**A CASE STUDY ON**

**Wedding Planner Management System**

**SUBMITTED BY:**

**Abdul Rahman**

1604-20-733-101

**CSE –B V-SEM**

**COMPUTER SCIENCE AND ENGINEERING DEPARTMENT**

**CERTIFICATE**

This is to certify that **\_\_\_\_\_\_\_\_\_\_\_\_\_\_** bearing the Roll No. **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** has completed the course of programs in the **SOFTWARE ENGINEERING LAB (PC 651 CS )** prescribed by Osmania University for **B.E V- SEM- CSE** in the academic year **2022-23.**

**INTERNAL EXAMINER EXTERNAL EXAMINER**

**Table Of Contents**

|  |  |  |
| --- | --- | --- |
| **S.No:** | **Topic:** | **Page No:** |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | ABSTRACT  SCOPE OF THE SYSYTEM  INTRODUCTION  UML  Rational Software Architect  LITERATURE SURVEY  MODULE SPECIFICATION  SOFTWARE REQUIREMENT SPECIFICATION  DATA DICTIONARY  UML DIAGRAMS  TEST DATA GENERATION  VERIFICATION  COST ESTIMATION & RESOURCE ESTIMATION  USER MANUAL  CONCLUSION  REFERENCES/BIBILOGRAPGHY |  |

**1.ABSTRACT**

The purpose of Online Wedding Planners is to automate the existing manual system by the help of computerized equipment and full-fledged computer software fulfilling their requirements, so that their valuable data/information can be stored for a longer period with easy accessing and manipulation of the same. The required software and hardware are easily available and easy to work with.

Online Wedding planners, as described above, can lead to error free, secure, reliable and fast management system. It can assist the user to concentrate on their other activities rather to concentrate on the record keeping. Thus it will help organization in better utilization of resources. The organization can maintain computerized records without redundant entries. That means that one need not be distracted by information that is not relevant, while being able to reach the information.

The aim is to automate it’s existing manual system by the help of computerized equipments and full-fledged computer software, fulfilling their requirements, so that their valuable data/information can be stored for a longer period with easy accessing and manipulation of the same. Basically the project describes how to manage for good performance and better services for the clients.

**2.Scope of the System**

**The various actors in the system are** :

* System User
* Super Admin
* Agents
* Customers

The role of Admin is to manage the users and staff , manage login logout, Change account password etc.

The role of System User is to make payment , login logout of the system etc.

**Scope of the system for wedding planner management system:**

* The scope of a wedding planner management system typically includes the following features:
* Client management: tracking client information, contacts, preferences, budgets, and event details.
* Vendor management: managing vendors, their services, prices, contracts and schedules.
* Task management: creating and tracking tasks, deadlines and appointments.
* Budget tracking: keeping track of expenses and budget allocation.
* Guest management: tracking guest information, RSVPs, seating arrangements, and meal choices.
* Schedule management: creating and managing event schedules, timelines, and vendor schedules.
* Communication management: tracking communication with clients, vendors, and guests.
* Document management: storing and organizing important documents related to the event.
* Reporting and analytics: generating reports on event details, budget, and vendor performance.

**Drawbacks of wedding planner management system:**

1. Complexity: UML diagrams can be complex and difficult to understand, especially for non-technical users.
2. Limitations: UML may not be able to fully capture all aspects of the system, especially when it comes to dynamic behavior.
3. Maintenance: UML diagrams need to be constantly updated and maintained to reflect changes in the system, which can be time-consuming and error-prone.
4. Over-documentation: UML diagrams can become too detailed, leading to over-documentation and making the diagrams difficult to read and understand.
5. Integration: UML diagrams may not be easily integrated with other tools and systems, such as project management or development tools.
6. Cost: Creating and maintaining UML diagrams can be costly, both in terms of time and resources.

**3. INTRODUCTION TO UML**

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object oriented software’s and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

***Goals of UML:***

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.

2. Provide extensibility and specialization mechanisms to extend the core concepts.

3. Be independent of particular programming languages and development processes.

4. Provide a formal basis for understanding the modeling language.

5. Encourage the growth of the OO tools market.

6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.

7. Integrate best practices.

**UML : (BASIC BUILDING BLOCKS)**

**Actor:**

Actors represent system users. They help delimit the system and give a clearer picture of what the system should do. It is important to note that an actor interacts with, but has no control over the use cases.

An actor is someone or something that:

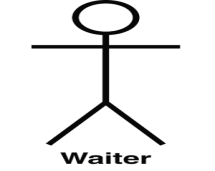
* Interacts with or uses the system
* Provides input to and receives information from the system
* Is external to the system and has no control over the use cases

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

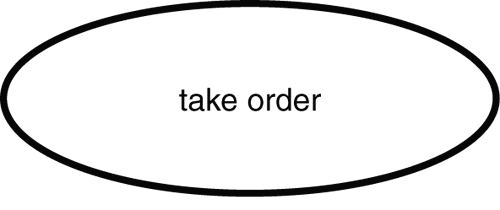
The needs of the actor are used to develop use cases. This insures that the system will be what the user expected.

An actor is something or someone outside the system that interacts directly with it-typically, a user of the system. An example of an actor is shown below.



### Use Case/Collaboration:

A use case is a sequence of interactions by an actor with the system, which yields observable value to the actor. According to UML, use cases are made to fall into a category of type of class. The UML calls it a "behavioral thing”. An example of a use case is shown below.



Collaboration is a collection of objects that interact to implement behavior. Typically, collaboration can be used to specify the realization of a use case or an operation. Collaboration can also be used to specify a software pattern, and a parameterized collaboration (that is, one with abstract participants that are replaced when the pattern is used) can specify an architectural pattern.

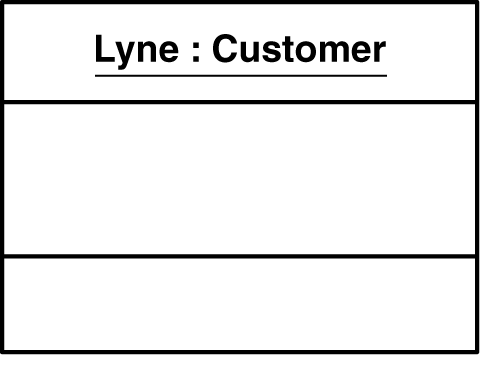
### Class/Object:

A class is an abstraction of a set of possible objects that share the same attributes, operations, methods, relationships, and semantics.

class

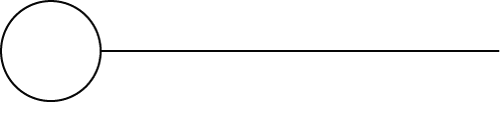
A class may use a set of interfaces to specify collections of operations it provides to its environment.

An object is an instance of a class or an example of a type-the same symbol as class, but with the name underlined. An example is shown below.



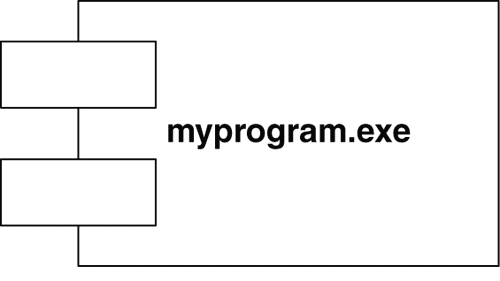
### Interface:

An interface describes the visible operations of a class, component, or package. It defines the services available from the implementing element.



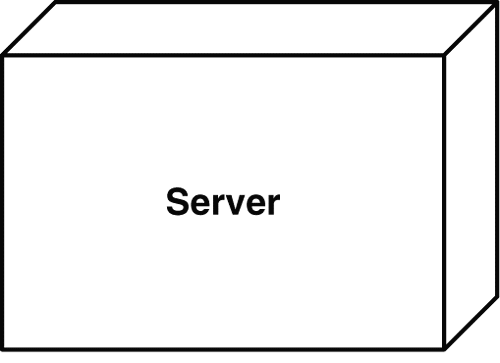
### Component:

A component is a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary, or executable) or equivalents such as scripts or command files. An example is shown below.



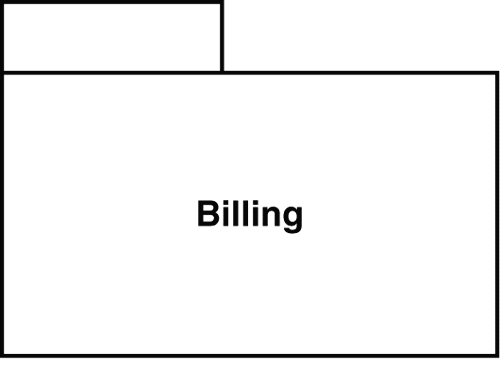
### Node:

A node represents a processing resource that exists at run time, with at least a memory and often processing capability as well. Nodes comprise computing devices and any other physical resources used in a system, such as people or machines. An example is shown below.



