USN | | | | | | | | | | |

13MCA51

Fifth Semester MCA Degree Examination, Dec.2016/Jan.2017
Object Oriented Modeling and Design Patterns

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. What is object orientation? Describe the characteristics of object orientation. **(10 Marks)**

b. Describe the stages of object oriented methodology, used in software development. **(10 Marks)**

2 a. With respect to object oriented terminology describe the following with suitable examples :
i) object ii) class iii) values and attributes iv) operations and methods. **(10 Marks)**

b. What do you mean by link and association? Hence explain with examples. **(06 Marks)**

c. Discuss the terms aggregation and composition. **(04 Marks)**

3 a. Discuss the significance of state diagram and draw the neat state diagram for telephone line system. **(10 Marks)**

b. Describe the following terms with suitable examples :
i) enumeration ii) multiplicity iii) scope iv) visibility. **(10 Marks)**

4 a. Draw the use case diagram for library management system. Hence describe the significance of use case diagram. **(10 Marks)**

b. Draw the sequence diagram for cash withdrawal process in ATM system. Hence describe the guidelines for sequence diagram. **(10 Marks)**

5 a. Explain well defined stages of software development stages. **(10 Marks)**

b. Elaborate and explain the questions to be answered for a good system concept. **(10 Marks)**

6 a. Explain the steps to construct a domain class model. **(10 Marks)**

b. Explain the steps to construct an application interaction model. **(10 Marks)**

7 a. What is pattern? Describe the categories architectural pattern, design pattern, idioms in detail. **(08 Marks)**

b. The following steps improve the organization of class design. Justify.
i) Information hiding
ii) Coherence of entities
iii) Fine tuning packages. **(12 Marks)**

8 Write short notes on :
a. Activity model
b. Publisher – subscriber pattern
c. Association class
d. Abstract and concrete class. **(20 Marks)**

* * * * *

1.a. What is object orientation? Describe the characteristics of object orientation.

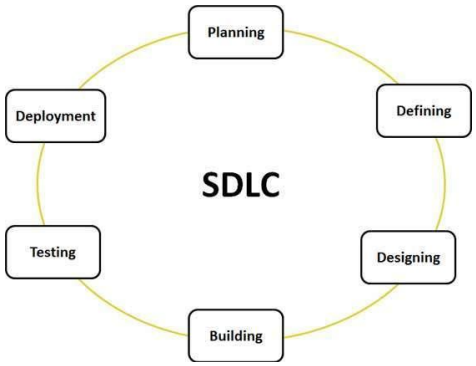
b. Describe the states of object oriented methodology, used in software development.

Ans: a. **Object-oriented** programming (OOP) is a programming language model organized around **objects** rather than "actions" and data rather than logic.

The characteristics of OOP are:

Class definitions – Basic building blocks OOP and a single entity which has data and operations on data together
Objects – The instances of a class which are used in real functionality – its variables and operations
Abstraction – Specifying what to do but not how to do ; a flexible feature for having a overall view of an object's functionality.
Encapsulation – Binding data and operations of data together in a single unit – A class adhere this feature
Inheritance and class hierarchy – Reusability and extension of existing classes
Polymorphism – Multiple definitions for a single name - functions with same name with different functionality; saves time in investing many function names Operator and Function overloading
Generic classes – Class definitions for unspecified data. They are known as container classes. They are flexible and reusable.
Class libraries – Built-in language specific classes
Message passing – Objects communicates through invoking methods and sending data to them. This feature of sending and receiving information among objects through function parameters is known as Message Passing.

b. The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development life cycle consists of the following stages:

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational, and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through .SRS. . Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the product architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity , budget and time constraints , the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometime product deployment happens in stages as per the organizations. business strategy. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

2. a. With respect to object oriented terminologies describe the following with suitable examples:
i) object: In computer science, an **object** can be a variable, a data structure, or a function or a method, and as such, is a location in memory having a value and possibly referenced by an identifier. In the class-based object-oriented programming paradigm, "object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures.
ii) class: In object-oriented programming, a **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (a subroutine that creates objects), and as the type of objects generated by instantiating the class; these distinct concepts are easily conflated.
iii) values and attributes: An **attribute-value system** is a basic knowledge representation framework comprising a table with columns designating "attributes" (also known as "properties", "predicates," "features," "dimensions," "characteristics", "fields", "headers" or "independent variables" depending on the context) and "rows" designating "objects" (also known as "entities," "instances," "exemplars," "elements", "records" or "dependent variables"). Each table cell therefore designates the value (also known as "state") of a particular attribute of a particular object.
iv) operations and methods
b. What do you mean by link and association? Hence explain with examples.

Links: In object modeling links provides a relationship between the objects. These objects or instance may be same or different in data structure and behavior. Therefore a link is a physical or conceptual connection between instances (or objects). For example: Ram works for HCL company. In this example “works for” is the link between “Ram” and “HCL company”. Links are relationship among the objects (instance)
Types of links:
1.One to one links
2.one to many and many to one links
3. many to many
Associations: The object modeling describes as a group of links with common structure and common semantics. All the links among the object are the forms of association among the same classes. The association is the relationship among classes.
1.Association
2.Association with inverse direction
3.Association between student and university
Degree of association:
1.Unary association(degree of one): the association can be defined on a single class. This
type of association called unary (or singular) association.
2.Binary Association (degree of two): The binary association contain the degree of two classes. The association uses two class.
3.Ternary Association(degree of three): The association which contain the degree of three classes is called ternary association. The ternary Association is an atomic unit and cannot be subdivided into binary association without losing information.
4.Quaternary Association(degree of four): The Quaternary Association exists when there are four classes associated.
5.Higher order association(more than four): The higher order association are more complicated to draw , implement because when more than four class need to be associated then it seems a hard task
c. Discuss the terms aggregation and composition.

