DROOT

Software Engineering

Project Report

Team-

Harsh Tonde (0827CS181085)

Isha Lodha (0827CS181093)

Ishika Shendge (0827CS181095)

Contents

Introduction

The Problem

There is nothing quite like traveling, like seeing a new place for the first time or returning to a favourite one. People of all ages, from all around the world, go to foreign places for different reasons – mainly, for work, family, and leisure. Whether by plane, train, ship or by automobile, traveling is generally a pleasurable experience.

While travelling we must have faced many problems due to which we get frustrated and have no idea what to do then. Some of the problems are listed below:

- People keep forgetting important stuff before travelling
- They face lot of troubles in organizing their travel documents
- Keeping live updates of your travel companions is irritating and difficult
- People don't know what other companions have packed for the trip, etc.

The Solution

In this new digital era, smartphones have reached in the hands of around 3.5 billion people in the world. Mobile phones are getting much smarter than the human mind, so it is obvious that we rely on them to make our life easier. Smartphones are our new personal assistant, so why not let them take care of the trip plan and management.

Imagine checking one place for all your travel details and getting a heads up as things happen throughout your trip. See why life without Droot is a distant memory for millions of travellers.

Droot is a personal travel assistant which will help the user before, after and during the trip. Let it be some business trip, educational trip, family trip or some trip which you enjoy with your friends, Droot will assist you everywhere.

Droot will help the user in:

- Making a common item list where all the companions will check the items they have packed already.
- Droot will send reminders for the travel check list and the documents.
- It will help the user to find the items in his/her list by recommending buying the items on a low price on an e-commerce website.
- It will show the live location of the companions
- It will give every possible information about the location where you are about to travel.
- Droot will keep the travel documents in an organized way.
- Droot will calculate the estimated expense which will be done by the user.
- Droot will recommend the accommodations and transportation platforms.
- Droot will help in sharing the trip photos and splitting the bills.
- Droot will recommend some travelling games and travel songs playlist.
- In case of emergency, Droot will send the live location of the person to all the other tripmates.

DROOT

Introduction to DROOT

1. You handle the booking; we'll take it from there

Unlike other travel apps, **Droot** can organize your travel plans no matter where you book. Simply forward your confirmed tickets and booking receipts in the Droot app and in a matter of seconds, Droot will create a master itinerary for every trip.

2. Helpful reminders and alerts so you never miss a beat

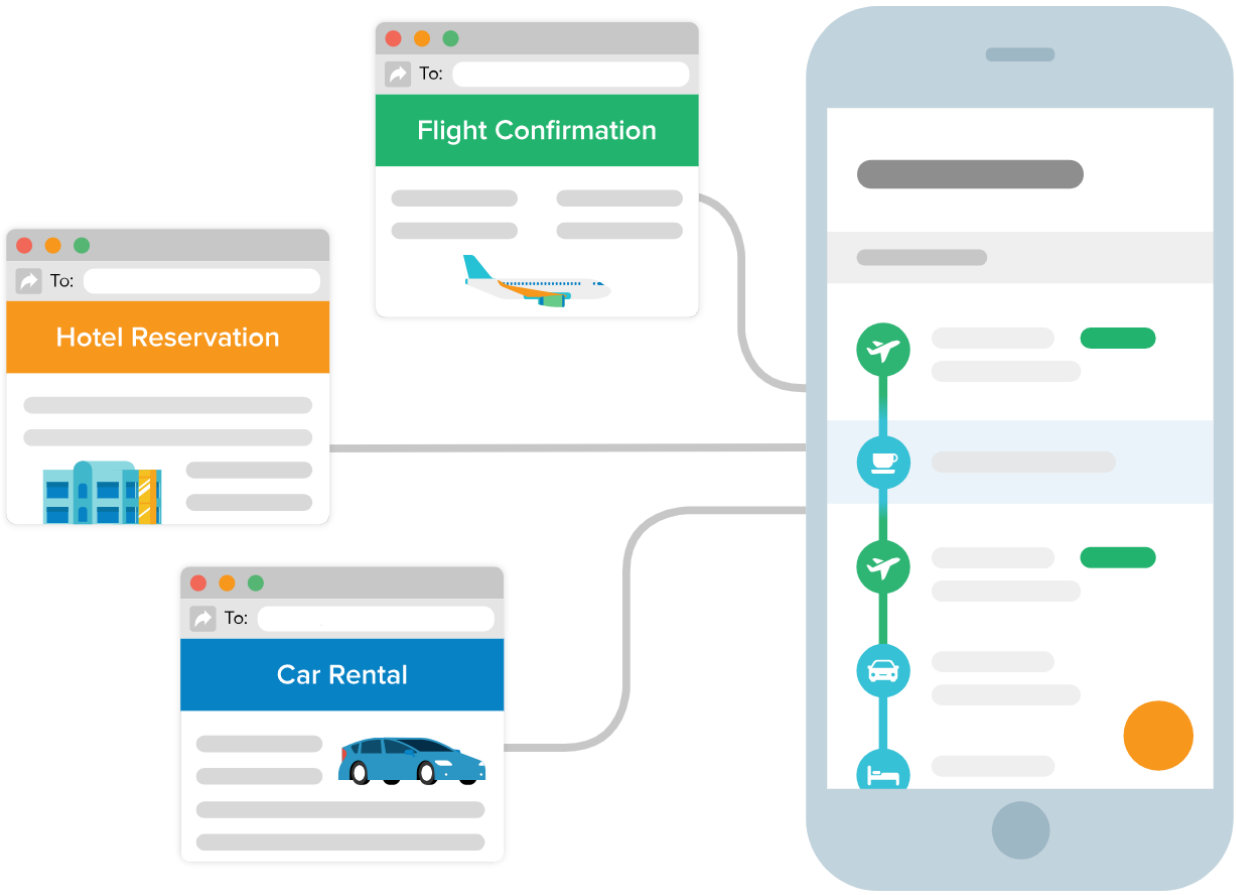
Packed with features that give you a leg up on changes and help you make the most of all your trips, Droot is where the magic happens.

3. Always know where to be and when

Need a reminder when it's time to leave for the airport? Not sure where to eat in the airport or near your hotel? Droot has you covered.

4. Hours of planning, organized in seconds

Learn how quickly Droot can make sense of all your travel plans and create a single itinerary for every trip.



While you're still making plans...

Droot kicks into gear as soon as you book, helping you get there more comfortably and earn something along the way.

Seat Tracker
Lets you know if a better seat becomes available

Fare Tracker
Notifies you if your airfare price drops after you book

Check-in Reminder
Gives you a heads up 24 hours before your flight

Point Tracker
Keeps track of your reward programs for you

Inner Circle
Instantly sends your travel plans to a select group of family, friends or colleagues

International Travel Tools
Provides country-specific travel information

Comparison of different Life Cycle Models

Classical Waterfall Model-

The Classical Waterfall model can be considered as the basic model and all other life cycle models are based on this model. It is an ideal model. However, the Classical Waterfall model cannot be used in practical project development, since this model does not support any mechanism to correct the errors that are committed during any of the phases but detected at a later phase. This problem is overcome by the Iterative Waterfall model through the inclusion of feedback paths.

Iterative Waterfall Model-

The Iterative Waterfall model is probably the most used software development model. This model is simple to use and understand. But this model is suitable only for well-understood problems and is not suitable for the development of very large projects and projects that suffer from many risks.

Prototyping Model-

The Prototyping model is suitable for projects, which either the customer requirements or the technical solutions are not well understood. These risks must be identified before the project starts. This model is especially popular for the development of the user interface part of the project.

Evolutionary Model-

The Evolutionary model is suitable for large projects which can be decomposed into a set of modules for incremental development and delivery. This model is widely used in object-oriented development projects. This model is only used if incremental delivery of the system is acceptable to the customer.