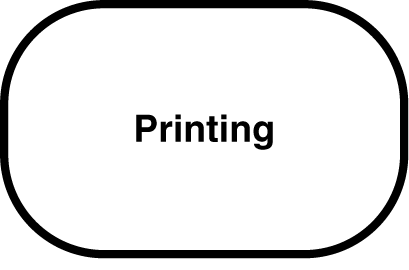
**Package:**

A package is a UML container used to organize model elements. An example of a package is shown below.



### State:

A state is the condition, status, or situation of an object as part of its lifecycle and/or as the result of an interaction. A state may also be used to model an ongoing activity. An example of a state is shown below.



### Note:

A note is used to provide explanatory text, such as comments in a model. A note can also be used in specific ways, depending on the modeling dialect you adopt.note.jpg

**Relationships:**

Relationships are used in structural models to show semantic connections between model elements.

**Dependency:** A dependency is a "using" relationship, one in which the connection between two things means that if one changes, it affects the other. Dependencies can be used to identify connections between varieties of model elements, packages being a notable example. These are unidirectional relationships.

generalization

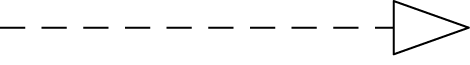
**Generalization:** A generalization is a relation between two elements, in which one is a more general form of the other. Class inheritance is represented this way, but generalization can be used more generally.

dependancy

**Association:** An association is a structural relationship, mapping one object to another set of objects. It is also used to identify the communication path between an actor and a use case.

An

**Realization:** A realization is a type of dependency relationship that identifies a contractual link between elements—a realizing element. For example, a class implements the behaviors in a specifying element; in this case, it is an interface. A realization also links use cases and collaborations.



**Aggregation:** An aggregation represents a whole-part relationship. This contrasts with a plain association, which shows a relationship among/between peers, depending on the number. In an aggregation, one element is the whole and the other(s) are the parts.

aggregation

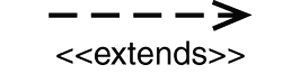
**Composition:** A composition is an aggregation that has strong ownership of its parts. Therefore, if the whole element disappears, the parts do, too.

composition

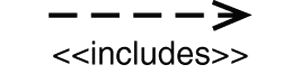
### Dependencies:

Dependency relationships are frequently stereotyped in the UML to support the needs of particular types of diagrams or model elements. The following are some examples:

**Extends** provides a way of handling behavior that is optional in a use case The optional behavior is packaged in an extending use case and connected via an <<extends>> dependency.



**Include** provides a way of handling behavior that is common to a number of use cases. The optional behavior is factored out, packaged in an included use case, and connected via an <<includes>> dependency.



**Diagrams:**

A diagram is a graphical presentation of a set of elements.

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in modeling tools include:

**Use Case Diagram** displays the relationship among actors and use cases.

**Class Diagram** models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.

**Interaction Diagrams**

**Sequence Diagram** displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).

**Communication Diagram** displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.

**State Diagram** displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.

**Activity Diagram** displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.

**Physical Diagrams**

• **Component Diagram** displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.

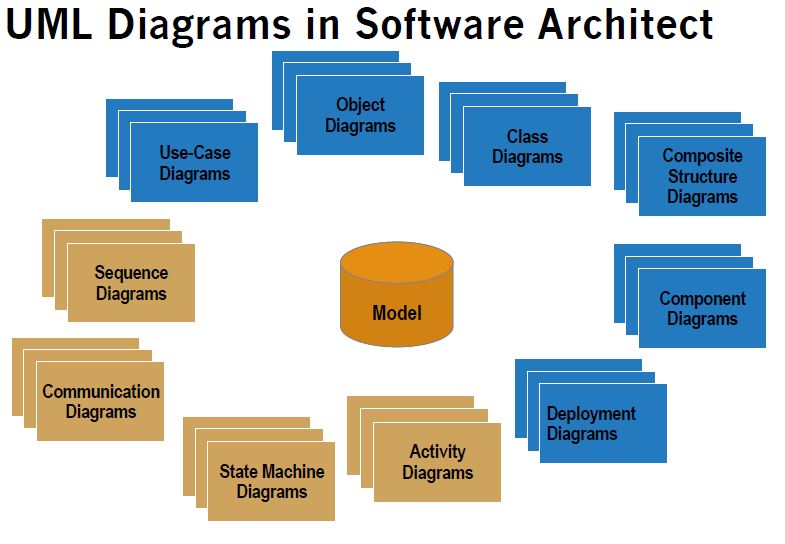
• **Deployment Diagram** displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

**Introduction to Rational Software Architect:**

Rational Software Architect is a tool that enables software architects to model and design the architecture of their applications. The content that can be created within Rational Software Architect includes all kinds of UML 2.2 diagrams. It also includes features for automatic code generation starting from the models developed within the application.

All content created in Rational Software Architect can be published to HTML and deployed to Web servers for distributed viewing.

Rational Software Architect can also be connected to a number of other Rational lifecycle process tools in order to be fully used into the software process



**4. Literature Survey**

**Diagrams:**

1. A diagram is a view into a model
   1. Presented from the aspect of a particular stakeholder
   2. Provides a partial representation of the system
   3. Is semantically consistent with other views
2. There are five types of views
   1. **Static views**: use case, class, object, component, deployment
   2. **Dynamic views**: sequence, collaboration, state chart, activity
   3. **Implementation view**: component diagrams
   4. **Deployment view:** Deployment diagram

**Use Case Diagrams**

A use case is a set of scenarios that describing an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors. An actor is represents a user or another system that will interact with the system you are modeling. A use case is an external view of the system that represents some action the user might perform in order to complete a task.



**Class Diagrams**

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

Classes are composed of three things: a name, attributes, and operations.

Class diagrams also display relationships such as containment, inheritance, associations and

others.Below is an example of an associative relationship:



The association relationship is the most common relationship in a class diagram. The

association shows the relationship between instances of classes. For example, the class Order

is associated with the class Customer. The multiplicity of the association denotes the number

of objects that can participate in then relationship .For example, an Order object can be

associated to only one customer, but a customer can be associated to many orders.

Another common relationship in class diagrams is a generalization. A generalization is used

when two classes are similar, but have some differences. Look at the generalization below:



**Interaction Diagrams**

Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are **sequence** and **communication** diagrams.

Interaction diagrams are used when you want to model the behavior of several objects in a use case. They demonstrate how the objects collaborate for the behavior. Interaction diagrams do not give a in depth representation of the behavior. If you want to see what a specific object is doing for several use cases use a state diagram. To see a particular behavior over many use cases or threads use an activity diagrams.

Sequence diagrams, communication diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Communication diagrams demonstrate how objects are statically connected.

Both diagrams are relatively simple to draw and contain similar elements.

**Sequence diagrams:**

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. the diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.



Below is a slightly more complex example. The light blue vertical rectangles the objects activation while the green vertical dashed lines represent the life of the object. The green vertical rectangles represent when a particular object has control. The represents when the object is destroyed. This diagrams also shows conditions for messages to be sent to other object. The condition is listed between brackets next to the message. For example, a [condition] has to be met before the object of class 2 can send a message() to the object of class 3.



**Communication diagrams:**

Communication diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML. A simple 1, 2, 3... format can be used, as the example below shows, or for more detailed and complex diagrams a 1, 1.1 ,1.2, 1.2.1... scheme can be used.



**State Diagrams**

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system. State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state.



All state diagrams begin with an initial state of the object. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.



**Activity Diagrams**

Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel Activity diagrams show the flow of activities through the system. Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities. A fork is used when multiple activities are occurring at the same time. The diagram below shows a fork after activity1. This indicates that both activity2 and activity3 are occurring at the same time. After activity2 there is a branch. The branch describes what activities will take place based on a set of conditions. All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch. After the merge all of the parallel activities must be combined by a join before transitioning into the final activity state.



**Physical Diagrams**

There are two common types of physical diagrams: **deployment diagrams** and **component diagrams.** Deployment diagrams show the physical relationship between hardware and software in a system. Component diagrams show the software components of a system and how they are related to each other. These relationships are called dependencies.

The deployment diagram contains nodes and connections. A node usually represents a piece of hardware in the system. A connection depicts the communication path used by the hardware to communicate and usually indicates a method such as TCP/IP.



The component diagram contains components and dependencies. Components represent the physical packaging of a module of code. The dependencies between the components show how changes made to one component may affect the other components in the system. Dependencies in a component diagram are represented by a dashed line between two or more components. Component diagrams can also show the interfaces used by the components to communicate to each other.

The combined deployment and component diagram below gives a high level physical description of the completed system. The diagram shows two nodes which represent two machines communicating through TCP/IP. Component2 is dependent on component1, so changes to component 2 could affect component1. The diagram also depicts component3 interfacing with component1. This diagram gives the reader a quick overall view of the entire system.