- **Association** is a relationship where all objects have their own lifecycle and there is no owner.

  Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers, but there is no ownership between the objects and both have their own lifecycle. Both can be created and deleted independently.

- **Aggregation** is a specialized form of Association where all objects have their own lifecycle, but there is ownership and child objects can not belong to another parent object.

  Let's take an example of Department and teacher. A single teacher can not belong to multiple departments, but if we delete the department, the teacher object will *not* be destroyed. We can think about it as a “has-a” relationship.

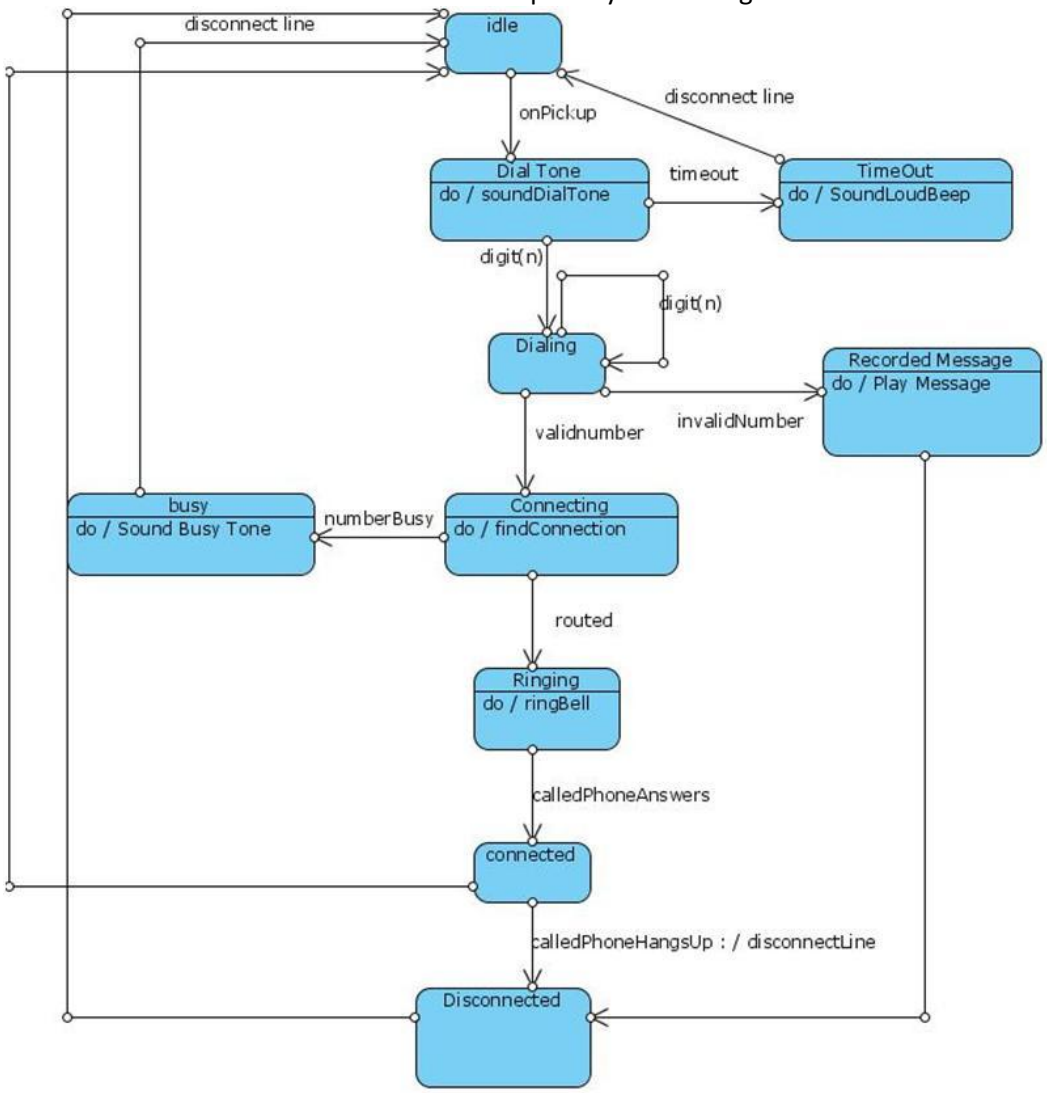
- **Composition** is again specialized form of Aggregation and we can call this as a “death” relationship. It is a strong type of Aggregation. Child object does not have its lifecycle and if parent object is deleted, all child objects will also be deleted.

  Let's take again an example of relationship between House and Rooms. House can contain multiple rooms - there is no independent life of room and any room can not belong to two different houses. If we delete the house - room will automatically be deleted.

  Let's take another example relationship between Questions and Options. Single questions can have multiple options and option can not belong to multiple questions. If we delete the questions, options will automatically be deleted.