Spiral Model-

The Spiral model is considered as a meta-model as it includes all other life cycle models. Flexibility and risk handling are the main characteristics of this model. The spiral model is suitable for the development of technically challenging and large software that is prone to various risks that are difficult to anticipate at the start of the project. But this model is very much complex than the other models.

Agile Model-

The Agile model was designed to incorporate change requests quickly. In this model, requirements are decomposed into small parts by that can be incrementally developed. But the main principle of the Agile model is to deliver an increment to the customer after each Time-box. The end date of iteration is fixed, it can't be extended. This agility is achieved removing unnecessary activities that waste time and effort.

Requirements

Tools:

Operating Systems	Windows
Programming Language	Flutter, Dart
Database	Firebase
Server	AWS
IDE	Android Studio

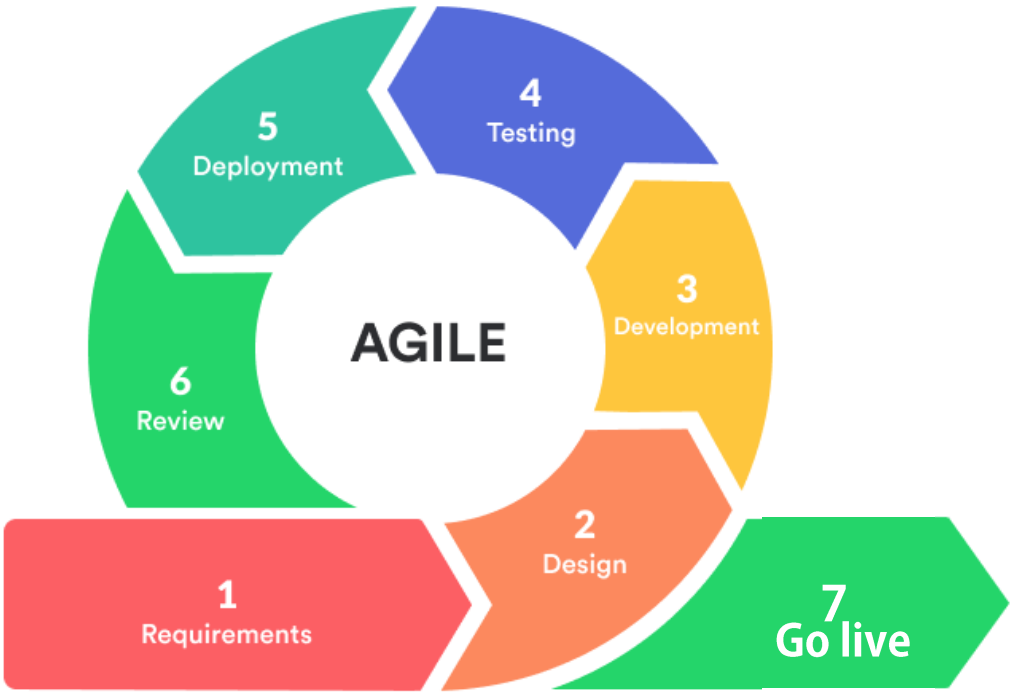
Method

This application works on a dynamic mobile system which acts as a personal travel assistant and will assist the user before, during and after the trip.

Model

In this application we will use **Agile SDLC model**.

Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product. Agile Methods break the product into small incremental builds.



System Design

Software Design

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

Once the requirements document for the software to be developed is available, the software design phase begins. While the requirement specification activity deals entirely with the problem domain, design is the first phase of transforming the problem into a solution. In the design phase, the customer and business requirements and technical considerations all come together to formulate a product or a system.

The design process comprises a set of principles, concepts and practices, which allow a software engineer to model the system or product that is to be built. This model, known as design model, is assessed for quality and reviewed before a code is generated and tests are conducted. The design model provides details about software data structures, architecture, interfaces and components which are required to implement the system. This chapter discusses the design elements that are required to develop a software design model. It also discusses the design patterns and various software design notations used to represent a software design.

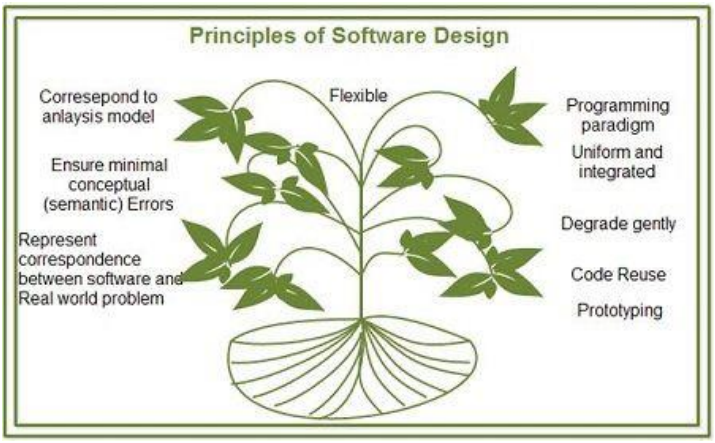
Basics of Software Design

Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system. **IEEE** defines software design as 'both a process of defining, the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.'

In the design phase, many critical and strategic decisions are made to achieve the desired functionality and quality of the system. These decisions are taken into account to successfully develop the software and carry out its maintenance in a way that the quality of the end product is improved.

Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice.



Some of the commonly followed design principles are as following.

Software design should correspond to the analysis model: Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.

Choose the right programming paradigm: A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.

Software design should be uniform and integrated: Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.

Software design should be flexible: Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.

Software design should ensure minimal conceptual (semantic) errors: The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.

Software design should be structured to degrade gently: Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.

Software design should represent correspondence between the software and real-world problem: The software design should be structured in such a way that it always relates with the real-world problem.

Software reuse: Software engineers believe on the phrase: 'do not reinvent the wheel'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.

Designing for testability: A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed

(designed and implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if any type of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.

Prototyping: Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick 'mock-up' of the system can be developed. This mock-up can be used as an effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

Note that design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures, and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references and maintenance.

Software Design Concepts

Every software process is characterized by basic concepts along with certain practices or methods. Methods represent the manner through which the concepts are applied. As new technology replaces older technology, many changes occur in the methods that are used to apply the concepts for the development of software. However, the fundamental concepts underlining the software design process remain the same, some of which are described here.

Abstraction

Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components. **IEEE** defines abstraction as 'a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.' The concept of abstraction can be used in two ways: as a process and as an entity. As a **process,** it refers to a mechanism of hiding irrelevant details and representing only the essential features of an item so that one can focus on important things at a time. As an **entity,** it refers to a model or view of an item.

Each step in the software process is accomplished through various levels of abstraction. At the highest level, an outline of the solution to the problem is presented whereas at the lower levels, the solution to the problem is presented in detail. For example, in the requirements analysis phase, a solution to the problem is presented using the language of problem environment and as we proceed through the software process, the abstraction level reduces and at the lowest level, source code of the software is produced.

There are three commonly used abstraction mechanisms in software design, namely, functional abstraction, data abstraction and control abstraction. All these mechanisms allow us to control the

complexity of the design process by proceeding from the abstract design model to concrete design model in a systematic manner.

Functional abstraction: This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as 'groups'. Within these groups there exist routines which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.

Data abstraction: This involves specifying data that describes a data object. For example, the data object window encompasses a set of attributes (window type, window dimension) that describe the window object clearly. In this abstraction mechanism, representation and manipulation details are ignored.

Control abstraction: This states the desired effect, without stating the exact mechanism of control. For example, if and while statements in programming languages (like C and C++) are abstractions of machine code implementations, which involve conditional instructions. In the architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

Architecture

Software architecture refers to the structure of the system, which is composed of various components of a program/ system, the attributes (properties) of those components and the relationship amongst them. The software architecture enables the software engineers to analyse the software design efficiently. In addition, it also helps them in decision-making and handling risks. The software architecture does the following.

