

JSPM'S
JAYAWANTRAO SAWANT COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER
ENGINEERING

LABORATORY MANUAL
Computer Laboratory – IV

BE (Computer)
SEMESTER – I
(2023-24)
2019 Pattern

Course coordinator
Prof. N. R. Zinzurke
Prof. R. N. Pawar

Teaching Scheme: Practical: 4 Hours/Week,

Credit: 01

Examination Scheme: TW: 50

Companion Course: Elective III (410244), Elective IV (410245)

Course Outcomes:

After completion of the course, students will be able to

CO1: Apply android application development for solving real life problems

CO2: Design and develop system using various multimedia components.

CO3: Identify various vulnerabilities and demonstrate using various tools.

CO4: Apply information retrieval tools for natural language processing

CO5: Develop an application using open source GPU programming languages

CO6: Apply software testing tools to perform automated testing

Part A

List of Experiments

Group 1

1. Draw state model for telephone line, with various activities.
2. Draw basic class diagrams to identify and describe key concepts like classes, types in your system and their relationships.
3. Draw one or more Use Case diagrams for capturing and representing requirements of the system. Use case diagrams must include template showing description and steps of the Use Case for various scenarios.
4. Draw activity diagrams to display either business flows or like flow charts
5. Draw component diagrams assuming that you will build your system reusing existing components along with a few new ones
6. Draw deployment diagrams to model the runtime architecture of your system.

Group 2

8. Mini Project: Draw all UML diagrams for your project work.
9. Mini Project: Draw following UML Diagrams for Bank Management application
 - a. Class Diagram
 - b. Object Diagram
 - c. ER Diagram
 - d. Component Diagram

Part B

410245(A) Information Retrieval Any 5 assignments from group 1 and 1 Mini project from group 2 is mandatory

Group 1:

- 1. Write a program to Compute Similarity between two text documents.**
- 2. Implement Page Rank Algorithm.**
- 3. Write a program for Pre-processing of a Text Document: stop word removal.**
- 4. Write a map-reduce program to count the number of occurrences of each alphabetic character in the given dataset. The count for each letter should be case-insensitive (i.e., include both uppercase and lower-case versions of the letter; Ignore non-alphabetic characters).**
- 5. Write a program to implement simple web crawler.**
- 6. Write a program to parse XML text, generate Web graph and compute topic specific page**

Group 2:

- 7. Mini project: Develop Document summarization system**
- 8. Mini Project: Develop Tweet sentiment analysis system**
- 9. Mini Project: Develop Fake news detection system**

Experiment No. 1

Title: Draw state model for telephone line, with various activities.

Theory:

Intention of this subject (object oriented modelling and design) is to learn how to apply object-oriented concepts to all the stages of the software development life cycle.

Object-oriented modelling and design is a way of thinking about problems using models organized around real world concepts. The fundamental construct is the object, which combines both data structure and behaviour.

WHAT IS OBJECT ORIENTATION?

Definition: OO means that we organize software as a collection of discrete objects (that incorporate both data structure and behaviour).

There are four aspects (characteristics) required by an OO approach Identity.

- Classification.
- Inheritance.
- Polymorphism.
- Identity

WHAT IS OO DEVELOPMENT?

Development refers to the software life cycle: Analysis, Design and Implementation. The essence of OO Development is the identification and organization of application concepts, rather than their final representation in a programming language.

It is a conceptual process independent of programming languages. OO development is fundamentally a way of thinking and not a programming technique.

OO methodology:

Here we present a process for OO development and a graphical notation for representing OO concepts. The process consists of building a model of an application and then adding details to it during design.

The methodology has the following stages System conception: Software development begins with business analysis or users conceiving an application and formulating tentative requirements.

Three models: We use three kinds of models to describe a system from different viewpoints.

1. **Class Model**—for the objects in the system & their relationships.

It describes the static structure of the objects in the system and their relationships.

Class model contains class diagrams- a graph whose nodes are classes and arcs are relationships among the classes.

2. State model—for the life history of objects.

It describes the aspects of an object that change over time. It specifies and implements control with state diagrams—a graph whose nodes are states and whose arcs are transition between states caused by events.

3. Interaction Model—for the interaction among objects.

It describes how the objects in the system co-operate to achieve broader results.

This model starts with use cases that are then elaborated with sequence and activity diagrams.

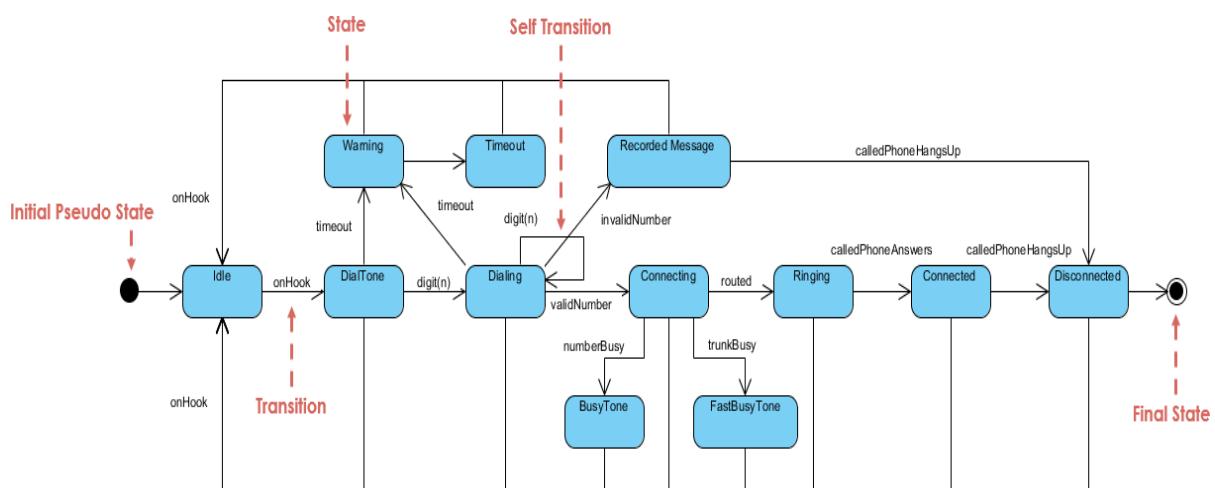
State Diagram Example – Phone Call

This is a UML state machine diagram example for telephone.

A state diagram is a graph whose node are state and whose directed arcs are transition between states which describes sequence caused by event sequences. A state diagram typically models the common behaviour of a class

In this example, the phone line is idle at the start of a call. When the phone is removed from the hook, it emits a dial tone and can accept the dialling of digits. Upon entry a valid number, the phone system tries to connect the call and route to the proper destination. The connection can fail if the number or trunk are busy. If the connection is successful the called phone begins ringing. When put on hook again, the phone line will go back to idle.

Result:



Conclusion: In this practical we conclude that how to draw the state diagram for Telephone Line.

Experiment No. 2

Title: Draw basic class diagrams to identify and describe key concepts like classes, types in your system and their relationships.

