MOBILE PAYMENT SYSTEM

1.PROBLEM STATEMENT

The project entitled Mobile payment system project is an application to assist mobile phone users to access the information of different bills, send and receive queries requested by user from anywhere and in anytime.

With the use of a single touch button the user is provided with the requested information as quickly as possible. Mobile payment system is focused for professionals as well as House wives and people who are busy can save their time to do their work in home. Mobile payment system project has lots of advanced accessibility features to facilitate users with a user-friendly as well as flexible interface.

Whenever user chooses to launch Mobile payment system project, he will be asked for credentials and after that he is logged on to a window. Where he is asked to select which option or service he would like to access. Upon choosing we will be presented with a list of the particular service bill details. If the user selects the options provided on the screen given he is instantly provide with the information pertaining to that particular service. Some of the most unique features of Mobile payment system are accessing the data at the server and retrieving the data from the database.

**ANALYSIS**

**2. Requirements Elicitation:**

|  |  |  |  |
| --- | --- | --- | --- |
| **RID** | **REQUIREMENTS** | **REQUIREMENT**  **NATURE** | **PRIORITY** |
| R1 | It shall allow user to  Register or Login to access various Online payments | Functional | Must Have |
| R2 | It shall collect User details like Email address, name | Functional | Must Have |
| R3 | It allows user to access payment for mobile recharge, Electricity bill, DTH recharge | Functional | Must Have |
| R4 | It allows user to recharge or payment within seconds | Functional | Must Have |
| R5 | It allows user to report problems and gives acknowledgement | Functional | Must Have |
| R6 | It allows user to access through debit card | Functional | Must Have |
| R7 | It provides facilities for mobile recharge, electricity bill payment, DTH recharge | Functional | Must Have |
| R8 | Database interface should be provided | Functional | Must Have |
| R9 | It provides tracking of complaints | Functional | Must Have |
| R10 | It stores details about recharge data | Functional | Should Have |
| R11 | It records Pay Status of the users | Functional | Should Have |
| R12 | Accepts all major debit cards | Functional | Should Have |
| R13 | It shall use a browser as its user interface | Non-Functional | Could Have |
| R14 | Checks the availability of balance when user requests for recharge by entering the amount | Non-Functional | Could Have |
| R15 | It provides high security to the user details | Non-Functional | Could Have |
| R16 | It authenticates user details after registering | Non-Functional | Could Have |
| R17 | It shall provide 24\*7 support and timely recharge services | Non-Functional | Could Have |
| R18 | It allows customer or user to login within 5 seconds | Non-Functional | Could Have |
|  |  |  |  |

**USECASE VIEW**

**3.IDENTIFICATION OF ACTORS**

1.Actors are NOT part of the system.

2.Actors represent anyone or anything that interacts with (input to or receive output from) the system.

3.An actor is someone or something that:

* + Interacts with or uses the system.
  + Provides input to and receives 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 needs of the actor are used to develop use cases. This insures that the system will be what the user expected.

**Graphical Depiction**

* An actor is a stereotype of a class and is depicted as a "stickman" on a use-case diagram.



Questions that help to identify actors

1. Who is interested in a certain requirement?
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?

From above mentioned information the actors mainly involved in the Mobile Payment System are:

1. Administrator

2. Customer

3.Bank Employee

**Brief description of actors:**

**Customer**:

The main objective of the users is to access the mobile payment system. He can utilize the services provided by the system by registering or login. He can recharge for mobile, pay electricity bill, DTH recharge. He can pay the amount through all types.



**Administrator:**

Administrator has the authority over the website. He can view details of the user and can delete the users. He can track complaints given by the users. He can view payment status of the users. He can update database.



**Bank Employee:**

Bank Employee has the authority to check the payments of the customer which are done by debit/credit card.



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

Use-cases diagrams graphically represent system behavior (use cases). 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 diagram may contain all or some of the use cases of a system.

Graphical Depiction:

* The basic 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 sequences of events between objects. Use-case names often start with a verb. For example, names of possible use cases for an ATM machine might include Dispense Cash or Transfer Funds.
* The name of the use case is displayed below the icon.



The set of questions used to identify the use-cases are:

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 this 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?

Use-cases identified for our system are:

* Login
* Register
* Select Category
* Payments
* Complaints
* Manage user details
* Manage user Pay status
* Track Complaints

1.Use-case name: **Login**

The customers must login into the Mobile Payment system.Only then,

they can access the services provided by the system.

UML Notation:



2.Use-case name: **Register**

The customers must register into the mobile payment system. Only then,

he can login and access the services.

UML Notation:



3. Use-case name: **Select Category**

User can select service among the displayed services.

UML Notation:



4.Use-case name: **Payments**

User can pay or recharge through various types of payment modes.

UML Notation:



5.Use-case name: **Complaints**

User can give complaints about the problems faced in accessing

services.

UML Notation:

Complaints

6. Use-case name: **Manage user details**

Administrator manages the user details and stores in the database.

UML Notation:

Manage user details

7.Use-case name: **Manage user Pay status**

Administrator makes list of user who access the services

and records pay status of users.

UML Notation:

Manage user Pay status

8. Use-case name: **Track Complaints**

Administrator tracks complaints given by the users and sends acknowledgement.

UML Notation:

Track Complaints

|  |  |
| --- | --- |
| Use case: | Payment |
| Id: | 001 |
| Brief description: | It is started by Customer. It provides customer to make payment by  Cash/Card. |
| Primary actor | Customer |
| Secondary actor | Bank Status Member |
| Pre-condition: | Bank Status Member checks balance of customer if payment is done by debit/credit card. |
| Post condition: | Customer checks balance left in the debit/credit card. |
| Flow of events: | 1.Customer logs into the system  2.Customer does the payment by cash/credit card. |
| Alternate flow: | 1.Payment by Credit/Debit Card becomes unsuccessful if Customer enters invalid Card Number/Name/Expiry date/CVV. |

**5.FLOW OF EVENTS**

|  |  |
| --- | --- |
| Use case: | Complaints |
| Id: | 002 |
| Brief description: | It is started by the Customer. It provides Customer to register any complaint regarding the payment. |
| Primary actor | Customer |
| Secondary actor | Administrator |
| Pre-condition: | Customer does the payment and if it becomes unsuccessful he registers a complaint. |
| Post condition: | Customer waits for the reply from the administrator regarding the complaint. |
| Flow of events: | 1.Customer does the payment by cash/card.  2.If the payment becomes unsuccessful then customer moves to complaint box. |
| Alternate flow: | None |

**6.USE-CASE DIAGRAM**

Use-case diagrams graphically represent system behavior (use cases). 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 diagram may contain all or some of the use cases of a system.

A use-case diagram can contain:

* Actors ("things" outside the system)
* Use cases (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.

Use-case diagrams can be used during analysis to capture the system requirements and to understand how the system should work. During the design phase, you can use use-case diagrams to specify the behavior of the system as implemented.

**RELATIONS:**

**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

By default, the association tool on the toolbox is uni-directional and drawn on a diagram with a single arrow at one end of the association. The end with the arrow indicates who or what is receiving the communication.

**Bi-directional association:**

If you prefer, you can also customize the toolbox to include the bi-directional tool to the use-case toolbox.

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 diagrams, a dependency relationship indicates that the operations of the client invoke operations of the supplier.

We can provide here

1. Include Relationship.

2. Extend Relationship

* There are two types of relationships that may exist between use cases: *include relationship* and *extend 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.

**Include Relationship:**

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



**Extended 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 relationships between use cases are modeled as dependencies by using the Extend stereotype.

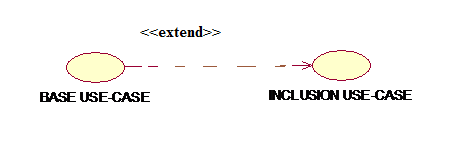
An *extend* relationship is used to show

* Optional behavior
* Behavior that is run only under certain conditions 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



Finally, we can conclude

«extend» is used when you wish to show that a use case provides 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.

