal state.
8 b What is **IDIOMS and view handler.**

**(10 marks)**

Variants:

*Spread controller functionality.* In this variant the role of the controller can be distributed over several components. For example, each user interface element such as a menu button could create a command object when activated. However, the role of the controller is not restricted to components of the graphical user interface.

*Combination with Interpreter pattern.* In this variant a scripting language provides a programmable interface to an application. The parser component of the script interpreter takes the role of the controller. Apply the Interpreter pattern and build the abstract syntax.
tree from command objects. The command processor is the client in the Interpreter pattern. It carries out interpretation by activating the commands.

Known Uses:

- **ET++** provides a framework of command processors that support unlimited, bounded, and single undo and redo.
- **MacApp** uses the Command Processor design pattern to provide undoable operations.
- **Interviews** includes an action class that is an abstract base class providing the functionality of a command component.
- **ATM-P** implements a simplified version of the Command Processor pattern.
- **SICAT** implements the Command Processor pattern to provide a well-defined undo facility in the control program and the graphical SDL editors.

**Benefits:**

- *Flexibility in the way requests are activated.*
- *Flexibility in the number and functionality of requests.*
- *Programming execution-related services.*
- *Testability at application level.*
- *Concurrency*

**Liabilities:**

- *Efficiency loss.*
- *Complexity in acquiring command parameters.*

**View Handler:**

The View Handler design pattern helps to manage all views that a software system provides. A view handler component allows clients to open, manipulate and dispose of views. It also coordinates dependencies between views and organizes their update.

Example: Multi-document editors allow several documents to be worked on simultaneously. Each document is displayed in its own window.
Context: A software system that provides multiple views of application-specific data or that supports working with multiple documents.

Problem: Software systems supporting multiple views often need additional functionality for managing them. Users want to be able to conveniently open, manipulate and dispose of views, such as windows and their contents.

Several forces drive the solution to their problem:

- Managing multiple views should be easy from the user's perspective, and also for client components within the system.
- Implementations of individual views should not depend on each other or be mixed with the code used to manage views.
- View implementations can vary, and additional types of views may be added during the lifetime of the system.

Solution:

Separate the management of views from the code required to present or control specific views. This general principle is found in three application areas:

- A view handler component manages all views that the software system provides.
- Specific views, together with functionality for their presentation and control, are encapsulated within separate view components—one for each kind of view. Suppliers provide views with the data they must present.
- The View Handler pattern adapts the idea of separating from functional core, as proposed by the Model-View-Controller

- You can consider the view handler component as an Abstract Factory and as a
Mediator. It is abstract factory because clients are independent of how specific views are created. It is mediator because clients independent of how views are coordinated.

Structure: The view handler is the central component of this pattern. It is responsible for opening new views and clients can specify the view they want.

The view handler also offers functions for closing views, both individual ones and all currently open views, as is needed when quitting the application.

The main responsibility of the view handler is to offer view management services. Examples include functions to quickly bring a specific view into the foreground, to tile all views, to split individual views into several parts, to refresh all views and to clone views to provide several views of the same document.

DEC 08

8 a What are the steps to implement the pattern

(20 marks)

To implement this pattern, carry out the following steps:

1 Define the interface of the abstract command. The abstract command class hides the details of all specific commands. An example is a method 'getNameAndParameters' for logging commands.

```cpp
class AbstractCommand {
    public:
        enum CmdType { no_change, normal, no_undo };
        virtual ~AbstractCommand();
        virtual void doit();
        virtual void undo();
        CmdType getType() const { return type; }
        virtual String getName() const { return "NONE";
            // gives name of command for selection
            // in undo/redo menu
        protected:
            CmdType type;
            AbstractCommand(CmdType t=no_change): type(t){}
}
```

2 Design the command components for each type of request that the application supports. An example of the second situation is a multi-document editor in which a command is connected to a specific document object.
3 Increase flexibility by providing macro commands that combine several successive commands. Apply the composite pattern to implement such a macro command component.

```java
class AbstractCommand {
public:
  enum CmdType { no_change, normal, no_undo };
  virtual ~AbstractCommand();
  virtual void doit();
  virtual void undo();
  CmdType getType() const { return type; }
  virtual String getName() const { return "NONAME"
    // gives name of command for selection
    // in undo/redo menu
protected:
  CmdType type;
  AbstractCommand(CmdType t=no_change): type(t) {}
};
```