1. a. Discuss the significance of state diagram and draw the neat state diagram for telephone line system.
A state diagram, also called a state machine diagram or state chart diagram, is an illustration of the states an object can attain as well as the transitions between those states in the Unified Modeling Language (UML). In this context, a state defines a stage in the evolution or behavior of an object, which is a specific entity in a program or the unit of code representing that entity. State diagrams are useful in all forms of object-oriented programming (OOP). The concept is more than a decade old but has been refined as OOP modeling paradigms have evolved. A state diagram resembles a flowchart in

which the initial state is represented by a large black dot and subsequent states are portrayed as boxes with rounded corners. There may be one or two horizontal lines through a box, dividing it into stacked sections. In that case, the upper section contains the name of the state, the middle section (if any) contains the state variables and the lower section contains the actions performed in that state. If there are no horizontal lines through a box, only the name of the state is written inside it. External straight lines, each with an arrow at one end, connect various pairs of boxes. These lines define the transitions between states. The final state is portrayed as a large black dot with a circle around it.



b. Describe the following terms with suitable examples:

i) Enumeration: An **enumeration** is a complete, ordered listing of all the items in a collection. The term is commonly used in mathematics and computer science to refer to a listing of all of the elements of a set. The precise requirements for an enumeration (for example, whether the set must be finite, or whether the list is allowed to contain repetitions) depend on the discipline of study and the context of a given problem.

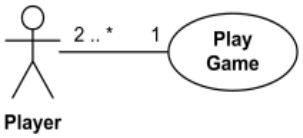
Some sets can be enumerated by means of a **natural ordering** (such as 1, 2, 3, 4, ... for the set of positive integers), but in other cases it may be necessary to impose a (perhaps arbitrary) ordering. In some contexts, such as enumerative combinatorics, the term *enumeration* is used more in the sense of *counting* – with emphasis on determination of the number of elements that a set contains, rather than the production of an explicit listing of those elements.

ii) multiplicity: **Multiplicity** is a definition of **cardinality** - i.e. **number of elements** - of some collection of elements by providing an inclusive interval of non-negative integers to specify the allowable number of instances of described element. Multiplicity interval has some **lower bound** and (possibly infinite) **upper bound**:

multiplicity-range ::= [lower-bound '..'] upper-bound
lower-bound ::= natural-value-specification
upper-bound ::= natural-value-specification | '*'

Lower and upper bounds could be natural constants or constant expressions evaluated to natural (non negative) number. Upper bound could be also specified as asterisk '*' which denotes unlimited number of elements. Upper bound should be greater than or equal to the lower bound.

Multiplicity	**Option**	**Cardinality**
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

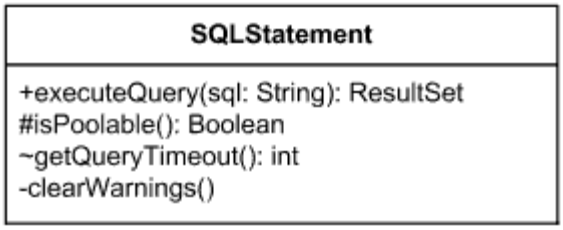


iii) scope: A **scope** is a region of the program. For example, speaking of scope of a variable, there are three places, where **variables** can be declared − Inside a function or a block which is called local **variables**, In the definition of function parameters which is called formal parameters, Outside of all functions which is called global **variables**.

iv) visibility: **Visibility** allows to constrain the usage of a **named element**, either in namespaces or in access to the element. It is used with classes, packages, generalizations, element import, package import.
UML has the following types of **visibility**:

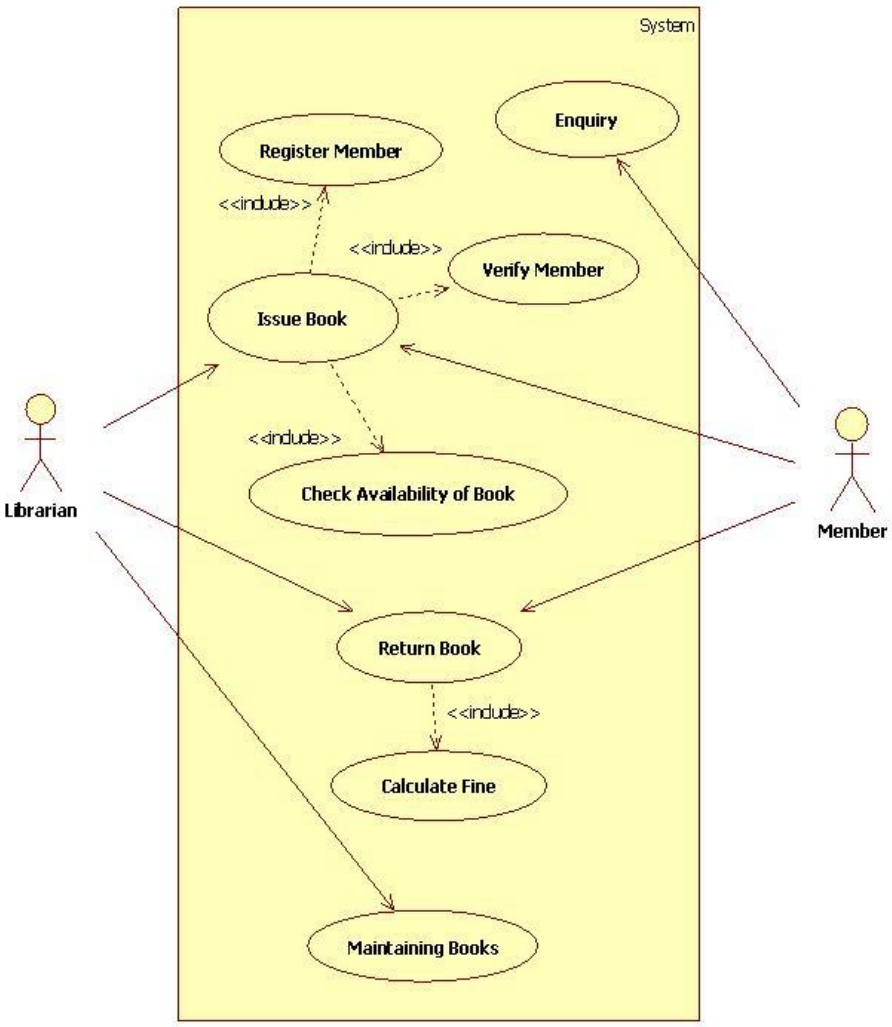
- public
- package
- protected
- private