Provides an insight to all the interested stakeholders that enable them to communicate with each other

Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase

Creates intellectual models of how the system is organized into components and how these components interact with each other.

Currently, software architecture is represented in an informal and unplanned manner. Though the architectural concepts are often represented in the infrastructure (for supporting particular architectural styles) and the initial stages of a system configuration, the lack of an explicit independent characterization of architecture restricts the advantages of this design concept in the present scenario.

Note that software architecture comprises two elements of design model, namely, data design and architectural design.

Patterns

A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again. The objective of each pattern is to provide an insight to a designer who can determine the following.

Whether the pattern can be reused

Whether the pattern is applicable to the current project

Whether the pattern can be used to develop a similar but functionally or structurally different design pattern.

Types of Design Patterns

Software engineer can use the design pattern during the entire software design process. When the analysis model is developed, the designer can examine the problem description at different levels of abstraction to determine whether it complies with one or more of the following types of design patterns.

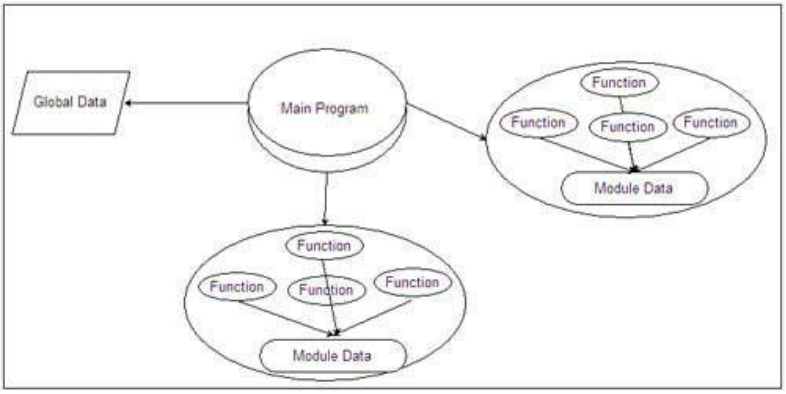
Architectural patterns: These patterns are high-level strategies that refer to the overall structure and organization of a software system. That is, they define the elements of a software system such as subsystems, components, classes, etc. In addition, they also indicate the relationship between the elements along with the rules and guidelines for specifying these relationships. Note that architectural patterns are often considered equivalent to software architecture.

Design patterns: These patterns are medium-level strategies that are used to solve design problems. They provide a means for the refinement of the elements (as defined by architectural pattern) of a software system or the relationship among them. Specific design elements such as relationship among components or mechanisms that affect component-to-component interaction are addressed by design patterns. Note that design patterns are often considered equivalent to software components.

Idioms: These patterns are low-level patterns, which are programming-language specific. They describe the implementation of a software component, the method used for interaction among software components, etc., in a specific programming language. Note that idioms are often termed as coding patterns.

Modularity

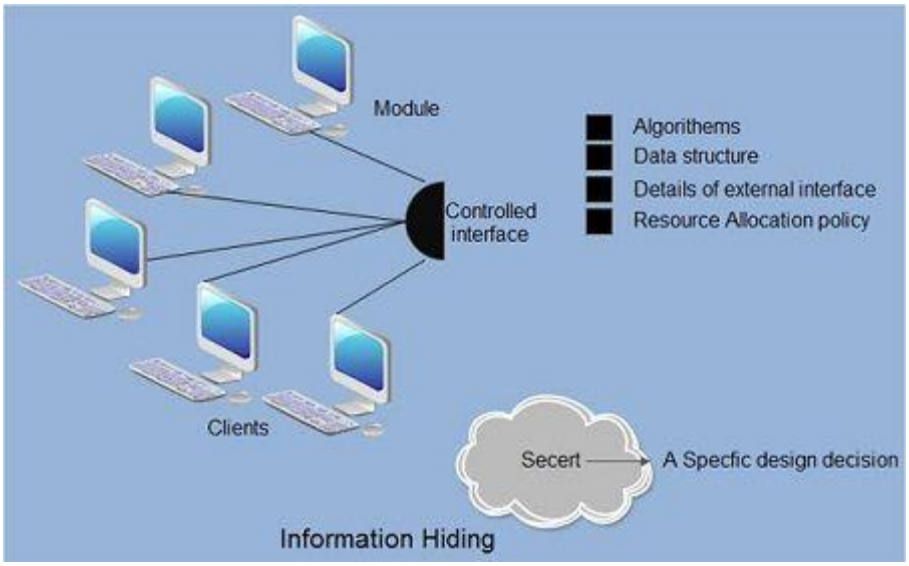
Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements. Note that larger the number of modules a system is divided into, greater will be the effort required to integrate the modules.



Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conduct maintenance work without adversely affecting the functioning of the software.

Information Hiding

Modules should be specified and designed in such a way that the data structures and processing details of one module are not accessible to other modules. They pass only that much information to each other, which is required to accomplish the software functions. The way of hiding unnecessary details is referred to as **information hiding. IEEE** defines information hiding as 'the technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus each module is a 'black box' to the other modules in the system.



Information hiding is of immense use when modifications are required during the testing and maintenance phase. Some of the advantages associated with information hiding are listed below.

Leads to low coupling

Emphasizes communication through controlled interfaces

Decreases the probability of adverse effects

Restricts the effects of changes in one component on others

Results in higher quality software.

Stepwise Refinement

Stepwise refinement is a top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.

Software designers start the stepwise refinement process by creating a sequence of compositions for the system being designed. Each composition is more detailed than the previous one and contains

more components and interactions. The earlier compositions represent the significant interactions within the system, while the later compositions show in detail how these interactions are achieved.

To have a clear understanding of the concept, let us consider an example of stepwise refinement. Every computer program comprises input, process, and output.

INPUT

Get user's name (string) through a prompt.

Get user's grade (integer from 0 to 100) through a prompt and validate.

PROCESS

OUTPUT

This is the first step in refinement. The input phase can be refined further as given here.

INPUT

Get user's name through a prompt.

Get user's grade through a prompt.

While (invalid grade)

Ask again:

PROCESS

OUTPUT

Note: Stepwise refinement can also be performed for PROCESS and OUTPUT phase.

Refactoring

Refactoring is an important design activity that reduces the complexity of module design keeping its behaviour or function unchanged. Refactoring can be defined as a process of modifying a software system to improve the internal structure of design without changing its external behaviour. During the refactoring process, the existing design is checked for any type of flaws like redundancy, poorly constructed algorithms and data structures, etc., in order to improve the design. For example, a design model might yield a component which exhibits low cohesion (like a component performs four functions that have a limited relationship with one another). Software designers may decide to refactor the component into four different components, each exhibiting high cohesion. This leads to easier integration, testing, and maintenance of the software components.

Structural Partitioning

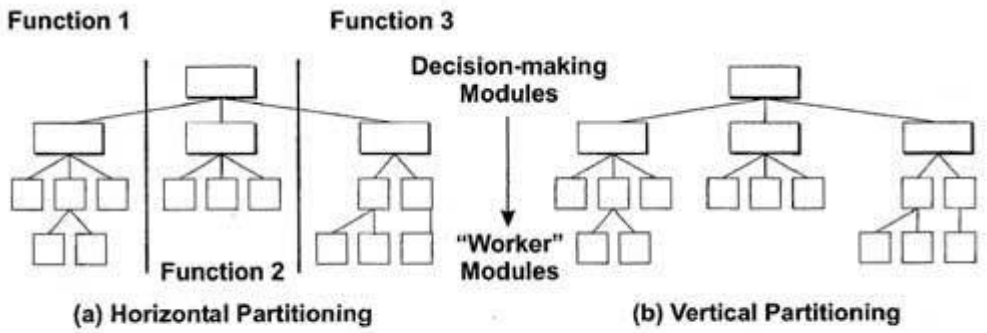
When the architectural style of a design follows a hierarchical nature, the structure of the program can be partitioned either horizontally or vertically. In **horizontal partitioning,** the control modules are used to communicate between functions and execute the functions. Structural partitioning provides the following benefits.