4. Implement the controller component. Command objects are created by the controller, for example with the help of the 'creational' patterns Abstract Factory and Prototype.

```java
void TEDDI_controller::deleteButtonPressed() {
  AbstractCommand *delcmd =
    new DeleteWordCommand(
      this->getCursor(), // pass cursor position
      this->getText());   // pass text
  theEditor->perform(delcmd);
}
```
5. **Implement access to the additional services of the command processor:** A user-accessible additional service is normally implemented by a specific command class.

```cpp
class UndoCommand : public AbstractCommand {
public:
    UndoCommand()
        : AbstractCommand(no_change) {
    virtual ~UndoCommand();
    virtual void doit() { theCP->undo_lastcmd(); }
}
```

6. **Implement the command processor component.** The command processor receives command objects from the controller and takes responsibility for them.

```cpp
class CommandProcessor {
public:
    CommandProcessor();
    virtual ~CommandProcessor();
    virtual void do_cmd(AbstractCommand *cmd) {
        // do cmd and push it on donestack
        cmd->doit();
        switch (cmd->getType()) {
        case AbstractCommand::normal:
            donestack.push(cmd); break;
        case AbstractCommand::no_undo:
            donestack.make_empty();
            undonestack.make_empty();
            // Fall through:
        case AbstractCommand::no_change:
            // take responsibility for command objec
delete cmd;
            break;
        }
    }
private:
    // this method is only used by UndoCommand
    virtual void undo_lastcmd();
    // pop cmd from donestack,
    // undo it, and push it on undonestack
    // this method is only used by RedoCommand
    virtual void redo_last undone() {
        AbstractCommand *last = undonestack.pop();
        if (last) this->do_cmd(last);
    }
private:
    Stack<AbstractCommand*> donestack, undonestack;
};
```
8 b Describe communication pattern

- The Forwarder-Receiver design pattern provides transparent inter-process communication for software systems with a peer-to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.

- The Client-Dispatcher-Server design pattern introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.

- While the Forwarder-Receiver pattern provides encapsulation, Client-Dispatcher-Server offers location transparency. If you need to support both encapsulation and location transparency, you could combine these patterns. Keeping cooperating components consistent is another problem in communication. we describe one pattern that addresses this issue:

- The Publisher-Subscriber pattern helps to keep the state of cooperating components synchronized. To achieve this it enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state.

May/June 2010

8 a) Explain the liabilities imposed by Command Processor Pattern?

The controller represents the interface of the application. It accepts requests, such as ‘paste text’, and creates the corresponding command objects.

The command processor manages command objects, schedules them and starts their execution. It is the key component that implements additional services related to the execution of commands. The command processor remains independent of specific commands because it only uses the abstract command interface.

The supplier components provide most of the functionality required to execute concrete commands. Related commands often share supplier components. When an undo mechanism is required, a supplier usually provides a means to save and restore its internal state.
Dynamics:

- 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.

b) Explain View Handler Design Pattern?

The View Handler design pattern helps to manage all views that a software system provides. A view handler component allows clients to open, manipulate and dispose of views. It also coordinates dependencies between views and organizes their update.

Example: Multi-document editors allow several documents to be worked on simultaneously. Each document is displayed in its own window.

Context: A software system that provides multiple views of application-specific data or that supports working with multiple documents.

Problem: Software systems supporting multiple views often need additional functionality for managing them. Users want to be able to conveniently open, manipulate and dispose of views, such as windows and their contents.

Several forces drive the solution to their problem:

- Managing multiple views should be easy from the user’s perspective, and also for client components within the system.
- Implementations of individual views should not depend on each other or be mixed with the code used to manage views.
- View implementations can vary, and additional types of views may be added during the lifetime of the system.

Solution:

Separate the management of views from the code required to present or control specific views. This general principle is found in three application areas:
A view handler component manages all views that the software system provides.

Specific views, together with functionality for their presentation and control, are encapsulated within separate view components—one for each kind of view. Suppliers provide views with the data they must present.

The View Handler pattern adapts the idea of separating from functional core, as proposed by the Model-View-Controller

You can consider the view handler component as an Abstract Factory and as a Mediator. It is abstract factory because clients are independent of how specific views are created. It is mediator because clients independent of how views are coordinated

Structure: The view handler is the central component of this pattern. It is responsible for opening new views and clients can specify the view they want.

The view handler also offers functions for closing views, both individual ones and all currently open views, as is needed when quitting the application.

The main responsibility of the view handler is to offer view management services. Examples include functions to quickly bring a specific view into the foreground, to tile all views, to split individual views into several parts, to refresh all views and to clone views to provide several views of the same document.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>View Handler</td>
<td>• Specific View</td>
</tr>
</tbody>
</table>

An additional responsibility of the view handler is coordination.

An abstract view component defines an interface that is common to all views. The view handler uses this interface for creating, coordinating and closing views.

Specific view components are derived from the abstract view and implement its interface.

