2018-19 Onwards (MR-18)	**MALLA REDDY ENGINEERING COLLEGE (Autonomous)**	**B.Tech. VI Semester**		
Code: 80514	**SOFTWARE ENGINEERING & UML LAB**	**L**	**T**	**P**
Credits: 2		**-**	**1**	**2**

Prerequisite: NIL

Course Objectives: To understand the software engineering methodologies involved in the phases for project development. To gain knowledge about open source tools used for implementing software engineering methods. To exercise developing product-startups implementing software engineering methods. Open source Tools: StarUML / UMLGraph / Top cased.

Prepare the following documents and develop the software project startup, prototype model, using software engineering methodology for at least two real time scenarios or for the sample experiments.

- Problem Analysis and Project Planning -Thorough study of the problem – Identify Project scope, Objectives and Infrastructure.
- Software Requirement Analysis – Describe the individual Phases/modules of the project and Identify deliverables. Identify functional and non-functional requirements.
- Data Modeling – Use work products – data dictionary.
- Software Designing – Develop use case diagrams and activity diagrams, build and test class diagrams, sequence diagrams and add interface to class diagrams.
- Prototype model – Develop the prototype of the product.

The SRS and prototype model should be submitted for end semester examination.

List of Sample Experiments:

Course management system (CMS)

A course management system (CMS) is a collection of software tools providing an online environment for course interactions. A CMS typically includes a variety of online tools and environments, such as:

- An area for faculty posting of class materials such as course syllabus and handouts
- An area for student posting of papers and other assignments
- A grade book where faculty can record grades and each student can view his or her grades
- An integrated email tool allowing participants to send announcement email messages to the entire class or to a subset of the entire class
- A chat tool allowing synchronous communication among class participants
- A threaded discussion board allowing asynchronous communication among participants

In addition, a CMS is typically integrated with other databases in the university so that students enrolled in a particular course are automatically registered in the CMS as participants in that course.

The Course Management System (CMS) is a web application for department personnel, Academic Senate, and Registrar staff to view, enter, and manage course information formerly submitted via paper. Departments can use CMS to create new course proposals, submit changes for existing courses, and track the progress of proposals as they move through the stages of online approval.

Easy Leave

This project is aimed at developing a web based Leave Management Tool, which is of importance to either an organization or a college.
The Easy Leave is an Intranet based application that can be accessed throughout the organization or a specified group/Dept. This system can be used to automate the workflow of leave applications and their approvals. The periodic crediting of leave is also automated. There are features like notifications, cancellation of leave, automatic approval of leave, report generators etc in this Tool.

Functional components of the project:
There are registered people in the system. Some are approvers. An approver can also be a requestor. In an organization, the hierarchy could be Engineers/Managers/Business Managers/Managing Director etc. In a college, it could be Lecturer/Professor/Head of the Department/Dean/Principal etc.

Following is a list of functionalities of the system: A person should be able to

- login to the system through the first page of the application
- change the password after logging into the system
- see his/her eligibility details (like how many days of leave he/she is eligible for etc)
- query the leave balance
- see his/her leave history since the time he/she joined the company/college
- apply for leave, specifying the from and to dates, reason for taking leave, address for communication while on leave and his/her superior’s email id
- see his/her current leave applications and the leave applications that are submitted to him/her for approval or cancellation
- approve/reject the leave applications that are submitted to him/her
- withdraw his/her leave application (which has not been approved yet)
- Cancel his/her leave (which has been already approved). This will need to be approved by his/her Superior
- get help about the leave system on how to use the different features of the system
- As soon as a leave application /cancellation request /withdrawal /approval /rejection /password-change is made by the person, an automatic email should be sent to the person and his superior giving details about the action

- The number of days of leave (as per the assumed leave policy) should be automatically credited to everybody and a notification regarding the same be sent to them automatically
- An automatic leave-approval facility for leave applications which are older than 2 weeks should be there. Notification about the automatic leave approval should be sent to the person as well as his superior

E-Bidding

Auctions are among the latest economic institutions in place. They have been used since antiquity to sell a wide variety of goods, and their basic form has remained unchanged. In this dissertation, we explore the efficiency of common auctions when values are interdependent the value to a particular bidder may depend on information available only to others-and asymmetric. In this setting, it is well known that sealed-bid auctions do not achieve efficient allocations in general since they do not allow the information held by different bidders to be shared.

Typically, in an auction, say of the kind used to sell art, the auctioneer sets a relatively low initial price. This price is then increased until only one bidder is willing to buy the object, and the exact manner in which this is done varies. In my model a bidder who drops out at some price can "reenter" at a higher price. With the invention of E-commerce technologies over the Internet the opportunity to bid from the comfort of one's own home has seen a change like never seen before. Within the span of a few short years, what may have began as an experimental idea has grown to an immensely popular hobby, and in some cases, a means of livelihood, the Auction Patrol gathers tremendous response every day, all day. With the point and click of the mouse, one may bid on an item they may need or just want, and in moments they find that either they are the top bidder or someone else wants it more, and you're outbid! The excitement of an auction all from the comfort of home is a completely different experience.

Society cannot seem to escape the criminal element in the physical world, and so it is the same with Auction Patrols. This is one area where in a question can be raised as to How safe Auction Patrols.

Proposed system

- To generate the quick reports
- To make accuracy and efficient calculations
- To provide proper information briefly
- To provide data security
- To provide huge maintenance of records
  Flexibility of transactions can be completed in time

Electronic Cash counter

This project is mainly developed for the Account Division of a Banking sector to provide better interface of the entire banking transactions. This system is aimed to give a better out look to the user interfaces and to implement all the banking transactions like:

- Supply of Account Information
- New Account Creations
- Deposits
- Withdraws
- Cheque book issues
- Stop payments
- Transfer of accounts
- Report Generations.

Proposed System:

The development of the new system contains the following activities, which try to automate the entire process keeping in view of the database integration approach.

User friendliness is provided in the application with various controls.The system makes the overall project management much easier and flexible. Readily upload the latest updates, allows user to download the alerts by clicking the URL. There is no risk of data mismanagement at any level while the project development is under process. It provides high level of security with different level of authentication

Objectives:The student should take up the case studies of ATM system, Online Reservation System and Model it in different views i.e. Use case view, logical view, component view, Deployment view.

Week 1

Design a Use case Diagram for ATM system, Online Reservation System

Week 2

Design a Sequence Diagram for ATM system, Online Reservation System. Design a Collaboration Diagram for ATM system, Online Reservation System.

Week 3

Design a Activity Diagram for ATM system, Online Reservation System.
Design a State Chart Diagram for ATM system, Online Reservation System.

Week 4

Design a Class Diagram for ATM system, Online Reservation System.

Week 5

Design a Component Diagram for ATM system, Online Reservation System.

Week 6

Design a Deployment Diagram for ATM system, Online Reservation System.

Course Outcomes:

Upon completion of this course, students should be able to:

1. Analyze the requirements through Use-Case View
2. Identify all structural and behavioral concepts of the entire system
3. Develop a model using UML concepts by different types of diagrams like Usecase Diagram, Class Diagram, Sequence Diagram etc

CO- PO Mapping

(3/2/1 indicates strength of correlation) 3-Strong, 2-Medium, 1-Weak

COs	Programme Outcomes(POs)												PSOS		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	2			2							3	2	
CO2		3	2	2	2	2							2	2	
CO3	2	3	3	2	2	2							2	2	3

MALLA REDDY ENGINEERING COLLEGE (A)

DEPARTMENTOF
INFORMATION TECHNOLOGY

SOFTWARE ENGINEERING & UML LAB MANUAL

CONTENT

Experiment No.1

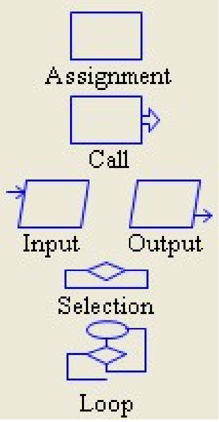
Aim: Develop Flow-Charts to understand basic problem solving technique by the help of Raptor tool.
Objective: To familiar with visual problem solving using Flow-Charts.
Software Required: Raptor 4.0

Procedure:

Introduction

One of the most important aspects of programming is controlling which statement will execute next. *Control structures* / *Control statements* enable a programmer to determine the order in which program statements are executed. These control structures allow you to do two things: 1) skip some statements while executing others, and 2) repeat one or more statements while some condition is true.

RAPTOR programs use six basic types of statements, as shown in the figure to the right. You have already learned about the four basic commands in a previous reading. In this reading you will learn about the **`Selection`** and **`Loop`** commands.

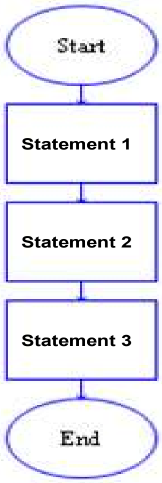


Basic Commands

Flow of Control Commands

Sequential Program Control

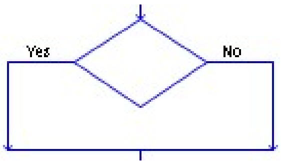
All of the RAPTOR programs you have seen in previous readings have used *sequential program control*. By sequential we mean "in sequence," one-after-the-other. Sequential logic is the easiest to construct and follow. Essentially you place each statement in the order that you want them to be executed and the program executes them in sequence from the `Start` statement to the `End` statement. As you can see by the example program to the right, the arrows linking the statements depict the execution flow. If your program included 20 basic commands then it would execute those 20 statements in order and then quit.



When you are solving a problem as a programmer, you must determine what statements are needed to create a solution to the problem and the order in which those statements must be executed. Writing the correct statements is one task. Determining where to place those statements in your program is equally important. For example, when you want to get and process data from the user you have to `GET` the data before you can use it. Switching the order of these statements would produce an invalid program.

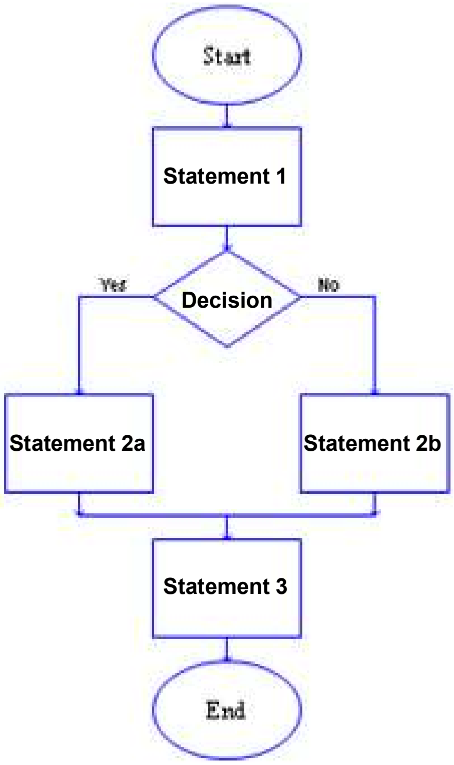
Sequential control is the "default" control in the sense that every statement automatically points to the next statement in the flowchart diagram. You do not need to do any extra work to make sequential control happen. However, using sequential control alone will not allow the development of solutions for most real-world problems. Most real world problems include "conditions" that determine what should be done next. For example, "If it is after taps, then turn your lights out," requires a decision to be made based on the time of day. The "condition" (i.e., the current time of day) determines whether the action should be executed or not executed. This is called "selection control" and is introduced next.

Selection Control



It is common that you will need to make a decision about some condition of your program's data to determine whether certain statements should be executed. For example, if you were calculating the slope of a line using the assignment statement, `slope ← dy / dx`, then you need to make sure that the value of `dx` is not zero (because division by zero is mathematically undefined and will produce a run-time error). Therefore, the decision you would need to make is, "Is `dx` zero?"