Theory:

Class Modelling

The first step in analyzing the requirements is to construct an class model. The class model shows the static data structure of the real-world system and organizes it into workable pieces. The class model describes real-world classes and their relationships to each other. Most crucial is the top level organization of the system into classes connected by associations: lower-level partitions within classes (generalizations) are less critical. The class models precedes the dynamic model and functional model because static structure evolves, and easier for human to understand.

Information for the class model comes from the problem statement, expert knowledge of the application domain, and general knowledge of the real world. If the designer is not a domain expert, the information must be obtained from the application expert and checked against the model repeatedly. Class model diagrams promote communication between computer professionals and applications-domain experts

The following steps are performed in constructing a class model:

Identify classes

Prepare a data dictionary

Identify associations (including aggregations) between class

Identify attributes of classes and links

Organize and simplify classes using inheritance

Verify that access paths exist for likely queries

Iterate and refine the model

Group classes into modules

Class:

A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as methods. Apart from business functionality, a class also has properties that reflect unique features of a class. The properties of a class are called as attributes. The UML representation of a class is a rectangle containing three compartments stacked vertically as shown in figure.

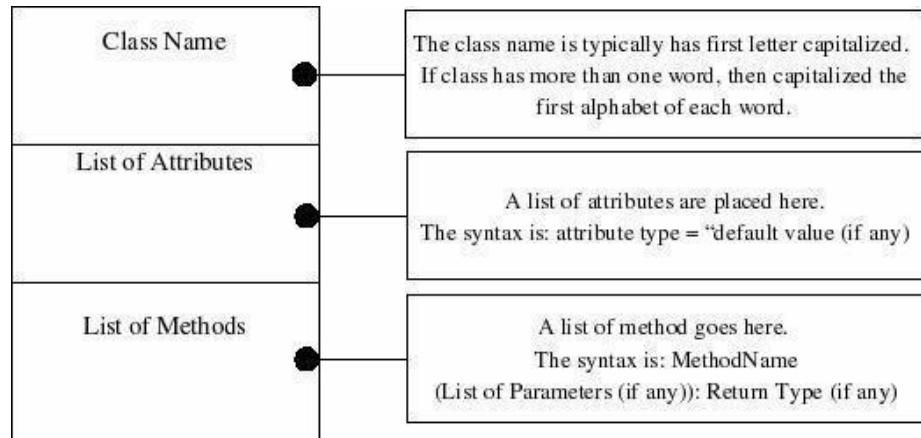
In the diagram, classes are represented with boxes which contain three parts:

The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.

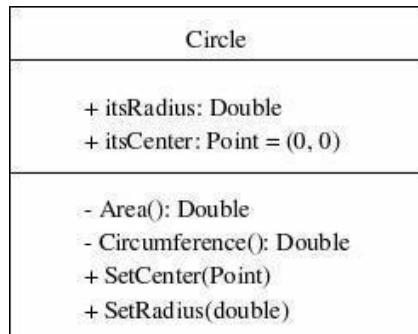
The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.

The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modeling, the classes of the conceptual design are often split into a number of subclasses.



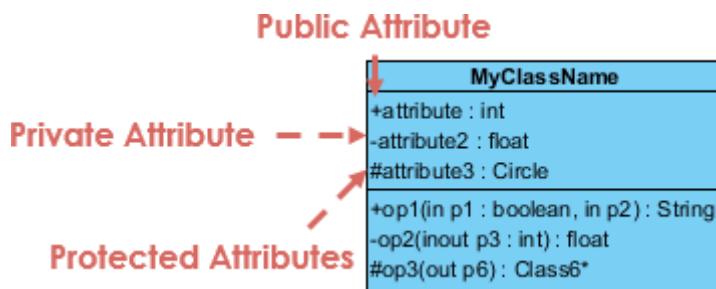
circle modelled as class



Circle class showing name, attributes, and methods

Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



+ denotes public attributes or operations

- denotes private attributes or operations

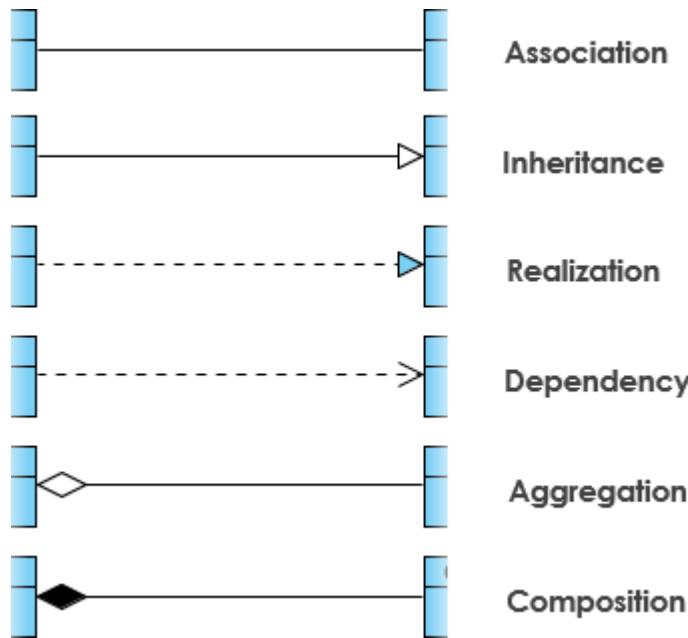
denotes protected attributes or operations

Relations

Relationships in class diagrams. In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between model elements. You can set properties and use keywords to create variations of these relationships.

Relationships between classes

A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

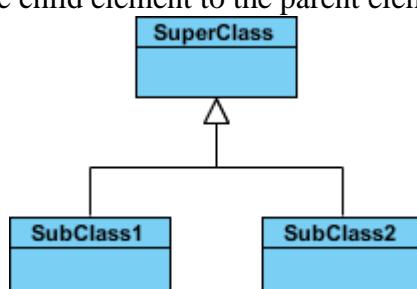


Inheritance (or Generalization):

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.

The figure below shows an example of inheritance hierarchy. SubClass1 and SubClass2 are derived from SuperClass. The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element.



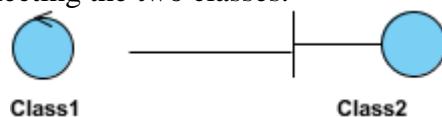
Association

Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real world problem domain.

Simple Association

- A structural link between two peer classes.
- There is an association between Class1 and Class2

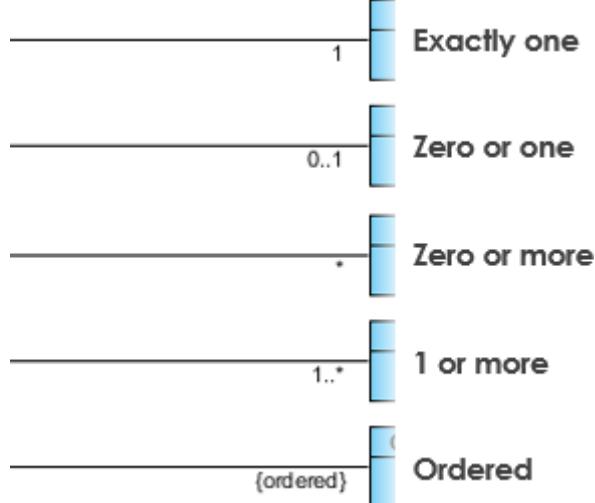
The figure below shows an example of simple association. There is an association that connects the <>control>> class Class1 and <>boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.