**EX: Basic Use-case diagram for customer in Mobile Payment System**





**Use-case Diagram for Administrator:**



****

**7. SAMPLE PROTOTYPES FOR APPLICATION:**

1) In software development, a prototype is a rudimentary working model of a product or information system, usually built for demonstration purposes or as part of the development process. In the systems development life cycle (SDLC) Prototyping Model, a basic version of the system is built, tested, and then reworked as necessary until an acceptable prototype is finally achieved from which the complete system or product can now be developed.

2) In prototype-based programming, a prototype is an original object; new objects are created by copying the prototype.

3) In hardware design, a prototype is a "hand-built" model that represents a manufactured (easily replicable) product sufficiently for designers to visualize and test the design.

The word prototype comes from the Latin words proto, meaning original, and typus, meaningform or model. In a non-technical context, a prototype is an especially representative example of a given category.

**Login Prototype:**

Submit

|  |
| --- |
| Password  Username  MOBILE PAYMENT SYSTEM |

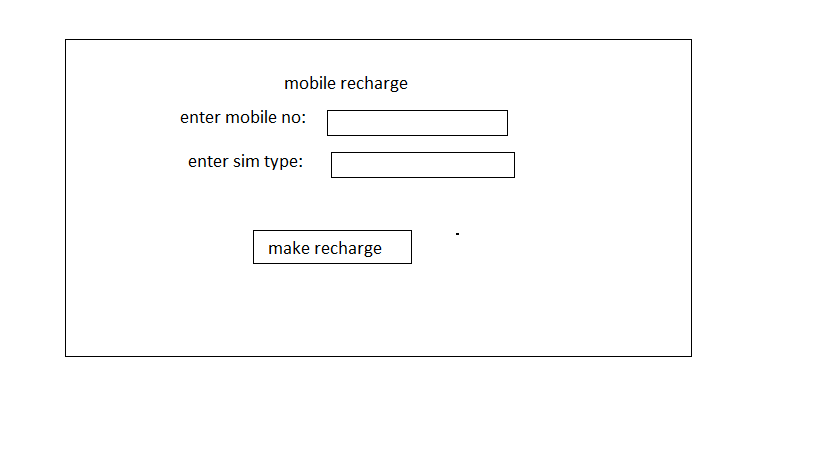
**Select Category:**

Select Category

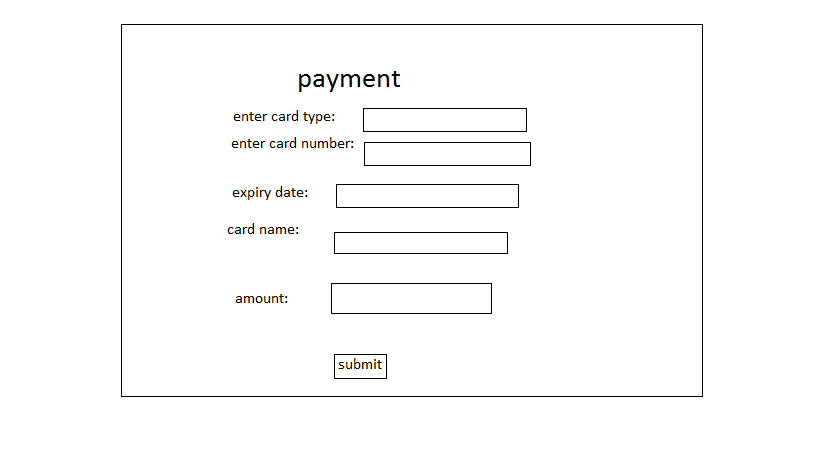
Cancel

OK

**prototype for mobile recharge:**



**prototype for payment:**



**8.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.

You can also use activity diagrams to model code-specific information such as a class operation.

Activity diagrams are very similar to a flowchart because you can model a workflow 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 diagrams also may be created at this stage in the life cycle. These diagrams represent 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 cases 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.

**ACTIVITIES:**

Activity diagrams contain activities, transitions between the activities, decision points, and synchronization bars. An activity represents the performance of some behavior in the workflow. In the UML, activities are represented as rectangles with rounded edges, transi­tions are drawn as directed arrows, decision points are shown as diamonds, and synchronization bars are drawn as thick horizontal or vertical bars as shown in the following.

The activity icon appears as a rectangle with rounded ends with a name and a component for actions



**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. Transition connects activities with other model elements and object flows connect activities with objects. They are typically triggered by the completion of the behavior in the originating activity.



**DECISIONS:**

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.

****

**Decision Point**

**END STATE:**

An end state represents a final or terminal state on an activity diagram or state chart diagram. Place an end state when you want t explicitly show the end of a workflow on an activity diagram or the end of a state chart diagram. Transitions can only occur into an end state; however, there can be any number of end states per context.

****

**End state**

**START STATE:**

A start state (also called an “initial state”) explicitly show the beginning of a workflow on an activity diagram or the beginning of the execution of a state machine on a state chart diagram.



**Start state**

**SWIMLANES:**

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.

Swim lanes are helpful when modeling a business workflow because they can represent organizational units or roles within a business model. Swim lanes are very similar to an object because they provide a way to tell who is performing a certain role. Swim lanes only appear on activity diagrams.

When a swim lane is dragged onto an activity diagram, it becomes a swim lane view. Swim lanes appear as small icons in the browse while a swim lane views appear between the thin, vertical lines with a header that can be renamed and relocated.

**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.

Horizontal synchronization 

 Vertical synchronization

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

* 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.
* 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.
* 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.
* Define and diagram any objects that are created or modified within your activity diagram. Connect the objects and activities with object flows.
* 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.
* Connect all elements on the diagram with transitions. Begin with the "main" workflow.
* 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.
* Evaluate your diagram and see if you have any concurrent workflows. If so, use synchronizations to represent forking and joining.
* Set all actions, triggers and guard conditions in the specifications of each model element.

**Activity diagram for Mobile payment system:**



**LOGICAL VIEW**

**9.IDENTIFICATION OF ANALYSIS CLASSES**

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

1) Noun phrase approach

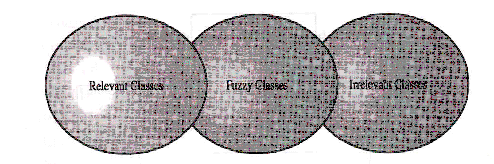
2) Common class pattern approach

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.

**10.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 t 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 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 they become apparent.

From a UML perspective, use of CRC cards is in analyzing the object interaction that 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.

1. Identify and record any missing or redundant objects.

**11.USE-CASE REALIZATIONS**

A use case realization is a graphic sequence of events, also referred as an instance of a use case. These realizations are represented using either a sequence or collaboration diagrams.

Use case Realization Diagrams:











**12.SEQUENCE DIAGRAM**

A sequence diagram is a graphical view of a scenario that shows object interaction in a time-based sequence-what happens first, what happens next….

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

A sequence diagram has two dimensions: the vertical dimension represents time; the horizontal dimension represents different objects. The vertical line is called the object’s lifeline. The lifelinerepresents the object’s existence during the interaction.

**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.

The sequence diagram is very simple and has immediate visual appeal—this is its great strength. A sequence diagram is an alternative way to understand the overall flow of control of a program. Instead of looking at the code and trying to find out the overall sequence of behavior

The following tools located on the sequence diagram toolbox which enable to model sequence diagrams:

**Object:** An object has state, behavior, and identity. The structure and behavior of similar 1objects are defined in their common class. Each object in a diagram indicates some instance of a class. An object that is not named is referred to as a class instance.

**Message Icons:** A message icon represents the communication between objects indicating that an action will follow. The message icon is a horizontal, solid arrow connecting two lifelines together.

**Focus of Control:** Focus of Control (FOC) is an advanced notational technique that enhances sequence diagrams. It shows the period of time during which an object is performing an action either directly or through an underlying procedure.

**Message to self:** A message to self is a tool that sends a message from one object back to the same object. It does not involve other objects because the message returns to the same object. The sender of a message is the same as the receiver.

**Note:** A note captures the assumptions and decisions applied during analysis and design. Notes may contain any information, including plan text, fragments of code, or references to other documents.