**5. Module Specification**

**A module specification for a wedding planner management system in UML(Unified Modeling Language) may include the following elements:**

* Class diagram - to model the system's object-oriented design and the relationships between classes. Each module should be represented as a class with attributes and methods that correspond to the module's functionality.
* Package diagram - to organize the system into related packages and show the dependencies between them. Each module should be placed in its own package, and the relationships between packages should reflect the dependencies between modules.
* Sequence diagram - to visualize the interactions between objects and the order in which they occur. Each module's behavior should be captured in a sequence diagram to demonstrate how it interacts with other modules.

**The modules in a wedding planner management system may include:**

1. User management - to handle user authentication and authorization.
2. Venue management - to manage the booking of wedding venues.
3. Vendor management - to manage the contracts and payments with vendors.
4. Planning management - to create and track wedding to-do lists and timelines.
5. Integration management - to integrate with email and calendar applications.
6. Reporting and analytics - to provide data analysis and reporting capabilities.

Each module should have well-defined interfaces, input, and output, and should be designed to be modular, reusable, and scalable. The UML diagrams should also reflect the performance, security, and accessibility requirements of the system**.**

**6. Software Requirement Specification**

**A software requirement specification for a wedding planner management system in UML (Unified Modeling Language) may include the following elements:**

1. Use case diagram - to represent the interactions between the system and the users.
2. Class diagram - to model the system's object-oriented design and the relationships between classes.
3. Activity diagram - to show the flow of activities and decision points in the system.
4. Sequence diagram - to visualize the interactions between objects and the order in which they occur.
5. State machine diagram - to model the states and transitions of the system's objects.

**These UML diagrams should capture the functional and non-functional requirements of the wedding planner management system, such as:**

* User authentication and authorization.
* Management of wedding venue bookings.
* Management of vendor contracts and payments.
* Creation and tracking of wedding to-do lists and timelines.
* Integration with email and calendar applications.
* Reporting and data analytics capabilities.

The UML diagrams can be complemented by a written document that provides a detailed description of the requirements, constraints, and design decisions for the wedding planner management system.

**7. DATA DICTIONARY**

**A data dictionary in a wedding planner management system typically includes the**

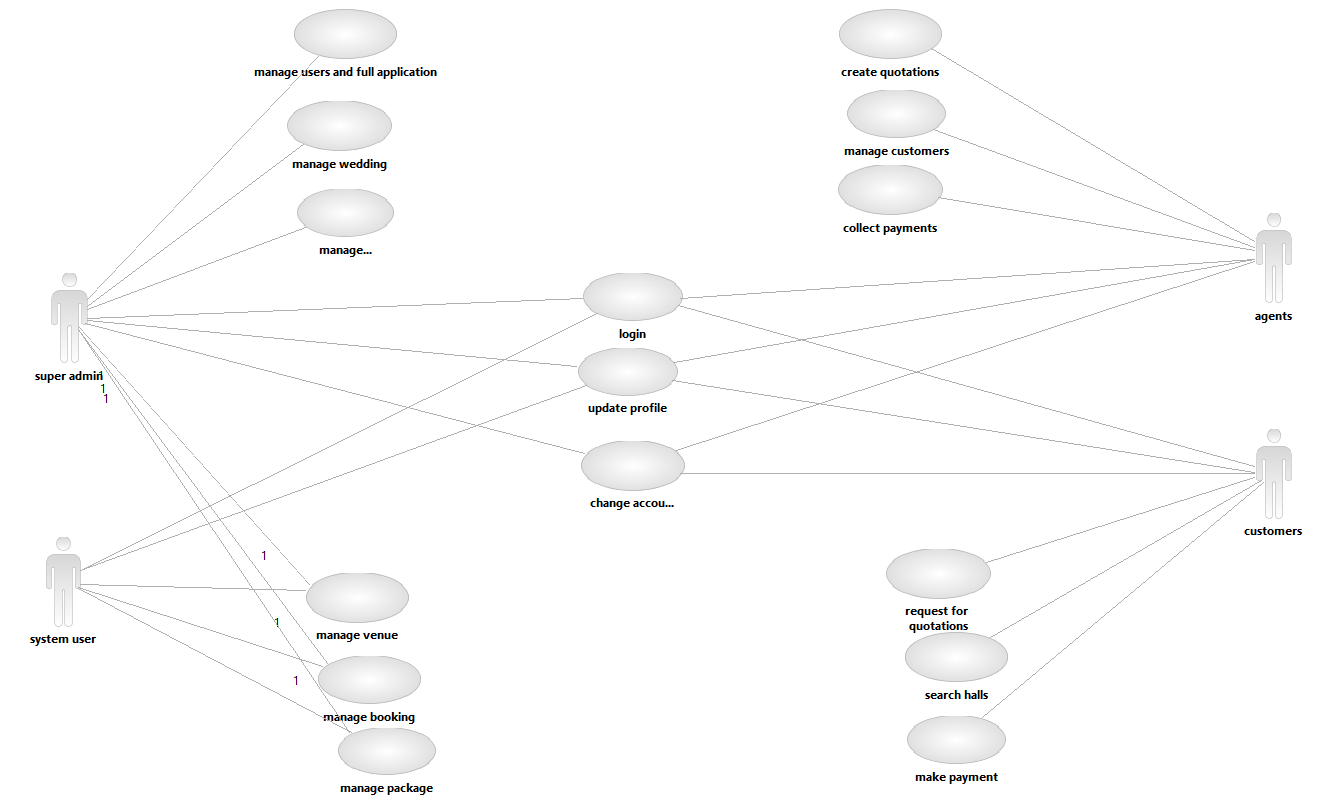
**following information:**

* Client Information: Name, contact details, wedding date, budget, venue, guest count, etc.
* Vendor Information: Name, contact details, services offered, pricing, portfolio, etc.
* Task Management: To-do list items, deadlines, and status updates for various tasks related to the wedding planning process.
* Budget Tracker: A record of all the expenses incurred for various wedding-related items such as venue rental, catering, decorations, etc.
* Guest List: Names, contact details, RSVP status, and meal preferences of guests invited to the wedding.
* Appointments: Records of appointments with vendors, clients, and other stakeholders.
* Contracts: Copies of contracts signed with vendors, venue, etc.
* Notes: Miscellaneous notes and information relevant to the planning of the wedding.
* Calendar: A visual representation of deadlines, appointments, and other important events related to the wedding planning process.
* Communication Log: A record of all communication between the wedding planner, clients, vendors, and other stakeholders