Cardinality

Cardinality is expressed in terms of:

- one to one
- one to many
- many to many

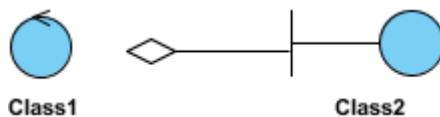


Aggregation

A special type of association.

- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.

The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.



Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.

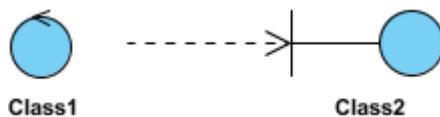


Dependency

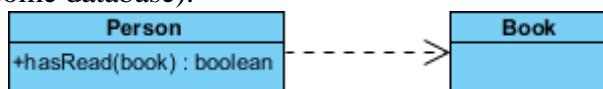
An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

- A special type of association.
- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2

The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.



The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).

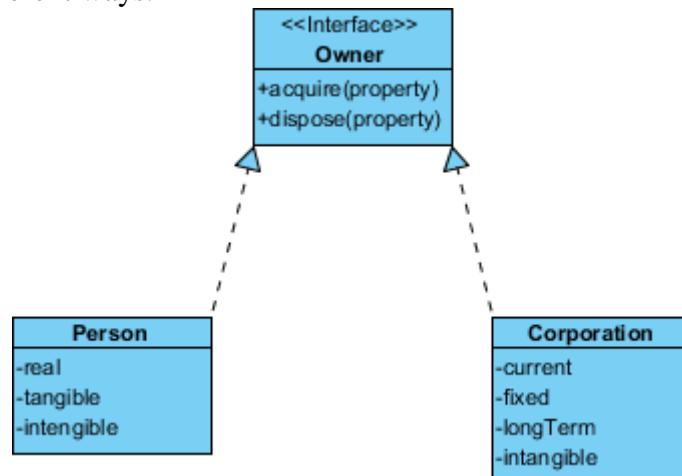


Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In

other words, you can understand this as the relationship between the interface and the implementing class.

For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.



Draw the class diagram for Telephone line system by using all the Class diagram symbols in Star UML software.

Conclusion: In this practical we conclude that how to draw the Class diagram for Telephone line system,

Experiment No. 3

Title: Draw one or more Use Case diagrams for capturing and representing requirements of the system. Use case diagrams must include template showing description and steps of the Use Case for various scenarios.

Theory:

Use case diagrams

Use case diagrams belong to the category of behavioural diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform, one or more actors, and dependencies among them.

Actor

An actor can be defined as [1] an object or set of objects, external to the system, which interacts with the system to get some meaningful work done. Actors could be human, devices, or even other systems.

For example, consider the case where a customer withdraws cash from an ATM. Here, customer is a human actor.

Actors can be classified as below [2], [i] :

Primary actor: They are principal users of the system, who fulfill their goal by availing some service from the system. For example, a customer uses an ATM to withdraw cash when he needs it. A customer is the primary actor here.

Supporting actor: They render some kind of service to the system. "Bank representatives", who replenishes the stock of cash, is such an example. It may be noted that replenishing stock of cash in an ATM is not the prime functionality of an ATM.

In a use case diagram primary actors are usually drawn on the top left side of the diagram.

Use Case

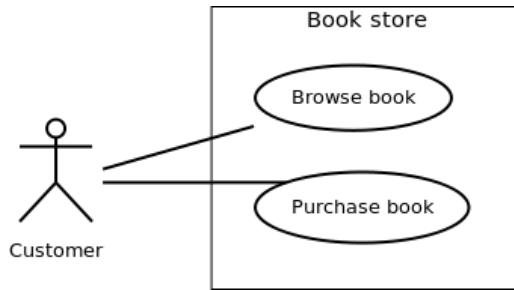
A use case is simply [1] functionality provided by a system.

Continuing with the example of the ATM, withdraw cash is a functionality that the ATM provides. Therefore, this is a use case. Other possible use cases include, check balance, change PIN, and so on.

Use cases include both successful and unsuccessful scenarios of user interactions with the system. For example, authentication of a customer by the ATM would fail if he enters wrong PIN. In such case, an error message is displayed on the screen of the ATM.

Graphical Representation

An actor is represented by a stick figure and name of the actor is written below it. A use case is depicted by an ellipse and name of the use case is written inside it. The subject is shown by drawing a rectangle. Label for the system could be put inside it. Use cases are drawn inside the rectangle, and actors are drawn outside the rectangle, as shown in figure - 01.



A use case diagram for a book store

Association between Actors and Use Cases

A use case is triggered by an actor. Actors and use cases are connected through binary associations indicating that the two communicate through message passing.

An actor must be associated with at least one use case. Similarly, a given use case must be associated with at least one actor. Association among the actors are usually not shown. However, one can depict the class hierarchy among actors.

Use Case Relationships

Three types of relationships exist among use cases:

Include relationship

Extend relationship

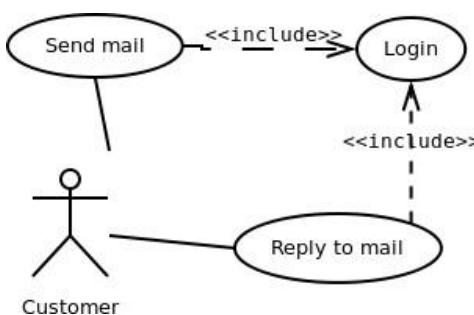
Use case generalization

Include Relationship

Include relationships are used to depict common behaviour that are shared by multiple use cases. This could be considered analogous to writing functions in a program in order to avoid repetition of writing the same code. Such a function would be called from different points within the program.

Example

For example, consider an email application. A user can send a new mail, reply to an email he has received, or forward an email. However, in each of these three cases, the user must be logged in to perform those actions. Thus, we could have a login use case, which is included by compose mail, reply, and forward email use cases. The relationship is shown in figure - 02.



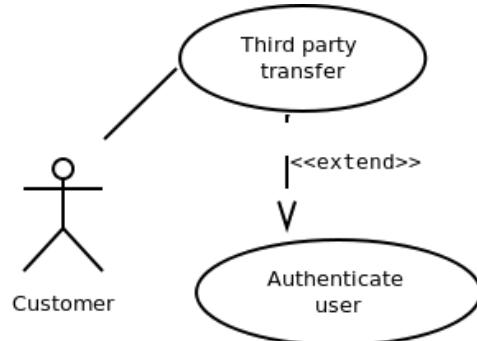
Include relationship between use cases

Extend Relationship

Use case extensions are used to depict any variation to an existing use case. They are used to specify the changes required when any assumption made by the existing use case becomes false [iv, v].

