**1.Problem Statement**

The Online Shopping is a web based application intended for online retailers. The main Objective of this application is to make it interactive and its ease of use. It would make

Searching, viewing and selection of a product easier. It contains a sophisticated search

Engine for users to search for products specific to their needs. The search engine provides

An easy and convenient way to search for products where a user can Search for a product

Interactively and the search engine would refine the products available based on the

User’s input. The user can then view the complete specification of each product. They can

Also view the product reviews and also write their own reviews. The application also

Provides a drag and drop feature so that a user can add a product to the shopping cart by

Dragging the item in to the shopping cart. The main emphasis lies in providing a user friendly

Search engine for effectively showing the desired results and its drag and drop

Behavior.

Customer registers for portal by signup page. Enter their name, age, address, phone number, credit information. Can update their profile at any time by login into the portal. Update includes changing their address, age, credit information.

Administrator maintains the product information in the portal stores the product and customer and billing information and orders info in a relational database. Admin adds an item to the product database, process the bills of customers. When order received from customer. His credit card is charged for the price of item by credit card Company, and then bill along with items catalog is sent to courier service.

Login security should be provided for registered customers. Web portal must be browser independent as the system is web based and should run on IE, opera, safari, chrome; Mozilla and it work in all pc. Search of an item must return the results in less than a second. Ease of future enhancement is very important.

Administrator generates order id for items and sends to courier service. Oracle database is maintained to store all product, customer and billing information. SQL queries are used to perform the queries against the database and transform the documents for display.

**USECASE VIEW**

**2. Identification Of Actors**

Actors are not part of system.

Actors represents anyone that interacts with (input to receive output from) the system.

An actor is someone or something that:

Interacts with or uses the system.

Provides input to and receive information from the system.

Is external to the system and has no control over the use cases.

Actors are discovered by examining:

Who directly uses the system?

Who is responsible for maintaining the system?

External hardware used by the system.

Other systems that need to interact with the system.

The need of the actor is used to develop use cases. This insures that the system will be what the user expected.

Graphical Depiction

An actor is a stereo type of a class and is depicted as “Stickman” on a use-case diagram.



Naming: The name of the actor is displayed below the icon.

Question that help to identify actors:

1. Who is interested in a certain requirements?
2. Where is the system used within the organization?
3. Who will benefit from the use of the system?
4. Who will supply the system with information, use this information, and remove this information?
5. Who will support and maintain the system?
6. Does the system use an external resource?
7. Does one person play several different roles?
8. Do several people play the same role?
9. Does the system interact with a legacy system?

Using the above questions we have identified four actors is online shopping system they are :

1. Customer.
2. Administrator.
3. Credit card Company.
4. Courier Company.

Customer: Customer can browse product, order for products, view shopping details and pay the bills.

UML notation:

Credit card Company: It is responsible for the charging the card submitted by the user. It is capable of accepting any type of card.

UML notation:



Administrator: She/he is responsible for maintaining customer details, maintain product catalog, generate ordered and bill and sends order to courier company.

UML notation:



Courier Company: It is responsible to track orders of customers and delivers order to customers.

UML notation:



**3. IDENTIFICATION OF USE-CASES AND SUB USE-CASES**

Use case diagrams graphically represent system behavior (Use case). These diagrams present a high level view of how the system is used as viewed from an outsider’s (actor’s) perspective. A use case is a sequence of transaction performed by a system that yields a measurable result of values for a particular actor. The use cases are all the ways the system may be used.

A use- case diagram can contain:

* Actors(“things” outside the system)
* Use case (system boundaries identifying what the system should do)
* Interactions or relationships between actors and use cases in the system including the associations, dependencies, and generalizations.

Graphical Depiction: The base shape of a use case is an ellipse:

Naming:

* A use case may have a name, although it is typically not a simple name. it is often written as an informal text description of the actors and the sequence of events between objects. Use case names often start with a verb.
* The name of the use case is displayed below the icon:

Maintain product catalog

Questions that help to find use cases:

1. What are the tasks of each actor?
2. Will any actor create, store, change, remove or read information in the system?
3. What use cases will create, store, change, remove or read information?
4. Will any actor need to inform the system about sudden, external changes?
5. Does any actor need to be informed about certain occurrences in the system?
6. What use cases will support or maintain the system?
7. Can all functional requirements be performed by the use cases?

By applying the above questions to the online shopping the following use cases are Identified they are:

1. Register: This use case is started by customer. It provides the facility for customer to register for web site

UML Notation:

Register

1. Login: This use case is started by the customer. It allows the customer to login to website.

UML Notation:

Login

1. Browse: This use case is started by the customer. It allows to browse for the products.

UML Notation:

Browse

1. Select items/ Add to card: This use case is started by the customer to select the items and add to his/ her shopping cart.

UML Notation:

Select items/ Add to card

1. Delete Items from cart: This use case is started by the customer to delete the items from the shopping cart.

UML Notation:

Delete Items from cart

1. Check Out: This use case is started by the customer to checkout from shopping.

UML Notation:

Check Out

1. Update profile: This use case is started by the customer and the system prompts the customer to make changes to profile if necessary.

UML Notation:

Update profile

1. Logout: This use case is started by the customer. It allows the customer to logout from the website.

UML Notation:

Logout

1. Maintain product catalog: This use case is started by administrator. It provides the capability for the administrator to maintain changes in the product information.

UML Notation:

Maintain product catalog

1. Generate Order ID / Bill: This use case is started by the administrator and generates on order ID and bill to the customer for the purchased products.

UML Notation:

Generate Order ID / Bill

1. Sends order to Courier Company: This use case is started by the administrator and it sends order to Courier Company.

UML Notation:

Sends order to Courier Company

**4. FLOW OF EVENTS**

Flow of events document is typically created in the elaboration phase each use case is documented with *flow of events*.

* A description of events needed to accomplish required behavior.
* Written in terms of what the system should do. Not how it should do it.
* Written in the domain languages, not in terms of the implementation.

Flow of events should include

* When and how the use case starts and ends.
* What interaction the use case has with the actors.
* What data is needed by the use case
* The normal sequence of events for the use case.
* The description of any alternate or exceptional flows.

The flow of events for a use case is contained in a document called the use case specification. Each project should use a standard template for the creation of the use case specification. Include the following:

1. Use case name, brief description
2. Flow of events
3. Basic flow.
4. Alternative flow.
5. Special requirements.
6. Pre conditions.
7. Post conditions.
8. Extension points.

|  |
| --- |
| **Use case specification for Register for portal:** |
| 1. *Use case name: -* *Register for portal.*   *Brief Description: -* This is use case is started by the customer. It provides the capability for the customer o register for website. |
| 1. *Flow of events:*    1. *Basic flow-* 2. Customer opens the system/website. 3. Customer clicks new user/register for. 4. System displays the registration form. 5. Customer enters the name, age, gender, address, credit and details. 6. Customer submits the form. 7. System registers the customer.    1. *Alternative flow-*   Registration has failed: - The customer can re-register or terminate the use case. |
| 1. *Special Requirements: -* There are no special requirements for this use case. |
| 1. *Pre Condition: -* The registration form should be provided to the user properly. |
| 1. *Post Condition: -* There are no Post Conditions. |
| 1. *Extension points:-* There is no extension points |

|  |
| --- |
| **Use case specification for Customer Login:** |
| 1. *Use case name- Customer Login.*   *Brief Description-* This is started by the customer. It allows the customer to login to website. |
| 1. *Flow of Events-*    1. *Basic Flow:-*       * 1. Customer opens the system/website.         2. The system prompts the customer to enter username and password.         3. Customer fills details and clicks login.         4. The system verifies whether customer is valid.         5. System allows customer to login.    2. *Alternative Flow: -* Login has failed and the customer is prompted to re-enter the password or terminate the use case. |
| 1. *Special Requirements*: - There are no special requirements for this use case. |
| 1. *Pre Conditions-* The customer should register for the website earlier. |
| 1. *Post Conditions: -* The customer is allowed to browse that particular website. |
| 1. *Extension points:-* There is no extension points |

|  |
| --- |
| **Use case specification for browse for products:** |
| 1. *Use case name- Browse for Products.*   *Brief Description: -* This use case is started by the customer and it allows to browse for the products. |
| 1. *Flow of events:*    1. *Basic Flow:-* 2. Customer clicks the browse option. 3. The system displays several categories to browse. 4. The customer selects option of his/her choice. 5. The system displays various items in that particular category.    1. *Alternative Flow:* Customer switches to the other category if he is not satisfied with that category. |
| 1. *Special Requirements: -* There are no special requirements for this use case. |
| 1. *Pre Conditions: -* This maintains products catalog use case is executed before this use case. |
| 1. *Post Conditions: -* Displays the search results for the customers. |
| 1. *Extension points:-* There is no extension points |