The testing and maintenance of software becomes easier.

The negative impacts spread slowly.

The software can be extended easily.

Besides these advantages, horizontal partitioning has some disadvantage also. It requires to pass more data across the module interface, which makes the control flow of the problem more complex. This usually happens in cases where data moves rapidly from one function to another.



Horizontal and Vertical Partitioning

In **vertical partitioning**, the functionality is distributed among the modules--in a top-down manner. The modules at the top level called **control modules** perform the decision-making and do-little processing whereas the modules at the low level called **worker modules** perform all input, computation and output tasks.

Concurrency

Computer has limited resources and they must be utilized efficiently as much as possible. To utilize these resources efficiently, multiple tasks must be executed concurrently. This requirement makes concurrency one of the major concepts of software design. Every system must be designed to allow multiple processes to execute concurrently, whenever possible. For example, if the current process is waiting for some event to occur, the system must execute some other process in the meantime.

However, concurrent execution of multiple processes sometimes may result in undesirable situations such as an inconsistent state, deadlock, etc. For example, consider two processes A and B and a data item Q1 with the value '200'. Further, suppose A and B are being executed concurrently and firstly A reads the value of Q1 (which is '200') to add '100' to it. However, before A updates es the value of Q1, B reads the value ofQ1 (which is still '200') to add '50' to it. In this situation, whether A or B first updates the value of Q1, the value of would definitely be wrong resulting in an inconsistent state of the system. This is because the actions of A and B are not synchronized with each other. Thus, the system must control the concurrent execution and synchronize the actions of concurrent processes.

One way to achieve synchronization is mutual exclusion, which ensures that two concurrent processes do not interfere with the actions of each other. To ensure this, mutual exclusion may use locking technique. In this technique, the processes need to lock the data item to be read or updated. The data item locked by some process cannot be accessed by other processes until it is unlocked. It implies that the process, that needs to access the data item locked by some other process, has to wait.

Developing a Design Model

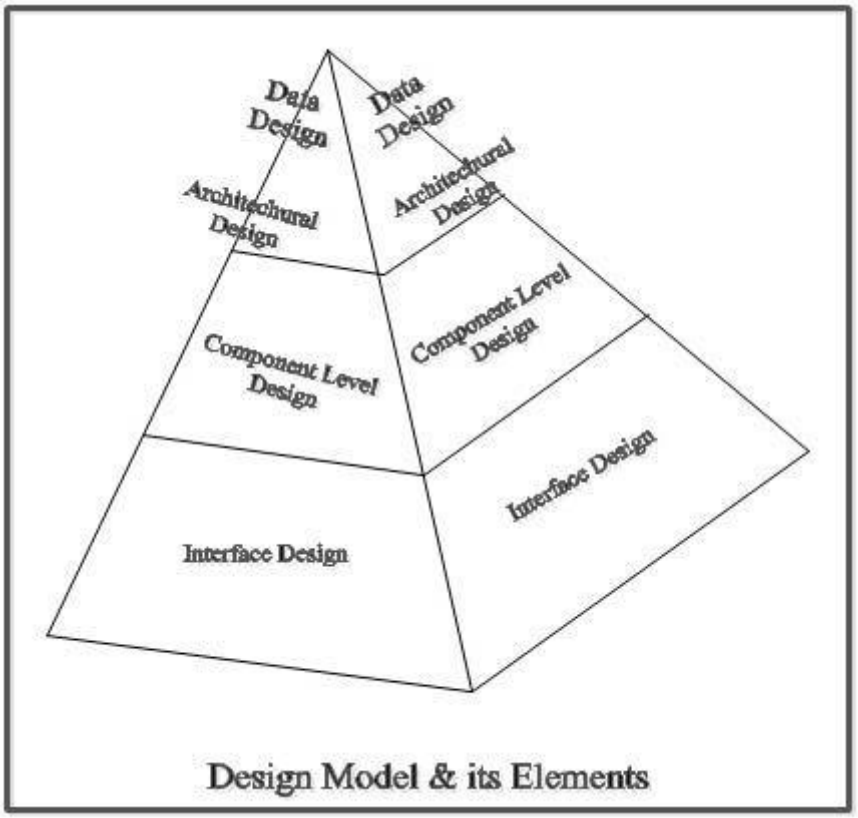
To develop a complete specification of design (design model), four design models are needed. These models are listed below.

Data design: This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design models.

Architectural design: This specifies the relationship between the structural elements of the software, design patterns, architectural styles, and the factors affecting the ways in which architecture can be implemented.

Component-level design: This provides the detailed description of how structural elements of software will actually be implemented.

Interface design: This depicts how the software communicates with the system that interoperates with it and with the end-users.



Design Model & its Elements

UML Diagram

What is a UML Diagram?

UML is a way of visualizing a software program using a collection of diagrams. The notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation to be used for object-oriented design, but it has since been extended to cover a wider variety of software engineering projects. Today, UML is accepted by the Object Management Group (OMG) as the standard for modelling software development.

What is Meant by UML?

UML stands for Unified Modelling Language. UML 2.0 helped extend the original UML specification to cover a wider portion of software development efforts including agile practices.

- Improved integration between structural models like class diagrams and behaviour models like activity diagrams.
- Added the ability to define a hierarchy and decompose a software system into components and sub-components.
- The original UML specified nine diagrams; UML 2.x brings that number up to 13. The four new diagrams are called: communication diagram, composite structure diagram, interaction overview diagram, and timing diagram. It also renamed state chart diagrams to state machine diagrams, also known as state diagrams.

Types of UML Diagrams

The current UML standards call for 13 different types of diagrams: class, activity, object, use case, sequence, package, state, component, communication, composite structure, interaction overview, timing, and deployment.

These diagrams are organized into two distinct groups: structural diagrams and behavioural or interaction diagrams.

Structural UML diagrams

- Class diagram
- Package diagram
- Object diagram
- Component diagram
- Composite structure diagram
- Deployment diagram

Behavioural UML diagrams

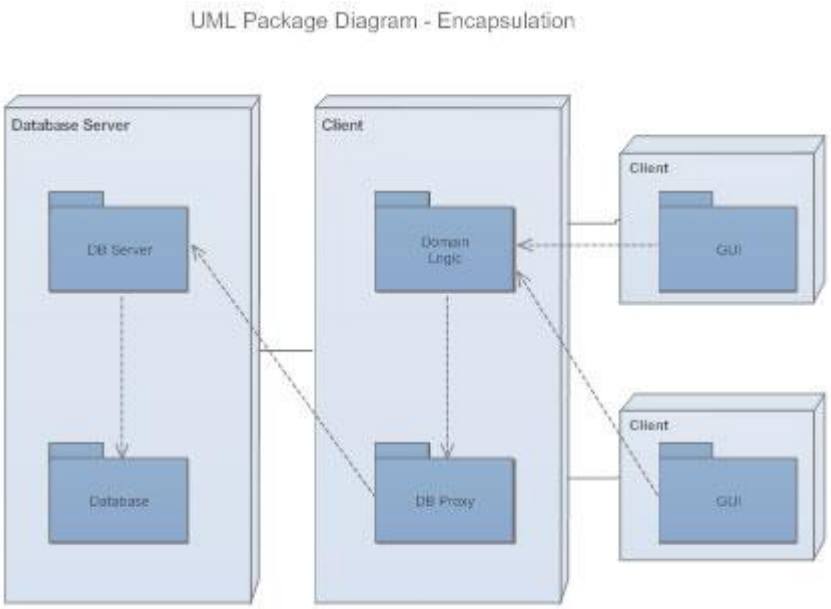
- Activity diagram
- Sequence diagram
- Use case diagram
- State diagram
- Communication diagram
- Interaction overview diagram
- Timing diagram

Class Diagram

Class Diagrams are the backbone of almost every object-oriented method, including UML. They describe the static structure of a system.