**8. UML DIAGRAMS**

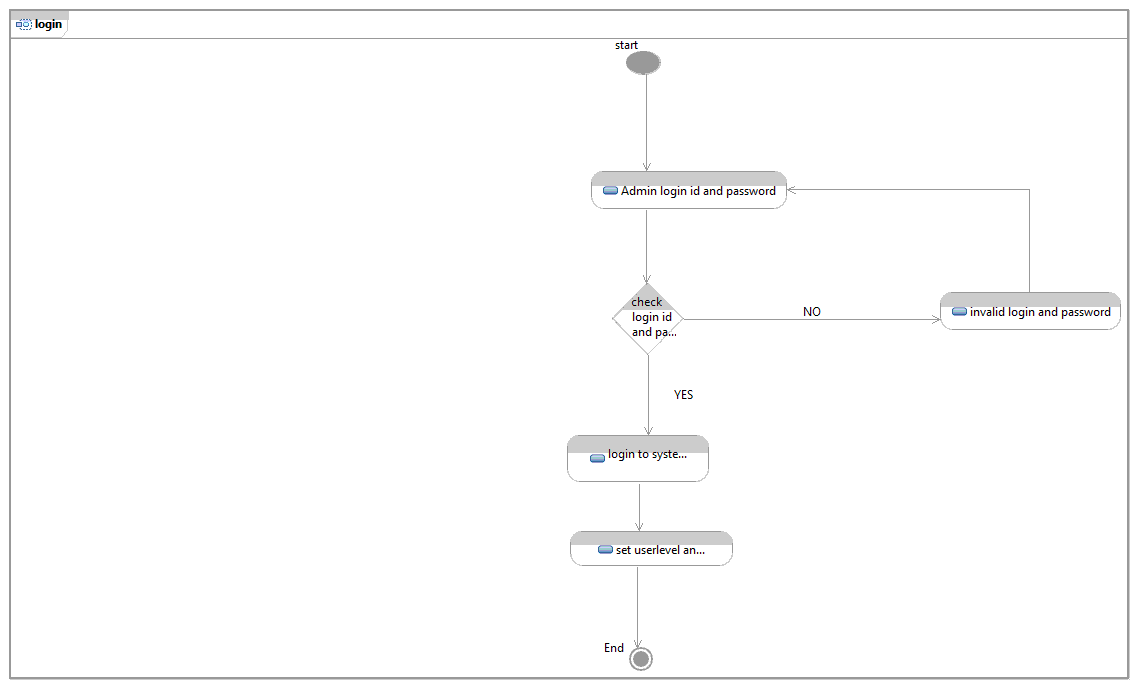
**1) USE-CASE DIAGRAM:**

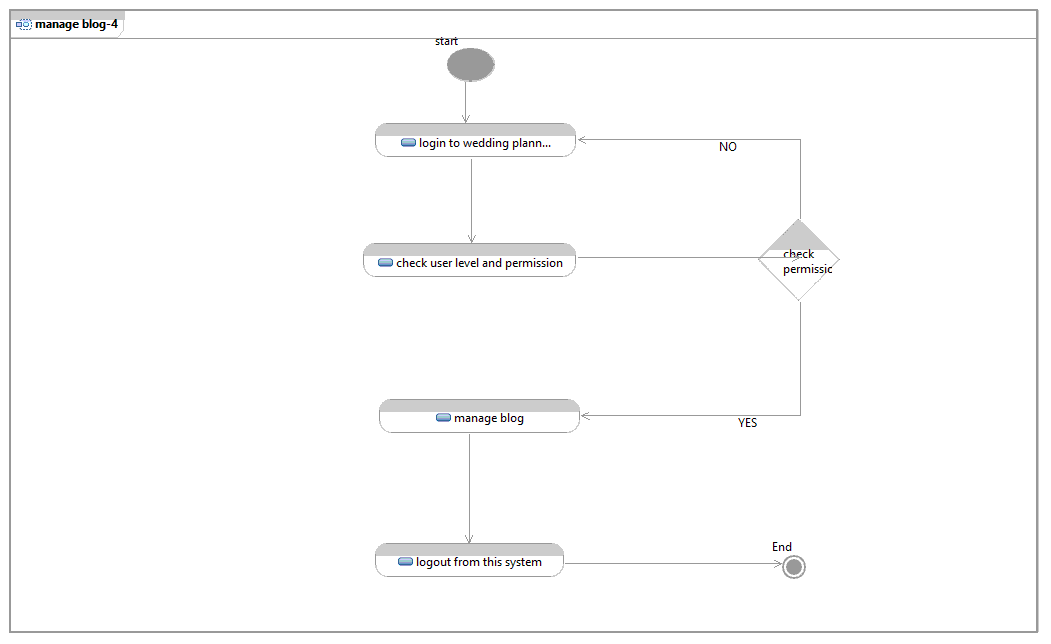
A use case diagram is a visual representation of the interactions between a systemand its users. It depicts the relationships between actors and use cases and helps to identify the functionality of a system from the users' perspective. It is a key component of the Unified Modeling Language(UML) and is used in software engineering to define the requirements of a system.

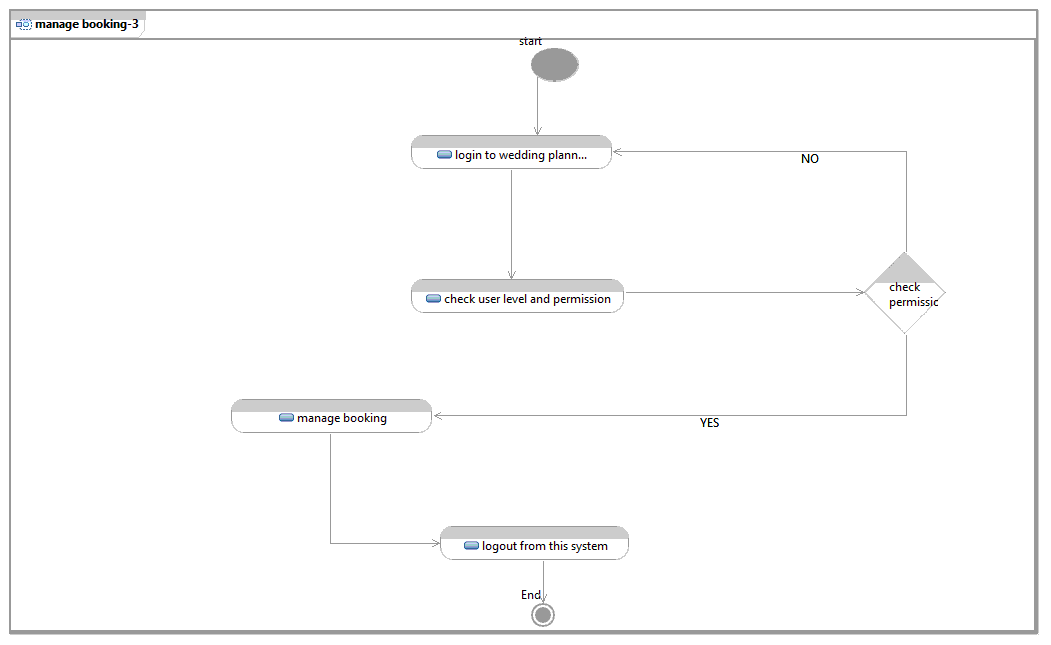
****

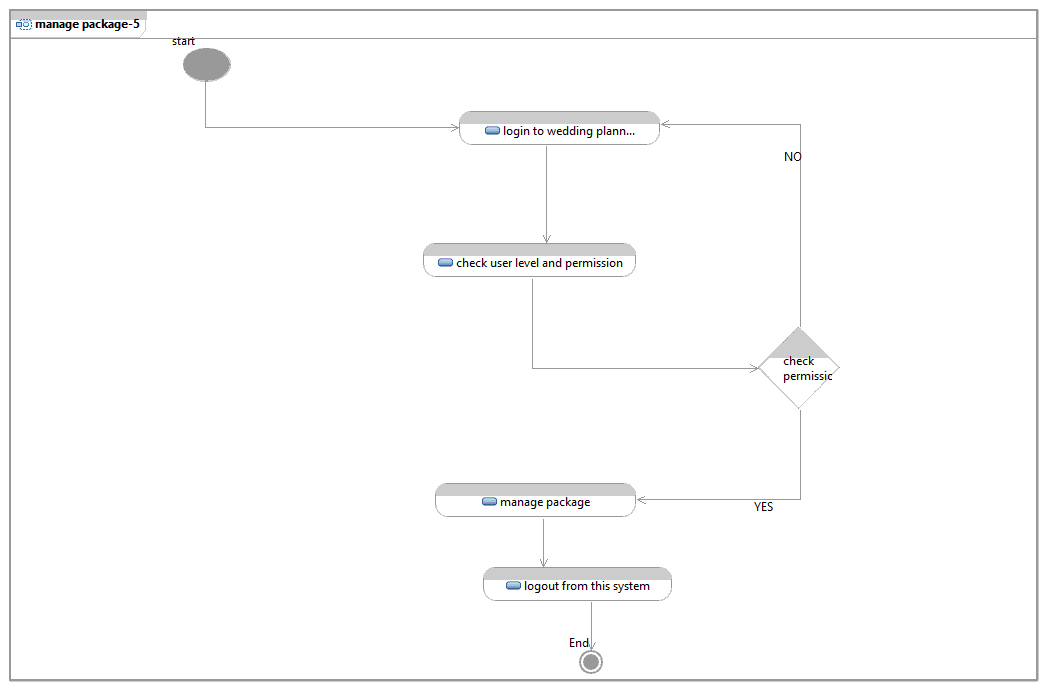
**2)ACTIVITY DIAGRAMS:**

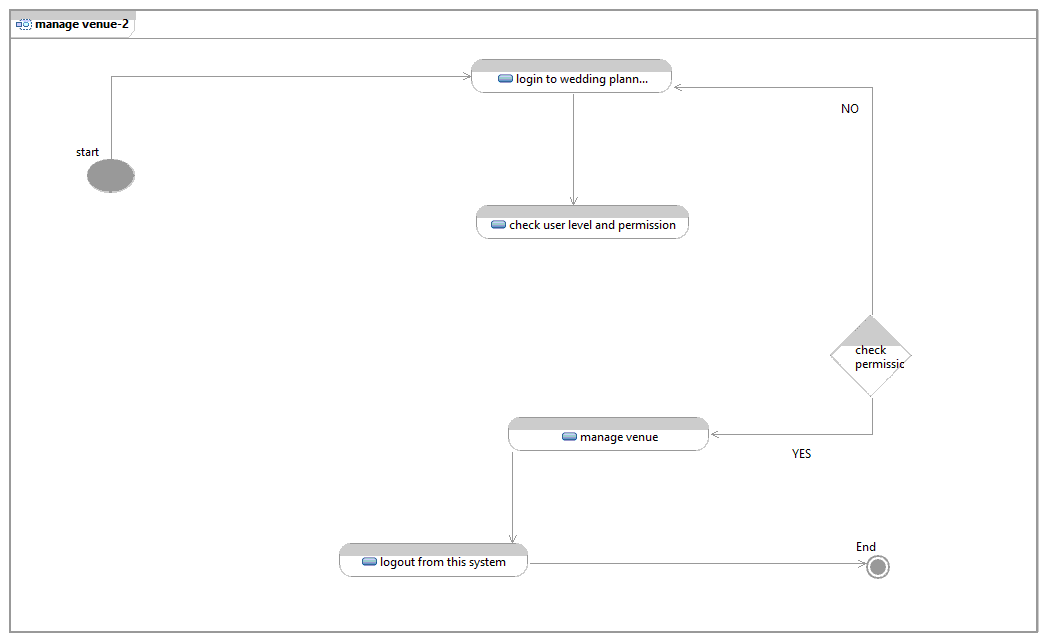
Activity diagrams are a type of flowchart that illustrate the flow of control in a system. They show the interactions between objects and the order of activities, including parallel and conditional branching, to represent the process flow from start to finish. Activity diagrams are used to model the behavior of a system and are commonly used in software development and business process modeling.