|  |
| --- |
| **Use case specification for select items/ add to cart:** |
| 1. *Use case name- select Items/ Add to cart.*   *Brief Description-* This use case is started by the customer to select the items and add to his /her shopping cart. |
| 1. *Flow of events:-*    1. *Basic Flow:* 2. Customer search products of his/ her interest in selected category. 3. Customer selects the products by enabling the checkbox. 4. The selected products are added to cart by clicking ADD TO CART button.    1. Alternative Flow: - Customer switches to other category if he is not satisfied. |
| 1. *Special Requirements -* There are no special requirements for this use case. |
| 1. *Pre Condition: -* The product view and price should be displayed to the customer. |
| 1. *Post Condition: -* System displays the shopping cart. |
| 1. *Extension points:-* There is no extension points |
| **Use case specification for the delete items from cart:** | |
| 1. *Use case name-* *Delete Items from cart.*   *Brief Description-* This use case is started by the customer to delete the items from shopping cart. | |
| 1. *Flow of events:-*    1. *Basic Flow:* 2. Customer search products by enabling the checkbox. 3. The selected products are deleted from cart.    1. *Alternative Flow: -* Customer is requested to delete the items from cart again or terminate use case. | |
| 1. *Special Requirements -* There are no special requirements for this use case. | |
| 1. *Pre Condition: -* select item/ add to cart use case is executed before this use case. | |
| 1. *Post Condition: -* System displays the shopping cart. | |
| 1. *Extension points:-* There is no extension points | |

|  |
| --- |
| **Use case specification for checkout:** |
| 1. *Use case name-* *Checkout.*  *Brief Description-* This use case is started by the customer, who allows to checkout from shopping. |
| 2. *Flow of events:-*  *2.1.Basic Flow:*   1. Customer clicks the checkbox button. 2. Administrator calculates bill and sends to customer.    1. *Alternative Flow: -* Customer prompted to check out properly. |
| 3. *Special Requirements -* There are no special requirements for this use case. |
| 4. *Pre Condition: -* Customer should finish the shopping. |
| 5. *Post Condition: -* The customer’s bill should be calculated. |
| 6.*Extension points:-* There is no extension points |

|  |
| --- |
| **Use case specification for update profile:** |
| 1. *Use case name-* *Update Profile.*   *Brief Description-* This use case is started by the customer and the system prompts the customer to make changes to profile if necessary. |
| 1. *Flow of events:-*    1. *Basic Flow:* 2. The system displays the profile form. 3. Customer makes changes for his desired column. 4. The system saves changes if customer wishes to do so. 5. The changed profile is updated.    1. *Alternative Flow: -* Customer can discard the form if he does not want to change. |
| 1. *Special Requirements -* There are no special requirements for this use case. |
| 1. *Pre Condition: -* The register for portal use case is executed before this use case. |
| 1. *Post Condition: -* System displays the updated profile to customer. |
| 1. *Extension points:-* There is no extension points |
| **Use case specification for logout:** | |
| 1. *Use case name-* *Logout.*   *Brief Description-* This use case is started by the customer. It allows customer to logout from website. | |
| 1. *Flow of events:-*    1. *Basic Flow:* 2. Customer clicks the logout button. 3. The system returns the customer to login form.    1. *Alternative Flow: -* Logout if failed. The customer is prompted to re-click the logout button. | |
| 1. *Special Requirements -* There are no special requirements for this use case. | |
| 1. *Pre Condition: -* The login use case must be executed before this use case. | |
| 1. *Post Condition: -* System displays Login form. | |
| 1. *Extension points:-* There is no extension points | |

|  |
| --- |
| **Use case Specification for send order to courier company: -** |
| 1. *Use case name: -* *Send Order to Courier Company*   *Brief Description: -* This use case is started by the administrator and it sends order to Courier Company. |
| 1. *Flow of Events:*    1. *Basic Flow: -* 2. The administrator sends order to Courier Company. 3. The Courier Company saves ordered and delivers to that particular customer.    1. *Alternative Flow: -* If the transaction is failed then it is retransmitted to the customer. |
| 1. *Special Requirements -* There are no special requirements for this use case. |
| 1. *Pre Conditions: -* Paying bill use case must be executed before this use case. |
| 1. *Post Conditions: -* The Courier Company should send order to customer. |
| 1. *Extension points:-* There is no extension points |

|  |
| --- |
| **Use case Specification for Maintain Product Catalog:-** |
| 1. *Use case Name: -* *Maintain Product Catalog.*   Brief Description: - This use case is started by administrator. It provides the capability for the administrator to maintain changes in the product information. |
| 1. *Flow of Events: -*    1. *Basic Flow: -* 2. The system prompts to selects the desired activity like ADD, DELETE, and VIEW, QUIT. 3. The activity selected is ADD, then the system shows the products form and he can add new products to database by filling details. 4. The activity selected is DELETE, then the system shows the product id that has to delete from data base. 5. The activity selected is VIEW, and then the system shows the details of the selected products. 6. The activity selected is QUIT, and then use case ends.    1. *Alternative Flow: -* 7. If entered details of new products are not added then system display option of not available at this time. 8. If selected product is not deleted then system display option of not available at this time. 9. If selected detailed products are not displayed, then the system display information not available at this time. |
| 1. *Special Requirements -* There are no special requirements for this use case. |
| 1. *Pre Conditions: -* Database must be maintained for products. |
| 1. *Post Conditions: -* System displays the update database. |
| 1. *Extension points:-* There is no extension points |

|  |
| --- |
| **Use case specification for generate order ID/ Bill: -** |
| 1. *Use case name: -* *Generate Order Id/ Bills.*   *Brief Description: -* This use case is started by the administrator and this generates an order ID and bill to the customer for the purchased products. |
| 1. *Flow of Events: -*    1. *Basic Flow: -* 2. The system displays the selected products to the customer. 3. An ordered is generated for the products. 4. The system calculates the bill for the purchased products.    1. *Alternative Flow: -* If the false combination is selected then it informs the user that combination is false. |
| 1. *Special Requirements -* There are no special requirements for this use case. |
| 1. *Pre Conditions: -* Select Item/ Add to cart use case must be executed before this use case. |
| 1. *Post Conditions: -* The generated bill is forwarded to customer. |
| 1. *Extension points:-* There is no extension points |

|  |
| --- |
| **Use case Specification for send order to courier company: -** |
| 1. *Use case name: -* *Send Order to Courier Company*   *Brief Description: -* This use case is started by the administrator and it sends order to Courier Company. |
| 1. *Flow of Events:*    1. *Basic Flow: -*       * 1. The administrator sends order to Courier Company.         2. The Courier Company saves ordered and delivers to that particular customer.    2. *Alternative Flow: -* If the transaction is failed then it is retransmitted to the customer. |
| 1. *Special Requirements -* There are no special requirements for this use case. |
| 1. *Pre Conditions: -* Paying bill use case must be executed before this use case. |
| 1. *Post Conditions: -* The Courier Company should send order to customer. |
| 1. *Extension points:-* There is no extension points |

**5. USE-CASE DIAGRAM**

**Definition:** A use-case diagram graphically represents system behavior (use cases). These diagrams presents a high level view of how the system is used as viewed from an outsider’s (actor’s) perspective. A use-case diagram may contain all or some of the use cases of a system.

**Association Relationship:** An association provides a pathway for communication. The communication can be between use cases, actors, classes or interfaces. Associations are the most general of all relationships and consequentially the most semantically weak. If two objects are usually considered independently, the relationship is an association. Associations are of two types:

Uni-directional association.

Bi-directional association.

Graphical Depiction: An association relationship is an orthogonal or straight solid line with an arrow at one end.

In an association relationship, we can provide stereotype COMMUNICATE also as shown below



**Dependency Relationship:** A dependency is a relationship between two model elements in which a change to one model element will affect the other model element. Use a dependency relationship to connect model elements with the same level of meaning. Typically, on class diagram, a dependency relation indicates that the operation of the client invoke operation of the supplier.

We can provide here

1. Include relationship.
2. Extend relationship.
3. *Include relationship:* Multiple use cases may share pieces of the same functionality. This functionality is placed in a separate use case rather than documenting it in every use case that needs it.

Include relationships are created between the new use case and any other use case that “uses” its functionality.

An include relationship is a stereotyped relationship that connects a base use case to an inclusion use case. An include relationship specific how behavior in the inclusion use case is used by the base use case.

1. *Extend Relationship:* An extend relationship is a stereotyped relationship that specifies how the functionality of one use case can be inserted into the functionality of another use case. Extend relationship between use cases are modeled as dependencies by using the extend stereotype.

An extend relationship is used to show

* + Optional behavior.
  + Behavior that is run only under certain condition such as triggering an alarm.
  + Several different flows that may be run based on actor selection.
  + An extend relationship is drawn as a dependency relationship that points from the extension to the base use case.

The extend relationship sample demonstrates how you can use an extend relationship to connect use cases. The sample illustrates two important aspects of extend relationships:

* An extend relationship shows optional functionality or system behavior.
* A base use case does not need to acknowledge any specific extended use cases. The same below shows a diagram containing an actor interacting with a web site. The customer has the option of buying merchandise online as shown through the extend relationship.

Finally we can conclude:

* Extend is used when you wish to show that a use case provide additional functionality that may be required in another use case.
* Include applies when there is a sequence of behavior that is used frequently in a number of use cases, and you want to avoid copying the same description of it into each use case in which it is used.