A *selection-control statement* allows you to make "decisions" in your code about the current state of your program's data and then to take one of two alternative paths to a "next" statement. The RAPTOR code on the right illustrates a selection-control statement, which is always drawn as a diamond. All decisions are stated as "yes/no" questions. When a program is executed, if the answer to a decision is "yes" (or true), then the left branch of control is taken. If the answer is "no" (or false), then the right branch of control is taken. In the example to the right, either statement 2a or statement 2b will be executed, but never both. Note that there are two possible executions of this example program:



Possibility 1	Possibility 2
Statement 1 Statement 2a Statement 3	Statement 1 Statement 2b Statement 3

Also note that either of the two paths of a selection-control statement could be empty or could contain several statements. It would be inappropriate for both paths to be empty or for both paths to have the exact same statements, because then your decision, Yes or No, would have no effect during program execution (since nothing different would happen based on the decision).

Decision Expressions

A selection-control statement requires an expression that can be evaluated into a "Yes/No" (or True/False) value. A decision expression is a combination of values (either constants or variables) and operators. Please carefully study the following rules for constructing valid decision expressions.

As you hopefully recall from our discussion of assignment statement expressions, a computer can only perform one operation at a time. When a decision expression is evaluated, the operations of the expression are not evaluated from left to right in the order that you typed them. Rather, the operations are performed based on a predefined "order of precedence." The order that operations are performed can make a radical difference in the final "Yes/No" value that is computed. You can always explicitly control the order in which operations are performed by grouping values and operators in parenthesis. Since decision expressions can contain calculations similar to those found in assignment statements, the following "order of precedence" must include assignment statement expression operators. The "order of precedence" for evaluating decision expression is:

1. compute all functions, then
2. compute anything in parentheses, then
3. compute exponentiation (^,**) i.e., raise one number to a power, then
4. compute multiplications and divisions, left to right, then
5. compute additions and subtractions, left to right, then
6. evaluate relational operators (= != /= < <= > >=), left to right,
7. evaluate any **not** logical operators, left to right,
8. evaluate any **and** logical operators, left to right,
9. evaluate any **xor** logical operators, left to right, then finally
10. evaluate any **or** logical operators, left to right.

In the above list, the *relational* and *logical* operators are new. These new operators are explained in the following table.

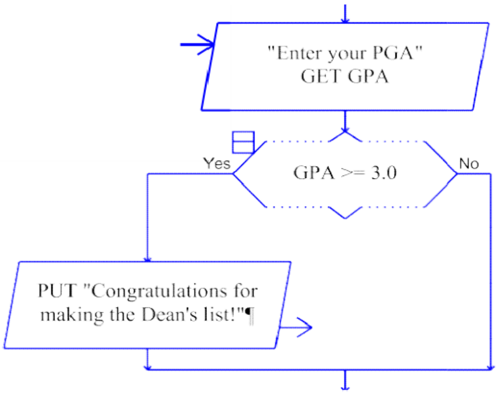
Operation	Description	Example
=	"is equal to"	3 = 4 is No(false)
!= /=	"is not equal to"	3 != 4 is Yes(true) 3 /= 4 is Yes(true)
<	"is less than"	3 < 4 is Yes(true)
<=	"is less than or equal to"	3 <= 4 is Yes(true)
>	"is greater than"	3 > 4 is No(false)
>=	"is greater than or equal to"	3 >= 4 is No(false)
and	Yes(true) if **both** are Yes	(3 < 4) and (10 < 20) is Yes(true)
or	Yes(true) if **either** are Yes	(3 < 4) or (10 > 20) is Yes(true)
xor	Yes(true) if the "yes/no" values are not equal	Yes xor No is Yes(true)
not	Invert the logic of the value Yes if No; No if Yes	not (3 < 4) is No(false)

The relational operators, (= != /= < <= > >=), must always compare two values of the same data type (either numbers, text, or "yes/no" values). For example, 3 = 4 or "Wayne" = "Sam" are valid comparisons, but 3 = "Mike" is invalid.

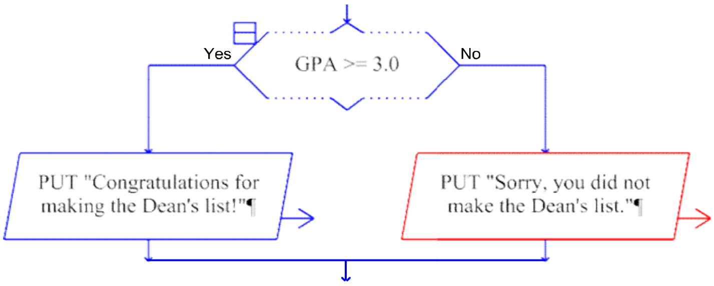
The logical operators, (and , or, xor), must always combine two Boolean values (true/false) into a single Boolean value. The logical operator, not, must always convert a single Boolean value into its opposite truth value. Several valid and invalid examples of decision expressions are shown below:

Example	Valid or Invalid?
`(3<4) and (10<20)`	Valid
`(flaps_angle < 30) and (air_speed < 120)`	Valid, assuming `flaps_angle` and `air_speed` both contain numerical data.
`5 and (10<20)`	Invalid - the left side of the "`and`" is a number, not a true/false value.
`5 <= x <= 7`	Invalid - because the `5 <= x` is evaluated into a true/false value and then the evaluation of `true/false <= 7` is an invalid relational comparison.

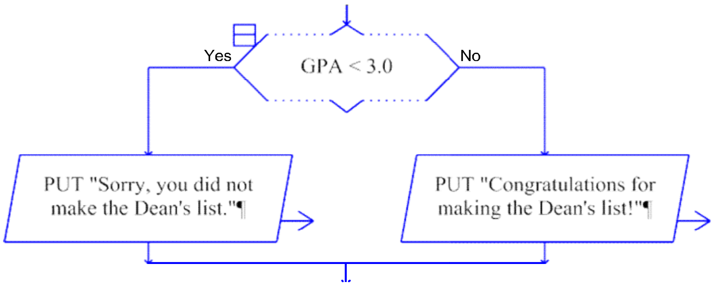
Selection Control Examples



To help clarify selection-control statements, please study the following examples. In the first example to the right, if a student has made the Dean's List, then a congratulations message will be displayed - otherwise nothing is displayed (since the "no" branch is empty).


Yes
GPA >= 3.0
No
PUT "Congratulations for making the Dean's list!"¶
PUT "Sorry, you did not make the Dean's list."¶

In the next example, one line of text is always displayed, with the value of the PGA variable determining which one.

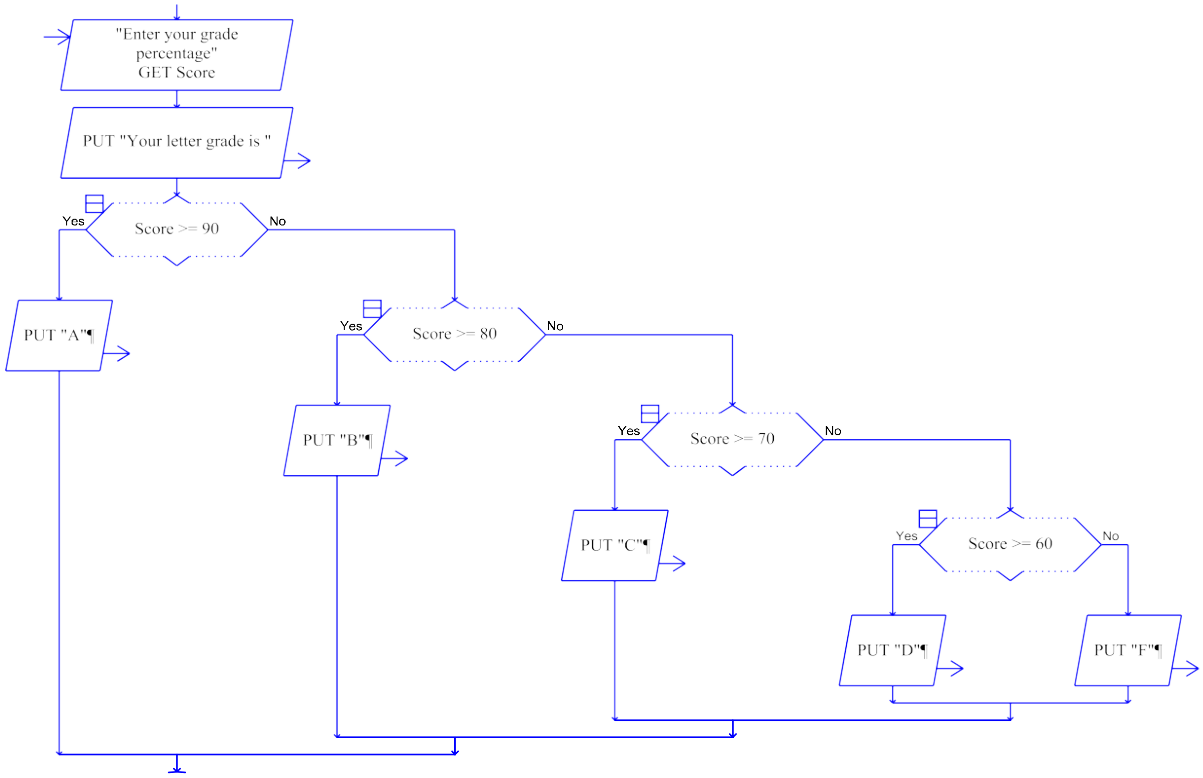

Yes
GPA < 3.0
No
PUT "Sorry, you did not make the Dean's list."¶
PUT "Congratulations for making the Dean's list!"¶

In the next example, if the student does not make the Dean's list then two lines of text are displayed, but only one line is displayed if they do.

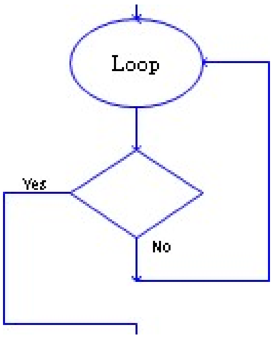
In the last example to the right, the logic of the decision expression has been inverted. This is perfectly acceptable as long as you make sure the inversion covers all possible cases. Note that the inversion of "greater than or equal to" is simply "less than."

Cascading Selection statements

A single selection-control statement can make a choice between one or two choices. If you need to make a decision that involves more than two choices, you need to have multiple selection control statements. For example, if you are assigning a letter grade (A, B, C, D, or F) based on a numeric score, you need to select between five choices, as shown below. This is sometimes referred to as "cascading selection control," as in water cascading over a series of water falls.



Loop (Iteration) Control

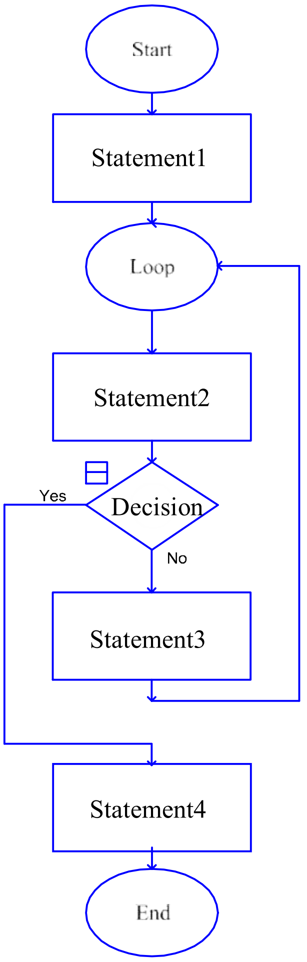


A Loop (iteration) control statement allows you to repeat one or more statements until some condition becomes true. This type of control statement is what makes computers so valuable. A computer can repeatedly execute the same instructions over-and-over again without getting bored with the repetition.

One ellipse and one diamond symbol is used to represent a loop in RAPTOR. The number of times that the loop is executed is controlled by the decision expression that is entered into the diamond symbol. During execution, when the diamond symbol is executed, if the decision expression evaluates to "no," then the "no" branch is taken, which always leads back to the Loop statement and repetition. The statements to be repeated can be placed above or below the decision diamond.

To understand exactly how a loop statement works, study the example RAPTOR program to the right and notice the follow things about this program:

- Statement 1 is executed exactly once before the loop (repetition) begins.
- Statement 2 will always be executed at least once because it comes before the decision statement.
- If the decision expression evaluates to "yes," then the loop terminates and control is passed to Statement 4.
- If the decision expression evaluates to "no," then control passes to Statement 3 and Statement 3 is executed next. Then control is returned back up to the Loop statement which re-starts the loop.
- Note that Statement 2 is guaranteed to execute at least once. Also note that Statement 3 is possibly never executed.