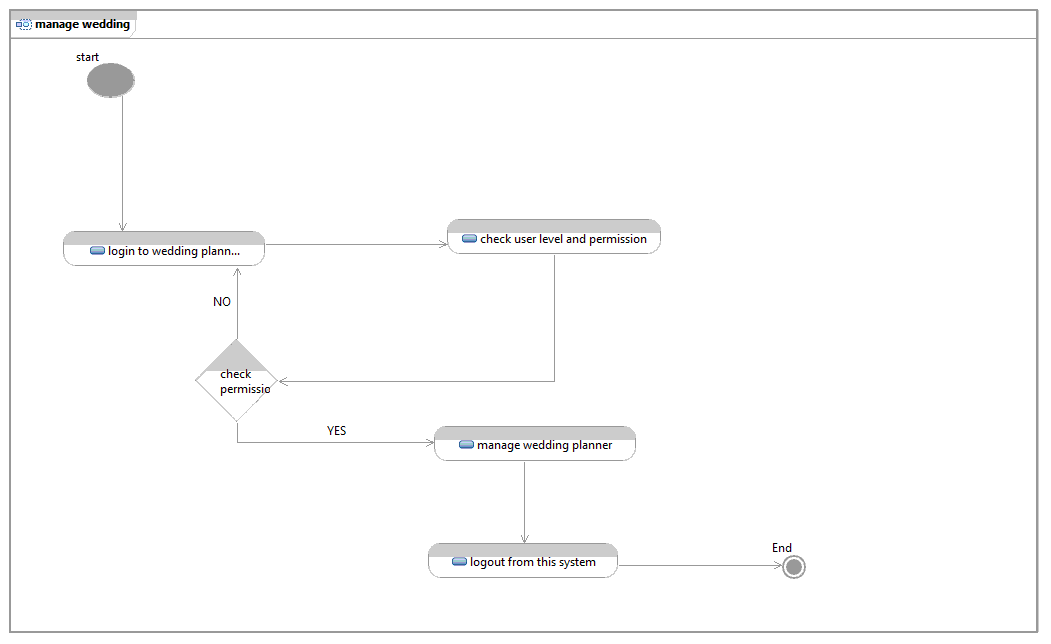
****

****

****

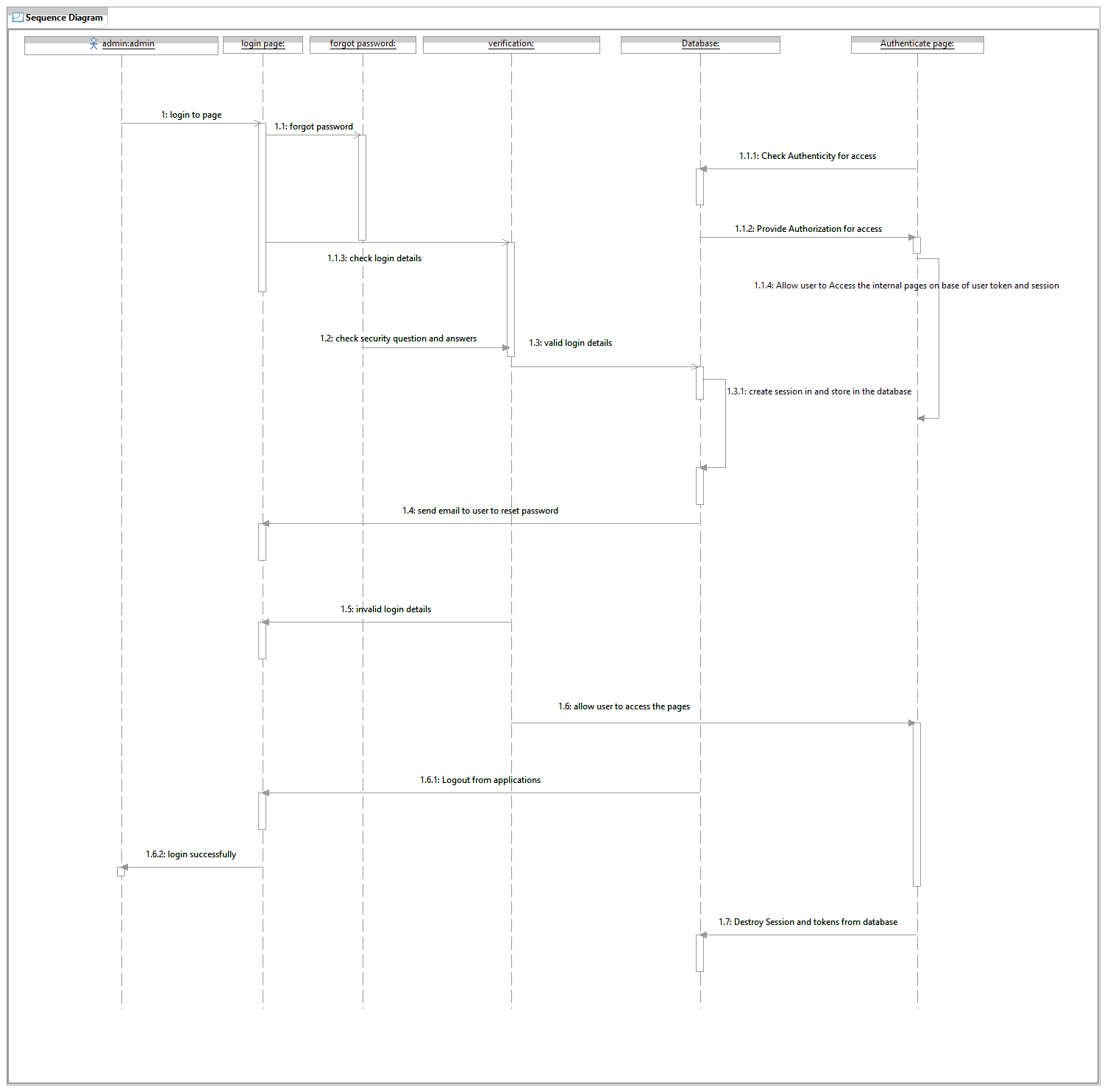
****

****

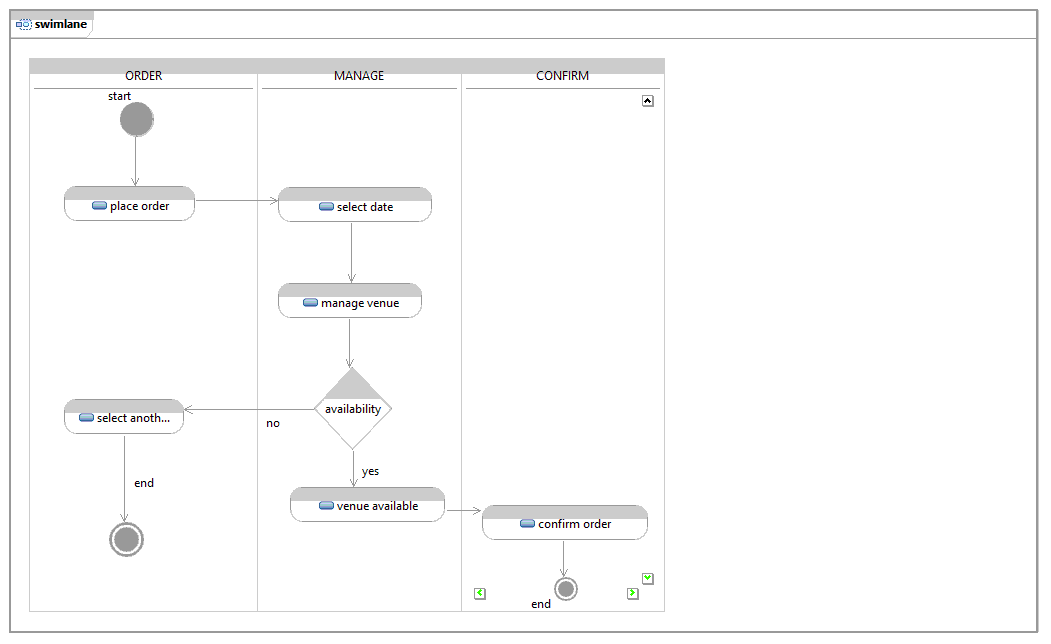
****

**3)Sequence Diagram:**

A sequence diagram is a type of UML diagram that shows the interactions between objects or components in a system over time, in a sequential order. It represents the flow of messages between objects and helps to visualize the order and timing of interactions between objects in a system.