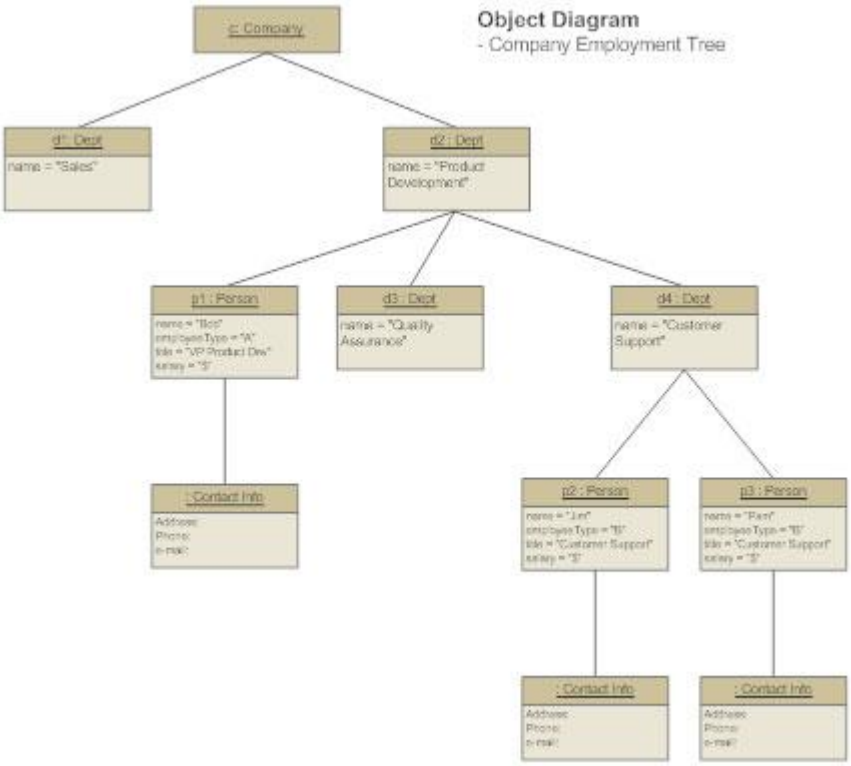
Package Diagram

Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.



Object Diagram

Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.

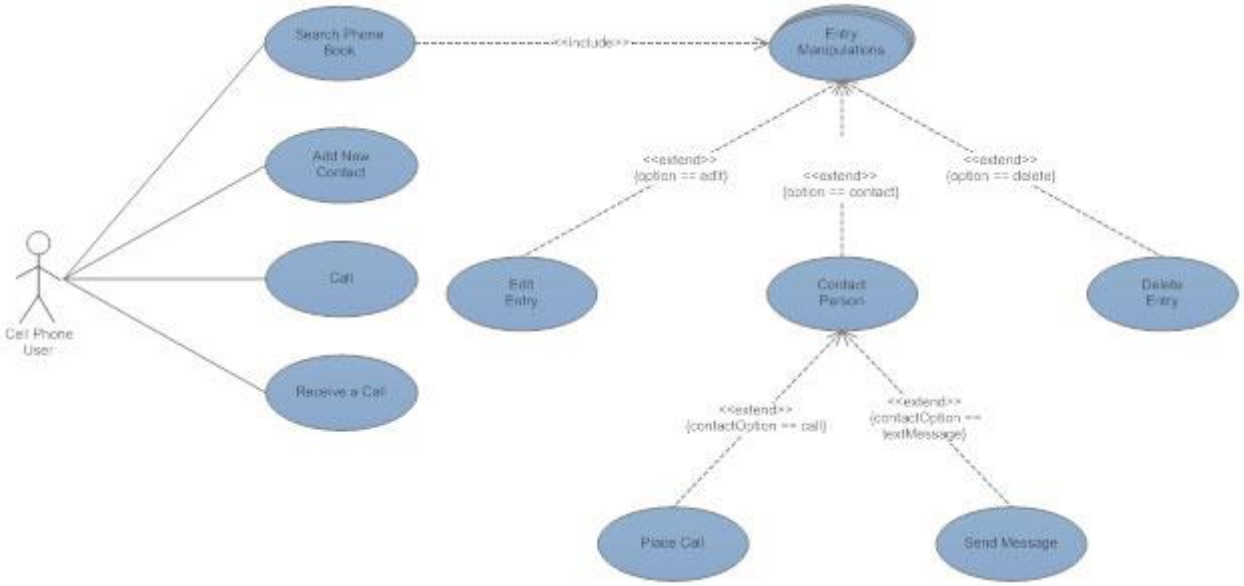


Composite Structure Diagram

Composite structure diagrams show the internal part of a class.

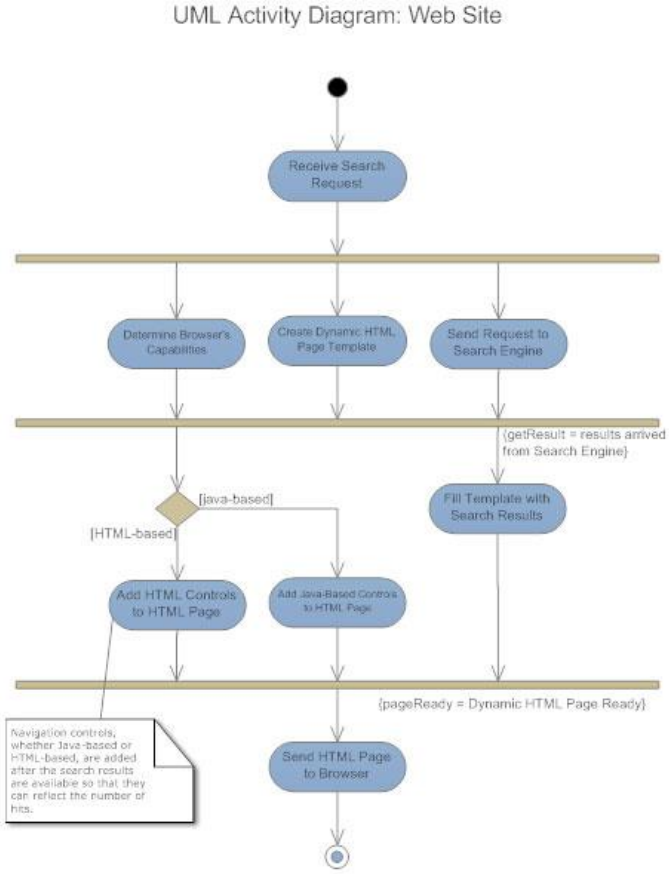
Use Case Diagram

Use case diagrams model the functionality of a system using actors and use cases.