Example

Let's consider an online bookstore. The system allows an authenticated user to buy selected book(s). While the order is being placed, the system also allows to specify any special shipping instructions [vii], for example, call the customer before delivery. This Shipping Instructions step is optional, and not a part of the main Place Order use case. Figure depicts such relationship.



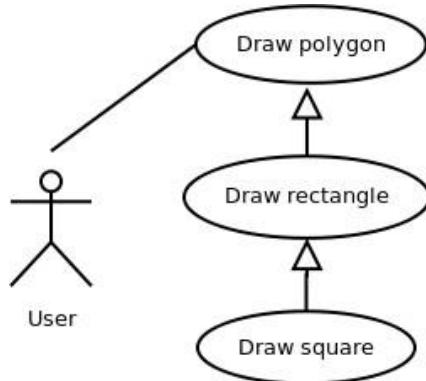
Extend relationship between use cases

Generalization Relationship

Generalization relationships are used to represent the inheritance between use cases. A derived use case specializes some functionality it has already inherited from the base use case.

Example

To illustrate this, consider a graphical application that allows users to draw polygons. We could have a use case "draw polygon". Now, rectangle is a particular instance of polygon having four sides at right angles to each other. So, the use case "draw rectangle" inherits the properties of the use case "draw polygon" and overrides its drawing method. This is an example of generalization relationship. Similarly, a generalization relationship exists between "draw rectangle" and "draw square" use cases. The relationship has been illustrated in figure.



Generalization relationship among use cases

Draw the Usecase diagram for Telephone line system by using all the Usecase diagram symbols in Star UML software.

Conclusion: In this practical we conclude that how to draw the Use Case diagram for Telephone Line System.

Experiment No. 4.

Title: Draw activity diagrams to display either business flows or like flow charts

Theory:

Activity Diagrams

Activity diagrams fall under the category of behavioural diagrams in Unified Modelling Language. It is a high level diagram used to visually represent the flow of control in a system. It has similarities with traditional flow charts. However, it is more powerful than a simple flow chart since it can represent various other concepts like concurrent activities, their joining, and so on.

Activity diagrams, however, cannot depict the message passing among related objects. As such, it can't be directly translated into code. These kind of diagrams are suitable for confirming the logic to be implemented with the business users. These diagrams are typically used when the business logic is complex. In simple scenarios it can be avoided entirely [ix].

Components of an Activity Diagram

Below we describe the building blocks of an activity diagram.

Activity

An activity denotes a particular action taken in the logical flow of control. This could simply be invocation of a mathematical function, alter an object's properties and so on [x]. An activity is represented with a rounded rectangle, as shown in table-01. A label inside the rectangle identifies the corresponding activity.

There are two special type of activity nodes: initial and final. They are represented with a filled circle, and a filled in circle with a border respectively (table-01). Initial node represents the starting point of a flow in an activity diagram. There could be multiple initial nodes, which means that invoking that particular activity diagram would initiate multiple flows.

A final node represents the end point of all activities. Like an initial node, there could be multiple final nodes. Any transition reaching a final node would stop all activities.

Flow

A flow (also termed as edge, or transition) is represented with a directed arrow. This is used to depict transfer of control from one activity to another, or to other types of components, as we will see below. A flow is often accompanied with a label, called the guard condition, indicating the necessary condition for the transition to happen. The syntax to depict it is [guard condition].

Decision

A decision node, represented with a diamond, is a point where a single flow enters and two or more flows leave. The control flow can follow only one of the outgoing paths. The outgoing edges often have guard conditions indicating true-false or if-then-else conditions. However, they can be omitted in obvious cases. The input edge could also have guard conditions. Alternately, a note can be attached to the decision node indicating the condition to be tested.

Merge

This is represented with a diamond shape, with two or more flows entering, and a single flow leaving out. A merge node represents the point where at least a single control should reach before further processing could continue.

Fork

Fork is a point where parallel activities begin. For example, when a student has been registered with a college, he can in parallel apply for student ID card and library card. A fork is graphically depicted with a black bar, with a single flow entering and multiple flows leaving out.

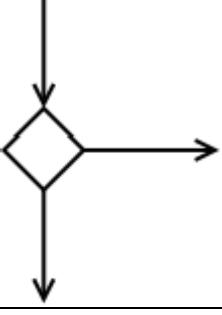
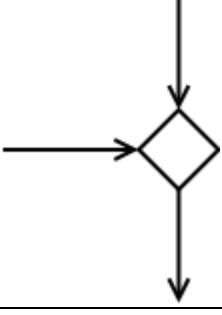
Join

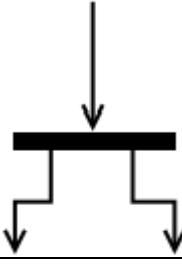
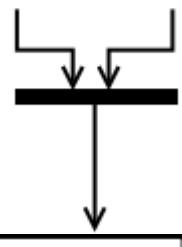
A join is depicted with a black bar, with multiple input flows, but a single output flow. Physically it represents the synchronization of all concurrent activities. Unlike a merge, in case of a join all of the incoming controls must be completed before any further progress could be made. For example, a sales order is closed only when the customer has received the product, and the sales company has received its payment.

Partition

Different components of an activity diagram can be logically grouped into different areas, called partitions or swimlanes. They often correspond to different units of an organization or different actors. The drawing area can be partitioned into multiple compartments using vertical (or horizontal) parallel lines. Partitions in an activity diagram are not mandatory.

The following table shows commonly used components with a typical activity diagram.

Component	Graphical Notation
Activity	
Flow	
Decision	
Merge	

Fork	
Join	
Note	A simple note

Typical components used in an activity diagram

Draw the Activity diagram for Telephone line system by using all the Activity diagram symbols in Star UML software.

Conclusion: In this practical we conclude that how to draw the Activity diagram for Telephone Line System.

Experiment No. 5.

Title: Draw component diagrams assuming that you will build your system reusing existing components along with a few new ones.

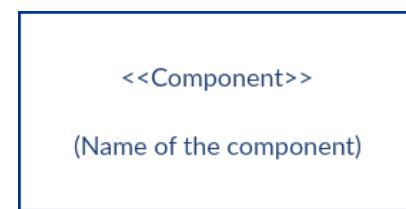
Theory:

Component diagrams are used to visualize the organization of system components and the dependency relationships between them. They provide a high-level view of the components within a system.

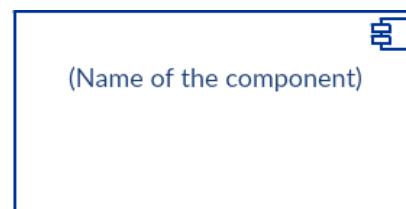
The components can be a software component such as a database or user interface; or a hardware component such as a circuit, microchip or device; or a business unit such as supplier, payroll or shipping.