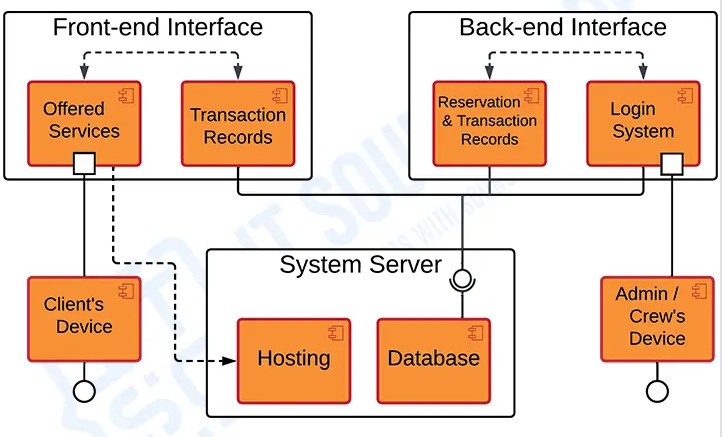
****

**4)Swimlane Diagrams:**

A swimlane diagram is a flowchart that visually separates processes into different lanes, each representing a specific functional unit or department involved in the process. It helps to visualize the flow of tasks and responsibilities among different groups, promoting better understanding and coordination****

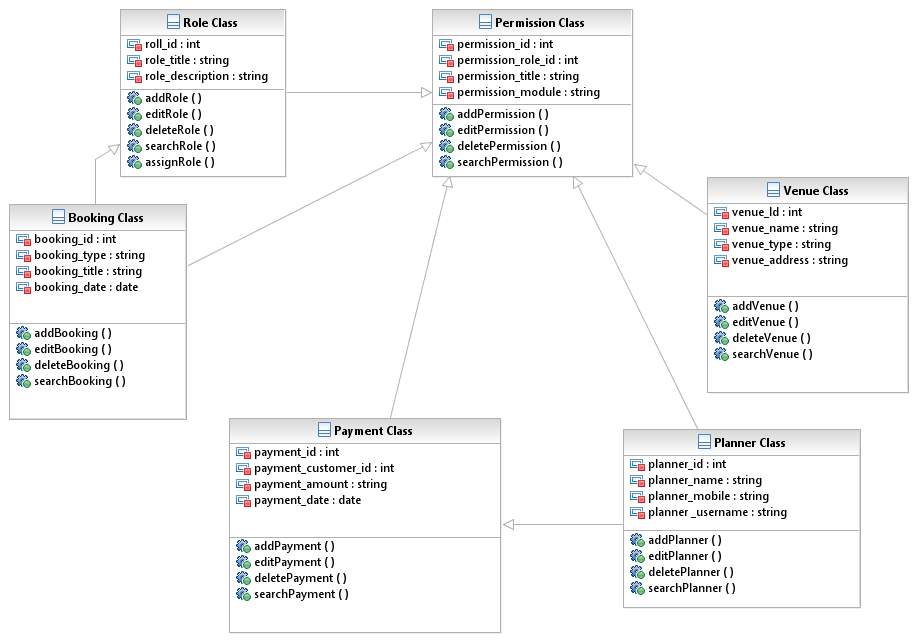
**4)Component Diagram:**

A component diagram is a visual representation of the components and their relationships in a software system. It shows the dependencies and interactions between different components, helping to understand and organize the structure of the system. Component diagrams are often used in software development to help clarify and simplify the design of complex systems.

****

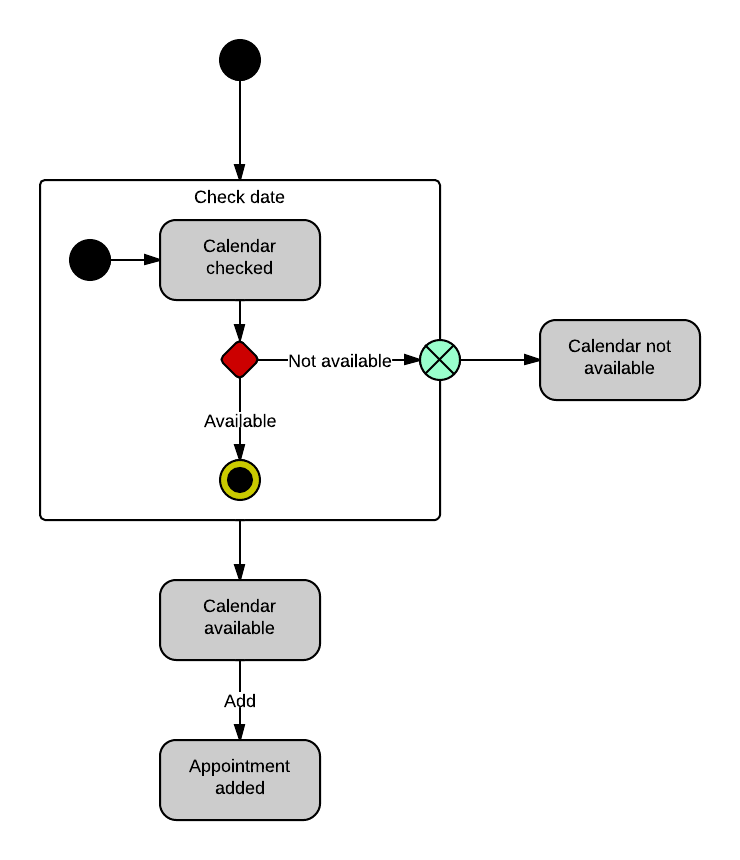
**5)Class Diagram:**

A class diagram is a visual representation of a software system's classes, objects, and their relationships, including inheritance, composition, and association. It helps to define the structure of a system, identify object interactions, and understand the relationships between objects. Class diagrams are used in object-oriented software design and modeling to depict the design of the system.

****

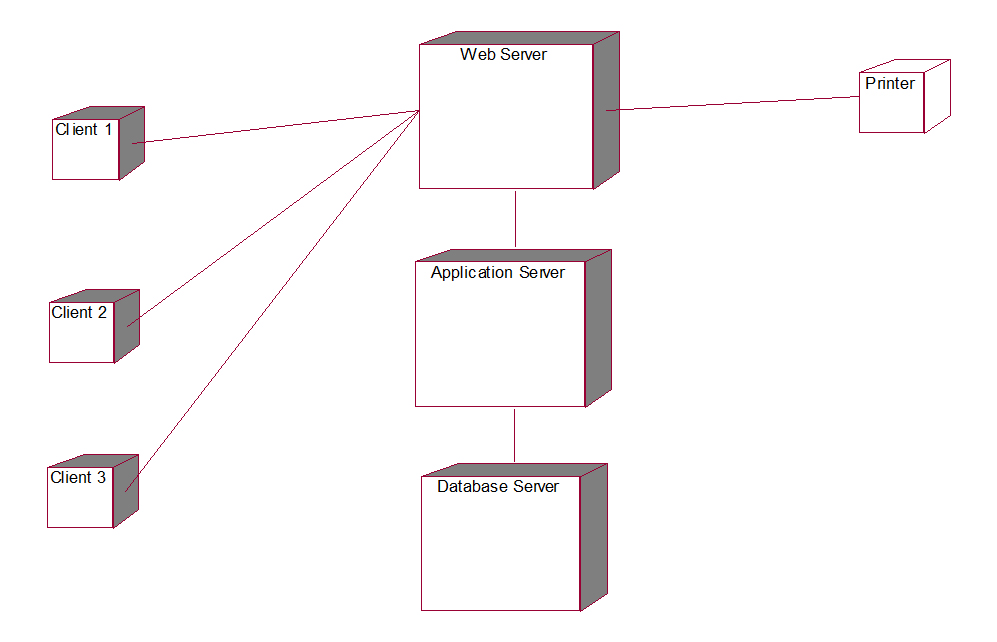
**6)State Diagram:**

A state diagram is a type of diagram used in computer science and related fields to describe the behavior of a system. It represents the various states that a system can be in and the transitions between those states, triggered by events or user actions. The state diagram helps to visualize the behavior and interactions of the system, making it easier to understand and design