**Use case Diagram for Customer:**

****

**Use case Diagram for Credit card company:**

****

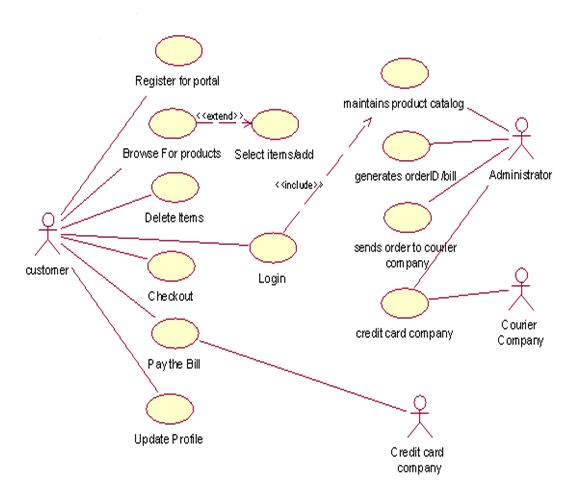
**Use case Diagram for Courier company:**

****

**Use case Diagram for Administrator:**

****

**Use case Diagram for Online Shopping:**



**6. ACTIVITY DIAGRAM**

An activity diagram is a variation of a special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. The purpose of activity diagram is to provide a view of flows and what is going on inside a use case or among several classes. We can also use activity diagrams to model code-specific information such as a class operation. Activity diagram are very similar to a flowchart because you can model a work flow from activity to activity.

An activity diagram is basically a special case of a state machine in which most of the states are activities and most of the transitions are implicitly triggered by completion of the actions in the source activities.

* Activity diagram represents the dynamics of the system.
* They are flow charts that are used to show the workflow of a system; that is, they show the flow of control from activity to activity in the system, what activities can be done in parallel, and any alternate paths through the flow.
* At this point in the life cycle, activity diagrams may be created to represent the flow across use case or they may be created to represent the flow within a particular use case.
* Later in the life cycle, activity diagrams may be created to show the workflow for an operation.

Activity diagram contains activities, transitions between the activities, decision points, and synchronization bars. In the UML, activities are represented as rectangles with rounded edges, transitions are drawn as directed arrows, decision points are shown as diamond, and synchronization bars are drawn as thick horizontal or vertical bars as shown in the following:

  Transition

Activity Decision point

Start state End state

Horizontal synchronization bar

Vertical synchronization bar

Activities: An activity represents the performance of some behavior in the workflow.

Transitions: Transitions are used to show the passing of the flow of control from activity to activity. They are typically triggered by the completion of the behavior in the originating activity.

Decision Points:

When modeling the workflow of a system it is often necessary to show where the flow of control branches based on a decision point. The transitions from a decision point contain a guard condition, which is used to determine which path from the decision point is taken. Decisions along with their guard conditions allow you to show alternate paths through a work flow.

Synchronization Bars

In a workflow there are typically some activities that may be done in parallel. A synchronization bar allows *you* to specify what activities may be done concurrently. Synchronization bars are also used to show joins in the workflow; that is, what activities must complete before processing may continue. Means, a synchronization bar may have many incoming transitions and one outgoing transition, or one incoming transition and many outgoing transitions.

Swim lanes

Swim lanes may be used to partition an activity diagram. This typically is done to show what person or organization is responsible for the activities contained in the swim lane.

Initial and Final Activities

There are special symbols that are used to show the starting and final activities in a workflow. The starting activity is shown using a solid filled circle and the final activities are shown using a bull's eye. Typically, there is one starting activity for the workflow and there may be more than one ending activity (one for each alternate flow in the workflow).

Modeling a workflow in an activity diagram can be done several ways; however, the following steps present just one logical process:

1. Identify a workflow objective. Ask, "What needs to take place or happen by the end of the workflow? What needs to be accomplished?" For example, if your activity diagram models the workflow of ordering a book from an online bookstore, the goal of the entire workflow could be getting the book to the customer.

2. Decide the pre and post-conditions of the workflow through a start state and an end state. In most cases, activity diagrams have a flowchart structure so start and end states are used to designate the beginning and ending of the workflow. State and end states clarify the perimeter of the workflow.

3. Define and recognize all activities and states that must take place to meet your objective. Place and name them on the activity diagram in a logical order.

4. Define and diagram any objects that are created or modified within your activity diagram. Connect the objects and activities with object flows.

5. Decide who or what is responsible for performing the activities and states through swim lanes. Name each swim lane and place the appropriate activities and states within each swim lane.

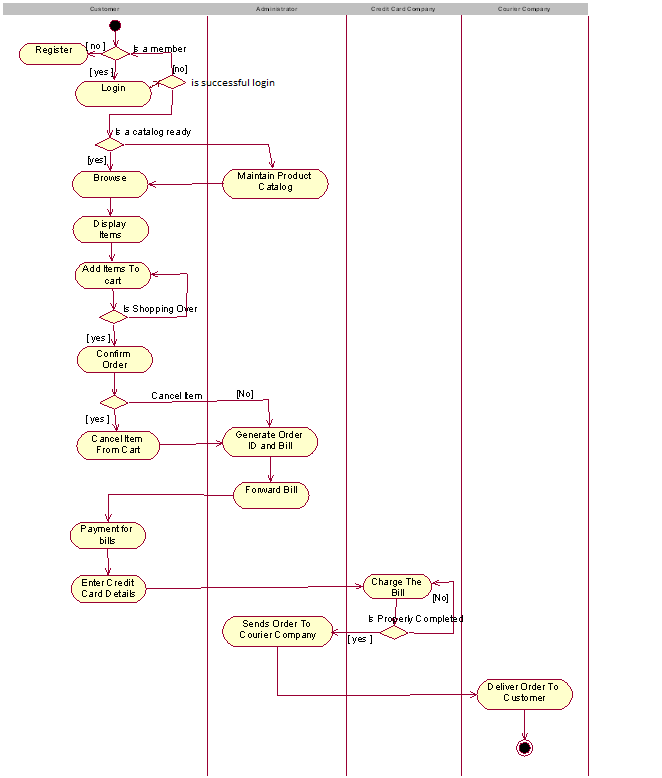
6. Connect all elements on the diagram with transitions. Begin with the "main" workflow.

7. Place decisions on the diagram where the workflow may split into an alternate flow. For example, based on a Boolean expression, the workflow could branch to a different workflow.

8. Evaluate your diagram and see if you have any concurrent workflows. If so, use synchronizations to represent forking and joining.

9. Set all actions, triggers and guard conditions in the specifications of each model element.

**Activity Diagram for Online Shopping:**

****

**Logical view**

**7. IDENTIFICATION OF ANALYSIS CLASSES**

For identification of analysis classes we have three approaches, they are :

1) Noun phrase approach

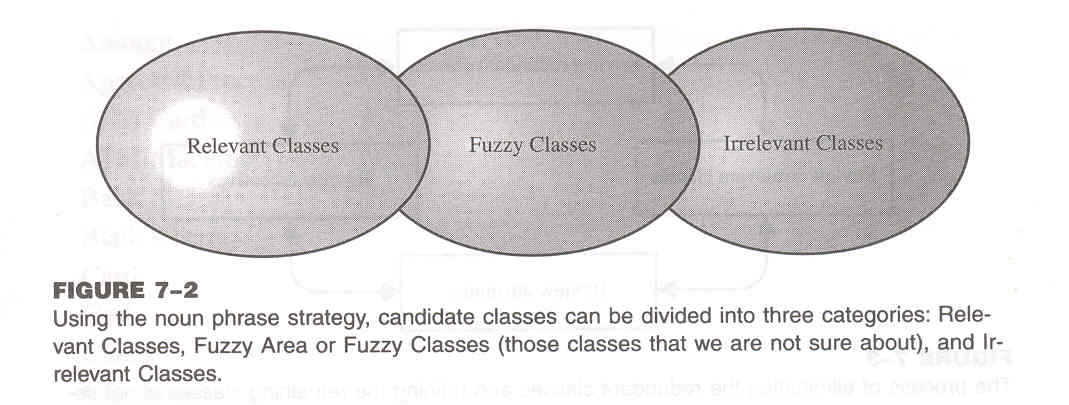
2) Common class pattern approach

3) Use case driven approach

In our application we used Use case driven approach for identifying analysis classes.

**1. NOUN PHRASE APPROACH.**

In this method, analysts read through the requirements or use cases looking for noun phrases. Nouns in the textual description are considered to be classes and verbs to be methods of the classes All plurals are changed to singular, the nouns are listed, and the list divided into three categories relevant classes, fuzzy classes (the "fuzzy area," classes we are not sure about), and irrelevant classes as shown below.



It is safe to scrap the irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories. Here identifying classes and developing a UML class diagram just like other activities is an iterative process. Depending on whether such object modeling is for the analysis or design phase of development, some classes may need to be added or removed from the model .Analyst must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

**1. Identifying Tentative Classes:**

The following are guidelines for selecting classes in an application:

* Look for nouns and noun phrases in the use cases.
* Some classes are implicit or taken from general knowledge.
* All classes must make sense in the application domain; avoid computer implementation classes-defer them to the design stage.
* Carefully choose and define class names.

Identifying classes is an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces, database standards, and even fourth-generation languages.

**2. Selecting Classes from the Relevant and Fuzzy Categories:**

The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

a)Redundant classes.

Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole. Choose your vocabulary carefully; use the word that is being used by the user of the system.