Component diagrams

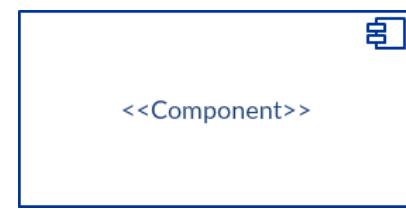
- Are used in Component-Based-Development to describe systems with Service-Oriented-Architecture
- Show the structure of the code itself
- Can be used to focus on the relationship between components while hiding specification detail
- Help communicate and explain the functions of the system being built to stakeholders
- Component Diagram Symbols
- We have explained below the common component diagram notations that are used to draw a component diagram.
- Component
- There are three ways the component symbol can be used.
- 1) Rectangle with the component stereotype (the text <<component>>). The component stereotype is usually used above the component name to avoid confusing the shape with a class icon.



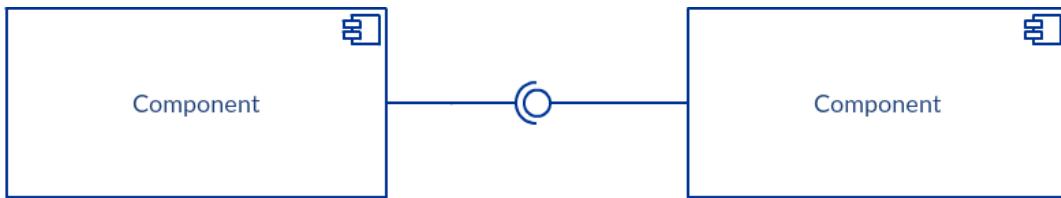
- 2) Rectangle with the component icon in the top right corner and the name of the component.



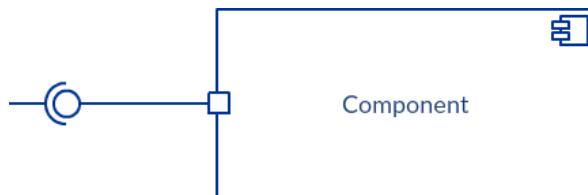
- 3) Rectangle with the component icon and the component stereotype.



- Provided Interface and the Required Interface



- Interfaces in component diagrams show how components are wired together and interact with each other. The assembly connector allows linking the component's required interface (represented with a semi-circle and a solid line) with the provided interface (represented with a circle and solid line) of another component. This shows that one component is providing the service that the other is requiring.
- Port



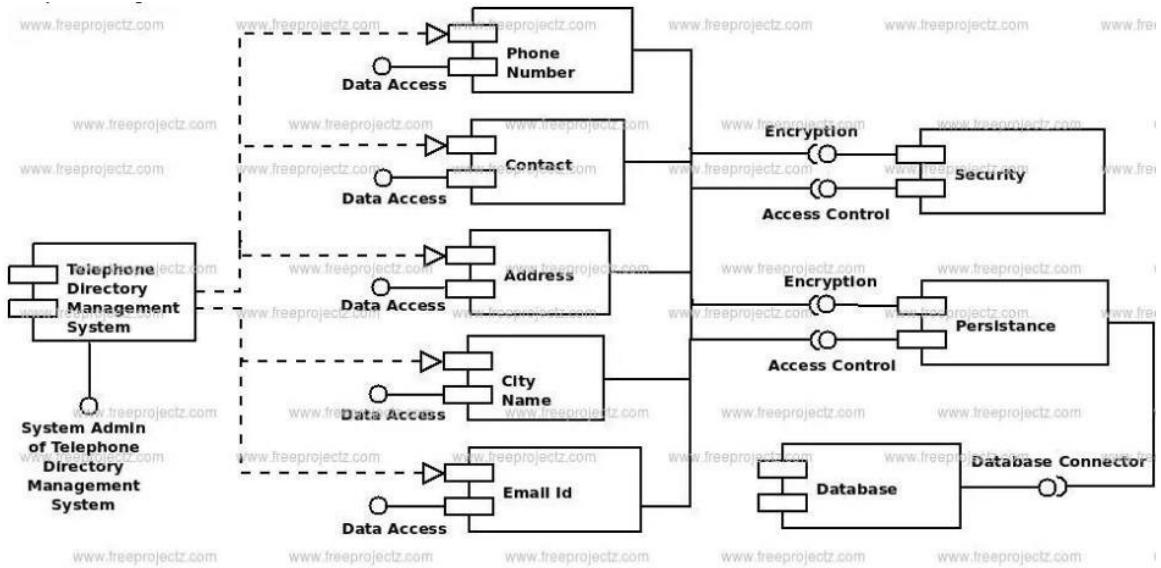
- Port (represented by the small square at the end of a required interface or provided interface) is used when the component delegates the interfaces to an internal class.
- Dependencies



- Although you can show more detail about the relationship between two components using the ball-and-socket notation (provided interface and required interface), you can just as well use a dependency arrow to show the relationship between two components.

• How to Draw a Component Diagram

- You can use a component diagram when you want to represent your system as components and want to show their interrelationships through interfaces. It helps you get an idea of the implementation of the system. Following are the steps you can follow when drawing a component diagram.
- **Step 1:** figure out the purpose of the diagram and identify the artifacts such as the files, documents etc. in your system or application that you need to represent in your diagram.
- **Step 2:** As you figure out the relationships between the elements you identified earlier, create a mental layout of your component diagram
- **Step 3:** As you draw the diagram, add components first, grouping them within other components as you see fit
- **Step 4:** Next step is to add other elements such as interfaces, classes, objects, dependencies etc. to your component diagram and complete it.
- **Step 5:** You can attach notes on different parts of your component diagram to clarify certain details to others.



Draw the Component diagram for Telephone line system by using all the Component diagram symbols in Star UML software.

Conclusion: In this practical we conclude that how to draw the Component diagram for Telephone Line System.

Experiment No. 6.

Title: Draw deployment diagrams to model the runtime architecture of your system.

Theory:

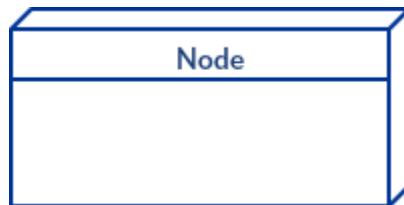
A deployment diagram is a UML diagram type that shows the execution architecture of a system, including nodes such as hardware or software execution environments, and the middleware connecting them.

Deployment diagrams are typically used to visualize the physical hardware and software of a system. Using it you can understand how the system will be physically deployed on the hardware. Deployment diagrams help model the hardware topology of a system compared to other UML diagram types which mostly outline the logical components of a system.

Deployment Diagram Notations

In order to draw a deployment diagram, you need to first become familiar with the following deployment diagram notations and deployment diagram elements.

Nodes



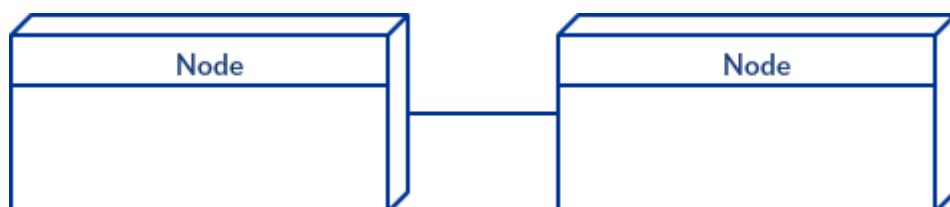
A node, represented as a cube, is a physical entity that executes one or more components, subsystems or executable. A node could be a hardware or software element.