****

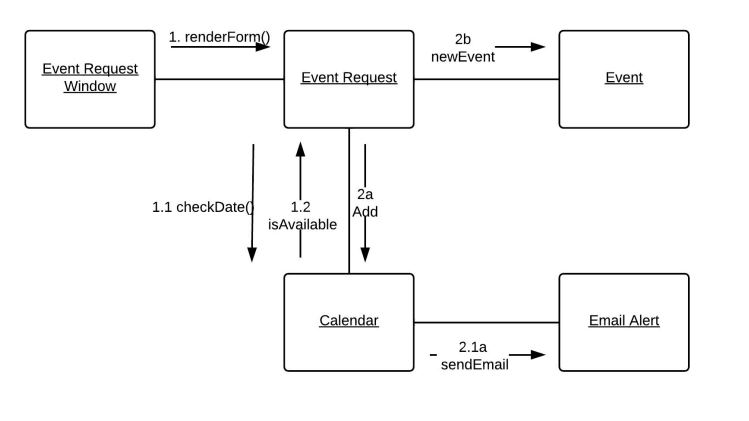
**7)Deployment Diagram:**

A Deployment Diagram is a diagram in software engineering that shows the physical arrangements and relationships among components in a distributed system. It represents the hardware and software components of a system and how they are connected.

****

**8)Communication Diagram:**

A Communication Diagram is a type of UML (Unified Modeling Language) diagram that shows the interactions between objects or components in a system. It represents the flow of messages between objects and their relationships in a graphical and organized manner.

**9. TEST DATA GENERATION**

Introduction:

Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. The engineer creates a series of test cases that are intended to “demolish” the software that has been built. In fact, testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.

Software engineers are by their nature constructive people. Testing requires that the developer discard preconceived notions of the “correctness” of software just developed and overcome a conflict of interest that occurs when errors are uncovered.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met. In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

**Testing principles:**

Before applying methods to design effective test cases, a software engineer must understand the basic principle that guide software testing:

All tests should be traceable to customer requirements

Tests should be planned long before testing begins80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Testing should being “in the small” and progress toward testing “in the large”.

Exhaustive testing is not possible

To be most effective an independent third party should conduct testing

A rich variety of test case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any engineered product (and most other things) can be tested in one of two ways:

Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational

While at the same time searching for errors in each function; (2) knowing the internal

Working of a product, tests can be conducted to ensure that “all gears mesh,” that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach is called black box testing and the second, white-box testing.

**10. VERIFICATION**

Testing against specification of system or component. Study it by examining its inputs and related outputs. Key is to devise inputs that have a higher likelihood of causing outputs that reveal the presence of defects. Use experience and knowledge of domain to identify such test cases. Failing this a systematic approach may be necessary. Equivalence partitioning is where the input to a program falls into a number of classes. E.g. positive numbers vs. negative numbers. Programs normally behave the same way for each member of a class. Partitions exist for both input and output. Partitions may be discrete or overlap. Invalid data (i.e. outside the normal partitions) is one or more partitions that should be tested. Test cases are chosen to exercise each portion. Also test boundary cases (atypical, extreme, zero) since these frequently show up defects. For completeness, test all combinations of partitions. Black box testing is rarely exhaustive (because one doesn't test every value in an equivalence partition) and sometimes fails to reveal corruption defects caused by "weird" combination of inputs. Black box testing should not be used to try and reveal corruption defects caused, for example, by assigning a pointer to point to an object of the wrong type. Static inspection (or using a better programming language!) is preferable for this.

Testing based on knowledge of structure of component (e.g. by looking at source code).Advantage is that structure of code can be used to find out how many test case need to be performed. Knowledge of the algorithm (examination of the code) can be used to identify the equivalence partitions. Path testing is where the tester aims to exercise every independent execution path through the component. All conditional statements tested for both true and false cases. If a unit has n control statements, there will be up to 2n possible paths through it. This demonstrates that it is much easier to test small program units than large ones. Flow graphs are a pictorial representation of the paths of control through a program (ignoring assignments, procedure calls and I/O statements). Use flow graph to design test cases that execute each path Static tools may be used to make this easier in programs that have a complex branching structure .Tools support. Dynamic program analyzers instrument a program with additional code.

Typically this will count how many times each statement is executed. At end, print out report showing which statements have and have not been executed. Problems with flow graph derived testing: Data complexity not taken into account.Does not test all paths in combination. Really only possible at unit and module testing stages because beyond that complexity is too high.

**Interface testing**

Usually done at integration stage when modules or sub-systems are combined. Objective is to detect errors or invalid assumptions about interfaces between modules. Reason these are not shown up in unit testing is that test case may perpetuate same incorrect assumption made by module designer. Particularly important when OO development has been used. Four types of interface:1. Parameter: data (or occasionally function references) passed from one unit to another.2. Shared memory: block of memory shared between units (e.g. global variable) .One places data there and the other retrieves it.3. Procedural: object-oriented or abstract data type form of interface, encapsulating several procedures.4. Message passing: one sub-system requests a service by passing a message. Client-server interface also used by some OO architectures.

**Testing process:**

Best testing process is to test each subsystem separately, as we have done in my project. Best done during implementation. Best done after small sub-steps of the implementation rather than large chunks. Once each lowest level unit has been tested, units are combined with related units and retested in combination. This proceeds hierarchically bottom-up until the entire system is tested as a whole. Typical levels of testing: Unit -procedure, function, method Module -package, abstract data type, class Sub-system - collection of related modules, cluster of classes, method-message paths Acceptance testing - whole system with real data (involve customer, user, etc)Alpha testing is acceptance testing with a single client (common for bespoke systems).Beta testing involves distributing system to potential customers to use and provide feedback. In, this project, Beta testing has been followed. This exposes system to situations and errors that might not be anticipated by us

**11. COST AND RESOURCE ESTIMATION**

Software cost estimation is the process of predicting the effort required to develop a software system. Many estimation models have been proposed over the last 20 years.

Models may be classified into 2 major categories: algorithmic and non-algorithmic. Each has its own strengths and weaknesses. A key factor in selecting a cost estimation model is the accuracy of its estimates. Unfortunately, despite the large body of experience with estimation models, the accuracy of these models is not satisfactory.

**Introduction**

Accurate cost estimation is important because:

• It can help to classify and prioritize development projects with respect to an overall business plan.

• It can be used to determine what resources to commit to the project and how well these resources will be used.

• It can be used to assess the impact of changes and support replanning.

• Projects can be easier to manage and control when resources are better matched to real needs.

• Customers expect actual development costs to be in line with estimated costs.

Software cost estimation involves the determination of one or more of the following estimates:

• effort (usually in person-months)

• project duration (in calendar time)

• cost (in dollars)

The three fundamental issues that need to be resolved are:

• Which software cost estimation model to use?

• Which software size measurement to use – lines of code (LOC), function points (FP), or feature point?

• What is a good estimate?

The widely practiced cost estimation method is expert judgment. For many years, project managers have relied on experience and the prevailing industry norms as a basis to develop cost estimate. However, basing estimates on expert judgment is problematic:

• This approach is not repeatable and the means of deriving an estimate are not explicit.

**Process of estimation**

Cost estimation is an important part of the planning process. For example, in the top-down planning approach, the cost estimate is used to derive the project plan:

1. The project manager develops a characterization of the overall functionality, size, process, environment, people, and quality required for the project.

2. A macro-level estimate of the total effort and schedule is developed using a software cost estimation model.

3. The project manager partitions the effort estimate into a top-level work breakdown structure. He also partitions the schedule into major milestone dates and determines a staffing profile, which together forms a project plan.

The actual cost estimation process involves seven steps :

1. Establish cost-estimating objectives

2. Generate a project plan for required data and resources

3. Pin down software requirements

4. Work out as much detail about the software system as feasible

5. Use several independent cost estimation techniques to capitalize on their combined strengths

6. Compare different estimates and iterate the estimation process

7. After the project has started, monitor its actual cost and progress, and feedback results to project management

**Software sizing**

The software size is the most important factor that affects the software cost. This section describes five software size metrics used in practice. The line of code and function point are the most popular metrics among the five metrics.

**Lines of Code:** This is the number of lines of the delivered source code of the software, excluding comments and blank lines and is commonly known as LOC . Although LOC is programming language dependent, it is the most widely used software size metric. Most models relate this measurement to the software cost. However, exact LOC can only be obtained after the project has completed. Estimating the code size of a program before it is actually built is almost as hard as estimating the cost of the program.