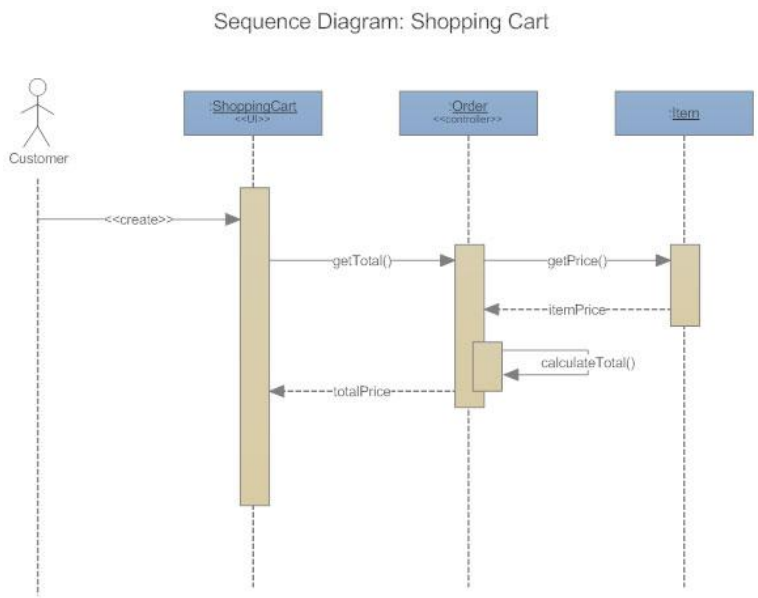
Activity Diagram

Activity diagrams illustrate the dynamic nature of a system by modelling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation. Learn more



Sequence Diagram

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.



Interaction Overview Diagram

Interaction overview diagrams are a combination of activity and sequence diagrams. They model a sequence of actions and let you deconstruct more complex interactions into manageable occurrences. You should use the same notation on interaction overview diagrams that you would see on an activity diagram.

Timing Diagram

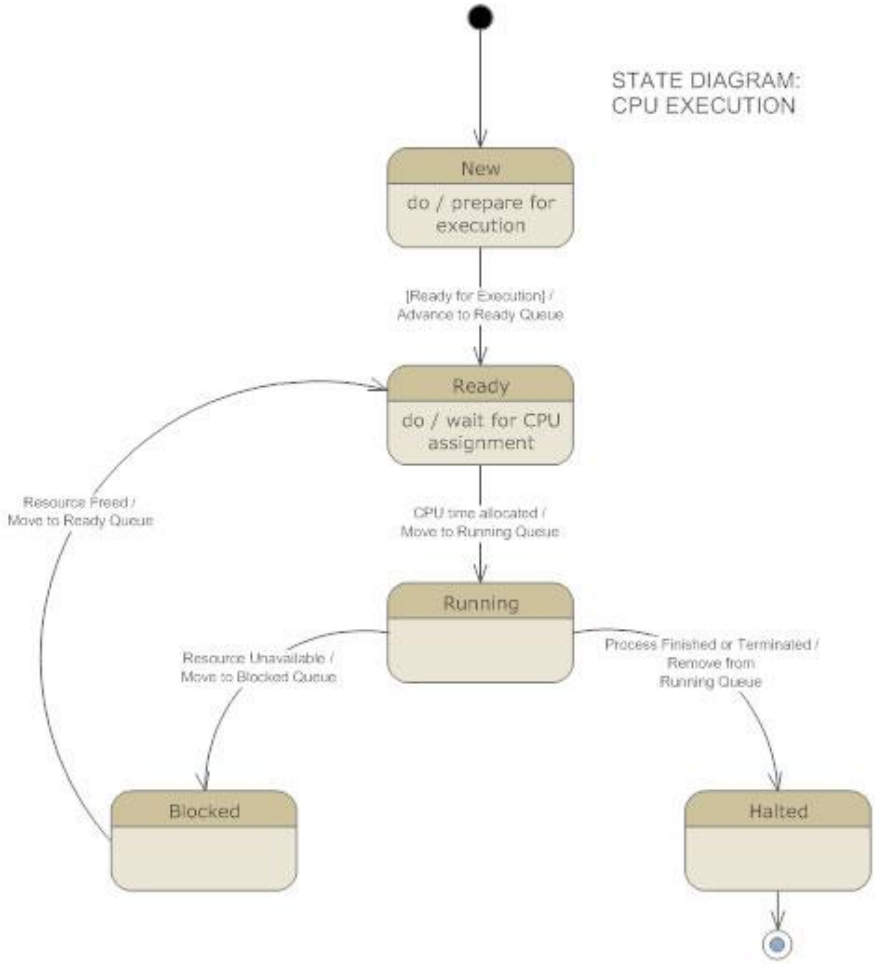
A timing diagram is a type of behavioural or interaction UML diagram that focuses on processes that take place during a specific period of time. They're a special instance of a sequence diagram, except time is shown to increase from left to right instead of top down.

Communication Diagram

Communication diagrams model the interactions between objects in sequence. They describe both the static structure and the dynamic behaviour of a system. In many ways, a communication diagram is a simplified version of a collaboration diagram introduced in UML 2.0.

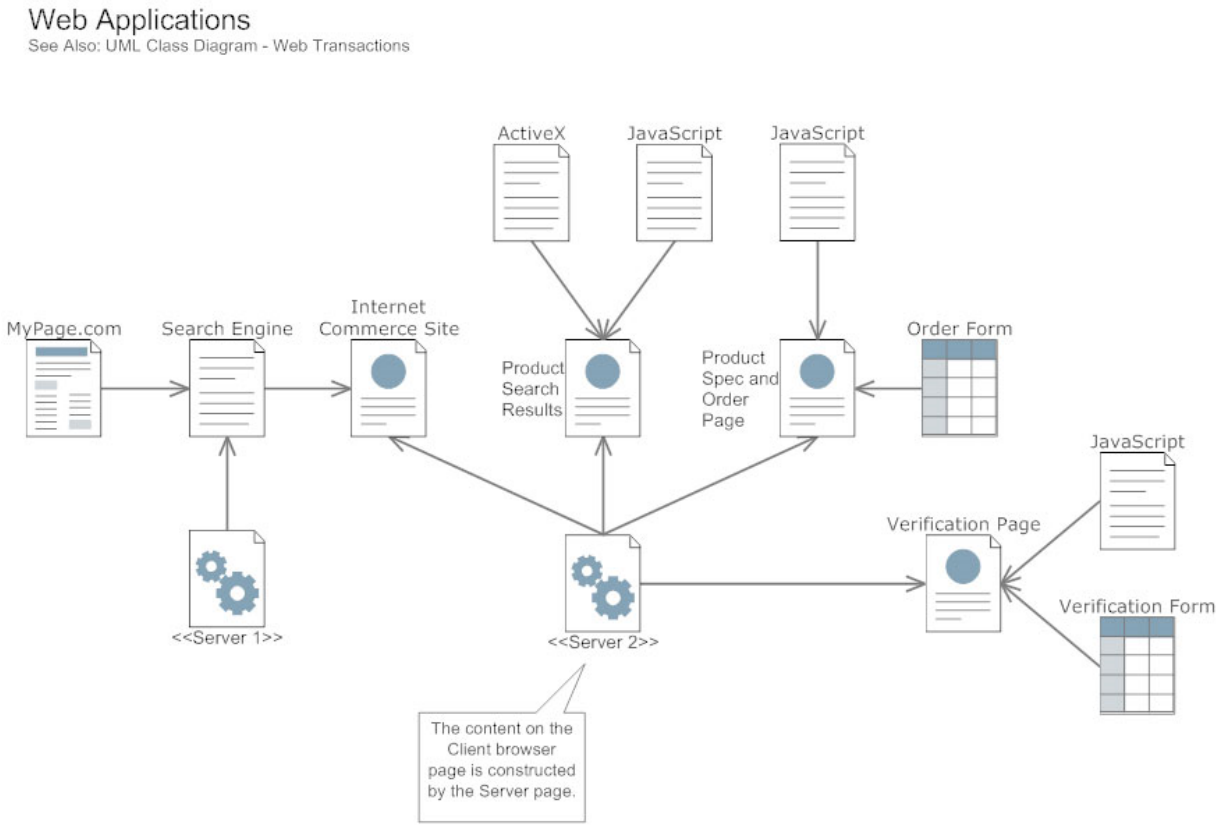
State Diagram

State chart diagrams, now known as state machine diagrams and state diagrams describe the dynamic behaviour of a system in response to external stimuli. State diagrams are especially useful in modelling reactive objects whose states are triggered by specific events.