E.g.: Registrar, University I/C

b) Adjectives classes.

"Be wary of the use of adjectives. Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, and then makes a new class".

For example : Single account holders behave differently than Joint account holders, so the two should be classified as different classes.

c) Attribute classes: Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Details of Client are not classes but attributes of the Client class.

d) Irrelevant classes: Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

As this is an incremental process. Some classes will be missing; others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably are missing more classes than you will eliminate. Although some classes ultimately may become super classes, at this stage simply identify them as individual, specific classes. Your design will go through many stages on its way to completion, and you will have adequate opportunity to revise it.

This refining cycle through the development process until you are satisfied with the results. Remember that this process (of eliminating redundant classes, classes containing adjectives, possible attributes, and irrelevant classes) is not sequential. You can move back and forth among these steps as often analysts likes.

**2. COMMON CLASS PATTERNS APPROACH**

The second method for identifying classes is using common class patterns, which is based on a knowledge base of the common classes.

The following patterns are used for finding the candidate class and object:

a) Concept class:

A concept is a particular idea or understanding that we have of our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communications.

E.g. Performance is an example of concept class object.

b) Events class

Events classes are points in time that must be recorded. Things happen, usually to something else at a given date and time or as a step in an ordered sequence. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why.

E.g.: Landing, interrupt, request, and order are possible events.

c) Organization class

An organization class is a collection of people, resources, facilities, or groups to which the users belong; their capabilities have a defined mission, whose existence is largely independent of the individuals.

E.g.: An accounting department might be considered a potential class.

d) People class (also known as person, roles, and roles played class)

The people class represents the different roles users play in interacting with the application.

E.g. Employee, client, teacher, and manager are examples of people.

e) Places class

Places are physical locations that the system must keep information about.

E.g.: Buildings, stores, sites, and offices are examples of places.

**3. USE-CASE DRIVEN APPROACH**

One of the first steps in creating a class diagram is to derive from a use case, via a collaboration (or collaboration diagram), those classes that participate in realizing the use case. Through further analysis, a class diagram is developed for each use case and the various use case class diagrams are then usually assembled into a larger analysis class diagram. This can be drawn first for a single subsystem or increment, but class diagrams can be drawn at any scale that is appro­priate, from a single use case instance to a large, complex system.

Identifying the objects involved in collaboration can be difficult at first, and takes some practice before the analyst can feel really comfortable with the process. Here collaboration (i.e. the set of classes that it comprises) can be identified directly for a use case, and that, once the classes are known, the next step is to consider the interaction among the classes and so build a collaboration diagram.

Consider the following use case **placing an order** to identify a set of classes:

Use case description: Placing an order

Actor Action

1. Log-in to the system.

2. Enter book details.

4. Go through with abstract.

5. Add select books to shopping cart

6. Place the order.

System Response

3. Displays the list of books.

7. Verify the list of books for the availability.

8. Generate and forward the order number.

Here objective is to find a set of classes that can interact to realize the use case. Inthis case, we know from the use case diagram that the user and administrator are the actors for this use case. The use case description tells us that the user logs on to the system and enter book details. The system displays the list of books. Customer or user selects the desired books and places an order. The Administrator verifies the order and generates the order number. Finally customer receives his order number. The objective of the use case is to allow the user to perform place an order for books.

First begin by picking out from the description all the important things or concepts in the application domain. Our first list might include: Applicant, username, password and company. But we are only interested in those about which the system must store some information or knowledge in order to achieve its objectives. The applicant will be modeled as actor.

For the purposes of this particular use case, it is unlikely the system will need to encapsulate any further knowledge about the actor. Here we can eliminate company, since this is just part of the description of model. This leaves applicant, company and database in the collaboration.

Collaboration is between individual object instances, not between classes. This is shown in the diagram by the convention of writing a colon before the class name, which indicates that this is an anonymous instance of the class, rather than the class itself. Messages between classes are shown by arrows, and their sequence is indicated by the number alongside. Inthis example, these are not yet labeled, although some those that can be most easily related to the use case description will probably soon be given names that correspond to responsibilities of the class to which the message is addressed.

In the collaboration diagram does not yet show any boundary or control objects, and these must be added. It is also based on certain assumptions about how the interaction between objects would take place, and we must make these assumptions explicit and question them.

Although the messages are not yet labeled, the numbers alongside the message arrows indicates their sequence. The diagram implies a linear flow of messages, along the following lines.

An initial message is directed from customer to administrator since customer starts this use case by logging into the system, selects the books and places an order to Administrator. Then Administrator performs the operations and sends back the order number for confirmation.

**From collaboration diagram to class** **diagram**

The next step in the development of a requirements model is usually to produce a class diagram that corresponds to each of the collaboration diagrams. The class diagram that corresponds to the use case E-Pay is shown below.

Collaboration diagrams are obtained by result of reasonably careful analysis, the transition is not usually too difficult.

The similarities & differences between Collaboration and class diagrams are:

First, consider the similarities:

Both show class or object symbols joined by connecting lines. In general, a class diagram has more or less the same structure as the corresponding collaboration diagram. In particular, both should show classes or objects of the same types. Any of the three analysis stereotype notations for a class can be used on either diagram, and stereotype labels can also be omitted from individual classes, or from an entire diagram.

Next, the differences are:

1. The difference is that an actor is almost always shown on a collaboration diagram, but not usually shown on a class diagram. This is because the collaboration diagram represents a particular interaction and the actor is an important part of this interaction. However, a class diagram shows the more enduring structure of associations among the classes, and frequently supports a number of different interactions that may represent several different use cases.

2. A collaboration diagram usually contains only object instances, while a class diagram usually contains only classes.

3. The connections between the object symbols on a collaboration diagram symbolize links between objects, while on a class diagram the corresponding connections stand for associations between classes.

4. A collaboration diagram shows the dynamic interaction of a group of objects and thus every link needed for message passing is shown. The labeled arrows alongside the links represent messages between objects. On a class diagram, the associations themselves are usually labeled, but messages are not shown.

5. Finally, any of the three stereotype symbols can be used on either diagram; there are also differences in this notation.

When the rectangular box variant of the notation is used in a collaboration diagram it represents object instances rather than classes, is normally undivided and contains only the class name .On a class diagram, the symbol is usually divided into three compartments that contain in turn the class name, its attributes and its operations.

**8. IDENTIFICATION OF RESPONSIBILITIES OF CLASSES**

**Class Responsibility collaboration Cards (CRC Cards)**

At the starting, for the identification of classes we need to concentrate completely on uses cases. A further examination of the use cases also helps in identifying operations and the messages that classes need to exchange. However, it is easy to think first in terms of the overall responsibilities of a class rather than its individual operations.

A *responsibility* is a high level description of something a class can do. It reflects the knowledge or information that is available to that class, either stored within its own attributes or requested via collaboration with other classes, and also the services that it can offer to other objects. A responsibility may correspond to one or more operations. It is difficult to determine the appropriate responsibilities for each class as there may be many alternatives that all appear to be equally justified.

*Class Responsibility Collaboration (CRC)* cards provide an effective technique for exploring the possible ways of allocating responsibilities to classes and the collaborations that are necessary to fulfill the responsibilities.

CRC cards can be used at several different stages of a project for different purposes.

1. They can be used early in a project to help the production of an initial class diagram

2. To develop a shared understanding of user requirements among the members of the team.

3. CRCs are helpful in modeling object interaction.

**The format of a typical CRC card is shown below**

|  |  |
| --- | --- |
| Class Name: | |
| Responsibilities | Collaborations |
| *Responsibilities of a class are listed in this section* | *Collaborations with other*  *classes are listed here, together*  *with a brief description of the*  *purpose of the collaboration* |

CRC cards are an aid to a group role-playing activity. Index cards are used in preference to pieces of paper due to their robustness and to the limitations that their size (approx. 15cm x 8cm) imposes on the number of responsibilities and collaborations that can be effectively allocated to each class.

A class name is entered at the top of each card and responsibilities and collaborations are listed underneath as they become apparent. For the sake of clarity, each collaboration is normally listed next to the corresponding responsibility.

From a UML perspective, use of CRC cards is in analyzing the object interaction that is triggered by a particular use case scenario. The process of using CRC cards is usually structured as follows.

1. Conduct a session to identify which objects are involved in the use case.

2. Allocate each object to a team member who will play the role of that object.

3. Act out the use case: This involves a series of negotiations among the objects to explore how responsibility can be allocated and to identify how the objects can collaborate with each other.

4. Identify and record any missing or redundant objects.

Before beginning a CRC session it is important that all team members are briefed on the organization of the session and a CRC session should be preceded by a separate exercise that identifies all the classes for that part of the application to be analyzed.

The team members to whom these classes are allocated can then prepare for the role playing exercise by considering in advance a first-cut allocation of responsibilities and identification of collaborations. Here, it is important to ensure that the environment in which the sessions take place is free from interruptions and free for the flow of ideas among team members.

During a CRC card session, there must be an explicit strategy that helps to achieve an appropriate distribution of responsibilities among the classes. One simple but effective approach is to apply the rule that each object should be as lazy as possible, refusing to take on any additional responsibility unless instructed to do so by its fellow objects.