**Note Anchor:** A note anchor connects a note to the element that it affects.

**Sequence diagram for mobile payment system:**



**13.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 predecessoris 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 conditionsare 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 recurrenceterm and terminated by a colon. A sequence-expression has the following syntax:

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

In this expression integerrepresents 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 ‘ ]’

**Collaboration diagram for mobile payment system:**

****

**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 show s object interaction in a time-based sequence.
* A collaboration diagram shows object interactions organized around the objects and their links to each other

**14.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.

Guidelines for identifying attributes of classes are as follows:

* Attributes usually correspond to nouns followed by prepositional phrases. Attributes also may correspond to adjectives or adverbs.
* Keep the class simple; state only enough attributes to define the object state.
* Attributes are less likely to be fully described in the problem statement.
* Omit derived attributes.
* Do not carry discovery attributes to excess.

Some questions are there which help in identifying the responsibilities of classes and deciding what data elements to keep track of:

* + What information about an object should we keep track of?
  + What services must a class provide?

Answering the first question helps us to identify the attributes of a class. Answering the second question helps us to identify class methods.

**15.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:

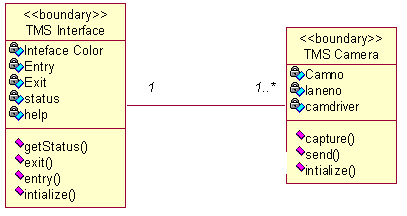
1. Associations Relationship
2. Aggregations Relationship

1. **Association Relationship:**

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 Searching system class and the Airline database means that objects in the class searching system are connected to objects in the Airline database.

Association Relationship with Multiplicity



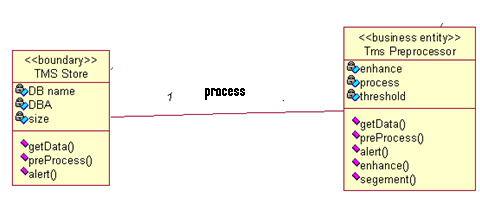
**2.** **Aggregation Relationship:**

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:



3. **Naming Relationship:**

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. 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.

**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

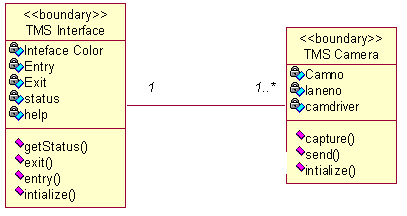
0.. \* Zero or more

1... \* One or more

0 … 1 Zero or one

5 …8 Specific range (5, 6, 7, or 8)

4 …7 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.



**16.UML CLASS DIAGRAM**

* 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 Airline database 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 as shown below.



**Analysis Class Stereotypes:**

Analysis class stereotypes represent 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 met model.
* Stereotypes must be based on certain existing types or classes in the met model.
* 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 stereotypes depend 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.

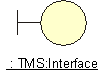
This may include interfaces with other software and also with physical devices such as printers, motors and sensors. Stereotyping these as boundary classes emphasizes that their main task is to manage the transfer of information across system boundaries. It also helps to partition the system, so that 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 TMS Interface is a typical boundary class. This style of writing the name shows that the class is TMS Interface 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

a) With stereotype



b) Symbol

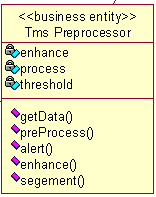


**2. Entity classes**

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 or 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

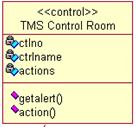


**3. Control classes**

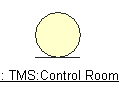
The third of the analysis class stereotypes is the control class, given by y the class Searching system in Search flight. Control classes ‘represent coordination, sequencing, transactions and control of other objects’ .In the USDP, as in the earlier methodology Objector. It is generally recommended that 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



**Class diagram for Mobile Payment System:**



**17.UML STATE CHART DIAGRAM**

**STATE MACHINE DIAGRAM:**

Use caseand 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 state of a single object, the events or messages that causes 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.

UML notation for STATE is



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

In our application the object for Account 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(arguments)[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.

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 send Event (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. Like-wise 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



**Purpose of state chart diagrams:**

* We use state chart diagram when working on real-time process control applications or systems that involve concurrent processing. It will also be used when showing the behavior of a class over several use cases.
* State chart diagrams are used to model dynamic view of a system.
* State chart diagrams are used to emphasizing the potential states of the objects and the transitions among those states.
* State chart diagrams are used to modeling the lifetime of an object.
* State chart diagrams are used to model the behavior of an interface. Although an interface may not have any direct instances, a class that realizes such an interface may. Those classes conform to behavior specified by the state machine of this interface.
* State chart diagrams are used to focus on the changing state of a system driven by events.
* This diagram is also for constructing executable systems through forward and reverse engineering.
* To model reactive objects, especially instances of a class, use cases, and the system as a whole.

**ELEMENTS OF STATE CHART DIAGRAMS:**

1. **State:** It is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
2. **Event:** It is the specification of significant occurrence that has a location in time and space.
3. **Transition:** It is a relation between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and conditions are satisfied.
4. **Action state:** An action state is shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action.
5. **Sequential sub state:** A submachine state represents the invocation of a state machine defined elsewhere. The submachine state is depicted as a normal state with the appropriate “include” declaration within its internal transitions compartment. As an option, the submachine state may contain one or more sub states, which represent already created states.
6. **Concurrent sub state:** A concurrent state is divided into two or more sub states. It is a state that contains other state vertices. Naturally, any sub state of a concurrent state may also be a composite state of either type. Any state enclosed within a composite state is called a sub state of that concurrent state.
7. **Initial state:** A pseudo state to establish the start of the event into an actual state.
8. **Final state:** The final state symbol represents the completion of the activity.
9. **History state:** History state is a state machine describes the dynamic aspects of an object whose current behavior depends on its past.
10. **Vertical Synchronization:** This merge branch bar symbol is also known as a “Synchronization Bar”. It merges concurrent transitions to a single target. It splits a single transition into parallel transitions.
11. **Horizontal Synchronization:** This merge branch bar symbol is also known as a “Synchronization Bar”. It merges concurrent transitions to a single target. It splits a single transition into parallel transitions.
12. **Guard conditions:** Activity and state diagrams express a decision when conditions are used to indicate different possible transitions that depend on Boolean conditions of container object. UML calls those conditions as guard conditions.

**Forks and joins:** A fork construct is used to model a single flow of control that divides into two or more separate, but simultaneous flows

**State chart diagram for Mobile payment system:**

****

**DESIGN**

**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.

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. |

**Interfaces:**

Generally, a class may present more than one external interface to other classes or the same interface may be required from more than one class. An interface in UML is a group of externally visible (i.e. public) operations. The interface contains no internal structure, it has no attributes, no associations and the implementation of the operations is not defined. Formally, an interface is equivalent to an abstract class that has no attributes, no associations and only abstract operations.

The following figure shows two alternative notations for an interface. The simpler of the two UML interface notations is a circle. This is attached by a solid line to the classes that support the interface.



The alternative notation uses a stereotyped class icon. As an interface only specifies the operations and has no internal structure, the attributes compartment is omitted. This notation lists the operations on the diagram. The *realize* relationship, represented by the dashed line with a triangular arrowhead, indicates that the class supports at least the operations listed in the interface



**Refining Attributes:**

In the analysis phase, the name of the attribute was sufficient. However, in the design phase, detailed information must be added to the model. There are three basic types of attributes. They are:

1) Single-value attributes.

2) Multiplicity or multi-value attributes.

3) Reference to another object, or instance connection.

Attributes represent the state of an object. The following is the attribute presentation:

**Visibility name: type-expression=initial-value**

Where visibility is one of the following:

+ public visibility

# protected visibility

- private visibility

During analysis, we identified the following attributes for classes:

Refining attributes for Librarian class:

* name
* number

At this stage we need to add more information to these attributes, such as

visibility and implementation type.

+ name: String

+ number: String

Refining attributes for member class:

name

id

After refining

+ id: String

+ name: String

**Refining methods:**

A class can provide several types of methods.

• Constructor: Method that creates instances of the class.

• Destructor: The method that destroys instances.

• Conversion method: The method that converts a value from one unit of measure

to another.

• Copy method: The method that copies the contents of one instance to another

instance.

• Attribute set: The method that sets the values of one or more attributes.