There are too many possible executions of this example program to list them all, but a few of the possibilities are listed in the following table. Make sure you can fill in the fourth column with the correct pattern.

Possibility 1	Possibility 2	Possibility 3	Possibility 4
Statement 1 Statement 2 Decision ("yes") Statement 4	Statement 1 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("yes") Statement 4	Statement 1 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("yes") Statement 4	(do you see the pattern?)

In the RAPTOR example above, "Statement2" could be removed, which means that the first statement in the loop would be the "Decision" statement. Or "Statement2" could be a block of multiple statements. In either case the loop executes in the same way. Similarly, "Statement3" could be deleted or be replaced by multiple statements. In addition, any of the statements above or below the "Decision" statement could be another loop statement! If a loop statement occurs inside a loop, we called these "nested loops."

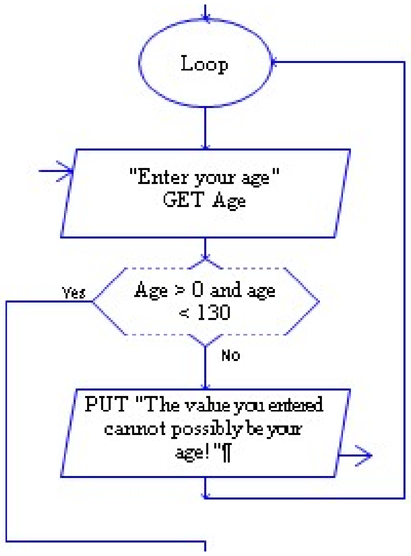
It is possible that the "Decision" statement never evaluates to "yes." In such a case you have an "infinite loop" that will never stop repeating. (If this ever happens, you will have to manually stop your program by selecting the "stop" icon in the tool bar.) You should never write statements that create an infinite loop. Therefore, one (or more) of the statements in the loop must change one or more of the variables in the "Decision" statement such that it will eventually evaluate to "yes."

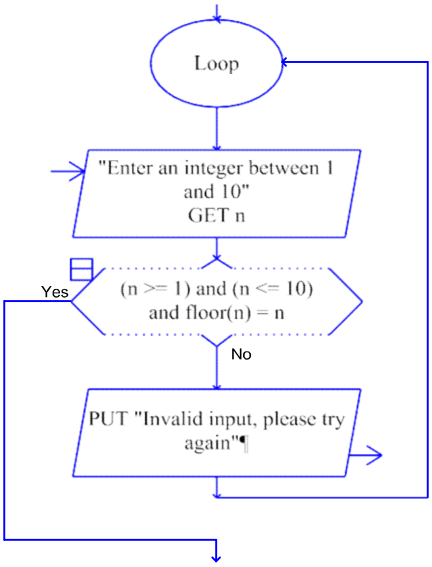
Input Validation Loops

One common use for a loop is to **validate user input**. If you want the user to input data that meets certain constraints, such as entering a person's age, or entering a number between 1 and 10, then validating the user input will ensure such constraints are met before those values are used elsewhere in your program. Programs that validate user input and perform other error checking at run-time are called **robust** programs.

A common mistake made by beginning programmers is to validate user input using a selection statement. This can fail to detect bad input data because the user might enter an invalid input on the second attempt. Therefore, you must use a loop to validate user input.

The two example RAPTOR programs below validate user input. Hopefully you see a pattern. Almost every validation loop that you write will include an input prompt, a decision, and an output error message.




Loop
"Enter an integer between 1 and 10"
GET n
Yes
(n >= 1) and (n <= 10) and floor(n) = n
No
PUT "Invalid input, please try again"¶

Counting Loops

Another common use of a loop is to execute a block of code a specific number of times. This type of loop is called a **counter-controlled loop** because it requires a variable that "counts by one" on each execution of the loop. Therefore, besides the loop statement, a counter-controlled loop requires a "counter" variable that is:

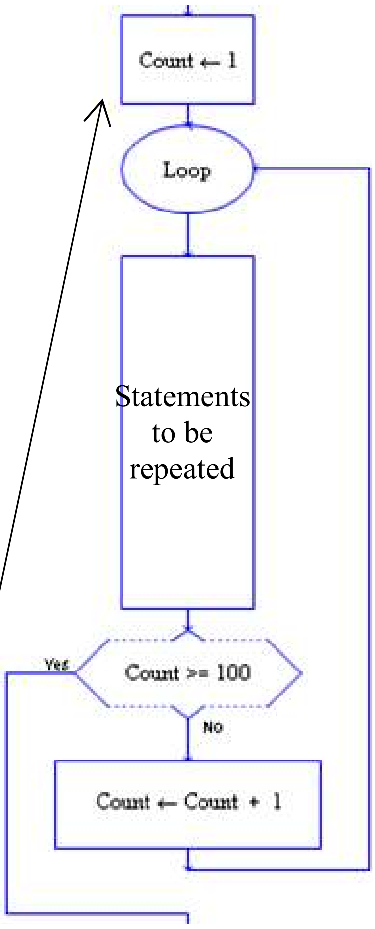
1. initialized before the loop starts,
2. modified inside the loop, and
3. used in the decision expression to stop the loop.

The acronym I.T.E.M (Initialize, Test, Execute, and Modify) can be used to check whether a loop and its counter variable are being used correctly.

An example of a count-controlled loop that executes exactly 100 times is shown to the right. As you study this example, please notice the following important points:

- In this example, the "counter" variable is called "Count." You can use any variable name, but try to make it descriptive and meaningful for your current task.

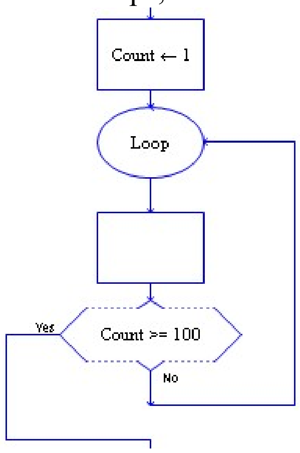
- The "counter" variable must be initialized to its starting value before the loop begins. It is common to start its value at one (1), but you could have a loop that executes 100 times by starting at 20 and counting to 119. Try to use a starting value that is appropriate for the problem you are solving.



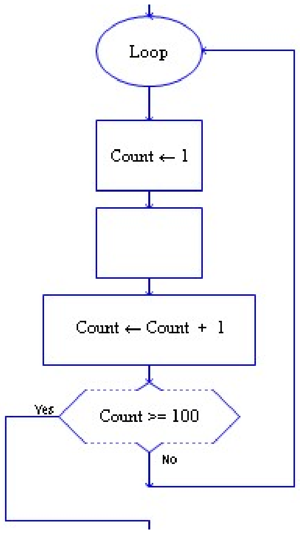
- The decision expression that controls the loop should typically test for "greater than or equal to." This is a safer test than just "equal to."

- A counter-controlled loop typically increments the counter variable by one on each execution of the loop. You can increment by a value other than one, but this will obviously change how many times the loop repeats.

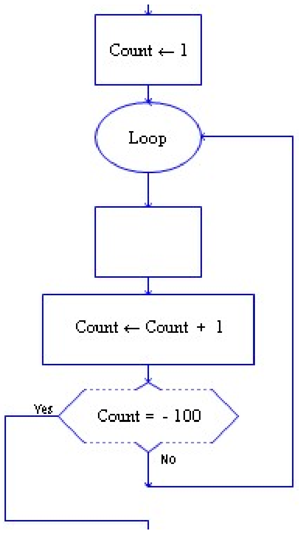
The following three RAPTOR programs demonstrate common errors that should be avoided when implementing loops. See if you can determine the error in each program. (If you can't find the errors, they are explained in footnotes at the bottom of the page.) All three of these problematic programs create an **infinite loop** – that is, a loop that never stops. To avoid writing infinite loops, avoid these common errors.



1



2

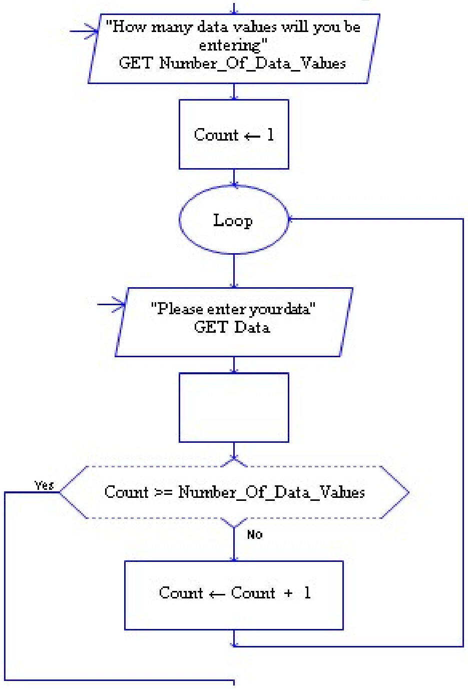


3

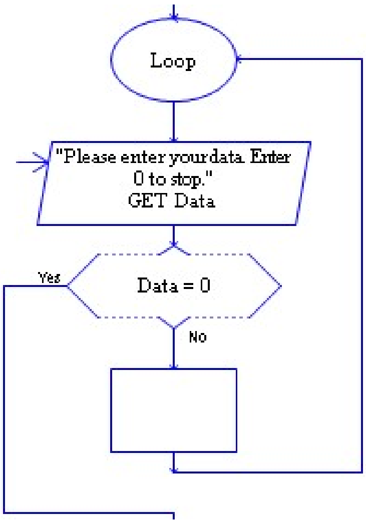
The following example programs show the six (6) possible variations of a counter-controlled loop. They all do the same thing -- they execute the statement(s) represented by the empty box `Limit` number of times. You can use the variation that makes the most sense to you. In each example, pay close attention to the starting (initial) value of `Count` and the Decision expression.

Input Loops

Sometimes you need a user to enter a series of values that you can process. There are two general techniques to accomplish this. The first method is to have the user enter a "special" value that signifies that the user is finished entering data. A second method is to ask the user, in advance, how many values they will be entering. Then that value can be used to implement a counter-controlled loop. These two methods are depicted in the following example programs. In both cases the empty boxes signify where the entered data would be

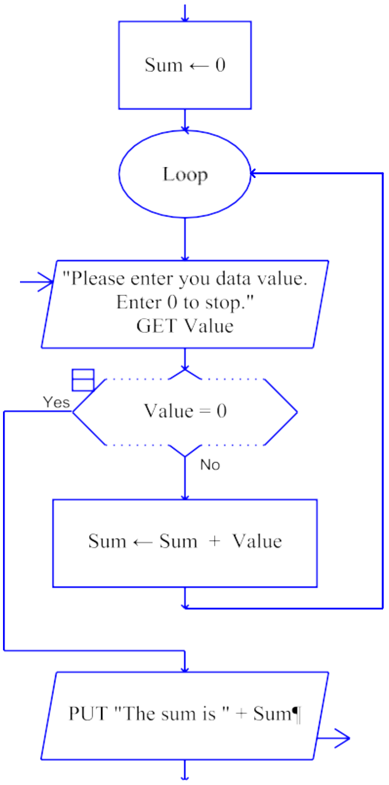

"How many data values will you be entering"
GET Number_Of_Data_Values
Count ← 1
Loop
"Please enter your data"
GET Data
Yes
Count >= Number_Of_Data_Values
No
Count ← Count + 1

processed. Don't worry about how the data is processed, just look at these examples to see how the user controls how much data is entered.



"Running Total" Loops

Another common use of loops is to calculate the sum of a series of data values, which is sometimes called a "**running total**" or a "**running sum**." The example program to the right produces a "running total" of a series of values entered by a user.