During a session conducted according to this rule, each role player identifies the object that they feel is the most appropriate to take on each responsibility, and attempts to persuade that object to accept the responsibility. For each responsibility that must be allocated, one object is eventually persuaded by the weight of rational argument to accept it. This process can help to highlight missing objects that are not explicitly referred to by the use case description. When responsibilities can be allocated in several different ways it is useful to role-play each allocation separately to determine which is the most appropriate. The aim normally is to minimize the number of messages that must be passed and their complexity, while also producing class definitions that are cohesive and well focused.

Consider the CRC exercise for the use case placing an order.

It consists of shop administrator, entity class, order UI boundary class and order control class.

CRC card for order UI Boundary class of the above use case:

|  |  |
| --- | --- |
| Class Name : Order UI | |
| Responsibilities | Collaborations |
| *Start The Interface*  *Take order details from the customer.* | *Sends the details to order control class* |

**9. USE CASE REALIZAION:**

Use case realization is nothing but an instance of a use case which involves the identification of a possible set of classes, together with an understanding of how those classes might interact to deliver the functionality of the use case. The set of classes is known as collaboration



**10. SEQUENCE DIAGRAM**

A sequence diagram is a graphical view of a scenario that shows object interaction in a time based sequence i.e. what happens first and what happens next.

Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces.

In Sequence diagram the vertical dimension represents time and all objects involved in the interaction are spread horizontally across the diagram.

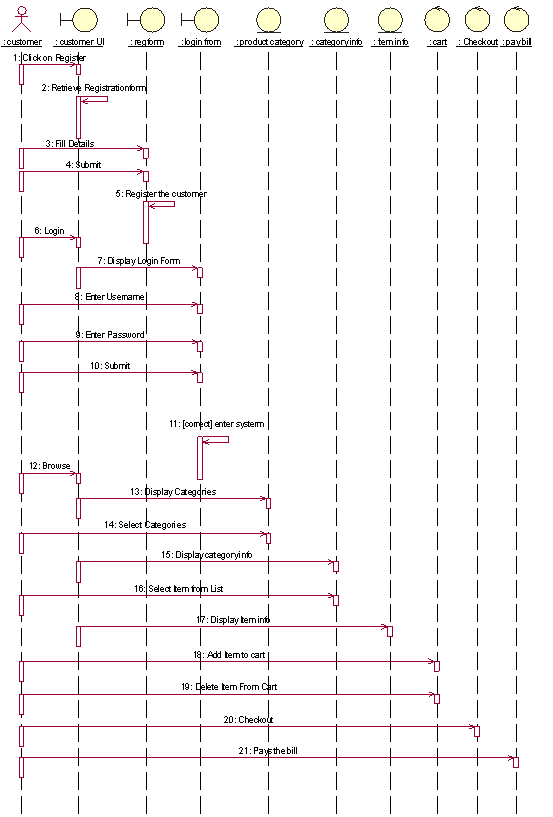
Time normally proceeds down the page. However, a sequence diagram may be drawn with a horizontal time axis if required, and in this case, time proceeds from left to right across the page. Each object is represented by a vertical dashed line, called a *lifeline,* with an object symbol at the top. A message is shown by a solid horizontal arrow from one lifeline to another and is labeled with the message name. Each message name may optionally be preceded by a sequence number that represents the sequence in which the messages are sent, but this is not usually necessary on a sequence diagram since the message sequence is already conveyed by their relative positions along time axis.

Steps:

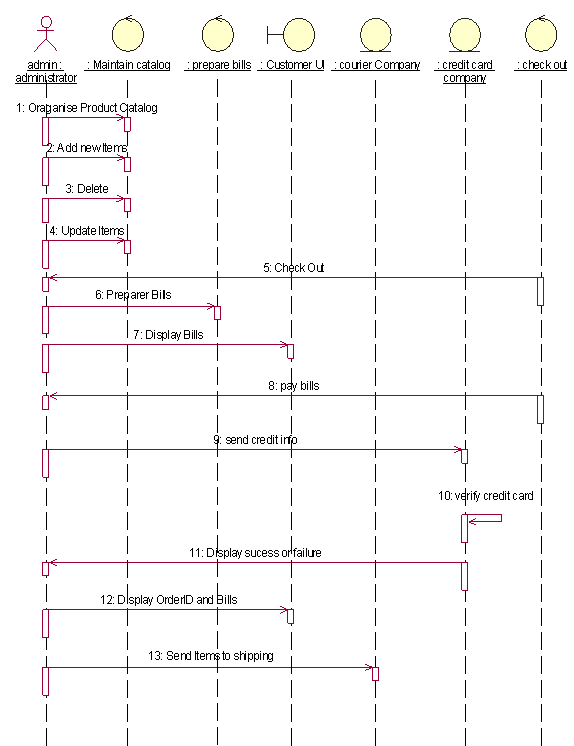
1. An object is shown as a box at the top of a dashed vertical line. Object names can be specific (e.g., Algebra 101, Section 1) or they can be general (e.g., a course offering). Often, an anonymous object (class name may be used to represent any object in the class.)

2. Each message is represented by an Arrow between the lifelines of two objects. The order in which these messages occur is shown top to bottom on the page. Each message is labeled with the message name.

**Sequence Diagram for “Customer”:-**

****

**Sequence Diagram for “Administrator”:-**

****

**11. COLLABORATION DIAGRAM**

A collaboration diagram is an alternate way to show a scenario. This type of diagram shows object interactions organized around the objects and their links to each other. A collaboration diagram contains:

* Objects drawn as rectangles
* Links between objects shown as lines connecting the linked objects
* Messages shown as text and an arrow that points from the client to the supplier

Message labels in collaboration diagrams:

Messages on a collaboration diagram are represented by a set of symbols that are the same as those used in a sequence diagram, but with some additional elements to show sequencing and recurrence as these cannot be inferred from the structure of the diagram. Each message label includes the message signature and also a sequence number that reflects call nesting, iteration, branching, concurrency and synchron­ization within the interaction.

The formal message label syntax is as follows:

[Predecessor] [guard-condition] sequence-expression [return-value ':='] message-name' (' [argument-list] ')'

A *predecessor* is a list of sequence numbers of the messages that must occur before the current message can be enabled. This permits the detailed specification of branching pathways. The message with the immediately preceding sequence number is assumed to be the predecessor by default, so if an interaction has no alternative pathways the predecessor list may be omitted without any ambiguity. The syntax for a predecessor is as follows:

Sequence-number { ',' sequence-number} *'I'*

The *'I'* at the end of this expression indicates the end of the list and is only included when an explicit predecessor is shown.

*Guard conditions* are written in Object Constraint Language (OCL), and are only shown where the enabling of a message is subject to the defined condition. A guard condition may be used to represent the synchronization of different threads of control.

A *sequence-expression is* a list of integers separated by dots ('.') optionally followed by a *name* (a single letter), optionally followed by a *recurrence* term and terminated by a colon. A sequence-expression has the following syntax:

Integer {‘.' integer} [name] [recurrence] ':'

In this expression *integer* represents the sequential order of the message. This may be nested within a loop or a branch construct, so that, for example, message 5.1 occurs after message 5.2 and both are contained within the activation of message 5.

The *name* of a sequence-expression is used to differentiate two concurrent messages since these are given the same sequence number. For example, messages 3.2.1a and 3.2.1b are concurrent within the activation of message 3.2. Recurrence reflects either iterative or conditional execution and its syntax is as follows:

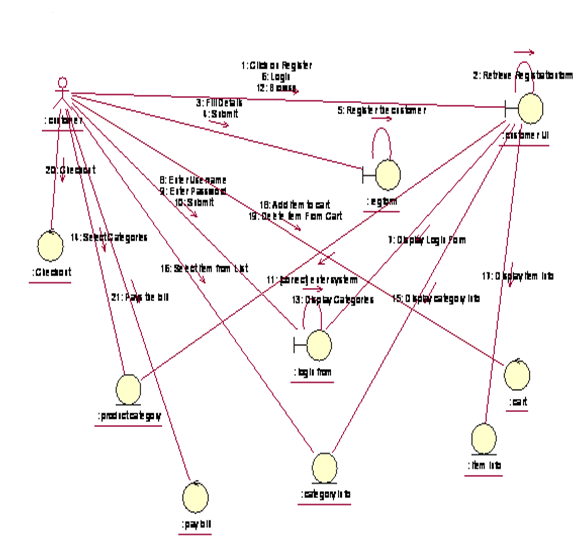
*Branching:* ‘[‘condition-clause‘ ],

*Iteration:* ‘\* ‘‘[‘iteration-clause ‘ ]’

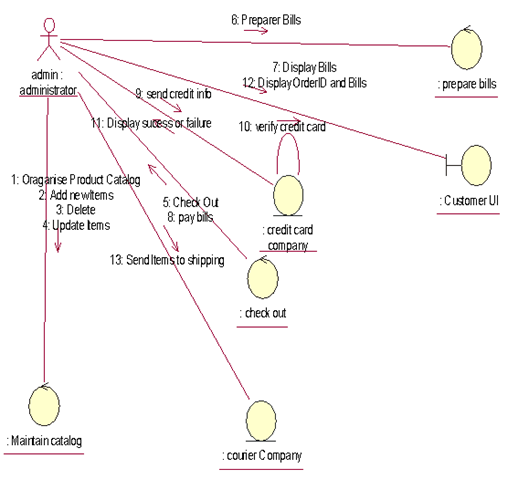
Difference between sequence and collaboration diagrams

* Sequence diagrams are closely related to collaboration diagrams and both are alternate representations of an interaction.
* Sequence diagrams show time-based object interaction while collaboration diagrams show how objects associate with each other.
* A sequence diagram is a graphical view of a scenario that shows object interaction in a time based sequence
* A collaboration diagram shows object interactions organized around the objects and their links to each other.