**Function points:** This is a measurement based on the functionality of the program and was first introduced by Albrecht . The total number of function points depends on the counts of distinct(in terms of format or processing logic) types in the following five classes:

1. User-input types: data or control user-input types

2. User-output types: output data types to the user that leaves the system

3. Inquiry types: interactive inputs requiring a response

4. Internal file types: files (logical groups of information) that are used and shared inside the system

5. External file types: files that are passed or shared between the system and other systems

Each of these types is individually assigned one of three complexity levels of {1 = simple, 2 =medium, 3 = complex} and given a weighting value that varies from 3 (for simple input) to 15(for complex internal files).

This initial function-point count is either directly used for cost estimation or is further modified by factors whose values depend on the overall complexity of the project. This will take into account the degree of distributed processing, the amount of reuse, the performance requirement, etc. The final function-point count is the product of the UFC unadjusted function-point counts and these project complexity factors. The advantage of the function-point measurement is that it can be obtained based on the system requirement specification in the early stage of software development

Object points: This measurement is based on the number and complexity of the following objects: screens, reports and 3GL components. Each of these objects is counted and given a weight ranging from 1 (simple screen) to 10 (3GL component) and the object point is the weighted sum of all these objects. It is easy to use at the early phase of the development cycle and also measures software size reasonably well, this measurement has been used in major estimation

models such as COCOMO II for cost estimation .

**Cost estimation**

There are two major types of cost estimation methods: algorithmic and non-algorithmic.

Algorithmic models vary widely in mathematical sophistication. Some are based on simple arithmetic formulas using such summary statistics as means and standard deviations. Others are based on regression models and differential equations . To improve the accuracy of algorithmic models, there is a need to adjust or calibrate the model to local circumstances. These models cannot be used off-the-shelf. Even with calibration the accuracy can be quite mixed

**Non-algorithmic Methods**

**Analogy costing:** This method requires one or more completed projects that are similar to the new project and derives the estimation through reasoning by analogy using the actual costs of previous projects. Estimation by analogy can be done either at the total project level or at subsystem level. The total project level has the advantage that all cost components of the system will be considered while the subsystem level has the advantage of providing a more detailed assessment of the similarities and differences between the new project and the completed projects. The strength of this method is that the estimate is based on actual project experience.

Expert judgment: This method involves consulting one or more experts. The experts provide estimates using their own methods and experience. Expert-consensus mechanisms such as Delphi technique or PERT will be used to resolve the inconsistencies in the estimates.

**Bottom-up:** In this approach, each component of the software system is separately estimated and the results aggregated to produce an estimate for the overall system. The requirement for this approach is that an initial design must be in place that indicates how the system is decomposed into different components.

**Top-down:** This approach is the opposite of the bottom-up method. An overall cost estimate for the system is derived from global properties, using either algorithmic or non-algorithmic methods. The total cost can then be split up among the various components. This approach is more suitable for cost estimation at the early stage.

**Algorithmic methods**

The algorithmic methods are based on mathematical models that produce cost estimate as a function of a number of variables, which are considered to be the major cost factors. Any algorithmic model has the form:

Effort = f(x1, x2, …, xn)

where {x1, x2, …, xn} denote the cost factors. The existing algorithmic methods differ in two aspects: the selection of cost factors, and the form of the function f..

**Cost factors**

Besides the software size, there are many other cost factors. The most comprehensive set of cost factors are proposed and used by Boehm et al in the COCOMO II model . These cost factors can be divided into four types:

Product factors: required reliability; product complexity; database size used; required reusability; documentation match to life-cycle needs;

Computer factors: execution time constraint; main storage constraint; computer turnaround constraints; platform volatility;

Personnel factors: analyst capability; application experience; programming capability; platform experience; language and tool experience; personnel continuity;

Project factors: multisite development; use of software tool; required development schedule.

The above factors are not necessarily independent, and most of them are hard to quantify. In many models, some of the factors appear in combined form and some are simply ignored. Also, some factors take discrete values, resulting in an estimation function with a piece-wise form.

**COCOMO (Constructive Cost Model) models**

This family of models was proposed by Boehm . The models have been widely accepted in practice. In the COCOMOs, the code-size S is given in thousand LOC (KLOC) and Effort is in person-month.

A) Basic COCOMO. This model uses three sets of {a, b} depending on the complexity of the software only:

(1) for simple, well-understood applications, a = 2.4, b = 1.05;

(2) for more complex systems, a = 3.0, b = 1.15;

(3) for embedded systems, a = 3.6, b = 1.20.

The basic COCOMO model is simple and easy to use. As many cost factors are not considered, it can only be used as a rough estimate.

B) Intermediate COCOMO and Detailed COCOMO. In the intermediate COCOMO, a nominal effort estimation is obtained using the power function with three sets of {a, b}, with coefficient a being slightly different from that of the basic COCOMO:

(1) for simple, well-understood applications, a = 3.2, b = 1.05

(2) for more complex systems, a = 3.0, b = 1.15

(3) for embedded systems, a = 2.8, b = 1.20

Then, fifteen cost factors with values ranging from 0.7 to 1.66 are determined . The overall impact factor M is obtained as the product of all individual factors, and the estimate is obtained by multiplying M to the nominal estimate.

**12.USER MANUAL**

**Introduction:**

* A wedding planner management system is a software application that helps wedding planners manage various aspects of a wedding, such as guest lists, vendor management, budgeting, and scheduling.

**Getting Started:**

* Install the wedding planner management system on your computer.
* Create an account or log in if you already have one.
* Enter your personal details, including name, email address, and phone number.

**Guest List Management:**

* Click on the "Guest List" tab.
* Add new guests by clicking the "Add Guest" button and entering their name, address, and contact information.
* Organize guests into categories, such as bridal party, family, or friends.
* Keep track of RSVPs and meal preferences.

**Vendor Management:**

* Click on the "Vendors" tab.
* Add new vendors by clicking the "Add Vendor" button and entering their name, contact information, and services offered.
* Keep track of vendor contracts, payments, and deadlines.
* Compare prices and services offered by different vendors to make informed decisions.

**Budget Management:**

* Click on the "Budget" tab.
* Enter your overall wedding budget and allocate funds to different categories, such as venue, catering, photography, and entertainment.
* Keep track of expenses and compare them to your budget to stay on track.
* Generate reports and charts to track your spending and make informed decisions.

**Scheduling:**

* Click on the "Schedule" tab.
* Enter important dates and deadlines, such as venue availability, vendor contracts, and RSVP deadlines.
* Keep track of appointments and meetings with vendors and other wedding-related events.
* Generate a detailed schedule of events to ensure that everything runs smoothly on the big day.

**13. Conclusion**

A wedding planner management system is an essential tool for wedding planners, as it helps them to streamline their planning process and keep track of all important details and tasks. This system allows wedding planners to manage schedules, budgets, vendors, guests, and other important aspects of the wedding planning process. With its user-friendly interface, robust features, and customizable options, a wedding planner management system can save time and increase efficiency for wedding planners, ensuring a successful and stress-free wedding planning experience for both the planner and the bride and groom. Wedding planner management system improves the efficiency and success of the wedding planning process by organizing schedules, budgets, vendors, guests, and other important details. With its user-friendly interface and customizable features, it saves time for wedding planners and leads to a stress-free experience for all parties involved. A UML-based wedding planner management system is a well-structured and organized approach to the wedding planning process. It utilizes Unified Modeling Language diagrams to represent the various components and interactions in the system. This system helps wedding planners to visualize and analyze the wedding planning process, ensuring that all important details and tasks are accounted for. The UML diagrams also provide a clear and concise representation of the system's functionality and structure, making it easier to maintain and modify in the future. Overall, a UML-based wedding planner management system provides a more comprehensive and effective approach to wedding planning. In conclusion, a wedding planner management system is a crucial tool for organizing and streamlining the various tasks involved in planning a wedding. The system's design, as depicted in the UML (Unified Modeling Language) diagrams, takes into account important aspects such as vendor management, budget management, task management, and more. The use of UML provides a clear and concise representation of the system's functionality, making it easier to understand and maintain. The implementation of this system would result in improved efficiency and organization for wedding planners, reducing the stress and workload associated with planning a wedding. Furthermore, the system's user-friendly interface and comprehensive features allow for seamless communication and collaboration between the wedding planner and their clients, ensuring that all aspects of the wedding planning process are well managed and executed. Overall, the proposed wedding planner management system in UML provides a solid foundation for the development of a functional and reliable system that helps wedding planners deliver unforgettable weddings for their clients.

**14. References**

1. [**https://www.freeprojectz.com/uml-diagram/wedding-planner-management-system-uml-diagram**](https://www.freeprojectz.com/uml-diagram/wedding-planner-management-system-uml-diagram)
2. **https://www.urbancompany.com/**
3. **https://www.marketing91.com/business-model-of-urbanclap/**
4. **https://p.urbanpro.com/**
5. **https://oyelabs.com/urbanclap-business-model/**