To create a "running sum" you must add two additional statements to a loop:

- An initialization statement, before the loop starts, that sets a "running sum" variable to zero (0).
  For example,
  `Sum ← 0`
- An assignment statement, inside the loop, that adds each individual value to the "running sum" variable.
  For example,
  `Sum ← Sum + Value`

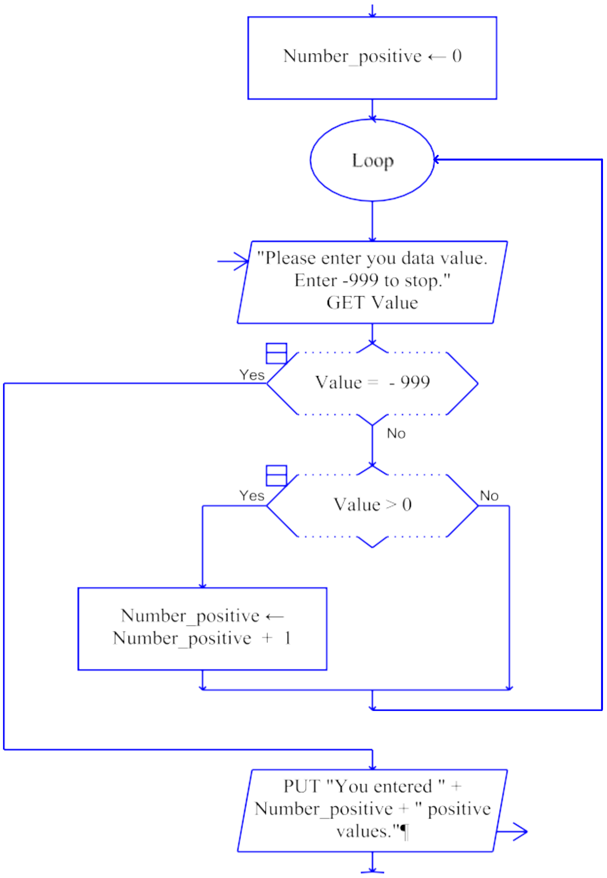
Make sure you understand the assignment statement, `Sum ← Sum + Value`. In English it says, calculate the expression on the right side of the arrow by taking the current value of `Sum` and adding `Value` to it. Then place the result into the variable Sum.

The variable name, `Sum`, is not magic. Any variable name could be used, such as `Total` or `Running_Sum`.

"Counting" Loops

Another common use of loops is for counting the number of times an event occurs. An example of this type of program logic is shown to the right. Note how similar this program is to the previous example.

The last two examples demonstrate how the same pattern of programming statements occurs over and over again and can be used to solve a variety of similar problems. By studying and understanding the simple examples in this reading you will be able to use these examples as the basis for solving additional, more complex problems.


Number_positive ← 0
Loop
"Please enter you data value. Enter -999 to stop." GET Value
Yes
Value = - 999
No
Yes
Value > 0
No
Number_positive ← Number_positive + 1
PUT "You entered " + Number_positive + " positive values."¶

Experiment No.2

AIM- Develop requirements specification for a given problem

Objective: To find the requirement specification(both functional and nonfunctional) of a given Problem.

Procedure:

Step 1:

Introduction:

Purpose

Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.

Intended Audience and Reading Suggestions

Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.

Project Scope

Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here. An SRS that specifies the next release of an evolving product should contain its own scope statement as a subset of the long-term strategic product vision.

Step 2:

Overall Description

Product Perspective

Describe the context and origin of the product being specified in this SRS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SRS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.

Product Features

Summarize the major features the product contains or the significant functions that it performs or lets the user perform. Only a high level summary is needed here. Organize the functions to make them understandable to any reader of the SRS. A picture of the major groups of related requirements and how they relate, such as a top level data flow diagram or a class diagram, is often effective.

User Classes and Characteristics

Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the favored user classes from those who are less important to satisfy.

Operating Environment
Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with
which it must peacefully coexist.

Design and Implementation Constraints
Describe any items or issues that will limit the options available to the developers. These might include: corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer‟s organization will be responsible for maintaining the delivered software).

Step 3:

System Features

This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional hierarchy, or combinations of these, whatever makes the most logical sense for your product.

System Feature 1

Don‟t really say "System Feature 1." State the feature name in just a few words.

1 Description and Priority Provide a short description of the feature and indicate whether it is of High,
Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).

2 Stimulus/Response Sequences List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.

3 Functional Requirements Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary.

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

REQ-1:
REQ-2:

System Feature 2 (and so on)

Step 4:

External Interface Requirements

User Interfaces
Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define

the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.

Hardware Interfaces

Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.

Software Interfaces

Describe the connections between this product and other specific software components (name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.

Communications Interfaces

Describe the requirements associated with any communications functions required by this product, including e-mail, web browser, network server communications protocols, electronic forms, and so on. Define any pertinent message formatting. Identify any communication standards that will be used, such as FTP or HTTP. Specify any communication security or encryption issues, data transfer rates, and synchronization mechanisms.

Nonfunctional Requirements

Performance Requirements

If there are performance requirements for the product under various circumstances, state them here and explain their rationale, to help the developers understand the intent and make suitable design choices. Specify the timing relationships for real time systems. Make such requirements as specific as possible. You may need to state performance requirements for individual functional requirements or features.

Safety Requirements

Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as actions that must be prevented. Refer to any external policies or regulations that state safety issues that affect the product’s design or use. Define any safety certifications that must be satisfied.

Security Requirements

Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.

Software Quality Attributes

Specify any additional quality characteristics for the product that will be important to either the customers or the developers. Some to consider are: adaptability, availability, correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative, and verifiable when possible. At the least, clarify the relative preferences for various attributes, such as ease of use
over ease of learning.

Other Requirements

Define any other requirements not covered elsewhere in the SRS. This might include database requirements, internationalization requirements, legal requirements, reuse objectives for the project, and so on. Add any new sections that are pertinent to the project.

Experiment No. 3

AIM : Develop DFD model (level-0, level-1 DFD and Data dictionary) of the project.

OBJECTIVE: To familiar with DFD models.

DESCRIPTION :
Data analysis attempts to answer four specific questions:

- What processes make up a system?
- What data are used in each process?
- What data are stored?
- What data enter and leave the system?

Data drive business activities and can trigger events (e.g. new sales order data) or be processed to provide information about the activity. Data flow analysis, as the name suggests, follows the flow of data through business processes and determines how organisation objectives are accomplished. In the course of handling transactions and completing tasks, data are input, processed, stored, retrieved, used, changed and output. Data flow analysis studies the use of data in each activity and documents the findings in data flow diagrams, graphically showing the relation between processes and data.

Physical and Logical DFDs

There are two types of data flow diagrams, namely *physical data flow diagrams* and *logical data flow diagrams* and it is important to distinguish clearly between the two:

Physical Data Flow Diagrams

An implementation-dependent view of the current system, showing what tasks are carried out and how they are performed. Physical characteristics can include:

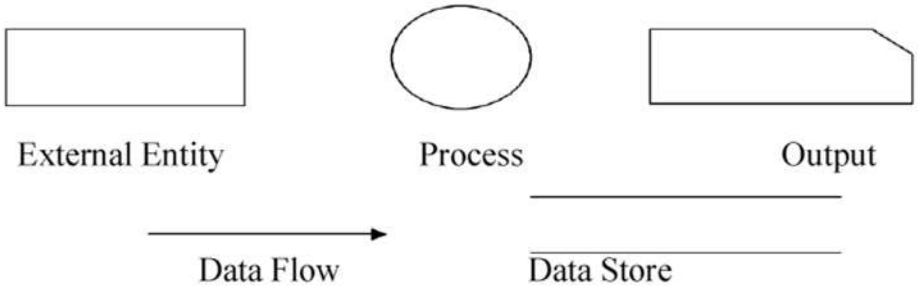
- Names of people
- Form and document names or numbers
- Names of departments
- Master and transaction files
- Equipment and devices used

Logical Data Flow Diagrams

An implementation-*independent* view of the a system, focusing on the flow of data between processes without regard for the specific devices, storage locations or people in the system. The physical characteristics listed above for physical data flow diagrams will not be specified.

Data Flow Diagram (DFD)

The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions. Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. A DFD model uses a very limited number of primitive symbols to represent the functions performed by a system and the data flow among these functions.

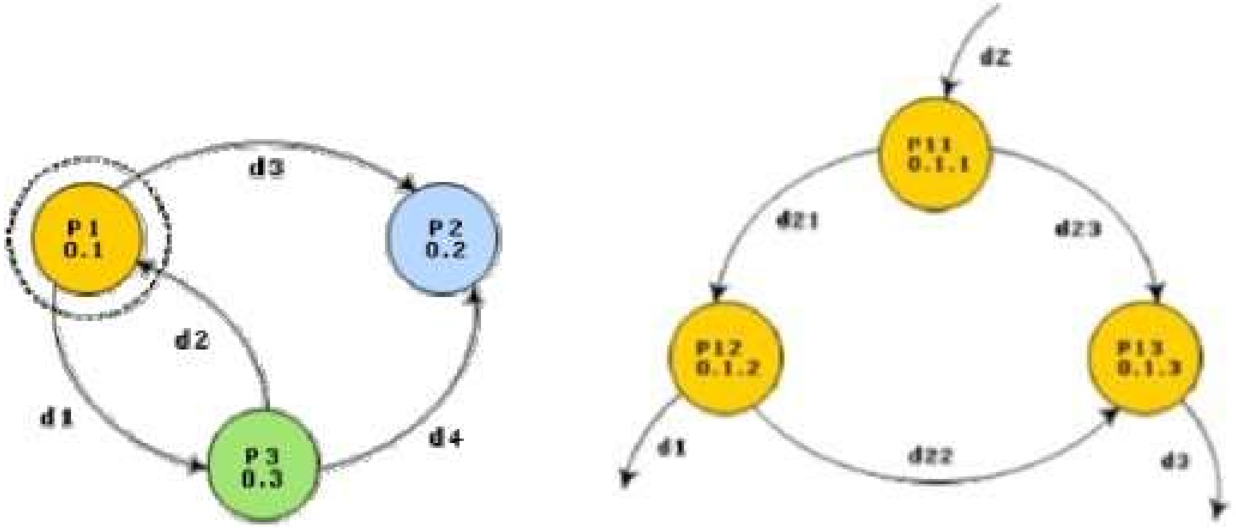


Importance of DFDs in a good software design

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub functions. In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization.

Balancing a DFD

The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD. The concept of balancing a DFD has been illustrated in fig. bellow. In the level 1 of the DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1. In the next level, bubble 0.1 is decomposed. The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.



Decomposition

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything between 3 to 7 bubbles. Too few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many bubbles, i.e. more than 7 bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

Numbering of Bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD by its bubble number.The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Data dictionary

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data **grossPay** consists of the components regularPay and overtimePay.

grossPay = regularPay + overtimePay

For the smallest units of data items, the data dictionary lists their name and their type.
Composite data items can be defined in terms of primitive data items using the following
data definition operators:

+: denotes composition of two data items, e.g. **a+b** represents data a and **b.**

[„]: represents selection, i.e. any one of the data items listed in the brackets can occur. For example, **[a,b]** represents either **a** occurs or **b** occurs.

(): the contents inside the bracket represent optional data which may or may not appear. e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.

{}: represents iterative data definition, e.g. **{name}5** represents five **name** data.

{name}* represents zero or more instances of **name** data.
=: represents equivalence, e.g. **a=b+c** means that **a** represents **b** and **c.**
/* */: Anything appearing within /* and */ is considered as a comment.

Experiment No. 4

Aim- Structured design for the developed DFD model.

Objective- To familiar with structure design.

Description-

Well-structured designs improves the maintainability of the system. A structured system is one that is developed from the top down and modular, that is, broken down into manageable components. The modules should be designed so that they have minimal effect on other modules in the system. The connections between modules are limited and the interaction of data is minimal. Such design objectives are intended to improe system quality while easing maintenance task. So, Structure design transform the results of structured analysis (i.e., a DFD representation) into a structure chart.

Purpose of Structure Chart

A Structure Chart is a design tool that visually displays the relationships between program modeles. It shows which modules with a system interact and also graphically depicts the data that are communicated between various modules.

Structure charts are developed prior to the writing of program code. They are not intended to express procedural logic a task left to flowcharts and pseudo code. Nor do they describe the actual physical interface between processing functions. Instead, they identify the data passes existing between individual modules that interact with one another.