• I/O methods: The methods that provide or receive data to or from a device.

• Domain specific: The method specific to the application.

The operation syntax is:

**Visibility name: (parameter-list): return-type-expression**

**19.Implementation diagrams**

**19.a. 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 another. 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 b/n 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 components diagram 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 components diagram uses the services 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. Here we have the following distinction b/n them

* Components in a component diagram should be the physical components of a system.
* During analysis and the early stages of design, package 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-systems.
* During implementation, package diagrams can be used to show the grouping of physical components into sub-systems.

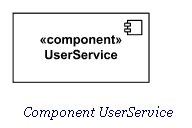
If component diagrams are used, it is better to keep separate sets of diagrams to show compile-time and run-time dependencies, however, this is likely to result in a large number of diagrams. Component diagrams show the components as types. If you wish to show instances of components of components you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

**Notations used in Component Diagram**

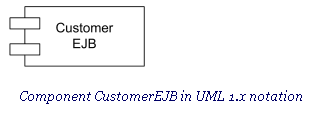
A component is shown as a classifier rectangle with the keyword **«component»**

****

Optionally, a component icon similar to the **UML** icons can be displayed in the right hand corner.

****

For backward compatibility reasons, the **UML** notations with protruding rectangles can still be used.



A component may be manifested by one or more **artifacts,** and in turn, that artifact may be deployed to its execution environment. A **deployment specification** may define values that parameterize the component’s execution.

Standard Component Stereotypes:There are several standard UML stereotypes that apply to components:

* [«subsystem»](http://www.uml-diagrams.org/component-diagrams.html#subsystem)
* [«process»](http://www.uml-diagrams.org/component-diagrams.html#process)
* [«service»](http://www.uml-diagrams.org/component-diagrams.html#service)
* [«specification»](http://www.uml-diagrams.org/component-diagrams.html#specification)
* [«realization»](http://www.uml-diagrams.org/component-diagrams.html#realization)
* [«implement»](http://www.uml-diagrams.org/component-diagrams.html#implement)

#### «subsystem»:

#### ****Subsystem**** is a [component](http://www.uml-diagrams.org/component-diagrams.html#component) representing unit of hierarchical decomposition for large systems, and is used to model large scale components. Definitions of subsystems may vary among different domains and software methods. It is expected that domain and method profiles will specialize this element. A subsystem is usually indirectly instantiated. A subsystem may have specification and realization elements.

#### «process»: UML Standard Profile defines ****process**** as a transaction based [component](http://www.uml-diagrams.org/component-diagrams.html#component).

#### «service»: ****Service**** is a stateless, functional [component](http://www.uml-diagrams.org/component-diagrams.html#component).

#### «specification»: Specification is a [classifier](http://www.uml-diagrams.org/uml-core.html#classifier) that specifies a domain of objects without defining the physical implementation of those objects. For example, a component stereotyped by «specification» will only have provided and required interfaces, and is not intended to have any realizing classifiers as part of its definition. This differs from «type» because a «type» can have features such as attributes and methods that are useful to analysts modeling systems. «Specification» and «realization» are used to model components with distinct specification and realization definitions, where one specification may have multiple realizations.

#### «realization»: ****Realization**** is a [classifier](http://www.uml-diagrams.org/uml-core.html#classifier) that specifies a domain of objects and that also defines the physical implementation of those objects. For example, a component stereotyped by «realization» will only have realizing classifiers that implement behavior specified by a separate [«specification»](http://www.uml-diagrams.org/component-diagrams.html#specification) component. This differs from «implementation class» because an «implementation class» is a realization of a class that can have features such as attributes and methods that are useful to system designers.

#### «implement»: ****Implement**** is a [component](http://www.uml-diagrams.org/component-diagrams.html#component) definition that is not intended to have a specification itself. Rather, it is an implementation for a separate [«specification»](http://www.uml-diagrams.org/component-diagrams.html#specification) to which it has a [dependency](http://www.uml-diagrams.org/dependency.html). It seems to duplicate [«realization»](http://www.uml-diagrams.org/component-diagrams.html#realization).

### Provided Interface: A ****provided interface**** is the one that is either

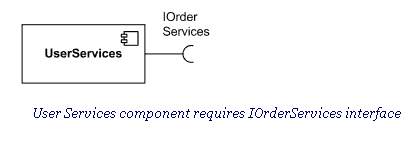
* [realized](http://www.uml-diagrams.org/abstraction.html#interface-realization) directly by the component itself, or
* [realized](http://www.uml-diagrams.org/abstraction.html#interface-realization) by one of the classifiers realizing component, or
* Is provided by a public port of the component.



### Required Interface

A **required interface** is either

* designated by usage dependency from the component itself, or
* designated by usage dependency from one of the classifiers realizing component, or
* Is required by a public port of the component.

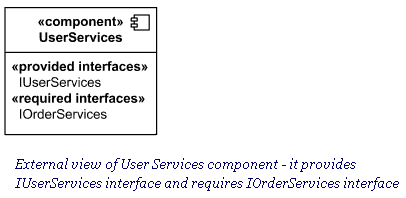


### External View of Component

A component has an **external view** or **"black-box"** view by means of interface symbols sticking out of the component box exposing its publicly visible properties and operations.

Optionally, a behavior such as a protocol state machine may be attached to an interface, port, and to the component itself, to define the external view more precisely by making dynamic constraints in the sequence of operation calls explicit. Other behaviors may also be associated with interfaces or connectors to define the "contract" between participants in collaboration (e.g., in terms of use case, activity, or interaction specifications).

Alternatively, the interfaces and/or individual operations and attributes can be listed in the compartments of a component box (for scalability, tools may offer a way of listing and abbreviating component properties and behavior).



For displaying the full signature of an interface of a component, the interfaces can also be displayed as typical classifier rectangles that can be expanded to show details of operations and events.

**Connector**

**Connector** in components extends connector from [internal structures](http://www.uml-diagrams.org/composite-structure-diagrams.html#internal-structure-diagrams). It specifies a **link** that enables communication between two or more instances. Connector was extended in the components to include **contracts** and specific notation.

Connector linking components could be either:

* Delegation connector or
* Assembly connector.

Kind of connector attribute is derived: a connector with one or more ends connected to a port that is not on a part and that is not a behavior port is a delegation; otherwise it is an assembly.

Connector's **contract** is set of behaviors that specify the valid interaction patterns across the connector.

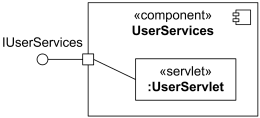
### Delegation Connector

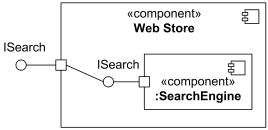
A **delegation connector** is a connector that links the external contract of a component (as specified by its ports) to the realization of that behavior. It represents the forwarding of events (operation requests and events): a signal that arrives at a port that has a delegation connector to one or more parts or ports on parts will be passed on to those targets for handling.

A delegation connector is a declaration that behavior that is available on a component instance is not actually realized by that component itself, but by one or more instances that have “compatible” capabilities. These situations are modeled through a delegation connector from a Port to compatible Ports or Parts.

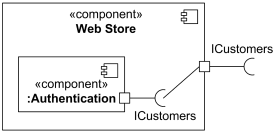
Delegation connectors can be used to model the hierarchical decomposition of behavior, where services provided by a component may ultimately be realized by one that is nested multiple levels deep within it. The word delegation suggests that concrete message and signal flow will occur between the connected ports, possibly over multiple levels. It should be noted that such signal flow is not always realized in all system environments or implementations (i.e., it may be design time only).

A port may delegate to a set of ports on subordinate components. In that case, these subordinate ports must collectively offer the delegated functionality of the delegating port. At execution time, signals will be delivered to the appropriate port. In cases where multiple target ports support the handling of the same signal, the signal will be delivered to all these subordinate ports. A delegation connector is notated as a connector from the delegating port to the handling port or part.



Delegation connector from the delegating port to the User Servlet part.If the delegation is handled by a simple port, then the connector may optionally be shown connected to the single lollipop or socket.