Note, that if a **named element** is not owned by any namespace, then it does not have a visibility.
A **public** element is visible to all elements that can access the contents of the namespace that owns it. **Public** visibility is represented by '+' literal.
A **package** element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace. Only named elements that are not owned by packages can be marked as having package visibility. Any element marked as having package visibility is visible to all elements within the nearest enclosing package (given that other owning elements have proper visibility). Outside the nearest enclosing package, an element marked as having package visibility is not visible. **Package** visibility is represented by '~' literal. A **protected** element is visible to elements that have a generalization relationship to the namespace that owns it. **Protected** visibility is represented by '#' literal. A **private** element is only visible inside the namespace that owns it. **Private** visibility is represented by '-' literal.



Operation executeQuery is public, isPoolable - protected,
getQueryTimeout - with package visibility, and clearWarnings is private.
If some named element appears to have multiple visibilities, for example, by being imported multiple times, public visibility overrides private visibility. If an element is imported twice into the same namespace, once using a public import and another time using a private import, resulting visibility is public.

4. a. Draw the use case diagram for library management system. Hence describe the significance of use case diagram.

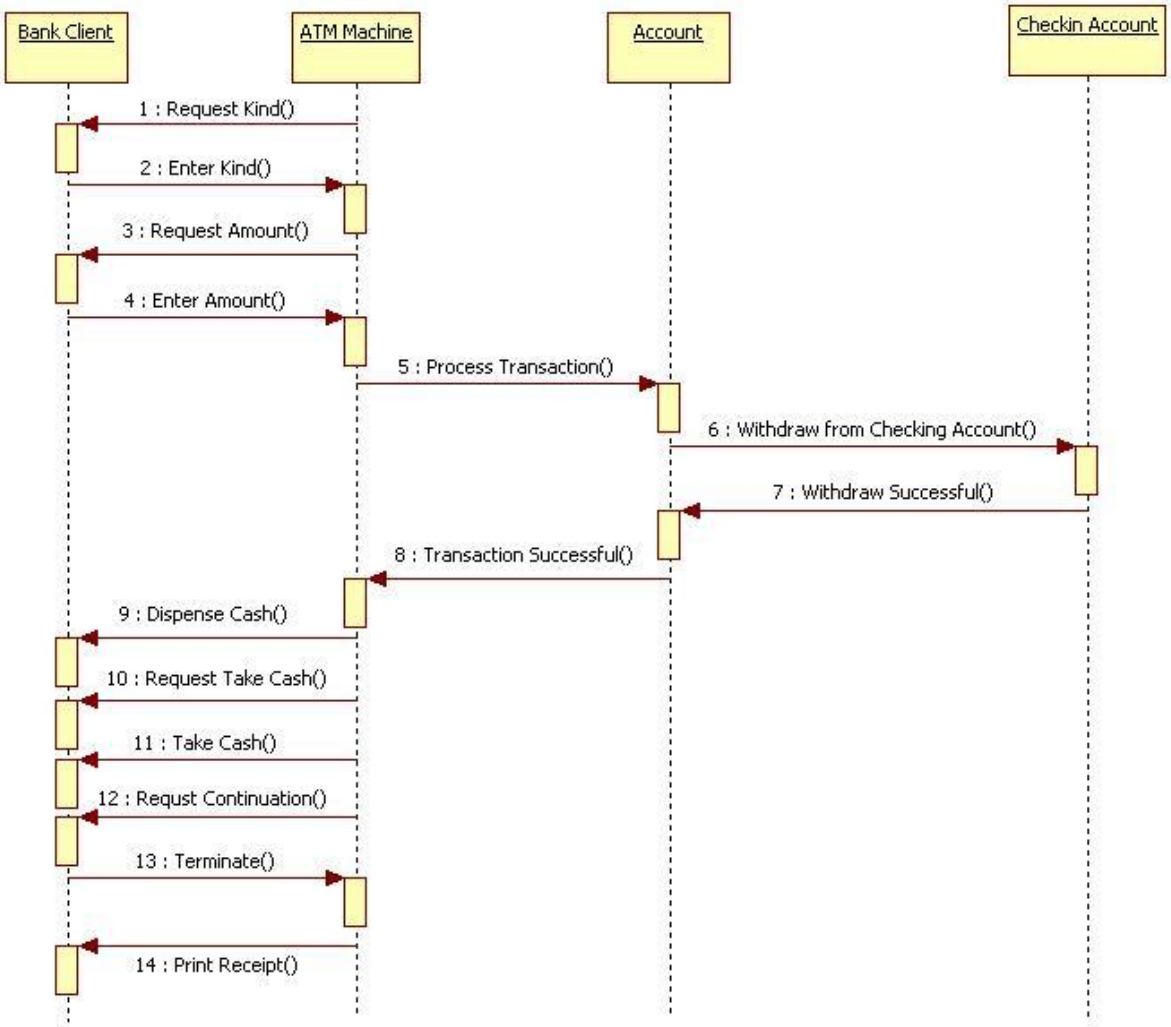


The purpose of use case diagram is to capture the dynamic aspect of a system. But this definition is too generic to describe the purpose. Because other four diagrams (activity, sequence, collaboration and Statechart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams. Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified. Now when the initial task is complete use case diagrams are modeled to present the outside view.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

b. Draw the sequence diagram for cash withdrawal process in ATM system. Hence describe the guidelines for sequence diagram.