A structure chart represents the software architecture:

- various modules making up the system
- module dependency (i.e. which module calls which other modules)
- parameters passed among different modules

Basic building blocks of structure chart

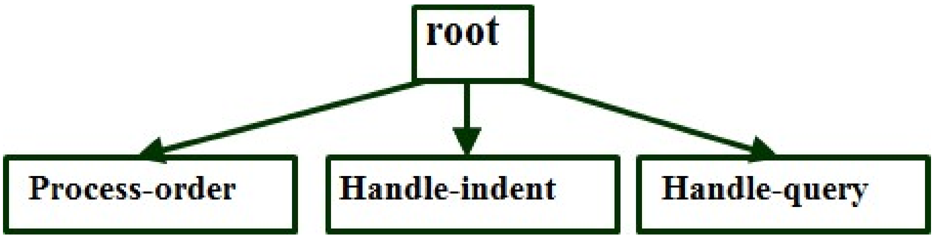
1. **Rectangular box:**

    - A rectangular box represents a module.
    - annotated with the name of the module it represents.

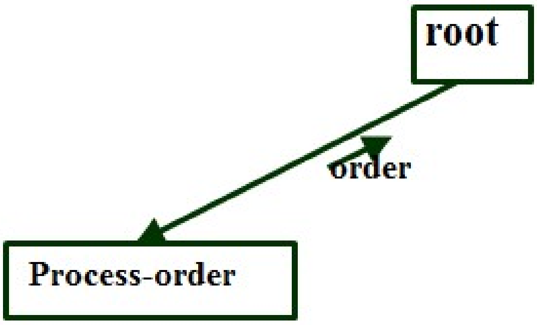
Process-order

2. **Arrows**

    An arrow between two modules implies: during execution control is passed from one module to the other in the direction of the arrow.

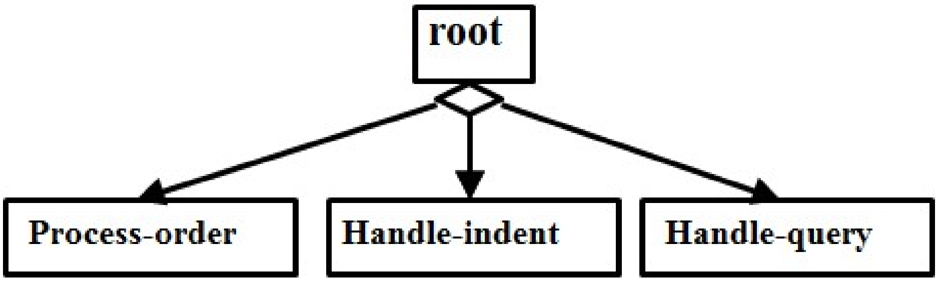


Data flow arrows represent: data passing from one module to another in the direction of the arrow.



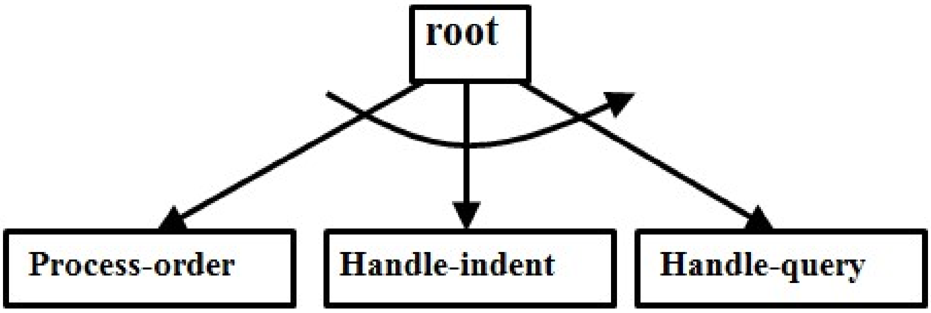
3. **Diamonds**

The diamond symbol represents: one module of several modules connected to the diamond symbol is invoked depending on some condition.



4. **Repetition**

A loop around control flow arrows denotes: that the concerned modules are invoked repeatedly.



Guidelines for design

- There is only one module at the top:
    - the root module.
- There is at most one control relationship between any two modules:
    - if module A invokes module B, module B cannot invoke module A.
- The main reason behind this restriction:
    - consider modules in a structure chart to be arranged in layers or levels.

Experiment No.5

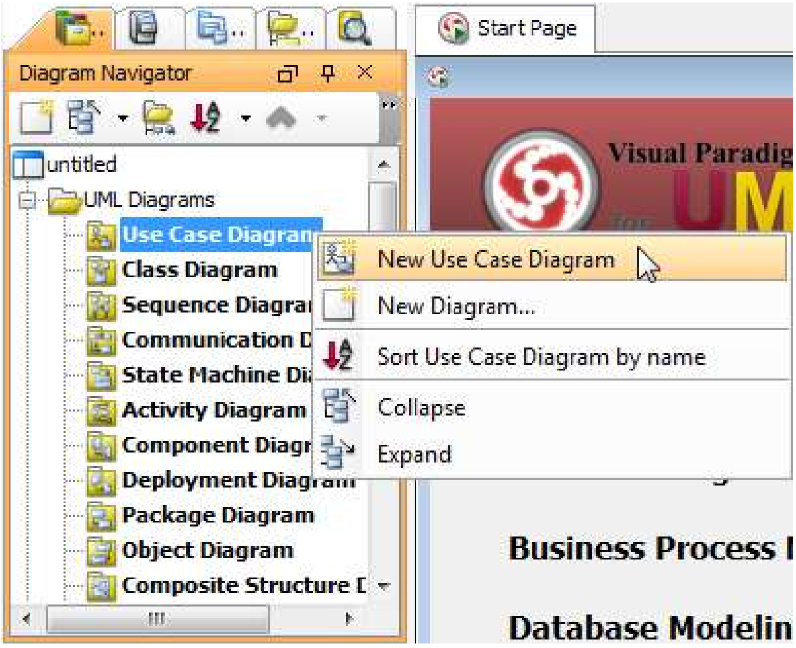
Aim: Develop UML Use case model for a problem.

Objective: To understand the users view of a project using Use case Diagram.

Software Required: Visual Paradigm for UML 8.2 or higher.

Procedure: You can use following steps to draw use case diagrams in VP-UML

Step 1:
Right click **Use Case Diagram** on **Diagram Navigator** and select **New Use Case Diagram** from the pop-up menu.



Step 2:
Enter name for the newly created use case diagram in the text field of pop-up box on the top left corner.

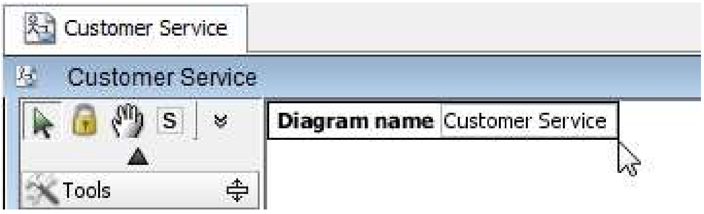
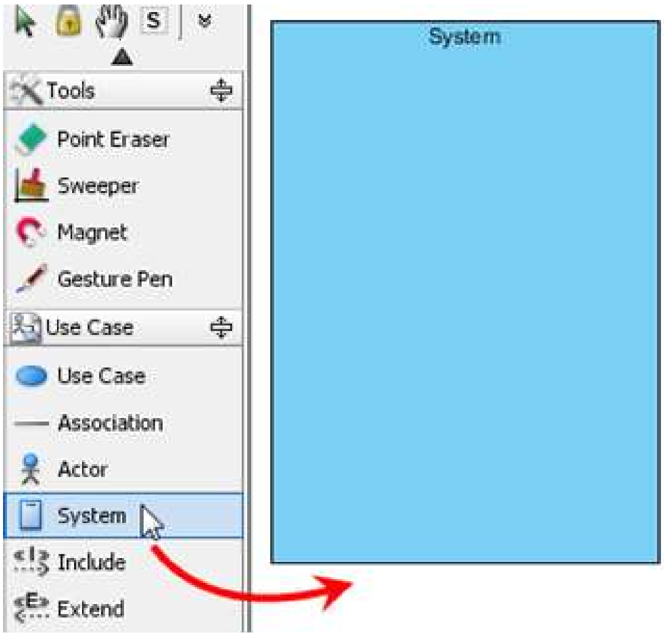


Diagram name: Customer Service

Step 3:

Drawing a system

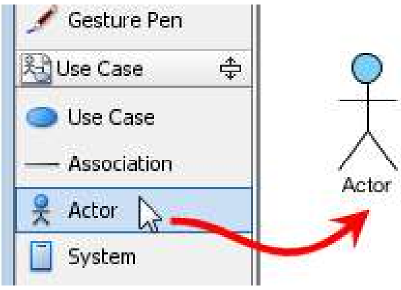
To create a system, select **System** on the diagram toolbar and then click it on the diagram pane. Finally, name the newly created system when it is created.



Step 4:

Drawing an actor

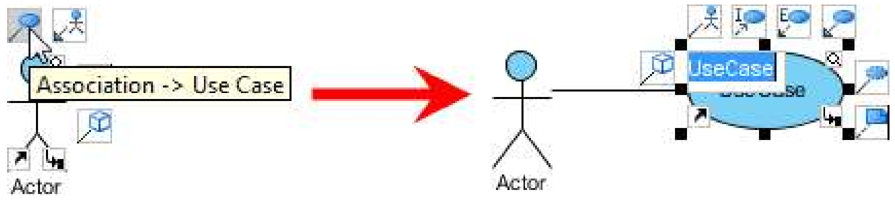
To draw an actor, select **Actor** on the diagram toolbar and then click it on the diagram pane. Finally, name the newly created actor when it is created.



Step 5:

Drawing a use case

Besides creating a use case through diagram toolbar, you can also create it through resource icon. Move the mouse over a shape and press a resource icon that can create use case. Drag it and then release the mouse button until it reaches to your preferred place. The source shape and the newly created use case are connected. Finally, name the newly created use case.

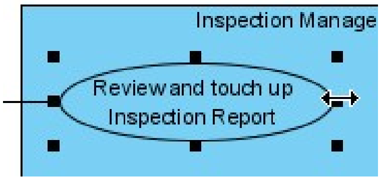


Step 6:-

Create a use case through resource icon

Line wrapping use case name

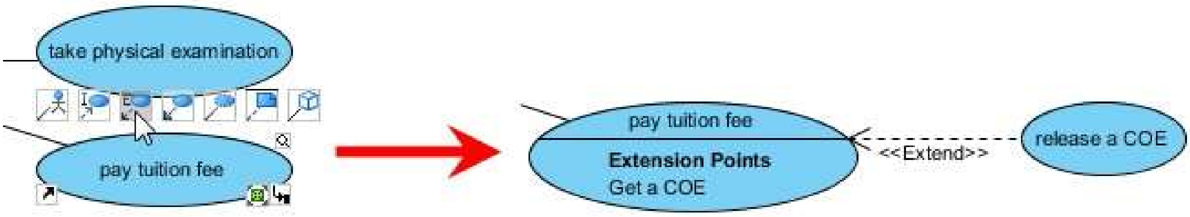
If a use case is too wide, for a better outlook, you may resize it by dragging the filled selectors. As a result, the name of use case will be line-wrapped automatically.



Step 7:

Resize a use case

To create an extend relationship, move the mouse over a use case and press its resource icon **Extend -> Use Case**. Drag it to your preferred place and then release the mouse button. The use case with extension points and a newly created use case are connected. After you name the newly created use case, a pop-up dialog box will ask whether you want the extension point to follow the name of use case. Click **Yes** if you want it to do so; click **NO** if you want to enter another name for extension point.