Delegation connector from the delegating port to the simple port of Search Engine



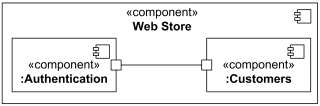
Delegation connector from the simple port of Authentication component to the delegating port

### Assembly Connector

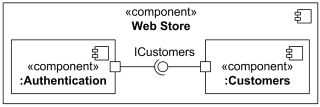
An **assembly connector** is a connector between two or more parts or ports on parts that defines that one or more parts provide the services that other parts use.

The execution time semantics for an assembly connector are that signals travel along an instance of a connector. Multiple connectors directed to and from different parts, or n-ary connectors where n > 2, indicates that the instance that will originate or handle the signal will be determined at execution time.

The interface compatibility between ports that are connected enables an existing component in a system to be replaced by one that (minimally) offers the same set of services. Also, in contexts where components are used to extend a system by offering existing services, but also adding new functionality, connectors can be used to link in the new component definition. Assembly connector is notated as a connector between two or more parts or ports on parts.

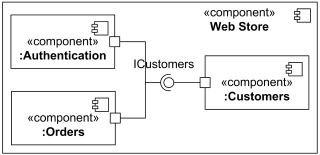


Assembly connector between ports of Authentication and Customers components When an assembly connector connects simple ports (ports that provide or require a single interface), it may be notated by a "ball-and-socket" connection between a provided interface and a required interface.



Assembly connector between simple ports of Authentication and Customers components Ball-and-socket notation may not be used to connect "complex" ports or parts without ports.

Where multiple components have simple ports that provide or require the same interface, a single symbol representing the interface can be shown, and lines from the components can be drawn to that symbol. This presentation option is applicable whether the interface is shown using "ball-and-socket" notation, or just using a required or provided interface symbol.

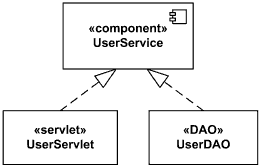


**Component Realization**

Component Realization is specialized [realization](http://www.uml-diagrams.org/abstraction.html#realization) dependency used to (optionally) define [classifiers](http://www.uml-diagrams.org/uml-core.html#classifier) that realize the contract offered by a component in terms of its [provided interfaces](http://www.uml-diagrams.org/class-diagrams.html#provided-interface) and [required interfaces](http://www.uml-diagrams.org/component-diagrams.html#required-interface).

A component’s behavior may typically be realized (or implemented) by a number of Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Component Realization Dependencies to these Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Realization Dependencies to these Classifiers.

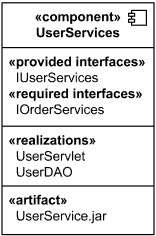
A component realization is notated in the same way as the [realization](http://www.uml-diagrams.org/abstraction.html#realization) dependency, i.e., as a general dashed line from implementing classifiers to realized component with hollow triangle as an arrowhead.



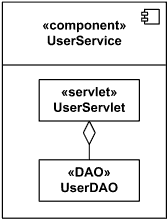
Component User Service realized by User Servlet and User DAO.

For the purpose of applications that require multiple different sets of realizations for a single component specification, a set of standard stereotypes are defined in the UML Standard Profile. In particular, «specification» and «realization» are defined there for this purpose.

Component could be shown using internal view or "white-box" view exposing its private properties and realizing classifiers. This view shows how the external behavior is realized internally. The realizing classifiers could be listed in an additional «realizations» compartment. Compartments may also be used to display a listing of any parts and connectors, or any implementing artifacts.



White box view of User Services component - it is realized by User Servlet and User DAO and manifested by UserService.jar artifact alternatively, the internal classifiers that realize the behavior of a component may be displayed nested within the component shape.



White box view of User Services component - it is realized by User Servlet and User DAO

**Component diagram for mobile payment system:**

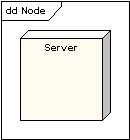


**19.b. Deployment Diagram**

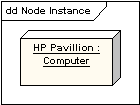
The second type of implementation diagram provided by UML is the deployment diagram they are used to show the configuration of runtime processing elements and the software components and processes that are located on them. They are made up of nodes and communication associations**.**

**Notations used in Deployment Diagram**

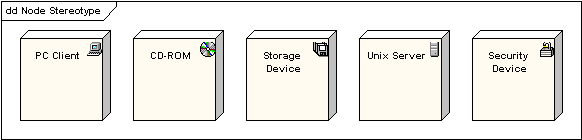
**Node**  
A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.



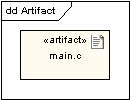
**Node Instance**: A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon. The following diagram shows a named instance of a computer.



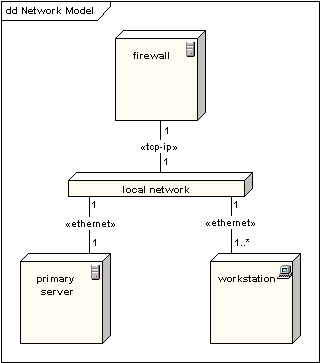
**Node Stereotypes**: A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «Unix server», «user pc». These will display an appropriate icon in the top right corner of the node symbol



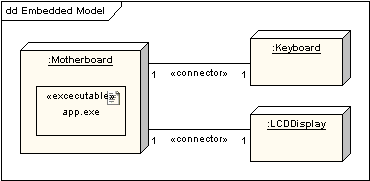
**Artifact**  
An artifact is a product of the [software development](http://www.sparxsystems.com/platforms/software_development.html) process. That may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals,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



**Node as Container**: A node can contain other elements, such as components or artifacts. The following diagram shows a deployment diagram for part of an embedded system, depicting an executable artifact as being contained by the motherboard node.



**Deployment diagram for mobile payment system:**



**19.c. IMPLEMENTATION OF DOMAIN OBJECTS LAYER AND**

**TECHNICAL SERVICE LAYER**

* **Code for DTH Recharge:**

<html>

<body background=bg3.jpg><center>

<h1>DTH RECHARGE</h1>

<form name="dthrecharge" method="post" >

Enter subscriber ID: <input type="text" name="mobile" maxlength="10">

<br/><br/>

Enter Amount: <input type="text" name="amount" maxlength="1000">

<br/></br>

Select Operator:

<select name="operator">

<option value="">Choose</option>

<option value="AT">TATASKY</option>

<option value="JO">VIDEOCON</option>

<option value="AL">HATHWAY</option>

<option value="BS">SUNDIRECT</option>

<option value="BSS">BIG TV</option>

</select>

<br/><br />

<a href="payment.html">Make recharge </a>

</form>

</center>

</body>

</html>

* **Code for Mobile Recharge:**

<html>

<head><title>mobilerecharge</title>

<style>

body{

background:url("bg2.jpg");

color:Blue;

}

</style>

</head>

<body>

<center>

<div style="font-size:45px;height:80px;">

MOBILE RECHARGE DETAILS

</div>

<form>

<label class="control-label"><b>Mobile:&nbsp;&nbsp;</b></label>

<input type="text" class="form-control" name="mobile" placeholder="Enter Mobile number here........"/><br/>

<br/>

select sim type:

<select>

<option>IDEA</option>

<option>VODAFONE</option>

<option>RELIANCE</option>

<option>BSNL</option>

<option>TATADOCOMO</option>

<option>AIRTEL</option>

</select>

</form>

<div id="menu">

<ul>

<li>

<a href="payment.html">Make Recharge</a></li></ul>

</div></center>

</body>

</html>

* **Code for Select Category:**

<html>

<head>

<title>payment type</title>

<link href="default.css" rel="stylesheet" type="text/css" />

<style type="text/css">

</style>

</head>

<!--<frameset cols="33%,33%,33%">

<frame src="mobrecg.html">

<frame src="elec.html">

<frame src="dth.html">

</frameset>--!>

<body>

<center>

<p style="text-align:center; color:#FF0000;"><strong><marquee behaviour="scroll">SELECT THE CATEGORY YOU WANT TO MAKE PAYMENT</marquee></strong></p>

<p class="style1"><a href="mobrecg.html"><img src="ss1.jpg" alt="" width="175" height="150" /></a></p>

<p class="style1"><a href="elec.html"><img src="ss2.jpg" alt="" width="175" height="150" /></a></p>