**Collaboration Diagram for “Customer”:**



**Collaboration Diagram for “administrator”:**

****

**12. IDENTIFICATION OF METHODS AND ATTRIBUTES OF CLASSES**

**Attributes**

Attributes are part of the essential description of a class. They belong to the class, unlike objects, which instantiate the class. Attributes are the common structure of what a member of the class can 'know'. Each object will have its own, possibly unique, value for each attribute.

**Operations**

Operations are the elements of common behavior shared by all instances of a class. They are actions that can be carried out by or carried on, an object. The classes modeled during requirements analysis represent real-world things and concepts, so their operations can be said to represent aspects of the behavior of the same things and concepts. Another way of defining operations, operations are services that objects may be asked to perform by other objects.

An operation is a specification for some aspect of the behavior of a class. Here Operations are eventually implemented by *methods,* and what a method actually does on any given occasion may be constrained by the value of object attributes and links when the method is invoked.

List of attributes that are identified in our application are:

* Username
* Password
* Product name
* Product ID
* Product cost
* No. of Products
* Customer ID
* Seller Id
* Card holder name
* Account number
* Account type
* Paid amount

Responsibilities that are identified for each class in our application:

**Client**: This is an entity class. The clients first browse the products to view the product details. He selects the desired products and submits those products to the administrator.

He views the bill and pay the bill. He views the shopping cart and tracks the status.

**Administrator:** This is an entity class. when the customer or seller register into the site, he maintains their information. When seller submits the product details, he maintains those product details and displays the product information to the customer. When the customer submitted the selected products, he will generate the bill and forward the bill to the customer. After the customer paid the bill, he adds those products to shopping cart of the customer. He maintains those completed shopping details.

**Seller**: It is an entity class. He submits the products to

**Billing System:** This is an entity class, when the customer pays the bill, he verifies the credit card details and performs the amount transactions.

**Customer UI**: It is a boundary class. It displays the fields for the customer to perform shopping.

**Shipping clerk**: He view the updated inventory and maintains the delivery information. He delivers the products to customer.

**13. IDENTIFICATION OF RELATIONSHIPS AMONG CLASSES**

**NEED FOR RELATIONSHIPS AMONG CLASSES:**

All systems are made up of many classes and objects. System behavior is achieved through the collaborations of the objects in the system. For example, a passenger can perform reservation operation by submitting form to reservation clerk. This is often referred to as an object sending a message to another object. Relationships provide the medium or tool for object interaction.

Two types of relationships in CLASS diagram are:

1. Associations Relationship 2. Aggregations Relationship

**1. ASSOCIATION RELATIONSHIPS**:

An association is a bidirectional semantic connection between classes. It is not a data flow as defined in structured analysis and design data may flow in either direction across the association. An association between classes means that there is a link between objects in the associated classes.

For example, an association between the Billing System class and the Reservation class means that objects in the Billing System class are connected to objects in the Reservation class.

Association Relationship without Multiplicity

****

**2. AGGREGATION RELATIONSHIPS:**

An aggregation relationship is a specialized form of association in which a whole is related to its part(s).

Aggregation is known as a "part-of" or containment relationship. The UML notation for an aggregation relationship is an association with a diamond next to the class denoting the aggregate (whole), as shown below:



*NAMING RELATIONSHIPS:*

An association may be named. Usually the name is an active verb or verb phrase that communicates the meaning of the relationship. Since the verb phrase typically implies a reading direction, it is desirable to name the association so it reads correctly from left to right or top to bottom. The words may have to be changed to read the association in the other direction (e.g., Buses are allotted to Routes). It is important to note that the name of the association is optional.

*ROLE NAMES:*

The end of an association where it connects to a class is called an association role. Role names can be used instead of association names.

A role name is a noun that denotes how one class associates with another. The role name is placed on the association near the class that it modifies, and may be placed on one or both ends of an association line.

* It is not necessary to have both a role name and an association name.
* Associations are named or role names are used only when the names are needed for clarity.

Product details

admin

*+product id*

MULTIPLICITY INDICATORS:

Although multiplicity is specified for classes, it defines the number of objects that participate in a relationship. Multiplicity defines the number of objects that are linked to one another. There are two multiplicity indicators for each association or aggregation one at each end of the line. Some common multiplicity indicators are

1 Exactly one

0 .. \* Zero or more

1 .. \* One or more

0 .. 1 Zero or one

5 .. 8 Specific range (5, 6, 7, or 8)

4 .. 7,9 Combination (4, 5, 6, 7, or 9)

****

REFLEXIVE RELATIONSHIPS:

Multiple objects belonging to the same class may have to communicate with one another. This is shown on the class diagram as a reflexive association or aggregation. Role names rather than association names typically are used for reflexive relationships.

0..n

0..n

Completed shopping details

Update inventory

seller

0..n

0..n

**14. UML CLASS DIAGRAM**

CLASS DIAGRAMS:

* Class diagrams are created to provide a picture or view of some or all of the classes in the model.
* The main class diagram in the logical view of the model is typically a picture of the packages in the' system. Each package also has its own main class diagram, which typically displays the "public" classes of the package.

A class diagram is a picture for describing generic descriptions of possible systems. Class diagrams and collaboration diagrams are alternate representations of object models.

Class diagrams contain icons representing classes, packages, interfaces, and their relationships. You can create one or more class diagrams to depict the classes at the top level of the current model; such class diagrams are themselves contained by the top level of the current model.

CLASS:

A class is a description of a group of objects with common properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. Thus, a class is a template to create objects. Each object is an instance of some class and objects cannot be instances of more than one class.

Classes should be named using the vocabulary of the domain. For example, the Bus class may be defined with the following characteristics:

Attributes - location, time offered

Operations - retrieve location, retrieve time of day, add a student to the offering.

Each object would have a value for the attributes and access to the operations specified by the Bus class.

UML REPRESENTATION:

* In the UML, classes are represented as compartmentalized rectangles.
* The top compartment contains the name of the class.
* The middle compartment contains the structure of the class (attributes).
* The bottom compartment contains the behavior of the class (operations) as shown below.

ANALYSIS CLASS STEROTYPES

Analysis class stereotypesrepresent three particular kinds of class that will be encountered again and again when carrying out requirements modeling.

UML DEFINITION:

Stereotype:

* A new type of modeling element that extends the semantics of the metamodel.
* Stereotypes must be based on certain existing types or classes in the metamodel.
* Stereotypes may extend the semantics but not the structure of preexisting classes.
* Certain stereotypes are defined in the UML, others may be user defined.

UML is designed to be capable of extension; developers can add new a stereotype depends on need. But this is only done when it is absolutely necessary. Three analysis class stereotypes to the UML are:

* Boundary classes,
* Control classes
* Entity classes.

1. Boundary classes:

Boundary classes, it is a 'model interaction between the system and its actors’. Since they are part of the requirements model, boundary classes are relatively abstract. They do not directly represent all the different sorts of interface that will be used in the implementation language. The design model may well do this later, but from an analysis perspective we are interested only in identifying the main logical interfaces with users and other systems.

This may include interfaces with other software and also with physical devices such as printers, motors and sensors. Stereotyping these as boundary classes emphasizes that their main task is to manage the transfer of information across system boundaries. It also helps to partition the system, so that any changes to the interface or communication aspects of the system can be isolated from those parts of the system that provide the information storage.

The class AddANewtrainUI is a typical boundary class. This style of writing the name shows that the class is AddNewtrainUI and it belongs to the User Interface package when we write the package name in this way before the class name, it means that this class is imported from a different package from the one with which we are currently working. In this case, the current package is the Agate application package, which contains the application requirements model, and thus consists only of domain objects and classes.

Alternative notations for Boundary class stereotype can be represented as shown below:

1. With stereotype

****

b) symbol

****

2. Entity classes

The second analysis class stereotype is the entity class, which is given in the class diagram of Allotting train to Routes by the two classes train and Route. Entity classes are used to model 'information and associated behavior of some phenomenon or concept such as an individual, a real-life object, or a real-life event’. As a general rule, entity classes represent something within the application domain, but external to the software system, about which the system must store some information.

Instances of an entity class will often require persistent storage of information about the things that they represent. This can sometimes help to decide whether an entity class is the appropriate modeling construct.

For example, an actor is often not represented as an entity class. This is in spite of the fact that all actors are within the application domain, external to the software system and important to its operation. But most systems have no need to store information about their users and to model their behavior.