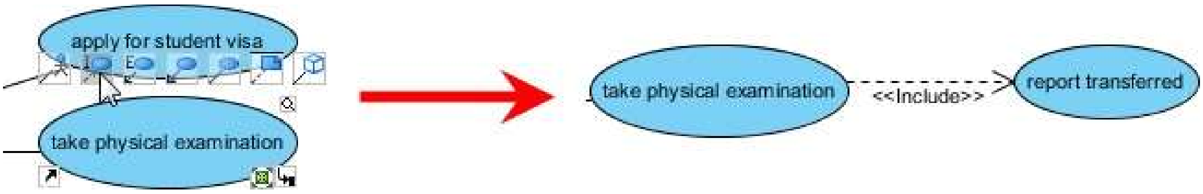


Step 8:

Create an extend relationship

Drawing <<Include>> relationship

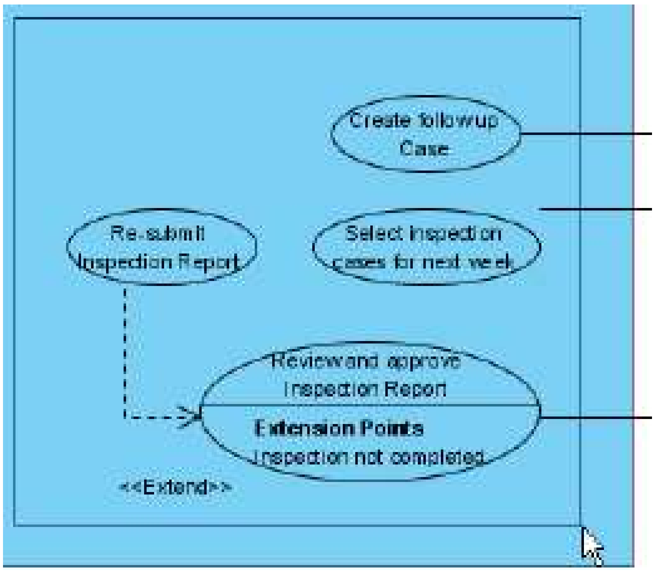
To create an include relationship, mouse over a use case and press its resource icon **Include -> Use Case**. Drag it to your preferred place and then release the mouse button. A new use case together with an include relationship is created. Finally, name the newly created use case.



Step 9:

Create a package

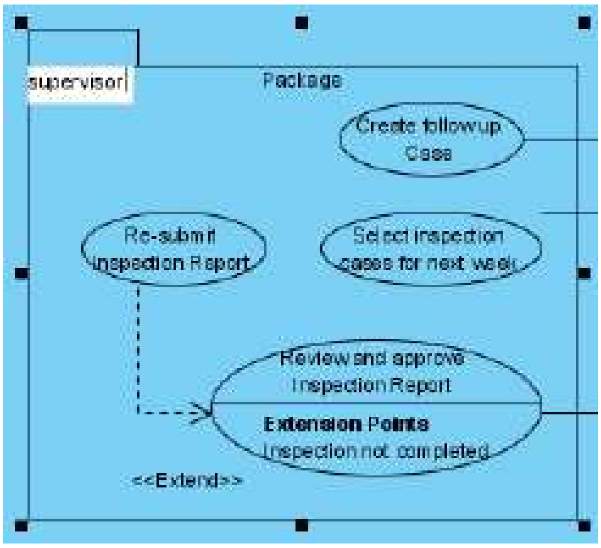
Drag the mouse to create a package surrounding those use cases.


Create followup Case
Re-submit Inspection Report
Select inspection cases for next week
Review and approve Inspection Report
Extension Points
Inspection not completed
<<Extend>>

Step 10:

Surround use cases with package

Finally, name the package.


supervisor
Package
Create followup Case
Re-submit Inspection Report
Select inspection cases for next week
Review and approve Inspection Report
Extension Points
Inspection not completed
<<Extend>>

Step 11:

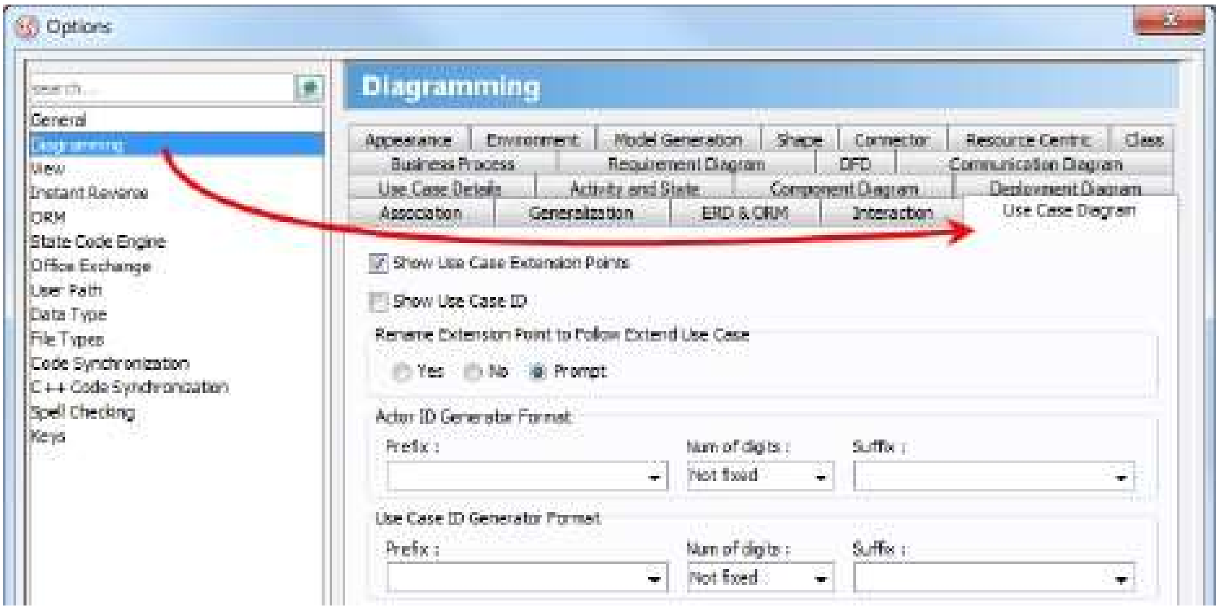
Name the package

Assigning IDs to actors/Use cases

You may assign IDs to actors and use cases. By default, IDs are assigned with the order of object creation, starting from one onwards. However, you can define the format or even enter an ID manually.

Defining the format of ID

To define the format of ID, select Tools > Options from the main menu to unfold the Options dialog box. Select Diagramming from the list on the left hand side and select the Use Case Diagram tab on the right hand side. You can adjust the format of IDs under Use Case Diagram tab. The format of ID consists of prefix, number of digits and suffix.



☑ Show Use Case Extension Points

□ Show Use Case ID

Rename Extension Point to Follow Extend Use Case

○ Yes ○ No ● Prompt

Actor ID Generator Format

Prefix : | Num of digits : Not fixed | Suffix :

Use Case ID Generator Format

Prefix : | Num of digits : Not fixed | Suffix :

Experiment No.6

AIM-Develop sequence diagram

Objective : To understand the interactions between objects that are represented as lifelines in a sequential order of a project using Sequence Diagram.

Software Required :-

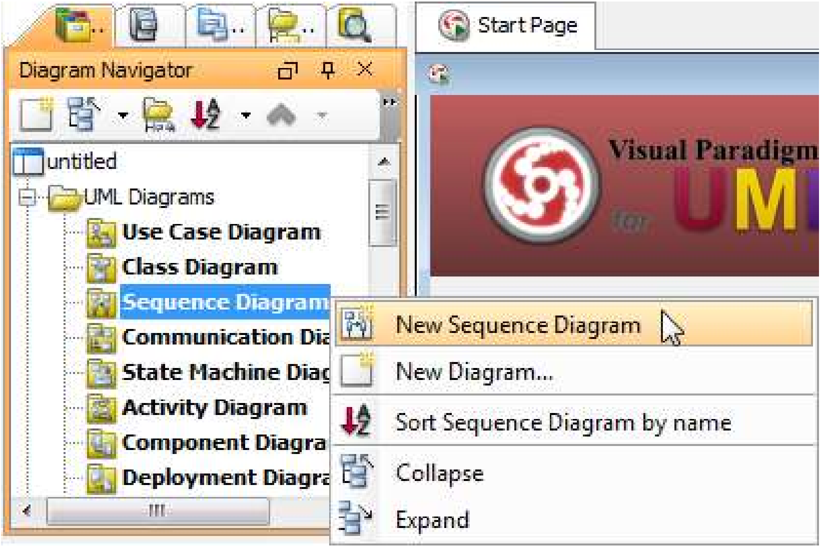
Visual Paradigm for UML 8.2 or higher.

Procedure :-

A sequence diagram is used primarily to show the interactions between objects that are represented as lifelines in a sequential order.

Step 1:-

Right click **Sequence diagram** on **Diagram Navigator** and select **New Sequence Diagram** from the pop-up menu to create a sequence diagram.

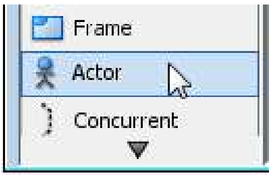


Step 2:-

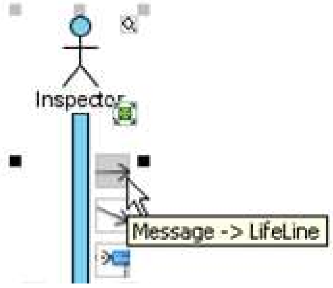
Enter name for the newly created sequence diagram in the text field of pop-up box on the top left corner.

Creating actor

To create actor, click **Actor** on the diagram toolbar and then click on the diagram.



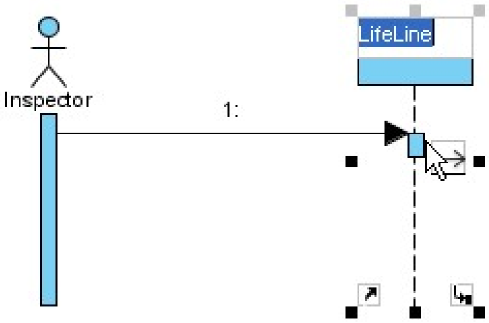
Creating lifeline
To create lifeline, you can click **LifeLine** on the diagram toolbar and then click on the diagram.



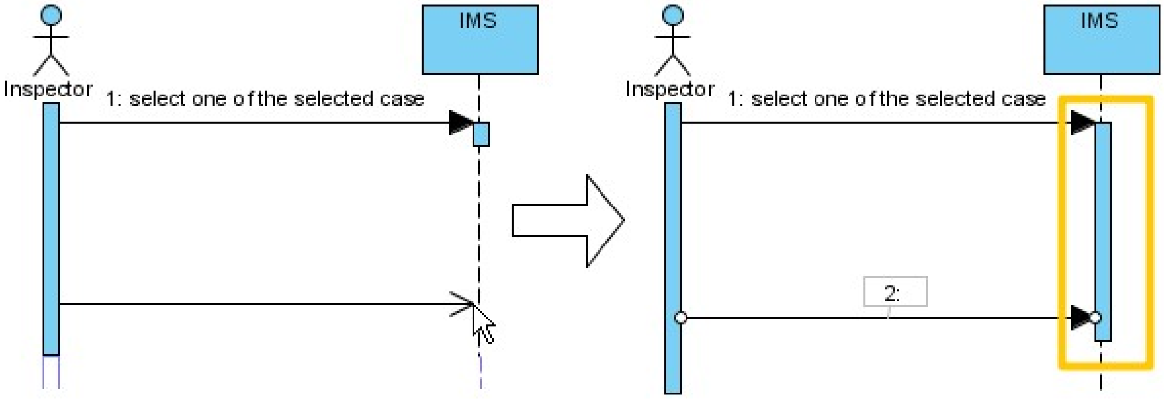
Alternatively, a much quicker and more efficient way is to use the resource-centric interface. Click on the **Message** -> **LifeLine** resource beside an actor/lifeline and drag.