<p class="style1"><a href="dthrecharge.html"><img src="ss3.jpg" alt="" width="175" height="150" /></a></p>

</center>

</body>

</html>

* **Code for Payment:**

<html>

<head>

<title>payment</title>

<style>

body{

background:url("5.jpg");

}

</style>

</head>

<body>

<center>

<div style=font-size:45px; height:80px;>

PAYMENT FORM

</div>

</center>

<section>

<form method=post id=payment action="pay.php">

<fieldset>

<legend>Card Details</legend>

<ul>

<li>

<fieldset>

<legend>Card Type</legend>

<ul><li>

<input id=visa name=cardtype type=radio />

<label for=visa>VISA</label>

</li>

<li>

<input id=maestro name=cardtype type=radio />

<label for=maestro>MAESTRO</label>

</li>

<li>

<input id=rupay name=cardtype type=radio />

<label for=rupay>RUPAY</label>

</li>

<li>

<input id=mastercard name=cardtype type=radio />

<label for=mastercard>MASTER</label>

</li>

</ul>

</li>

<li>

<label for=cardnumber>Card Number</label>

<input id=cardnumber name=cardnumber type=text maxlength=16 required /><br/><br/>

</li>

<li>

<label for=namecard>Name on Card</label>

<input id=namecard name=namecard type=text placeholder="Exact name as on the card" required /><br/><br/>

</li>

<li>

<label for=expiry>Expiry Date</label>

<input id="date" name=date type="date"><br/><br/>

</li>

<li>

<label for=secure>CVV</label>

<input id=secure method=post maxlength=3 name=secure type=password required /><br/><br/>

</li>

<li>

<label for=amount>AMOUNT</label>

<input id=amount name=amount type=number required /><br/><br/>

</li>

</ul>

</fieldset>

</fieldset>

<center>

<input type="submit" value="submit" name="submit" class="btn btn-lg btn-default" />

</center>

</form>

</body>

</html>

* **Code for Logout:**

<html>

<head>

<title>|| MOBILE PAYMENT SYSTEM||</title>

<link href="default.css" rel="stylesheet" type="text/css" />

<style type="text/css">

</style></head>

<body>

<div id="header">

<table width="200" align="center">

<tr>

<td height="112"><img src="1.jpg" alt="" width="500" height="195" /></td>

</tr>

</table>

</div>

<div id="footer">

<p style="text-align:center; color:#FF0000;"><strong><marquee behaviour="scroll">YOU HAVE SUCCESSFULLY LOGGED OUT</marquee></strong></p>

<p class="style1"><a href="home.html"><img src="th.jpg" alt="" width="486" height="243" /></a></p>