Artifacts



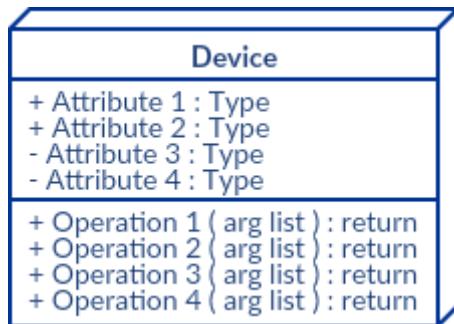
Artifacts are concrete elements that are caused by a development process. Examples of artifacts are libraries, archives, configuration files, executable files etc.

Communication Association



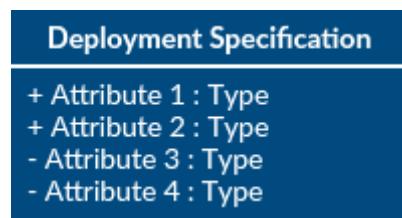
This is represented by a solid line between two nodes. It shows the path of communication between nodes.

Devices



A device is a node that is used to represent a physical computational resource in a system. An example of a device is an application server.

Deployment Specifications



Deployment specifications is a configuration file, such as a text file or an XML document. It describes how an artifact is deployed on a node.

Draw the Deployment diagram for Telephone line system by using all the deployment diagram symbols in Star UML software.

Conclusion: In this practical we conclude that how to draw the Deployment diagram for Telephone Line System.

410245(A) Information Retrieval Any 5 assignments from group 1 and 1 Mini project from group 2 is mandatory

Group 1:

Experiment No. 7.

Title: Write a program to Compute Similarity between two text documents.

Theory:

Document similarities are measured based on the content overlap between documents. With the large number of text documents in our life, there is a need to automatically process those documents for information extraction, similarity clustering, and search applications. There exist a vast number of complex algorithms to solve this problem. One of such algorithms is a cosine similarity - a vector based similarity measure. The cosine distance of two documents is defined by the angle between their feature vectors which are, in our case, word frequency vectors. The word frequency distribution of a document is a mapping from words to their frequency count.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

where " \cdot " denotes the dot-product of the two frequency vectors A and B, and $\|A\|$ denotes the length (or norm) of a vector.

Objectives

Gain experience in working with List, Set and Map interfaces.

Gain experience working with more realistic problems that are open-ended, with black-box specifications and requirements.

Example:

In this lab you are to compute a distance (an angle) between two given documents or between two strings using the cosine similarity metric. You start with reading in the text and counting frequency of each word. The word frequency distribution for the text D is Java's Map from words to their frequency counts, which we'll denote as freq(D). We view freq(D) as a vector of non-negative integers in N-dimensional space. For example, reading the string

"To be or not to be" results in the following map

{be=2, not=1, or=1, to=2}

These 4 distinct words make a document representation as a 4-dimensional vector {2, 1, 1, 2} in term space.

A word is a sequence of letters [a..zA..Z] that might include digits [0..9] and the underscore character. All delimiters are thrown away and not kept as part of the word. Here are examples of words:

abcd
abcd12
abc_
a12cd
15111

We'll treat all upper-case letters as if they are lower-case, so that "CMU" and "cmu" are the same word.

The Euclidean norm of the frequency vector is defined by

$$\|\mathbf{x}\| := \sqrt{x_1^2 + \dots + x_n^2}$$

where x_k denote frequencies for each word in the text. For the above example, the norm is

$$\sqrt{2^2 + 1^2 + 1^2 + 2^2} = 3.16228$$

The Dot product (or the inner product) of two frequency vectors X and Y is defined by

$$X = \{x_1, x_2, \dots, x_n\}; Y = \{y_1, y_2, \dots, y_n\};$$

$$X \cdot Y = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$$

Here we multiply frequencies x_k and y_k of the same word in both text documents.

Finally, we define a distance between two documents D_1 and D_2 by using cosine similarity measurement:

$$\text{dist}(D_1, D_2) = \arccos \left(\frac{\text{freq}(D_1) \cdot \text{freq}(D_2)}{\|\text{freq}(D_1)\| \cdot \|\text{freq}(D_2)\|} \right)$$

Observe, the distance between two identical documents is 0, and the distance is $\pi/2 = 1.57\dots$ if two documents have no common words.

Your task is to write a program to compute a distance (an angle) between two given documents using the formula above.

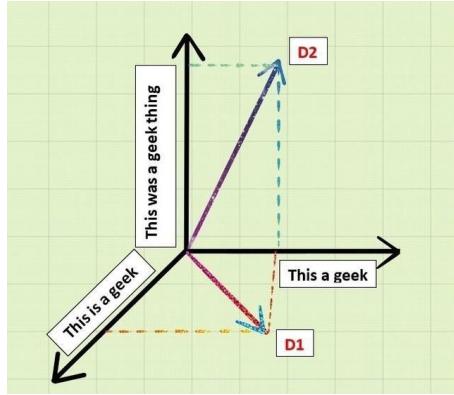
D1: —This is a geek!

D2: —This was a geek thing!

The similar words in both these documents then become:

"This a geek"

If we make a 3-D representation of this as vectors by taking D_1 , D_2 and similar words in 3 axis geometry, then we get:



Now if we take dot product of D1 and D2,

$$D1 \cdot D2 = "This". "This" + "is". "was" + "a". "a" + "geek". "geek" + "thing". 0$$

$$D1 \cdot D2 = 1 + 0 + 1 + 1 + 0$$

$$D1 \cdot D2 = 3$$

Now that we know how to calculate the dot product of these documents, we can now calculate the angle between the document vectors:

$$\cos d = D1 \cdot D2 / |D1| |D2|$$

Here d is the document distance. Its value ranges from 0 degree to 90 degrees. Where 0 degree means the two documents are exactly identical and 90 degrees indicate that the two documents are very different.

Conclusion: In this practical we implement to Compute Similarity between two text documents.

Experiment No. 8.

Title: Implement Page Rank Algorithm.

Theory:

PageRank (PR) is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages.