Step 3:-

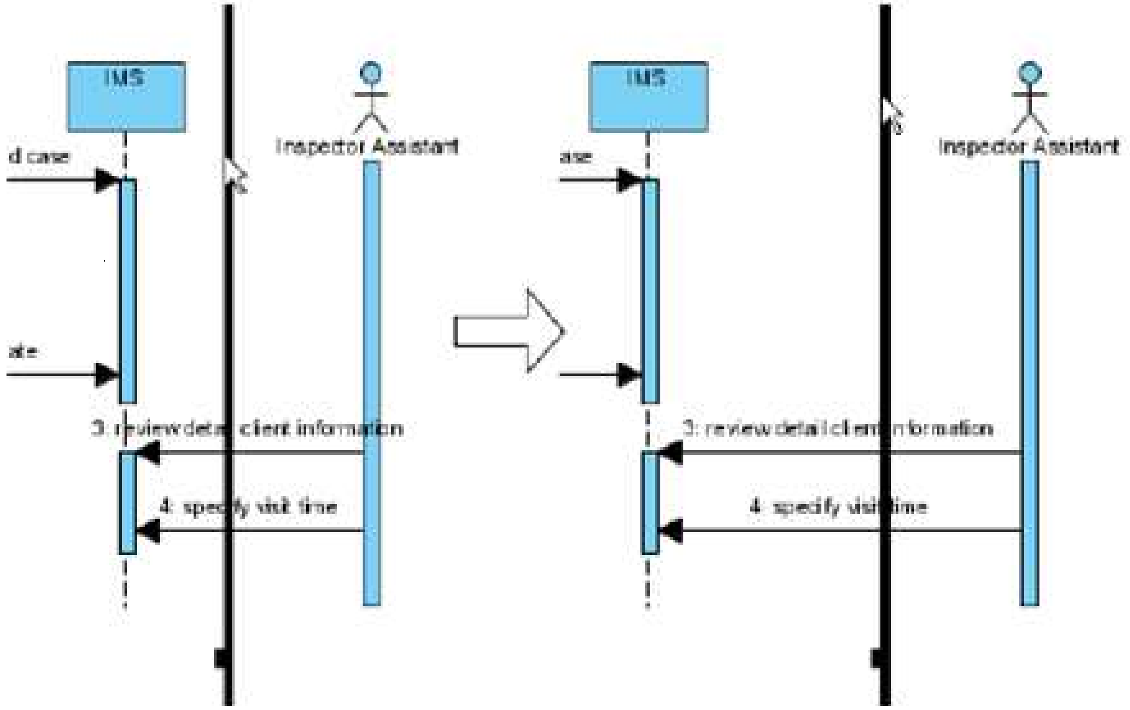
Move the mouse to empty space of the diagram and then release the mouse button. A new lifeline will be created and connected to the actor/lifeline with a message.



Auto extending activation
When create message between lifelines/actors, activation will be automatically extended.

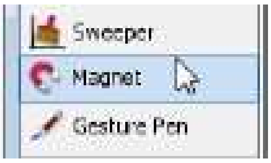

Inspector
IMS
1: select one of the selected case
Inspector
IMS
1: select one of the selected case
2:

Step 4:-
Using sweeper and magnet to manage sequence diagram
Sweeper helps you to move shapes aside to make room for new shapes or connectors. To use sweeper, click **Sweeper** on the diagram toolbar (under the **Tools** category).



Step 5:-

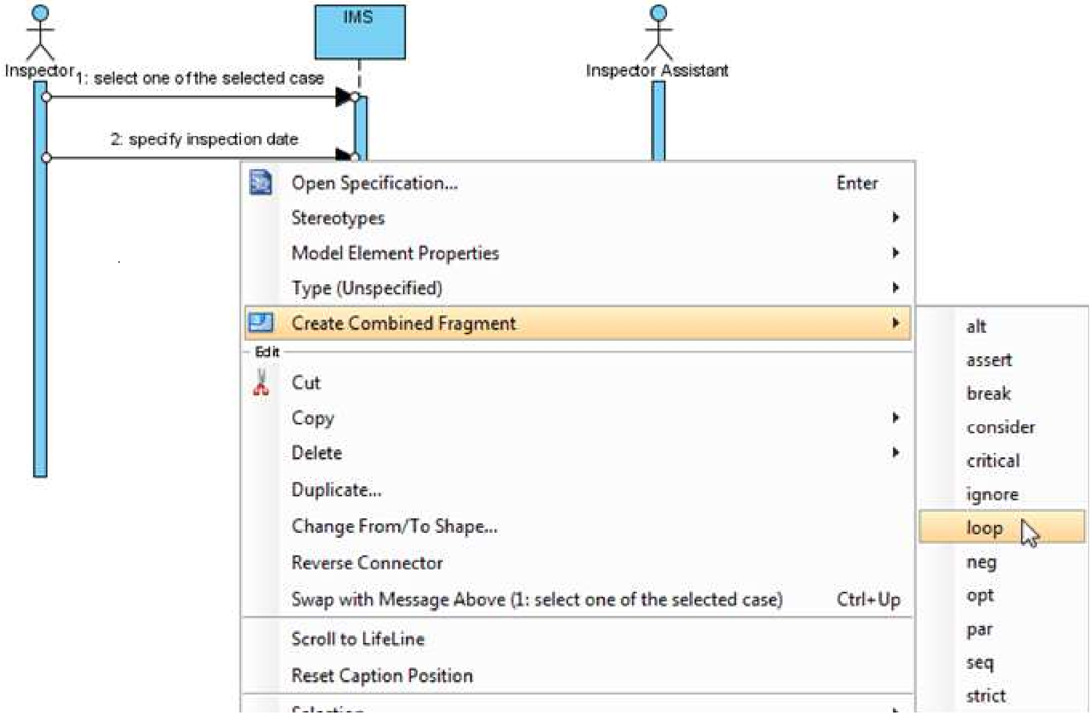
You can also use magnet to pull shapes together. To use magnet, click **Magnet** on the diagram toolbar (under the **Tools** category).



Step 6:-

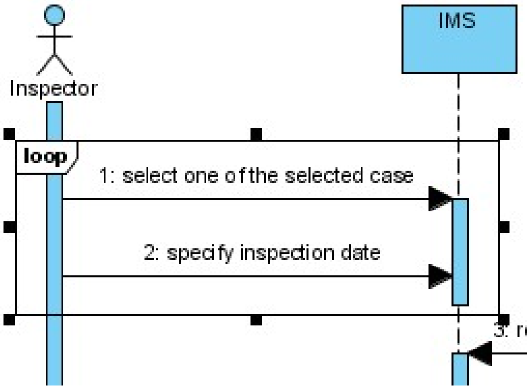
Creating combined fragment for messages

To create combined fragment to cover messages, select the messages, right-click on the selection and select **Create Combined Fragment**, and then select a combined fragment type (e.g. loop) from the popup menu.



Step 7:-

A combined fragment of selected type will be created to cover the messages.

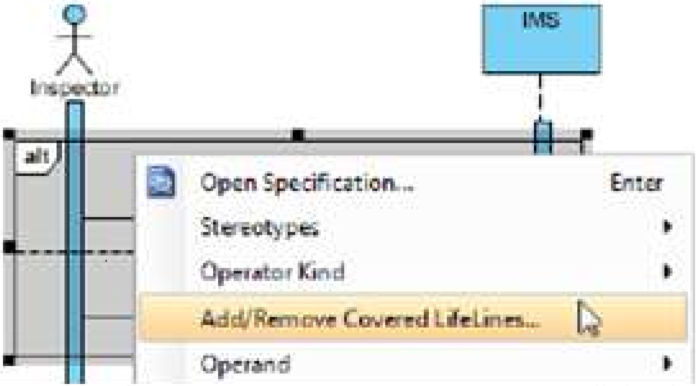

Inspector
IMS
loop
1: select one of the selected case
2: specify inspection date
3: r

Step 8:-

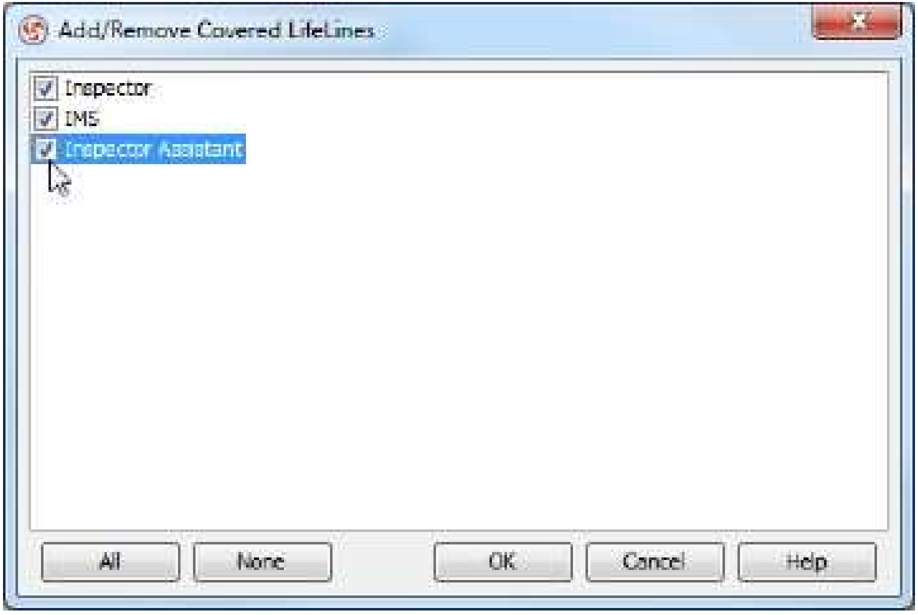
Adding/removing covered lifelines

After you've created a combined fragment on the messages, you can add or remove the covered lifelines.

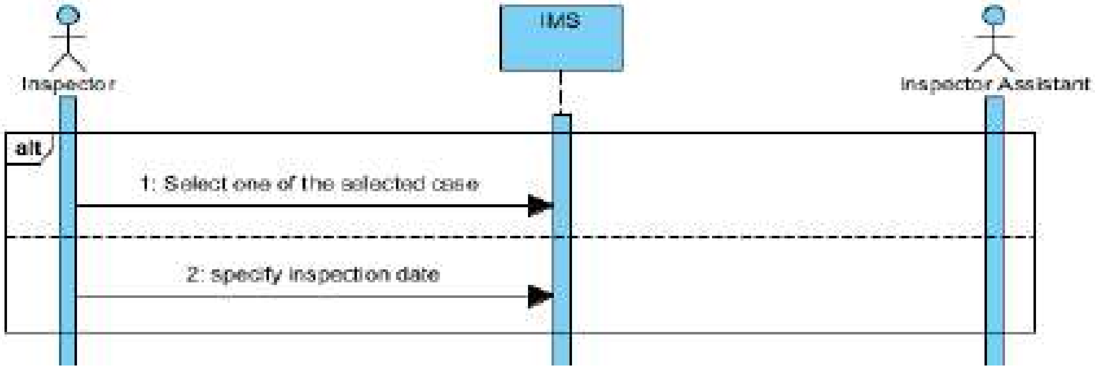
1. Move the mouse over the combined fragment and select **Add/Remove Covered Lifeline...** from the pop-up menu.



2. In the **Add/Remove Covered Lifelines** dialog box, check the lifeline(s) you want to cover or uncheck the lifeline(s) you don't want to cover. Click **OK** button.



3. As a result, the area of covered lifelines is extended or narrowed down according to your selection.


Inspector
IMS
Inspector Assistant
alt
1: Select one of the selected case
2: specify inspection date

Experiment No. 7

AIM- Develop Class diagram

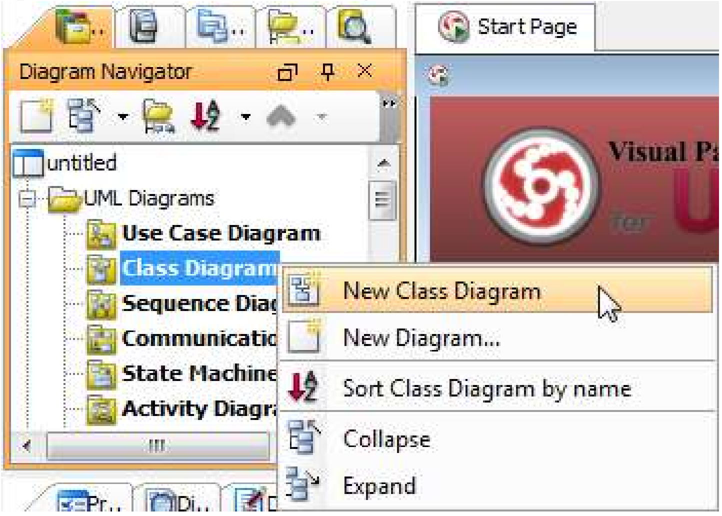
Objective:- To show diagrammatically the objects required and the relationships between them while developing a software product.