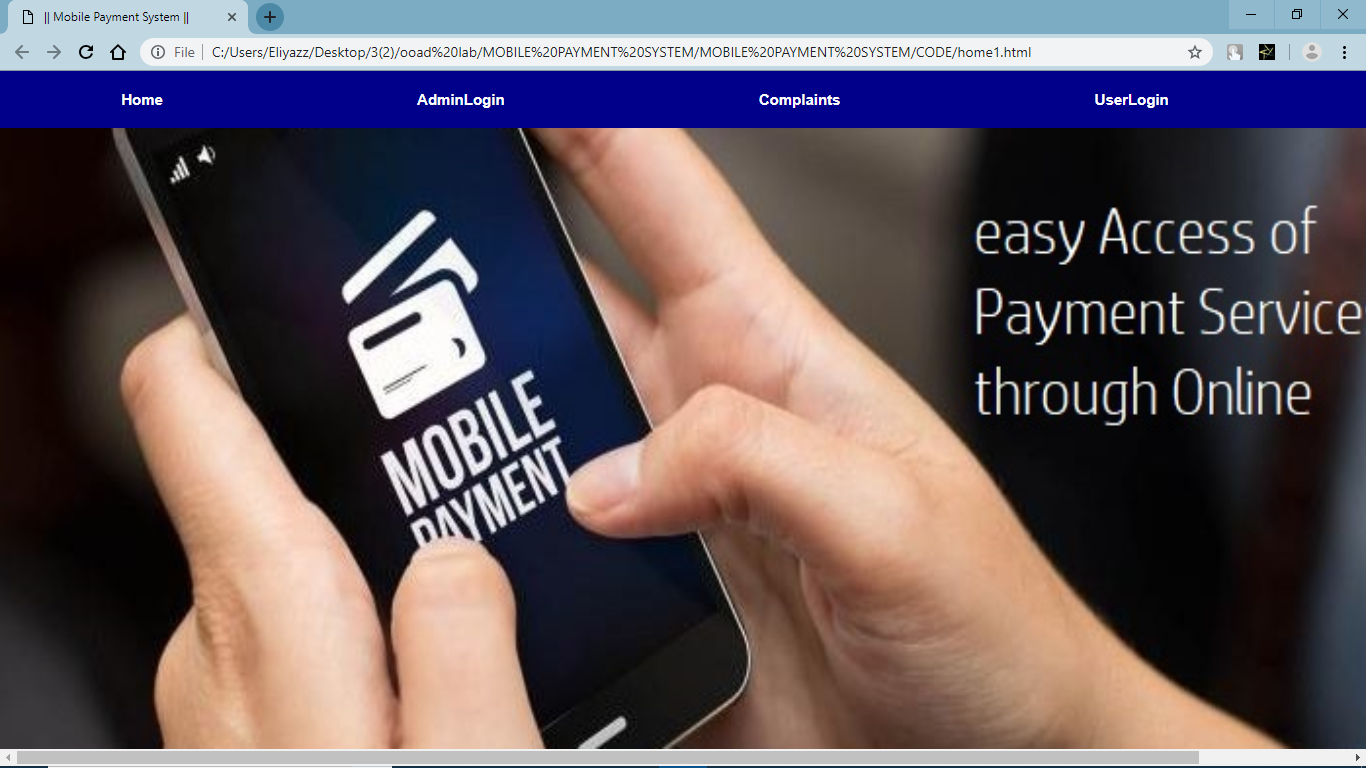
</div>

</body>

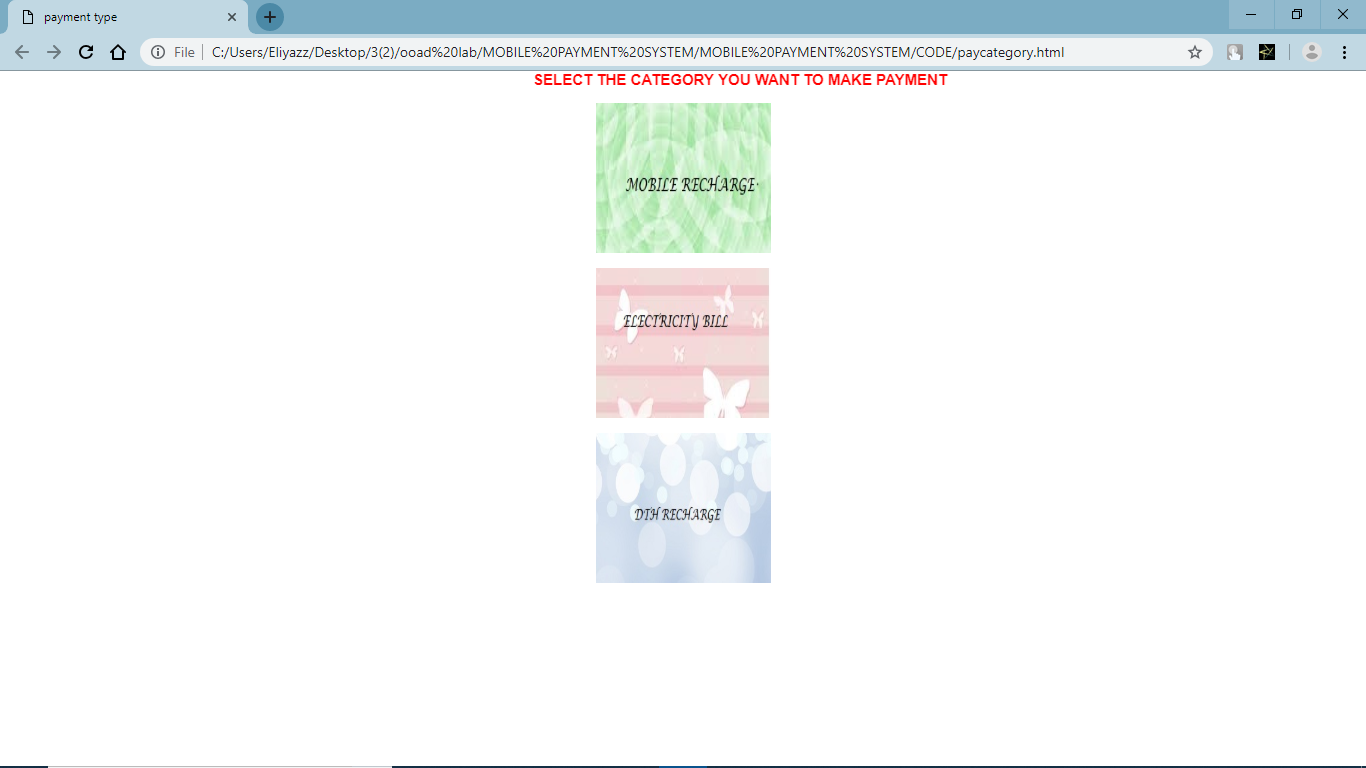
</html>

**19.d.** **IMPLEMENTATION OF USER INTERFACE LAYER**

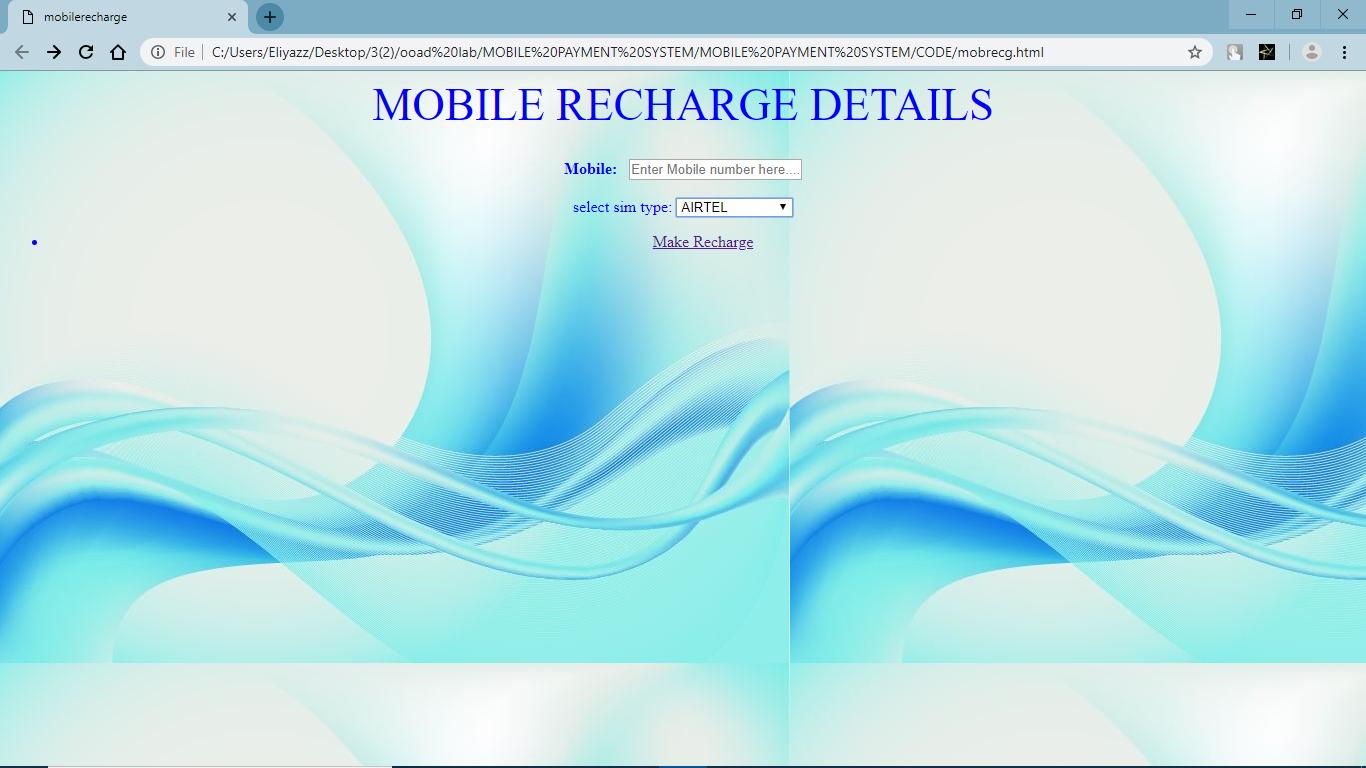
**Mobile Payment System Home Page:**



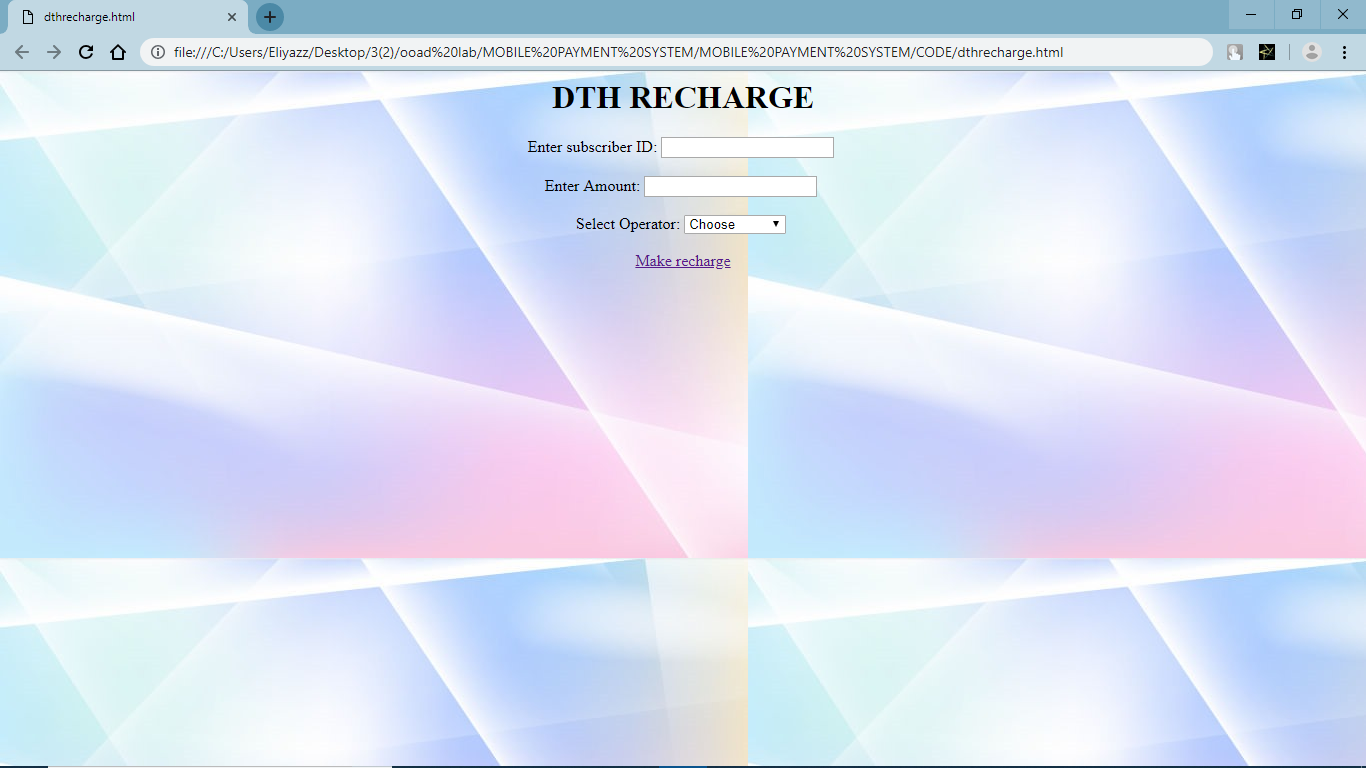
**User selects one of the following categories:**