Algorithm

The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called “iterations”, through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

Simplified

algorithm

Assume a small universe of four web pages: A, B, C, and D. Links from a page to itself, or multiple outbound links from one single page to another single page, are ignored. PageRank is initialized to the same value for all pages. In the original form of PageRank, the sum of PageRank over all pages was the total number of pages on the web at that time, so each page in this example would have an initial value of 1. However, later versions of PageRank, and the remainder of this section, assume a probability distribution between 0 and 1. Hence the initial value for each page in this example is 0.25. The PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links. If the only links in the system were from pages B, C, and D to A, each link would transfer 0.25 PageRank to A upon the next iteration, for a total of 0.75.

$$PR(A) = PR(B) + PR(C) + PR(D)$$

Suppose instead that page B had a link to pages C and A, page C had a link to page A, and page D had links to all three pages. Thus, upon the first iteration, page B would transfer half of its existing value, or 0.125, to page A and the other half, or 0.125, to page C. Page C would transfer all of its existing value, 0.25, to the only page it links to, A. Since D had three outbound links, it would transfer one-third of its existing value, or approximately 0.083, to A. At the completion of this iteration, page A will have a PageRank of approximately 0.458.

$$PR(A) = PR(B)/2 + PR(C)/1 + PR(D)/3$$

In other words, the PageRank conferred by an outbound link is equal to the document's own PageRank score divided by the number of outbound links $L(\)$.

$$PR(A) = PR(B)/L(B) + PR(C)/L(C) + PR(D)/L(D)$$

In the general case, the PageRank value for any page u can be expressed as:

$$PR(u) = \sum_{v \in Bu} \frac{PR(v)}{L(v)}$$

i.e. the PageRank value for a page u is dependent on the PageRank values for each page v contained in the set B_u (the set containing all pages linking to page u), divided by the number $L(v)$ of links from page v . The algorithm involves a damping factor for the calculation of the PageRank.

Implement the algorithm in any programming language.

Conclusion: In this practical we Implement Page Rank Algorithm.

Experiment No. 9.

Title:3. Write a program for Pre-processing of a Text Document: stop word removal.

Theory:

In computer search engines, a stop word is a commonly used word (such as "the") that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query.

Stop Words: A stop word is a commonly used word (such as -the\|, -all\|, -an\|, -in\|) that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query.

Document Representative:

Documents in a collection are frequently represented through a set of index terms or keywords. Such keywords might be extracted directly from the text of the document or might be specified by a human subject. Modern computers are making it possible to represent a document by its full set of words. With very large collections, however, even modern computers might have to reduce the set of representative keywords. This can be accomplished through the elimination of stopwords (such as articles and connectives), the use of stemming (which reduces distinct words to their common grammatical root), and the identification of noun groups (which eliminates adjectives, adverbs, and verbs). Further, compression might be employed. These operations are called text operations (or transformations). The full text is clearly the most complete logical view of a document but its usage usually implies higher computational costs. A small set of categories (generated by a human specialist) provides the most concise logical view of a document but its usage might lead to retrieval of poor quality. Several intermediate logical views (of a document) might be adopted by an information retrieval system as illustrated in Figure. Besides adopting any of the intermediate representations, the retrieval system might also recognize the internal structure normally present in a document. This information on the structure of the document might be quite useful and is required by structured text retrieval models. As illustrated in Figure we view the issue of logically representing a document as a continuum in which the logical view of a document might shift (smoothly) from a full text representation to a higher level representation specified by a human subject.

The document representative is one consisting simply of a list of class names, each name representing a class of words occurring in the total input text. A document will be indexed by a name if one of its significant words occurs as a member of that class.

The removal of high frequency words, 'stop' words or 'fluff' words. This is normally done by comparing the input text with a 'stop list' of words which are to be removed. The advantages of the process are not only that non-significant words are removed and will therefore not interfere during retrieval, but also that the size of the total document file can be reduced by between 30 and 50 per cent.

List of stop words are:

{_ourselves‘, _hers‘, _between‘, _yourself‘, _but‘, _again‘, _there‘, _about‘, _once‘, _during‘, _out‘, _very‘, _having‘, _with‘, _they‘, _own‘, _an‘, _be‘, _some‘, _for‘, _do‘, _its‘, _yours‘, _such‘, _into‘, _of‘, _most‘, _itself‘, _other‘, _off‘, _is‘, _s‘, _am‘, _or‘, _who‘, _as‘, _from‘, _him‘, _each‘, _the‘, _themselves‘, _until‘, _below‘, _are‘, _we‘, _these‘, _your‘, _his‘,

`_through‘, _don‘, _nor‘, _me‘, _were‘, _her‘, _more‘, _himself‘, _this‘, _down‘, _should‘, _our‘,
_their‘, _while‘, _above‘, _both‘, _up‘, _to‘, _ours‘, _had‘, _she‘, _all‘, _no‘, _when‘, _at‘, _any‘,
_before‘, _them‘, _same‘, _and‘, _been‘, _have‘, _in‘, _will‘, _on‘, _does‘, _yourselves‘, _then‘,
_that‘, _because‘, _what‘, _over‘, _why‘, _so‘, _can‘, _did‘, _not‘, _now‘, _under‘, _he‘, _you‘,
_herself‘, _has‘, _just‘, _where‘, _too‘, _only‘, _myself‘, _which‘, _those‘, _i‘, _after‘, _few‘,
_whom‘, _t‘, _being‘, _if‘, _theirs‘, _my‘, _against‘, _a‘, _by‘, _doing‘, _it‘, _how‘, _further‘,
_was‘, _here‘, _than‘}`

Implement the program by using file handling concept and remove the stop words.

Program:

Output:

Conclusion: In this practical we implement to Compute Similarity between two text documents.

Experiment No. 10.

Title:4. Write a program to implement simple web crawler.

Theory:

Search Engines

A program that searches documents for specified keywords and returns a list of the documents where the keywords were found is a search engine. Although search engine is really a general class of programs, the term is often used to specifically describe systems like Google, Alta Vista and Excite that enable users to search for documents on the World Wide Web and USENET newsgroups.

Typically, a search engine works by sending out a spider to fetch as many documents as possible. Another program, called an indexer, then reads these documents and creates an index based on the words contained in each document. Each search engine uses a proprietary algorithm to create its indices such that, ideally, only meaningful results are returned for each query. Search engines are special sites on the Web that are designed to help people find information stored on other sites.

There are differences in the ways. Various search engines work, but they all perform three basic tasks:

They search the Internet - based on important words.

They keep an index of the words they find, and where they find them.