Component Diagram

Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.



Deployment Diagram

Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.

UML Diagram Symbols

There are many different types of UML diagrams and each has a slightly different symbol set.

Class diagrams are perhaps one of the most common UML diagrams used and class diagram symbols centre around defining attributes of a class. For example, there are symbols for active classes and interfaces. A class symbol can also be divided to show a class's operations, attributes, and responsibilities.

Visibility of any class members are marked by notations of

\+ *For Public*

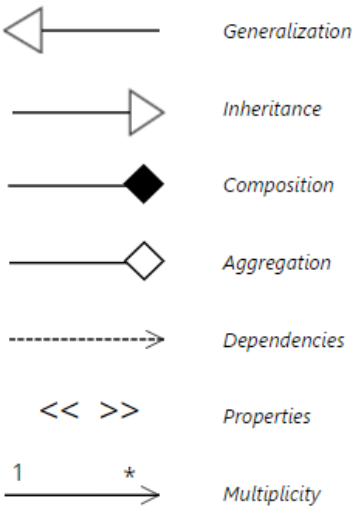
\- *For Private*

\# *For Protected*

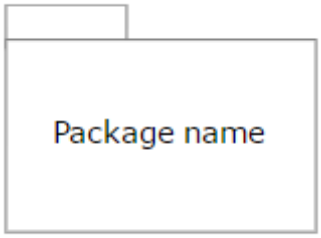
/ *For Derived*

~ *For Package*

Lines are also important symbols to denote relationships between components. Generalization and Inheritance are denoted with empty arrowheads. Composition is shown with a filled in diamond. Aggregation is shown with an empty diamond. Dependencies are marked with a dashed line with an arrow. Using << >> allows you to indicate properties of that dependency. Multiplicity is usually shown with a number at one end of the arrow and a * at the other.



Package diagrams have symbols defining a package that look like a folder.



Activity diagrams have symbols for activities, states, including separate symbols for an initial state and a final state. The control flow is usually shown with an arrow and the object flow is shown with a dashed arrow.

Use case diagrams have symbols for actors and use cases.

Why Do We Use UML?

A complex enterprise application with many collaborators will require a solid foundation of planning and clear, concise communication among team members as the project progresses.

Visualizing user interactions, processes, and the structure of the system you're trying to build will help save time down the line and make sure everyone on the team is on the same page.