Supplier components provide the data that is displayed by view components. Suppliers offer an interface that allows clients such as views to retrieve and change data.
Dynamics:

**Scenario I** 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.

**Scenario II** 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 before redisplaying themselves.
Implementation: the implementation of a View Handler structure can be divided into 4 steps:

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

2. **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.

   ```
   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() { /* ... */ ;
   }
   ```

3. **Implement the views.** Derive a separate class from the `abstract view` 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.

4. **Define the view handler;** Implement functions for creating views as Factory Methods. Clients can specify the view they want, but they do not control how it is created. The view handler is responsible for instantiating and initializing the correct view component.

   ```
   class ViewHandler {
       // Data structures
       struct ViewInfo {
           AbstractView* view;
           Rectangle boundary;
           bool iconized;
       };
   ```
The following code illustrates the creation of new views.

```cpp
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);
}
```

Variants:

View Handler with Command objects. This variant uses command objects to keep the view handler independent of specific view interfaces. Instead of calling view functionality directly, the view handler creates an appropriate command and executes it. The command itself knows how to operate on the view.

- Macintosh Window Manager
Microsoft Word: The Microsoft Word word-processing system offers functions for cloning, splitting, and tiling windows, and also for bringing an open window into the foreground. Quitting Word closes all open windows; dialogs are displayed requesting the desired action if a window contains data that has been changed but not saved. This provides an example of how a View Handler system can appear to the user, and the functionality it can provide.

Benefits:

- **Uniform handling of views.**
- **Extensibility and changeability of views.**
- **Application-specific view coordination.**

Liabilities:

- **Restricted applicability.**
- **Efficiency.**

C) Explain Problem, Context, and solution for Client Server Dispatcher?

**Context:** A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.

**Problem:** The Client-Dispatcher-Server pattern is useful when you need to balance the following *forces*:

A component should be able to use a service independent of the location of the service provider.

The code implementing the functional core of a service consumer should be separate from the code used to establish a connection with service providers.
8a) What is a pattern? Explain the Pattern description template?

Ans  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.

Pattern description template is as follows:

**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 pattern 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.

**Dynamics**  Typical scenarios describing the run-time behavior of the pattern.

**Implementation:** Guidelines for implementing the pattern. These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs, by adding different, extra, or more detailed steps, or by re-ordering the steps.

**Example:** Discussion of any important aspects for resolving the example that are not yet resolved: covered in the Solution, Structure, Dynamics and Implementation sections.

**Variants:** A brief description of variants or specializations 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 to patterns that help us
refine the pattern we are describing.

8 b) Consider any example problem definition **design** and **implement** proxy pattern ?

. Design Part of Proxy pattern.

**Class Diagram:**

<table>
<thead>
<tr>
<th>Class</th>
<th>Diagram:</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloClient</td>
<td><img src="image" alt="Class Diagram" /></td>
</tr>
<tr>
<td>HelloServerImpl</td>
<td><img src="image" alt="Class Diagram" /></td>
</tr>
</tbody>
</table>

**Use Case Diagram:**

<table>
<thead>
<tr>
<th>Use Case Diagram:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Use Case Diagram" /></td>
</tr>
</tbody>
</table>

**Sequence Diagram:**

<table>
<thead>
<tr>
<th>Sequence Diagram:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Sequence Diagram" /></td>
</tr>
</tbody>
</table>
Activity Diagram:

Client Request for Service

Preprocessing by Proxy

Server Accepts and Processes Request

Responses Back to Client

Proxy Forwards the Request to Server

Response Received & Post Processing...

Response Back to Proxy

Success

Failure

Success

Failure
Implementation:

HelloInterface.java

```java
import java.rmi.*;

public interface HelloInterface extends Remote
{
    public String getMessage() throws RemoteException;
    public void display() throws RemoteException;
}
```

HelloServerImpl.java

```java
import java.rmi.*;
import java.rmi.server.*;

public class HelloServerImpl extends UnicastRemoteObject implements HelloInterface
{
    public HelloServerImpl() throws RemoteException
    {
    }

    public String getMessage() throws RemoteException
    {
        return "Hello Welcome to PESIT";
    }

    public void display() throws RemoteException
    {
        System.out.println("How are u");
    }

    public static void main(String a[])
    {
        Try {
            HelloServerImpl server=new HelloServerImpl();
            Naming.rebind("rmi://localhost.1099/server",server);
        }
    }
```
public class Helloclient
{
    public static void main(String a[])
    {
        try{
            HelloInterface server=(HelloInterface)Naming.lookup("rmi://localhost:1099/server");
            String msg=server.getMessage();
            System.out.println(msg);
            server.display();
        }
        catch(Exception e)  {}  
    }
}