UML Sequence diagrams are a dynamic modeling technique, as are collaboration diagrams and activity diagrams. UML sequence diagrams are typically used to:

1. Validate and flesh out the logic of a usage scenario. A usage scenario is exactly what its name indicates - the description of a potential way that your system is used. The logic of a usage scenario may be part of a use case, perhaps an alternate course; 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; or a pass through the logic contained in several use cases, for example a student enrolls in the university then immediately enrolls in three seminars.
2. Explore your design because they provide a way for you to visually step through invocation of the operations defined by your classes.
3. To detect bottlenecks within an object-oriented design. By looking at what messages are being sent to an object, and by looking at roughly how long it takes to run the invoked method, you quickly get an understanding of where you need to change your design to distribute the load within your system. In fact some CASE tools even enable you to simulate this aspect of your software.
4. Give you a feel for which classes in your application are going to be complex, which in turn is an indication that you may need to draw state chart diagrams for those classes.

There are guidelines for:

General Issues:

1. Strive for Left-To-Right Ordering Of Messages
2. Layer The Classifiers

3. Name Actors Consistently With Your Use Case Diagrams
4. Name Classes Consistently With Your Class Diagrams
5. An Actor Can Have The Same Name as a Class
6. Include a Prose Description of the Logic
7. Place Human and Organization Actors On the Left-Most Side of Your Diagram
8. Place Reactive System Actors on the Right-Most Side of Your Diagram
9. Place Proactive System Actors on the Left-Most Side of Your Diagram
10. Avoid Modeling Object Destruction

Classifiers:

1. Name Objects When You Refer To Them In Messages
2. Name Objects When Several of the Same Type Exist
3. Apply Textual Stereotypes Consistently
4. Apply Visual Stereotypes Sparingly
5. Focus on Critical Interactions

Messages:

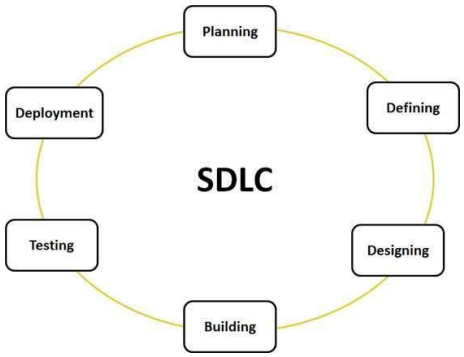
1. Justify Message Names Beside the Arrowhead
2. Create Objects Directly
3. Apply Operation Signatures for Software Messages
4. Use Prose for Messages Involving Human and Organization Actors
5. Prefer Names Over Types for Parameters
6. Indicate Types as Parameter Placeholders
7. Messages to Classes are Implemented as Static Operations
8. Apply the <<include>> Stereotype for Use Case Invocations

Return Values:

1. Do Not Model a Return Value When it is Obvious What is Being Returned
2. Model a Return Value Only When You Need to Refer to It Elsewhere
3. Justify Return Values Beside the Arrowhead
4. Model Return Values As Part of a Method Invocation
5. Indicate Types as Return Value Placeholders
6. Explicitly Indicate The Actual Value for Simple Values

5. a. Explain well defined stages of software development stages.

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development life cycle consists of the following stages:

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational, and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through .SRS. . Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the product architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity , budget and time constraints , the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometime product deployment happens in stages as per the organizations. business strategy. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

b. Elaborate and explain the questions to be answered for a good system concept.

A good system concept must answer the following questions.

- Who is the application for ?
  - Understand the stakeholders of the system;
  - Usually two important ones are:
    - Financial sponsor (client)
    - End Users
- What problems will it Solve?
  - Bound the size of effort and scope of system
  - Determine what feature is in and what is out
- Where will it be used ?
  - Characterize the environment the system will be used, e,g
    - Mission-critical?
    - Experimental?
    - Enhancements to existing system?
  - For commercial products, characterize the customer base.
- When is it needed?
  - Feasible time (T_f)
    - The time in which the system can be developed within the constraints of cost and available resource
  - Required time (T_r)
    - The time that the system is needed to meet the business goals
  - If (T_r < T_f), work with technologists and business experts to trim the system
- Why is it needed ?
  - Prepare a business case
    - Financial justification including cost, tangible and intangible benefits, risks and alternatives.
    - For a commercial product, estimate the number of units you can sell and determine a reasonable price.
- How will it work
  - Investigate feasibility of the problem
  - Build prototype, if it helps clarifying a concept or removing a technological risk

6. a. Explain the steps to construct a domain class model

The first step in analyzing the requirement is to correct a domain model.
The domain model describes the real world classes and their relationships to each other.

To construct a domain class model we need to perform the following:

Find classes
Prepare a data dictionary
Find associations
Find attributes of objects and links
Organize and simplify classes using inheritance
Verify that access paths exists for likely queries
Iterate and refine the model
Reconsider the level of abstraction
Group classes into packages.

Finding Classes:- Classes often correspond to nouns for example . In the statement “a reservation system to sell tickets to performances at various theaters tentative classes would be Reservation , System , Ticket ,Performances and Theater

Keeping the right classes:

We need to discard the unnecessary and incorrect classes:
Redundant classes: if two classes express same
Irrelevant classes: It has little or nothing to do with the problem
Vague class: too broad in scope
Attributes: describe individual objects
Operations
Roles
Implementation constructs: CPU, subroutine, process and algorithm

Derived classes: omit class that can be derived from other class
Prepare data dictionary: Information regarding the data is maintained
Keeping the right associations:
Associations between eliminated classes
Irrelevant or implementation associations
Actions
Ternary association
Derived Association

b. Explain the steps to construct an application interaction model.