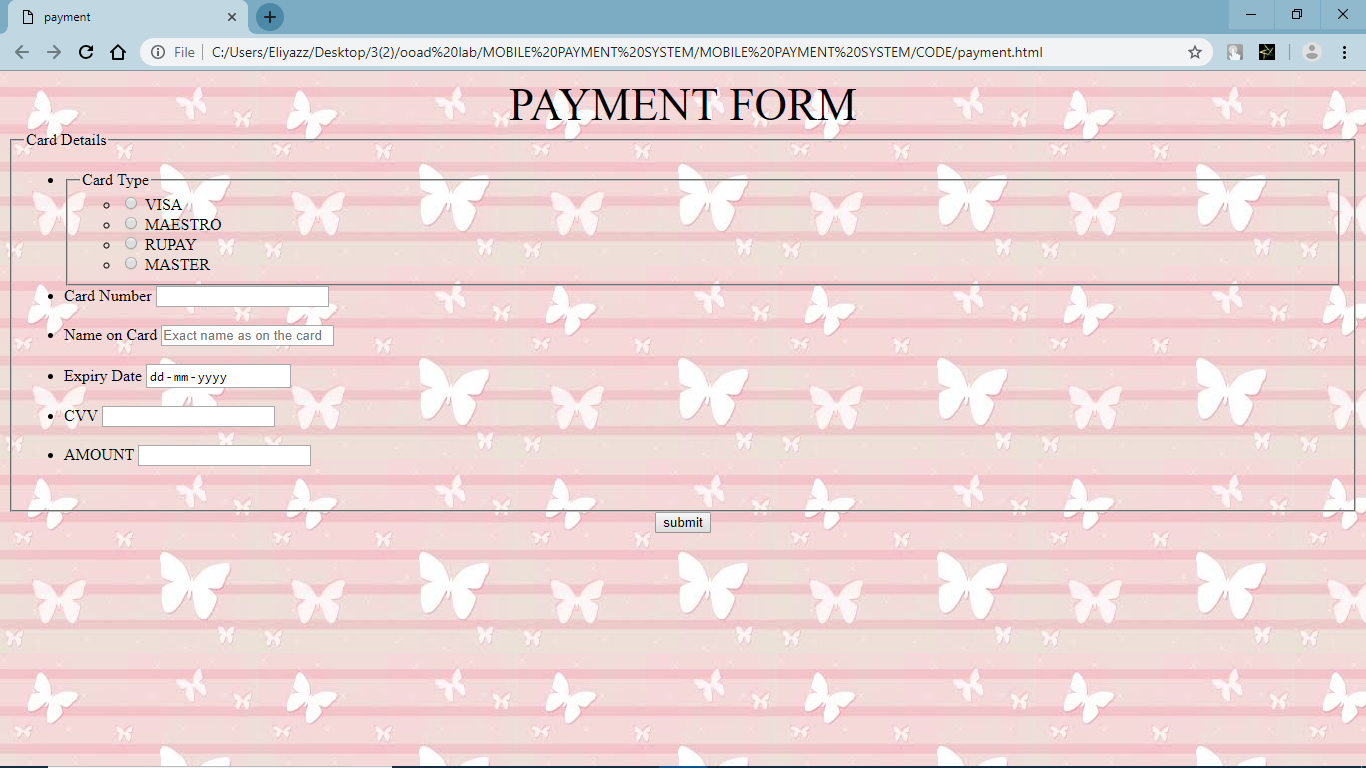
**If user select the category Mobile Recharge:**



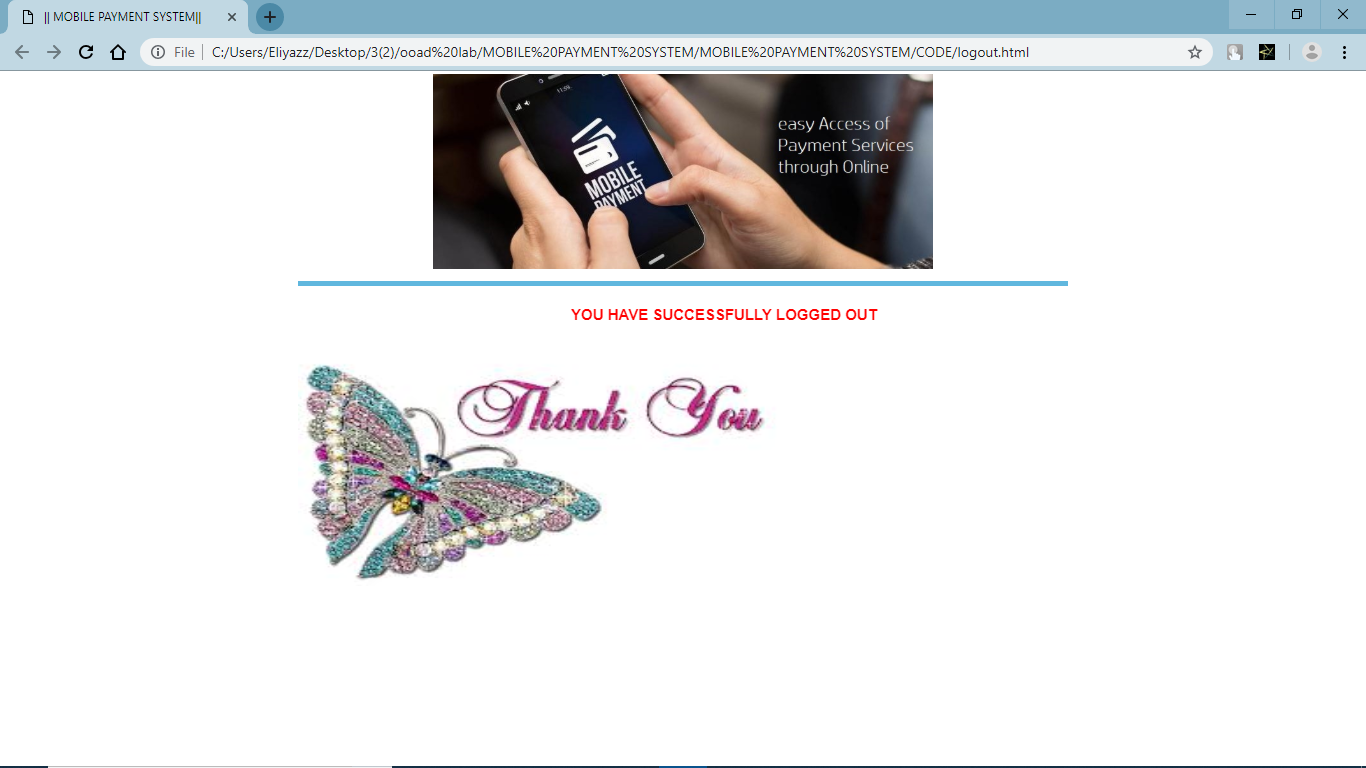
**If user select the category DTH Recharge:**



**Payment Form:**



**Logout Page:**



**20. CONCLUSION:**

The new mobile payment system maintains user payment details, updates the status of the payment. The system enables the customer to use various payment category such as mobile recharge, electricity bill, DTH recharge.

The system displays all the forms related to the category of payment.

The new system is an improvement on manual bidding process in the following ways,

* No need for physical appearance for payment.
* People can pay anything related to various categories from a computer connected to internet from home.
* Reduces strain for individuals.
* easy to use and user friendly.
* Issue only with security, but with only few websites.

**21.REFERENCES:**

1. Object Oriented Analysis and Design – Simon Bennet
2. Visual modeling with Rational Rose 2002 – Terry Quartan
3. Object Oriented systems Development - Ali Bahram
4. The Unified Modelling Language User Guide by Grady Booch, James Rumbaugh, Ivar Jacobson

**Web Links:**

**1.** [www.w3schools.com/php](http://www.w3schools.com/php)

**2.**  <https://stackoverflow.com>

**3.** [C:/Users/Eliyazz/Desktop/home1.html](file:///C:\Users\Eliyazz\Desktop\home1.html)