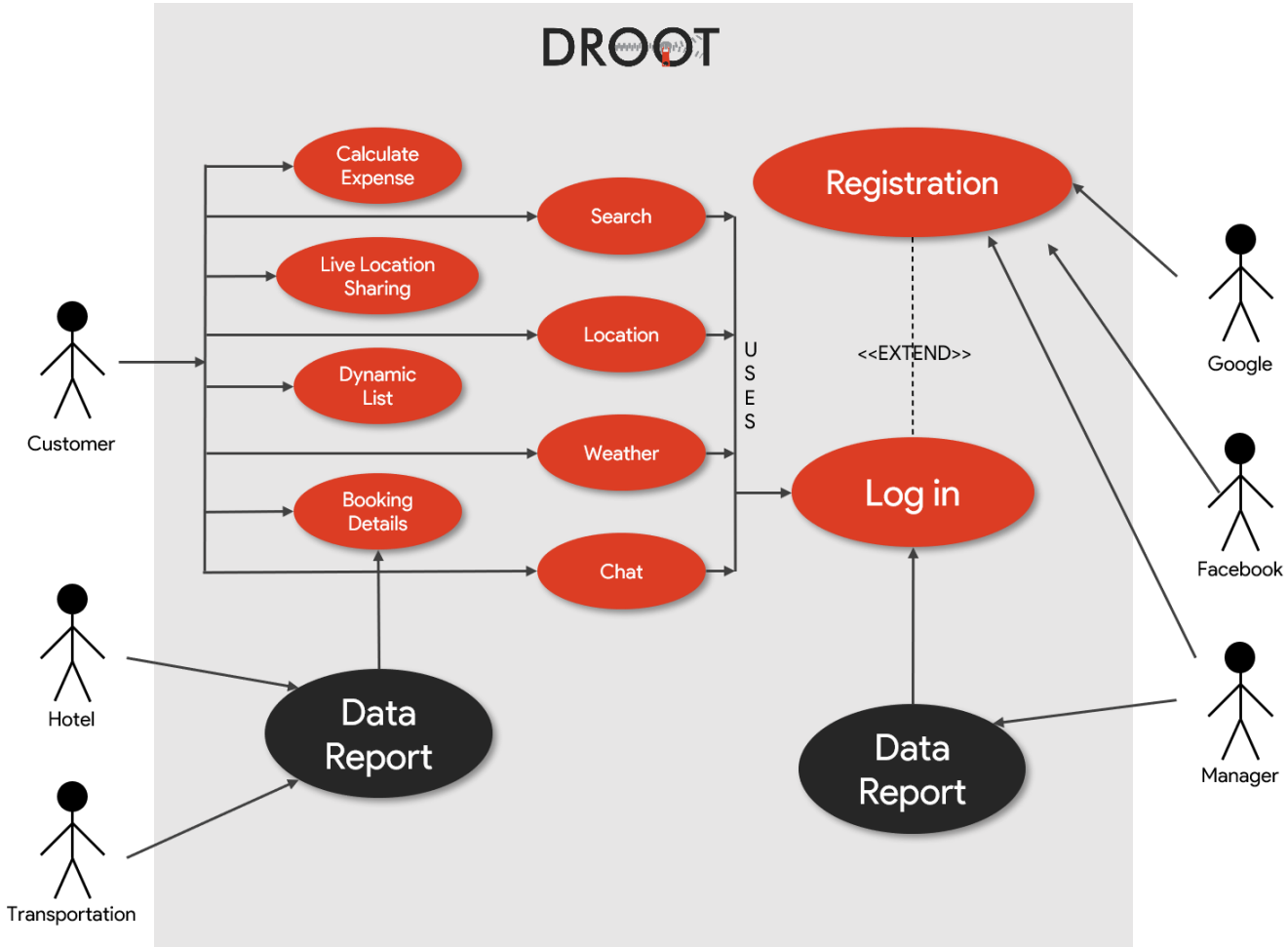
Diagrams from Data

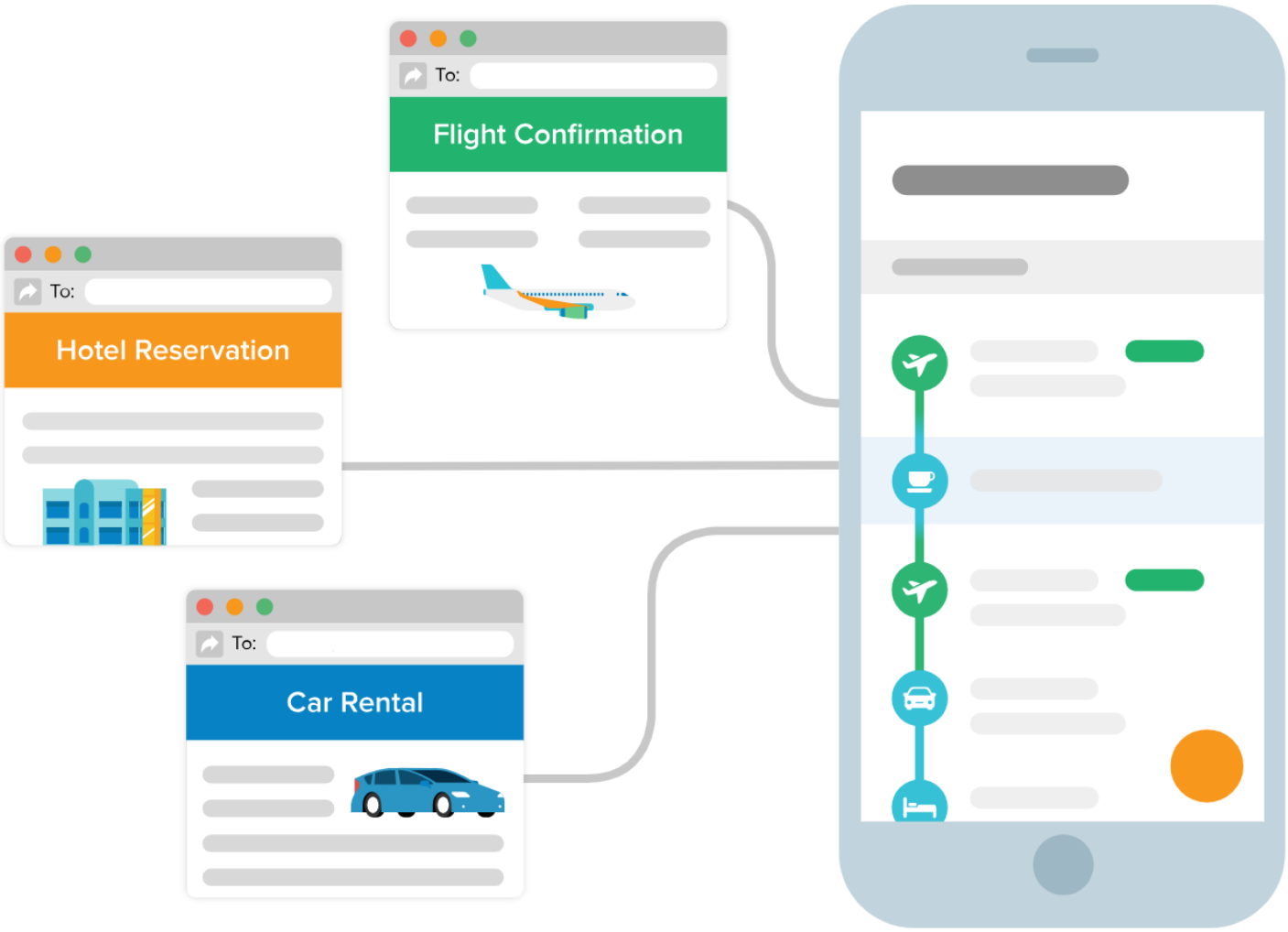
Smart Draw has an extension to generate UML class diagrams automatically using a GitHub repo or a local repository. Learn more about how to build a class diagram without drawing at all using Smart Draw's Class Diagram Extension.

You can also write your own extension to generate other UML and software design diagrams using Smart Draw's Open API

UML Diagram of Droot

Use case diagram




To:
Flight Confirmation
To:
Hotel Reservation
To:
Car Rental

Implementation

The implementation of this idea will be done by making a mobile app which will work on both android and iOS. For the development of this application Flutter (Google's SDK) and Dart (language) will be used. Flutter is an open-source UI software development kit created by Google. It is used to develop applications for Android, iOS, Windows, Mac, Linux, Google Fuchsia and the web.

After the application is built, it will go on the App Store and Google Play. Regular updates to this app will be provided which will include some feature updates.

Software Testing

Introduction to Testing

Testing is a process, which reveals errors in the program. It is the major quality measure employed during software development. During software testing, the program is executed with a set of test cases and the output of the program for the test cases of evaluated to determine if the program is performing as it is expected to perform.

Testing Strategies

In order to make sure that the system does not have errors, the different levels of testing strategies that are applied at different phases of software development are:

Unit Testing:

Unit Testing is done on individual modules as they are completed and become executable. It is confined only to the designer's requirements.

Each module can be tested using the following two Strategies:

Black Box Testing:

In this strategy some test cases are generated as input conditions that fully execute all functional requirements for the program. This testing has been uses to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structure or external database access
- Performance errors
- Initialization and termination errors.

In this testing only the output is checked for correctness.

The logical flow of the data is not checked.

White Box testing:

In this the test cases are generated on the logic of each module by drawing flow graphs of that module and logical decisions are tested on all the cases. It is been used to generate the test cases in the following cases:

- Guarantee that all independent paths have been Executed.
- Execute all logical decisions on their true and false sides.
- Execute all loops at their boundaries and within their operational bounds.
- Execute internal data structures to ensure their validity.

Integrating Testing:

Integration testing ensures that software and subsystems work together a whole. It tests the interface of all the modules to make sure that the modules behave properly when integrated together.

System Testing:

Involves in-house testing of the entire system before delivery to the user. It's aim is to satisfy the user the system meets all requirements of the client's specifications.

Test Approach:

Testing can be done in two ways:

- Bottom up approach
- Top down approach

Bottom up approach:

Testing can be performed staring from smallest and lowest modules and proceeding one at a time. For each module in bottom up testing a short program executes the module and provides the needed data so that the module is asked to perform the way it will when embedded with in the larger system. When bottom level modules are tested attention turns to those on the next level that use the lower level ones they are tested individually and then linked with previously examined lower level modules.

Top down approach:

This type of testing starts from upper level modules. Since the detailed activities usually performed in the lower level routines are not provided stubs are written. A stub is a module shell called by upper level module and that when reached properly will return a message to the calling module indicating that proper interaction occurred. No attempt is made to verify the correctness of the lower level module.

Validation:

The system has been tested and implemented successfully and thus ensured that all the requirements as listed in the software requirements specification are completely fulfilled. In case of erroneous input corresponding error messages are displayed.

Cost estimation and maintenance

For the Cost estimation of our Project we are going to use COCOMO Basic Model. COCOMO is one the most widely used software estimation models in the world. The Constructive Cost Model (COCOMO) is a procedural software cost estimation model. COCOMO is used to estimate size, effort and duration based on the cost of the software.

COCOMO predicts the effort and schedule for a software product development based on inputs relating to the size of the software and a number of cost drivers that affect productivity. COCOMO has three different models that reflect the complexities:

- Basic Model: This model would be applied early in a project development. It will provide a rough estimate early on that should be refined later with one of the other models.

- Intermediate Model: This model would be used after you have more detailed requirements for a project.

- Detailed Model: When design of the project is complete you can apply this model to further refine your estimate.

Within each of these models there are also three different modes. The mode you choose will depend on your work environment, and the size and constraints of the project itself.

Within these models we are going to COCOMO basic model for our cost estimation process. The mode of work will be **organic** i.e. relatively small software teams developing software in a highly familiar, in-house environment.

Basic Model: The basic COCOMO model estimates the software development effort using only Lines Of Code (LOC). Various equations in this model are:

$$\text{Effort Applied (E)} = ab(\text{KLOC})^{bb} \qquad \text{[man-months]}$$

$$\text{Development Time (D)} = cb(\text{Effort Applied})^{db} \qquad \text{[months]}$$

$$\text{People required (P)} = \text{Effort Applied} / \text{Development Time} \qquad \text{[count]}$$

Where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for project. The coefficients ab, bb, cb and db are given in the following table:

Software Projects	Ab	bb	cb	db
Organic	2.40	1.05	2.50	0.38
Semi detached	3.00	1.12	2.50	0.35
Embedded	3.60	1.20	2.50	0.32

Cost Estimation of our Project:

For our Project KLOC i.e. total line of code is about **5000 lines** .

1.

$$KLOC=5000$$

$$\text{Effort Applied (E)} = ab(5000)^{bb}$$

$$=2.40X(5000)^{1.05}$$

Effort Applied (E) = 18370.87 PM

2.

$$\text{Development Time (D)} = cb(\mathbf{18370.87})^{db}$$

$$=2.50X(\mathbf{18370.87})^{0.38}$$

Development Time (D)= 104.30 M

3.

$$\text{People required (P)} = \text{Effort Applied} / \text{Development Time}$$

$$=15680.23/98.2$$

People required (P) =176.13