We can construct an application interaction model with the following steps:
Determine the system boundary
Find actors
Find use cases
Find initial and final events
Prepare normal scenarios
Add variation and exception scenarios
Find external events
Prepare activity diagrams for complex use cases
Organize actors and use cases
Check against the domain class model
Determine the system boundary:
We must decide what the system includes and more than that what it omits.
If the system boundary is drawn correctly, we can treat the system as a black box in its interactions with the outside world.
We can regard the system as a single object, whose internal details are hidden and changeable.
Example: ATM, Cashier transaction and ATM transaction follows same domain but have its own distinct application model.
Finding Actors:
We must identify the external objects that interacts directly with the system
Actors include humans, external devices, and other software systems.
Ex: A particular person may be both a bank teller and a customer of the same bank.
For the ATM application, the actors are customer,Bank and Consortium
Finding Usecase:
For each actor, list the fundamentally different ways in which the actor uses the system. Each of these ways is a use case.Each use case should represent a kind of service that the
system provides – something that provide value to the actor.
Initiate session: The ATM establishes the identity of the user and makes available a list of accounts and actions.
Query account: The system provides general data for an account, such as the current balance, date of last transaction and date of mailing for last statement
Finding initial and final state: Initiate session: The initial event is the customer insertion of a cash card. 2 final events: system keeps the card and returns the cash card.
Query account: The initial event is the customer's request for account data. The final event is the systems delivery of account data to the customer.
Process transaction: The initial event is the customers initiation of a transaction. There are two final events: committing or aborting the transaction.
Transmit data: the initial event is request for account data. Final event is successful transmission of data.

7. a. What is pattern? Describe the categories architectural pattern, design pattern, idioms in detail.

Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns.
Each pattern is a three part rule:
• a relation between a certain context
• a problem
• and a solution

Architectural Pattern:

- Architectural Patterns express a fundamental structural organization for software systems

- It provides a set of predefined sub-systems, specifies their responsibilities, and includes rules for establishing relationships between them
- Example: 3-tier database systems (Database, Intermediate DB Layer, User-Application), Client/Server, Component (Module) -Based Software Systems, Feedback Systems, Event-Driven Systems

Design Pattern:

- Popularized by the 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also called the *Gang of Four Book* or *GOF)*
- Their work largely based on Alexander's *The Timeless Way of Building*
- OOP is made up of objects, where an object packages both data and operations that may be performed on that data
- Ideally, the only way to change the data of an object is via a request to that object (normally a method call) – we call this encapsulation; it's representation is invisible to the outside world
- The hard part of object-oriented design is decomposing a system into objects. Many factors come into play: granularity, encapsulation, dependency, flexibility, performance, reusability, and each affects the nature of the decomposition, often in conflicting ways

Idioms:

- **Idioms are lower level patterns than general design patterns; they are often specific to a programming language, and usually only encompass a few lines of code**
- **Idioms however are important to understand, and it's often crucial to pick up the particular idioms of a language to use it properly**

b. The following steps improve the organization of class design. Justify.

i) Information hiding:

15.10.1 Information Hiding

During analysis we did not consider information visibility—rather our focus was on understanding the application. 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***. Then you can change the implementation of a class without requiring that clients of the class modify code. In addition, the resulting "firewalls" around classes limit the effects of changes so that you can better understand them.

There are several ways to hide information.

- **Limit the scope of class-model traversals**. Taken to an extreme, a method could traverse all associations of the class model to locate and access an object. Such unconstrained visibility is appropriate during analysis, when you are trying to understand a problem, but methods that know too much about a model are fragile and easily invalidated by changes. During design you should try to limit the scope of any one method [Lieberherr-88]. An object should access only objects that are directly related (directly connected by an association). An object can access indirectly related objects via the methods of intervening objects.
- **Do not directly access foreign attributes**. Generally speaking, it is acceptable for subclasses to access the attributes of their superclasses. However, classes should not access the attributes of an associated class. Instead, call an operation to access the attributes of an associated class.
- **Define interfaces at a high a level of abstraction**. It is desirable to minimize class couplings. One way to do this is by raising the level of abstraction of interfaces. It is fine to call a method on another class for a meaningful task, but you should avoid doing so for minutia.
- **Hide external objects**. Use boundary objects to isolate the interior of a system from its external environment. A ***boundary object*** is an object whose purpose is to mediate requests and responses between the inside and the outside. It accepts external requests in a client-friendly form and transforms them into a form convenient for the internal implementation.
- **Avoid cascading method calls**. Avoid applying a method to the result of another method, unless the result class is already a supplier of methods to the caller. Instead consider writing a method to combine the two operations.

ii) Coherence of entities

15.10.2 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 parts fit together toward a common goal. An entity should have a single major theme; it should not be a collection of unrelated parts.

A method should do one thing well. A single method should not contain both policy and implementation. ***Policy*** is the making of context-dependent decisions. ***Implementation*** is the execution of fully-specified algorithms. Policy involves making decisions, gathering global information, interacting with the outside world, and interpreting special cases. A policy method contains I/O statements, conditionals, and accesses data stores. A policy method does not contain complicated algorithms but instead calls the appropriate implementation methods. An implementation method encodes exactly one algorithm, without making any decisions, assumptions, defaults, or deviations. All its information is supplied as arguments, so the argument list may be long.

Separating policy and implementation greatly increases the possibility of reuse. The implementation methods do not contain any context dependencies, so they are likely to be reusable. Usually you must rewrite policy methods in a new application, but they are often simple and consist mostly of high-level decisions and low-level calls.