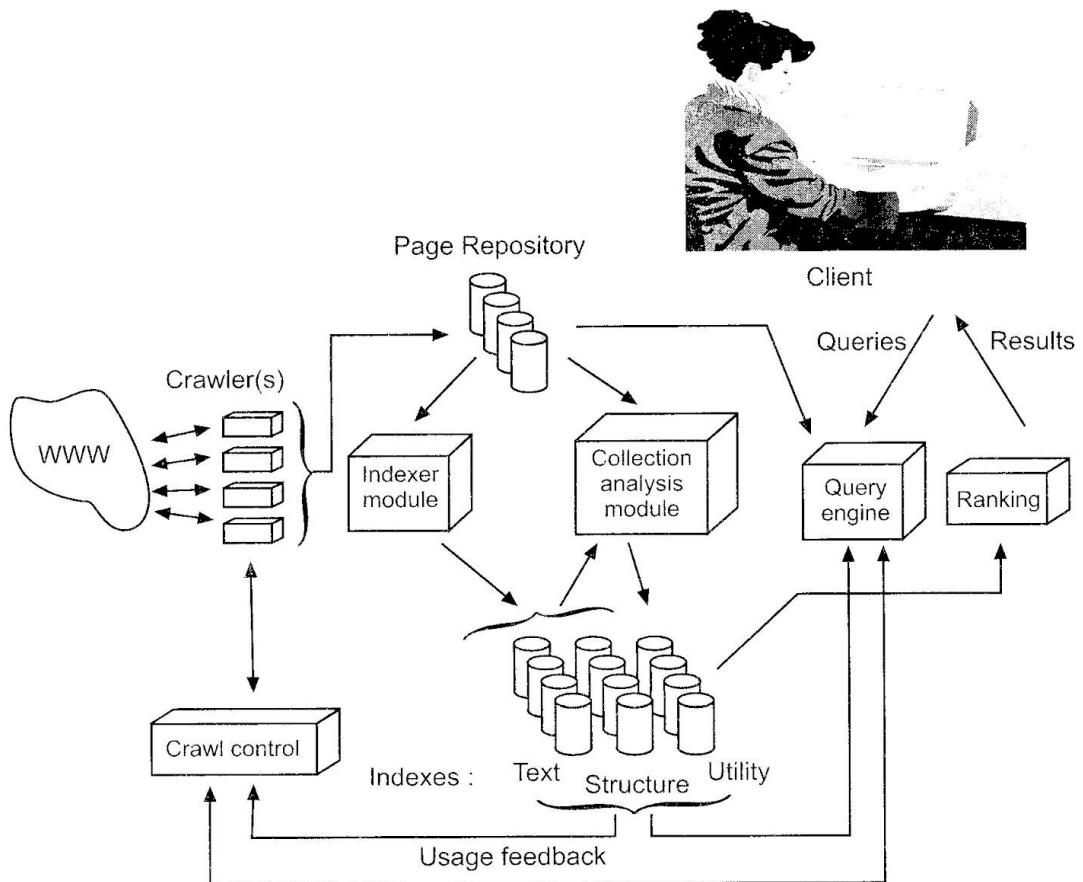
They allow users to look for words or combinations of words found in that index.

Fig.1 shows general search engine architecture. Every engine relies on a crawler module to provide the grist for its operation. Crawlers are small programs that browse the Web on the search engine's behalf, similar to how a human user would follow links to reach different pages. The programs are given a starting set of URLs, whose pages they retrieve from the Web.

The crawlers extract URLs appearing in the retrieved pages, and give this information to the crawler control module. This module determines what links to visit next, and feeds the links to visit back to the crawlers. (Some of the functionality of the crawler control module may be implemented by the crawlers themselves.) The crawlers also pass the retrieved pages into a page repository. Crawlers continue visiting the Web, until local resources, such as storages, are exhausted.

Web Crawlers

Web crawlers are programs that exploit the graph structure of the Web to move from page to page. It may be observed that the noun 'crawler' is not indicative of the speed of these programs, as they can be considerably fast. A key motivation for designing Web crawlers has been to retrieve Web pages and add them or their representations to a local repository. Such a repository may then serve particular application needs such as those of a Web search engine. In its simplest form, a crawler starts from a seed page and then uses the external links within it to attend to other pages.



The Crawler is the means by which WebCrawler collects pages from the Web. It operates by iteratively downloading a web page, processing it, and following the links in that page to other Web pages, perhaps on other servers. The end result of crawling is a collection of Web pages, HTML or plain text at a central location. The collection policy implemented in the crawler determines what pages are collected, and which of those pages are indexed. Although at first glance the process of crawling appears simple, many complications occur in practice. In a more traditional IR system, the documents to be indexed are available locally in a database or file system. However, since Web pages are distributed on millions of servers, we must first bring them to a central location before they can be indexed together in a single collection. WebCrawler's first information retrieval system was based on Salton's vector-space retrieval model.

The first system used a simple vector-space retrieval model. In the vector-space model, the queries and documents represent vectors in a highly dimensional word space. The component of the vector in a particular dimension is the significance of the word to the document. For example, if a particular word is very significant to a document, the component of the vector along that word's axis would be strong. In this vector space, then, the task of querying becomes that of determining what document vectors are most similar to the query vector. Practically speaking, this task amounts to comparing the query vector, component by component, to all the document vectors that have a word in common with the query vector. WebCrawler determined a similarity number for each of these comparisons that formed the basis of the relevance score returned to the user. WebCrawler's first IR system had three pieces: a query processing module, an inverted full-text index, and a metadata store. The query processing module parses the searcher's query, looks up the words in the inverted index, forms the result list, looks up the metadata for each result, and builds the HTML for the result page. The query processing module used a series of data structures and algorithms to generate results for a given query. First, this module put the query in a canonical form, and parsed each space-separated word in the query. If necessary, each word was converted to its

singular form using a modified Porter stemming algorithm and all words were filtered through the stop list to obtain the final list of words. Finally, the query processor looked up each word in the dictionary, and ordered the list of words for optimal query execution. WebCrawler's key contribution to distributed systems is to show that a reliable, scalable, and responsive system can be built using simple techniques for handling distribution, load balancing, and fault tolerance.

Conclusion: In this practical we implement simple web crawler.

Experiment No. 11.

Title: 5. Write a map-reduce program to count the number of occurrences of each alphabetic character in the given dataset. The count for each letter should be case-insensitive (i.e., include both uppercase and lower-case versions of the letter; Ignore non-alphabetic characters).

Theory:

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model.

A MapReduce program has two basic components provided by the user, a Mapper and a Reducer; these in turn have methods map and reduce. The MapReduce class calls the methods and manages the data.

map has two arguments, a String key and a String value, plus the MapReduce mr. For our purposes, the key will be a line number (sometimes used, sometimes not used), and the value will be a line of text. map will emit zero or more items of data by calling mr.collect_map with arguments of a key and a value that is a list of one String (potentially more). Note that the key in the call to mr.collect_map is often different from the key parameter to the map method. As an example, suppose that the mapper program detects some conditions in the lines of input text and wants to count those conditions; for condition foo, the call to emit this output would be mr.collect_map("foo", list("1")); In this case, "foo" is the key, and the value is a list of one String value, in this case denoting one occurrence of foo.

The reduce method gets a key that is a String, and a value that is a list of lists of (usually one) String. The value contains all the values emitted for that key by the map method, collected into a single list. The task of reduce is to produce an answer from the list of values and output that answer by calling mr.collect_reduce . The answer is of type Cons, a list of the key and combined value. As an example, if there are 3 occurrences of foo, the value would be (("1") ("1") ("1")); the quotation marks are shown here to emphasize that the values are String and must be parsed to get integer values, e.g. by Integer.decode().

Given a string, write a program to count the occurrence of Lowercase characters, Uppercase characters, Special characters, and Numeric values.

Examples:

Input: #GeeKs01fOr@gEEks07

Output:

Upper case letters: 5

Lower case letters: 8

Numbers: 4

Special Characters: 2

Conclusion: In this practical we count the number of occurrences of each alphabetic character by using map-reduce.