While there are some obvious exceptions to this (consider a system that monitors user access for security purposes), these are typically separate, specialist applications in their own right. In such a context, an actor would be modeled appropriately as an entity class, since the essential requirements for such a system would include storing information about users, monitoring their access to computer systems and tracking their actions while logged on to a network. But it is more commonly the case that the software we develop does not need to know anything about the people that use it, and so actors are not normally modeled as classes. The following are representations for Entity classes.

a) With stereotype



b. symbol



3. Control classes

The third of the analysis class stereotypes is the control class, given by y the class Allot train in Allotting train to Routes.

Control classes 'represent coordination, sequencing, transactions and control of other objects' .In the USDP, as in the earlier methodology Objectory, it is generally recommended that there should be a control class for each use case.

In a sense, then, the control class represents the calculation and scheduling aspects of the logic of the use case at any rate, those parts that are not specific to the behavior of a particular entity class, and that *are* specific to the use case. Meanwhile the boundary class represents interaction with the user and the entity classes represent the behavior of things in the application domain and storage of information that is directly associated with those things. The following are the notations can be used to represent Control class

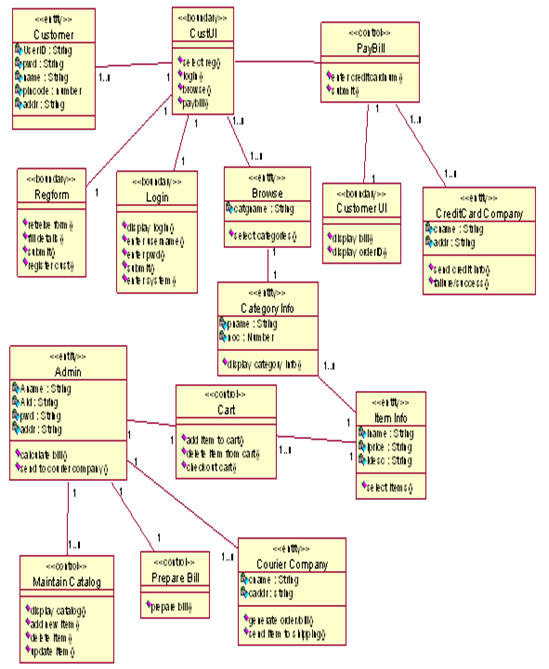
a.With stereotype

****

b. symbol

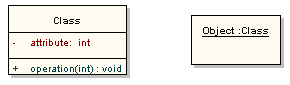
Add a customer

**Class Diagram for Online Shopping:**

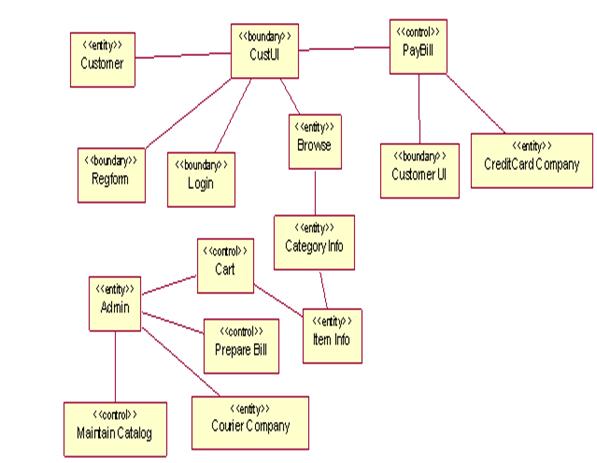


**15. OBJECT DIAGRAMS**   
 An object diagram may be considered a special case of a class diagram. Object diagrams use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes at some point in time. They are useful in understanding class diagrams. They don’t show anything architecturally different to class diagrams, but reflect multiplicity and roles.

Class and Object Elements:  
The following diagram shows the differences in appearance between a class element and an object element. Note that the class element consists of three parts, being divided into name, attribute and operation compartments; by default, object elements don’t have compartments. The display of names is also different: object names are underlined and may show the name of the classifier from which the object is instantiated.



**Object Diagram for online shopping:**

****

**16. UML STATE CHART DIAGRAM**

Use cases and scenarios provide a way to describe system behavior in the form of interaction between objects in the system. Sometimes it is necessary to consider inside behavior of an object.

A state chart diagram shows the states of a single object, the events or messages that cause a transition from one state to another and the actions that result from a state change. As in Activity diagram, state chart diagram also contains special symbols for start state and stop state.

State chart diagram cannot be created for every class in the system; it is only for those class objects with significant behavior.

STATE:

A state represents a condition or situation during the life of an object during which it satisfies some condition, performs some action or waits for some event.

UML notation for STATE is



To identify the states for an object its better to concentrate on sequence diagram.

In an Online Shopping application the object for Course Offering may have in the following states, initialization, open and closed state. These states are obtained from the attribute and links defined for the object. Each state also contains a compartment for actions.

ACTIONS:

Actions on states can occur at one of four times:

* on entry
* on exit
* do
* on event.

On entry: What type of action that object has to perform after entering into the state.

On exit : What type of action that object has to perform after exiting from the state.

Do : The task to be performed when object is in this state, and must to continue until it leaves the state.

On event : An on event action is similar to a state transition label with the following

syntax:

event(args)[condition] : the Action

STATE TRANSITION:

A state transition indicates that an object in the source state will perform certain specified actions and enter the destination state when a specified event occurs or when certain conditions are satisfied. A state transition is a relationship between two states, two activities, or between an activity and a state.

We can show one or more state transitions from a state as long as each transition is unique. Transitions originating from a state cannot have the same event, unless there are conditions on the event.

Graphical Representation

Active

Repair

Provide a label for each state transition with the name of at least one event that causes the state transition. You do not have to use unique labels for state transitions because the same event can cause a transition to many different states or activities.

Transitions are labeled with the following syntax:

event (arguments) [condition] / action ^ target.sendEvent (arguments)

Only one event is allowed per transition, and one action per event.

STATE DETAILS:

Actions that accompany all state transitions into a state may be placed as an entry action within the state. Likewise that accompany all state transitions out of a state may be placed as exit actions within the state. Behavior that occurs within the state is called an activity.

An activity starts when the state is entered and either completes or is interrupted by an outgoing state transition. The behavior may be a simple action or it may be an event sent to another object.

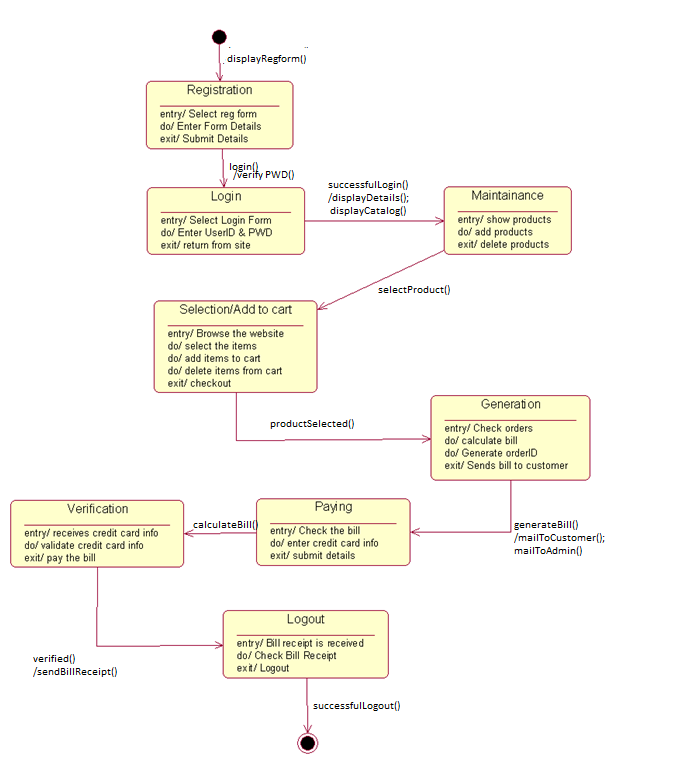
UML notation for State Details



Difference between activity diagrams and state chart diagrams

State chart diagrams are closely related to activity diagrams. The main difference between the two diagrams is state chart diagrams are state centric, while activity diagrams are activity centric. A state chart diagram is typically used to model the discrete stages of an object’s lifetime, whereas an activity diagram is better suited to model the sequence of activities in a process.

State chart Diagram For Online Shopping:

****

**DESIGN**

**17. DESIGNING CLASSES BY APPLYING DESIGN AXIOMS**

**Criteria for Good Design**

**Coupling and cohesion**

The factors coupling and cohesion are important factors for good design.

Coupling describes the degree of interconnectedness between design components and is reflected by the number of links an object has and by the degree of interaction the object has with other objects.

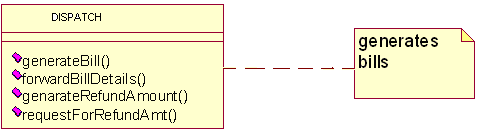
Cohesion is a measure of the degree to which an element contributes to a single purpose. The concepts of coupling and cohesion are not mutually exclusive but actually support each other. This criterion can be used within object-orientation as described below.

***Interaction Coupling*** *is* a measure of the number of message types an object sends to other objects and the number of parameters passed with these message types. Interaction coupling should be kept to a minimum to reduce the possibility of changes rippling through the interfaces and to make reuse easier. When an object is reused in another application it will still need to send these messages and hence needs objects in the new application that provide these services. This complicates the reuse process as it requires groups of classes to be reused rather than individual classes.