For example, consider an operation to credit interest on a checking account. Interest is compounded daily based on the balance, but all interest for a month is lost if the account is closed. The interest logic consists of two parts: an implementation method that computes the interest due between a pair of days, without regard to any forfeitures or other provisions; and a policy method that decides whether and for what interval the implementation method is called. The implementation method is complex, but likely to be reused. Policy methods are less likely to be reusable, but simpler to write.

A class should not serve too many purposes at once. If it is too complicated, you can break it up using either generalization or aggregation. Smaller pieces are more likely to be reusable than large complicated pieces. Exact numbers are somewhat risky, but as a rule of thumb consider breaking up a class if it contains more than about 10 attributes, 10 associations, or 20 operations. Always break a class if the attributes, associations, or operations sharply divide into two or more unrelated groups.

iii) Fine tuning packages

15.10.3 Fine-Tuning Packages

During analysis you partitioned the class model into packages. This initial organization may not be suitable or optimal for implementation. You should 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.

You can use the connectivity of the class model as a guide for forming packages. As a rough rule of thumb, classes that are closely connected by associations should be in the same package, while classes that are unconnected, or loosely connected, may be in separate packages. Of course there are other aspects to consider. Packages should have some theme, functional cohesiveness, or unity of purpose.

The number of different operations that traverse a given association is a good measure of its coupling strength. We are referring to the number of different ways that the association is used, not the frequency of traversal. Try to keep strong coupling within a single package.

8. Write short notes on:
a. Activity model

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 are intended to model both computational and organizational processes (i.e. workflows). Activity diagrams show the overall flow of control.

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

- *rounded rectangles* represent *actions*;
- *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.

Activity diagrams may 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.

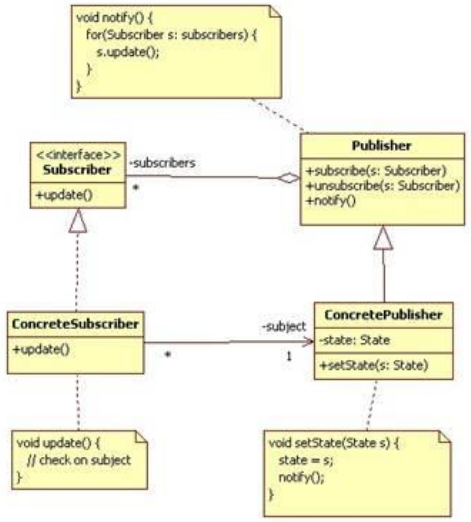
b. Publisher – Subscriber pattern

In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead characterize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

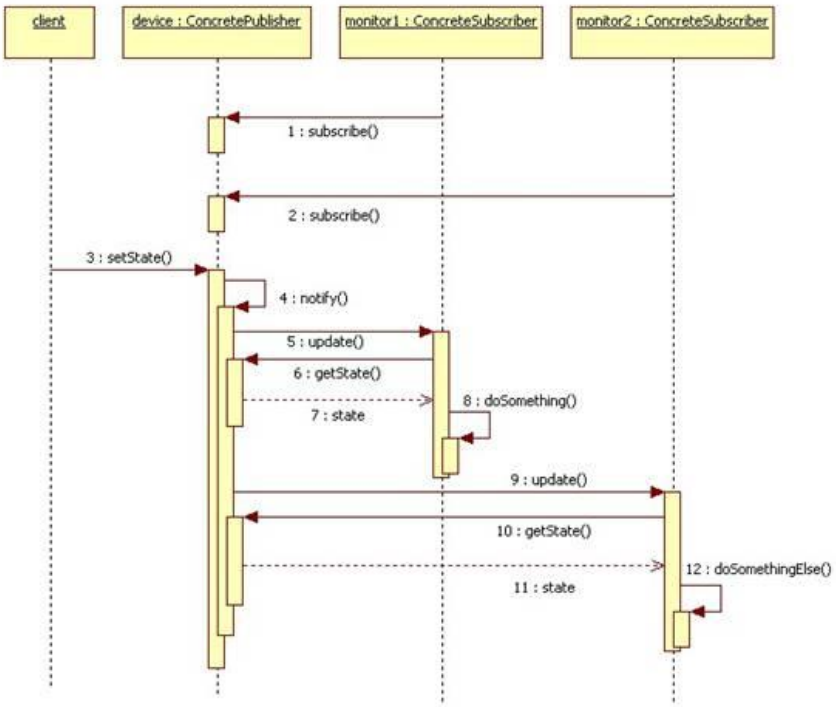
Publish–subscribe is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the pub/sub and message queue models in their API, e.g. Java Message Service (JMS).

This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data.

Structure

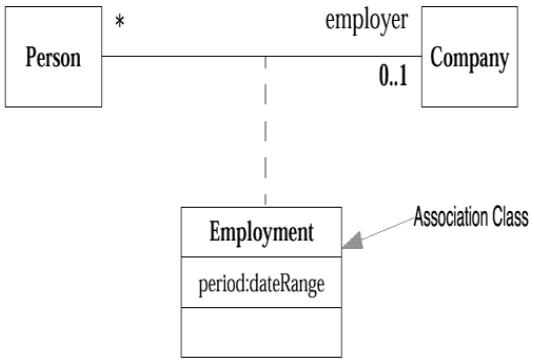


Behavior



c. Association Class: **Association classes** allow you to add attributes, operations, and other features to associations, as shown in Figure 6-14.

Figure 6-14. Association Class

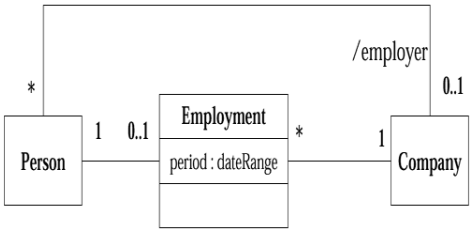


We can see from the diagram that a Person may work for a single Company. We need to keep information about the period of time that each employee works for each Company.

We can do this by adding a *dateRange* attribute to the association. We could add this attribute to the Person class, but it is really a fact about a Person's relationship to a Company, which will change should the person's employer change.

Figure 6-15 shows another way to represent this information: make Employment a full class in its own right. (Note how the multiplicities have been moved accordingly.) In this case, each of the classes in the original association has a single-valued association end with regard to the Employment class. The "employer" end now is derived, although you don't have to show this.

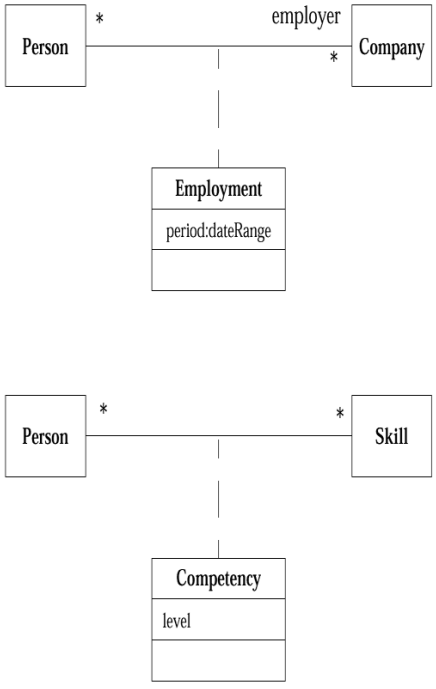
Figure 6-15. Promoting an Association Class to a Full Class



What benefit do you gain with the association class to offset the extra notation you have to remember? The association class adds an extra constraint, in that there can be only one instance of the association class between any two participating objects. I feel the need for an example.

Take a look at the two diagrams in Figure 6-16. These diagrams have much the same form. However, we could imagine a Person working for the same Company at different periods of timethat is, he or she leaves and later returns. This means that a Person could have more than one Employment association with the same Company over time. With regard to the Person and Skill classes, it would be hard to see why a Person would have more than one Competency in the same Skill; indeed, you would probably consider that an error.

Figure 6-16. Association Class Subtleties



In the UML, only the latter case is legal. You can have only one Competency for each combination of Person and Skill.

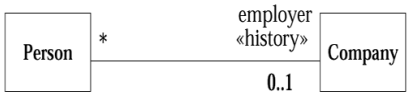
The top diagram in Figure 6-16 would not allow a Person to have more than one Employment with the same Company. If you need to allow this, you need to make Employment a full class, in the style of Figure 6-15.

In the past, modelers made various assumptions about the meaning of an association class in these circumstances. Some assumed that you can have only unique combinations, such as competency, whereas others did not assume such a constraint.

Many people did not think about it at all and may have assumed the constraint in some places and not in others. So when using the UML, remember that the constraint is always there.

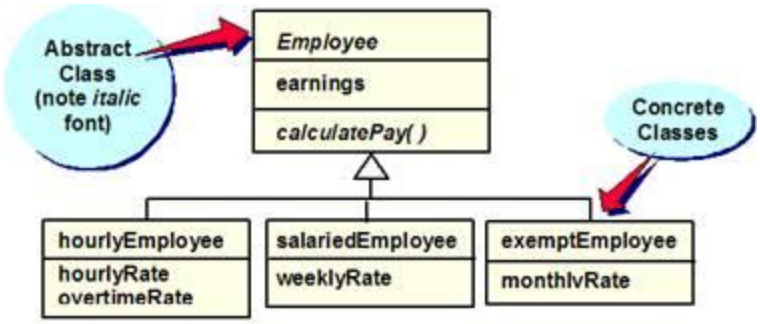
You often find this kind of construct with historical information, such as in the preceding Employment case. A useful pattern here is the *Historic Mapping* pattern described in Fowler (1997). We can use this by defining a history stereotype (see Figure 6-17).

Figure 6-17. History Stereotype for Associations



d. Abstract and concrete class:

All classes can be designated as either abstract or concrete. Concrete is the default. This means that the class can have (direct) instances. In contrast, abstract means that a class cannot have its own (direct) instances. Abstract classes exist purely to generalize common behaviour that would otherwise be duplicated across (sub)classes. We'll illustrate this with the following diagram.



In the above diagram there is an abstract class called Employee. In UML, the name of an abstract class is written in an italic font. This class contains one abstract method called calculatePay, it is written in a italic font. An abstract method has no implementation. Typically an abstract class contains one or more abstract method. The diagram also shows three subclasses that inherit behaviour and data attributes from the Employee class. These are concrete classes that can be instantiated; abstract classes cannot directly be instantiated. Think of abstract classes are providing some generic behaviour but also delegating some behaviour to a subclass. The subclass will have to provide the delegated behaviour or it will also have to be marked as abstract and hence will not be able to be instantiated.

Key point: When extending an abstract class the subclass must implement all of the abstract methods in order to be classified as a concrete class.

Returning to the diagram, each of the subclasses provides its own implementation of the calculatePay method. This means that each class provides an implementation of the abstract behaviour but each subclass does that in a slightly different way.

It is often assumed that all superclasses must be abstract. This is not so. Instances of a concrete superclass represent the 'normal' cases; instances of its subclasses represent special exceptions. E.g. Instances of SavingsProduct are 'normal' products; instances of TaxExemptSavingsProduct are also savings products - but ones which encapsulate special rules.

Key point: The bottom-level classes must be concrete, otherwise no instances will ever be created that exhibit the defined behaviour!!