***Inheritance Coupling***describes the degree to which a subclass actually needs the features it inherits from its base class.

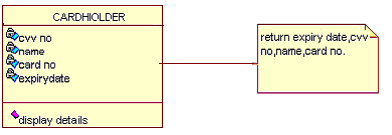
*Operation Cohesion*

It *measures* the degree to which an operation focuses on a single functional requirement. Good design produces highly cohesive operations, each of which deals with a single functional requirement. For example in the following figure , the operation genarateBills() is highly cohesive.



*Class Cohesion*

It reflects the degree to which a class is focused on a single requirement. The class Lecturer in the figure below exhibits low levels of cohesion as it has three attributes (roomNumber, roomLength and roomWidth and one operation calculate RoomSpace () ) that would be more appropriate in a class Room. The class Lecturer should only have attributes that describe a Lecturer object (e.g. lecturerName and lecturerAddress) and operations that use them.

**

*Specialization Cohesion*

It addresses the semantic cohesion of inheritance hierarchies. For example in the following figure all the attributes and operations of the Address base class are used by the m derived classes - this hierarchy has high inheritance coupling. However, it is neither true that a person is a kind of address nor that a company is a kind of address. The example is only using inheritance as a syntactic structure for sharing attributes and operations. This structure has low specialization cohesion and is poor design. It does not reflect meaningful inheritance in the problem domain.

**Further design guidelines**

The following are the further guidelines which are to be considered for good design.

*Design Clarity.*

A design should be made as easy to understand as possible. This reinforces the need to use design standards or protocols that have been specified.

*Don't Over-Design.*

Developers are on occasions tempted to produce designs that may not only satisfy current requirements but may also be capable of supporting a wide range of future requirements. Designing flexibility into a system has a cost, the system may take longer to design and construct but this may be offset in the future by easier and less expensive modification. However, it is not feasible to design for every eventuality. Systems that are over­designed in first instance are more difficult to extend if the modifications are not sympathetic to the existing structure.

*Control Inheritance Hierarchies.*

Inheritance hierarchies should be neither too deep nor too shallow. If a hierarchy is too deep it is difficult for the developer to understand easily what features are inherited. There is a tendency for developers new to 00 to produce over-specialized hierarchies, thus adding complexity rather than reducing it.

*Keep Messages and Operations Simple.*

In general it is better to limit the number of parameters passed in a message to no more than three .Ideally an operation should be capable of specification in no more than one page.

*Design Volatility.*

A good design will be stable in response to changes in requirements. It is reasonable to expect some change in the design if the requirements are changed. However, any change in the design should be commensurate with the change in requirements. Enforcing encapsulation is a key factor in producing stable systems.

*Evaluate by Scenario.*

An effective way of testing the suitability of a design is to role play it against the use cases using CRC cards.

*Design by Delegation*

A complex object should be decomposed (if possible) into component objects forming a composition or aggregation. Behaviour can then be delegated to the component objects producing a group of objects that are easier to construct and maintain. This approach also improves reusability.

*Keep Classes Separate.*

In general, it is better not to place one class inside another. The internal class is encapsulated by the other class and cannot be accessed independently. This reduces the flexibility of the system.

**18. REFINING ATTRIBUTES, METHODS AND RELATIONSHIPS**

Attributes

During analysis Stage we need to consider in detail the data types of the attributes also. Common primitive data types include Boolean (true or false), Character (any alphanumeric or special character), Integer (whole numbers) and Floating-Point (decimal numbers). In most object-oriented languages more complex data types, such as Money, String, Date, or Name can be constructed from the primitive data types or may be available in standard libraries. An attribute's data type is declared in UML using the following syntax:

name ':' type-expression '=' initial-value '{'property-string'}'

The name is the attribute name, the type-expression is its data type, the initial value is the value the attribute is set to when the object is first created and the property-string describes a property of the attribute, such as constant or fixed. The characters in single quotes are literals.

The following is a class Staff which is shown along with attribute data types declared.

Attribute declarations can also include arrays also. For example, an Employee class might include an attribute to hold a list of qualifications that would be declared using the syntax:

Qualification [O ... 10]: String

*Operations*

Each operation also has to be specified in terms of the parameters that it passes and returns. The syntax used for an operation is:

Operation name' ('parameter-list ') “: “return-type-expression

An operation's *signature* is determined by the operation's name, the number and type of its parameters and the type of the return value if any.

*Object visibility*

The concept of encapsulation is one of the fundamental principles of object-orientation. During analysis various assumptions have been made regarding the encapsulation boundary for an object and the way that objects interact with each other.

For example, it is assumed that the attributes of an object cannot be accessed directly by other objects but only via 'get' and 'set' operations (primary operations) that are assumed to be available for each attribute. Moving to design involves making decisions regarding which operations (and possibly attributes) are publicly accessible. In other words we must define the encapsulation boundary.

The following are the different kinds of visibilities, their symbols and their meaning.

|  |  |  |
| --- | --- | --- |
| Visibility symbol | Visibility | Meaning |
| + | Public | The feature (an operation or an attribute) is directly accessible by an instance of any class. |
| - | Private | The feature may only be used by an instance of the class that includes it. |
| # | protected | The feature may be used either by instances of the class that includes it or of a subclass or descendant of that class. |
| ~ | Package | The feature is directly accessible only by instances of a class in the same package. |

**19. COMPONENT DIAGRAM:**

Two type’s implementation diagrams in UML terminology are

1. component Diagrams
2. Deployment diagrams

In a large project there will be many files that make up the system. These files will have dependencies on one other. The nature of these dependencies will depend on the language or languages used for the development and may exist at compile-time, at link-time or at run-time. There are also dependencies between source code files and the executable files or byte code files that are derived from them by compilation. Component diagrams are one of the two types of implementation diagram in UML. Component diagrams show these dependencies between software components in the system. Stereotypes can be used to show dependencies that are specific to particular languages also.

A component diagram shows the allocation of classes and objects to components in the physical design of a system. A component may represent all or part of the component architecture of a system along with dependency relationships.

The dependency relationship indicates that one entity in a component diagram uses the service or facilities of another.

* Dependencies in the component diagram represent compilation dependencies.
* The dependency relationship may also be used to show calling dependencies among components, using dependency arrows from components to interfaces on other components.

Different authors use component diagrams in different ways.

FOWLER suggests that component correspond exactly to packages of the system. MULLER suggests that packages contain components as well as other packages. Strictly in UML packages should be used for model management for organizing models into convenient parts that contain types of diagram or sub-system.

Here we have the following distinctions between them.

* Components in a component diagram should be the physical components of a system.
* During analysis and the early stages of design, package diagrams can be used to show the logical grouping of class diagrams or of models that use other kinds of diagrams into packages relating to sub-system.
* During implementation, package diagrams can be used to show the grouping of physical components into sub-system.
* Component diagrams can also be combined with deployment diagrams to show the physical location of components of the system. The classes in one logical packages may be distributed across physical locations in a physical system, and the component diagram and deployment diagram can be used to show this.

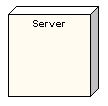
If component diagrams are used , it is better to keep separate sets of diagrams to show compile-time and run-time dependencies. However, this is likely to result in a large number of diagrams. Component diagrams show the components as types. If you wish to show instances of components you can use a deployment diagram.

**Component Diagram for Online Shopping:**

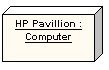
****

**20. Deployment Diagram:**  
A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

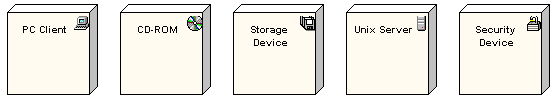
Node:  
A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.



Node Instance:  
A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon. The following diagram shows a named instance of a computer.

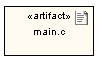


Node Stereotypes:  
A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc». These will display an appropriate icon in the top right corner of the node symbol

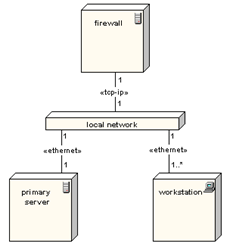


Artifact:  
An artifact is a product of the [software development](http://www.sparxsystems.com/platforms/software_development.html) process. That may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals, etc.

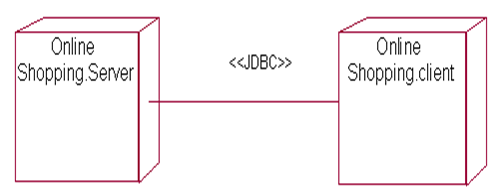
An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon, as shown below.



Association:  
In the context of a deployment diagram, an association represents a communication path between nodes. The following diagram shows a deployment diagram for a network, depicting network protocols as stereotypes, and multiplicities at the association ends.



**Deployment Diagram:**

****

**REFERENCES**

1. Object Oriented Analysis and Design – Simon Benett
2. Visual modeling with Rational Rose 2002 – Terry Quatrani
3. Object Oriented systems Development - Ali Bahrami

**WEB REFERENCES**

1. [http://www.Wikipedia.com](http://www.wikipedia.com/)

2. <http://cpwdpims.nic.in/pims/download/PMSHelpFile.pdf>

3. <http://124.107.59.117/PROJMON>**/**