Software Required :- Visual Paradigm for UML 8.2 or higher

Procedure :-

Step 1:-

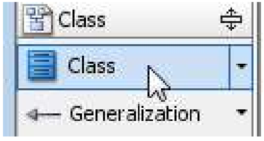
Right click Class Diagram on Diagram Navigator and select New Class Diagram from the pop-up menu to create a class diagram.



Step 2:-

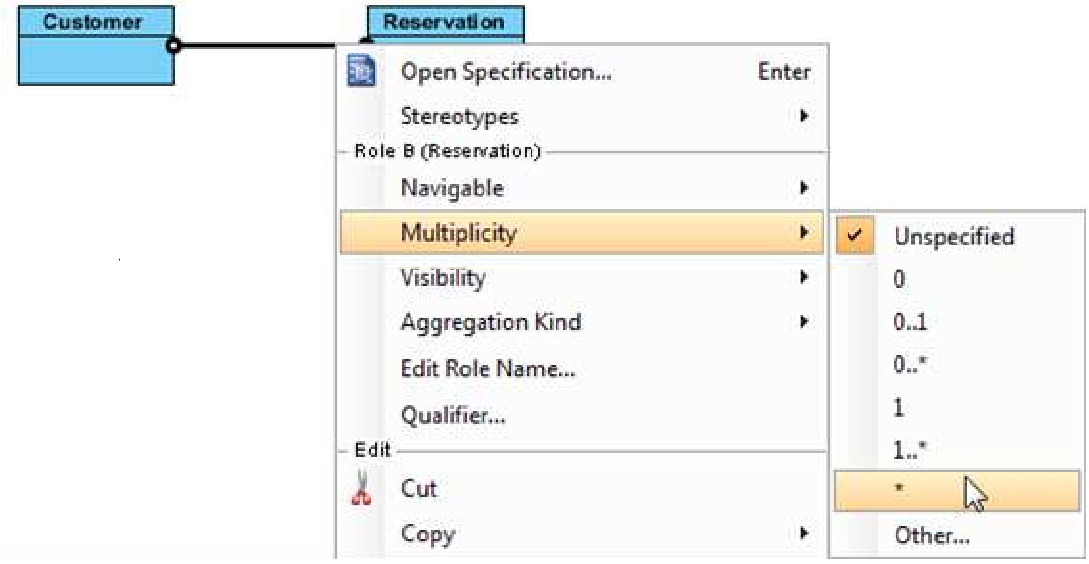
Creating class

To create class, click Class on the diagram toolbar and then click on the diagram.

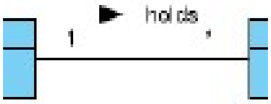


Step 3:-

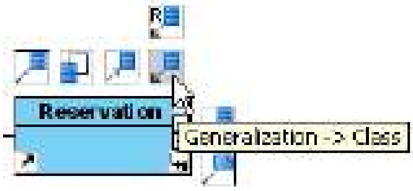
To edit multiplicity of an association end, right-click near the association end, select Multiplicityfrom the popup menu and then select a multiplicity.



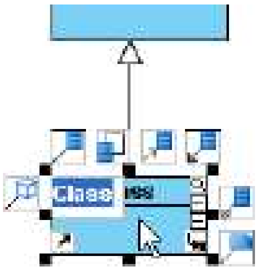
Step 4:-
The direction arrow is shown beside the association.



Creating generalization
To create generalization from class, click the **Generalization** -> **Class** resource beside it and drag.

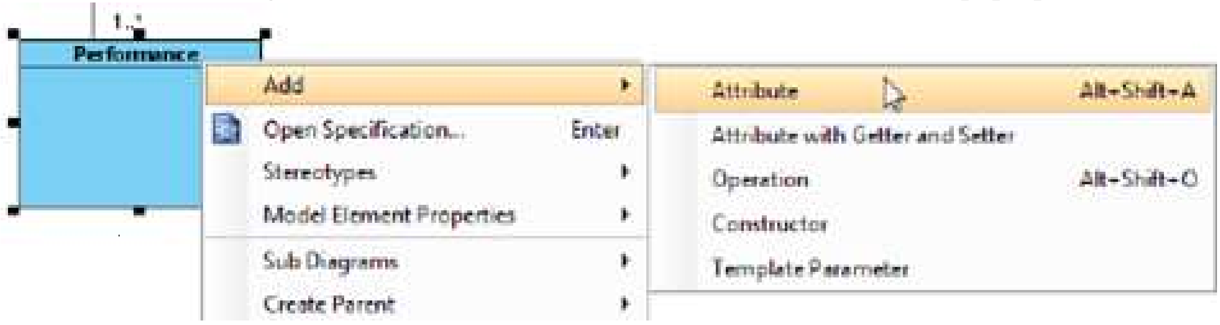


Drag to empty space of the diagram to create a new class, or drag to an existing class to connect to it.
Release the mouse button to create the generalization.

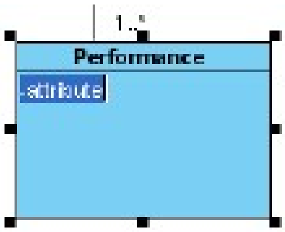


Creating attribute

To create attribute, right click the class and select **Add** > **Attribute** from the pop-up menu.

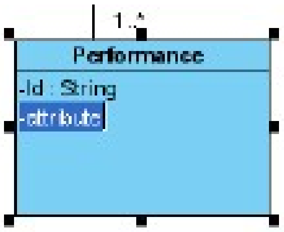


An attribute is created.



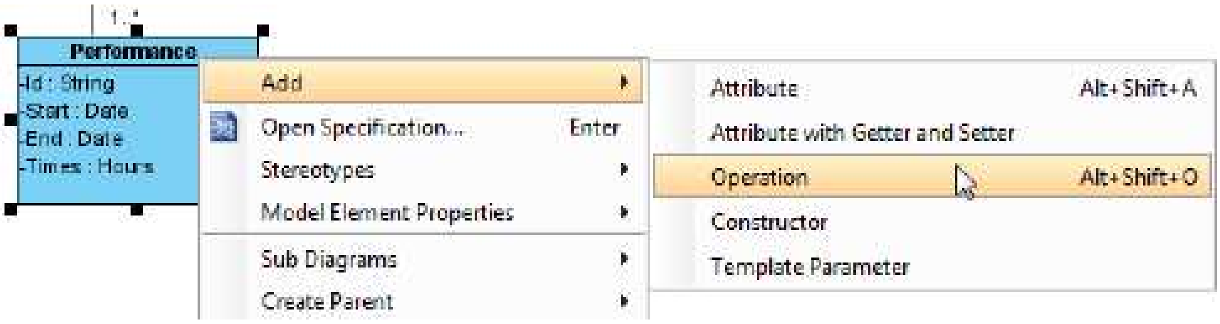
Creating attribute with enter key

After creating an attribute, press the Enter key, another attribute will be created. This method lets you
create multiple attributes quickly and easily.

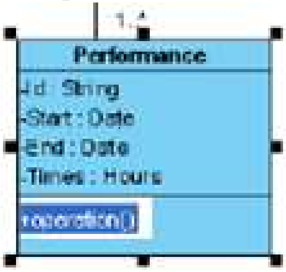


Creating operation

To create operation, right click the class and select **Add** > **Operation** from the pop-up menu.



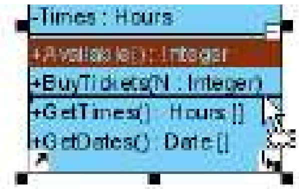
An operation is created.



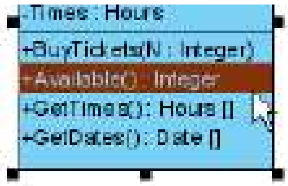
Similar to creating attribute, you can press the Enter key to create multiple operations continuously.

Drag-and-Drop reordering, copying and moving of class members

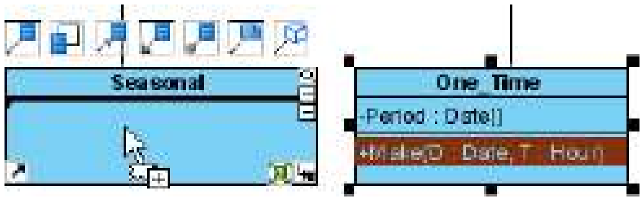
To reorder a class member, select it and drag within the compartment, you will see a thick black line
appears indicating where the class member will be placed.



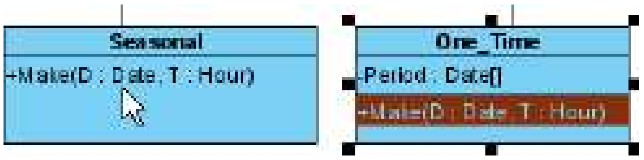
Release the mouse button, the class member will be reordered.



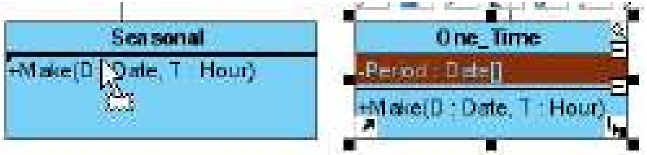
To copy a class member, select it and drag to the target class while keep pressing the Ctrl key, you will
see a thick black line appears indicating where the class member will be placed. A plus sign is shown
beside the mouse cursor indicating this is a copy action.



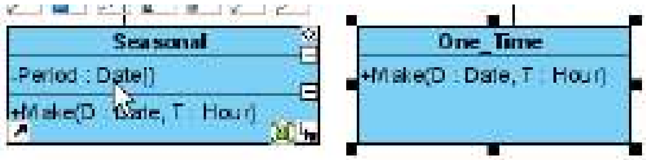
Release the mouse button, the class member will be copied.



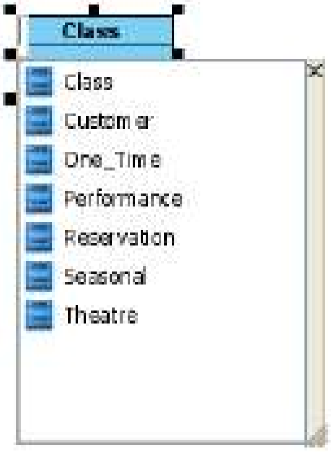
To move a class member, select it and drag to the target class, you will see a thick black line appears
indicating where the class member will be placed. Unlike copy, do not press the Ctrl key when drag, the
mouse cursor without the plus sign indicates this is a move action.



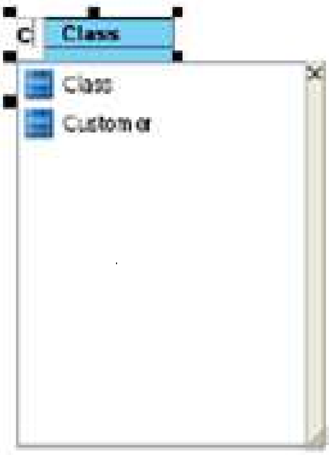
Release the mouse button, the class member will be moved.



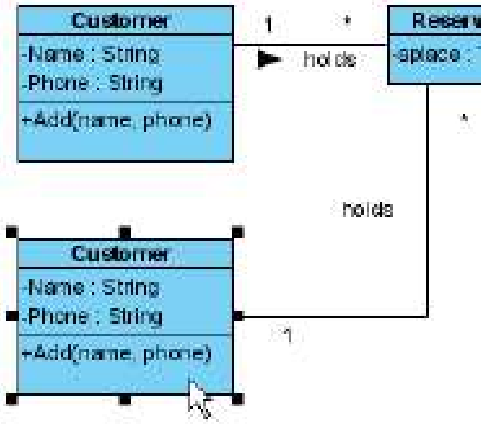
Model name completion for class
The model name completion feature enables quick creation of multiple views for the same class model.
When create or rename class, the list of classes is shown.



Type text to filter classes in the list.



Press up or down key to select class in the list, press Enter to confirm. Upon selecting an existing class,
all class members and relationships are shown immediately.



Experiment No. 8

Aim- Develop java programming language code for sample class diagram.

Objective- To familiar with java coding conventions.

Descirption:

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file rganization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc. These are guidelines for software structural quality. Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier. Coding conventions are only applicable to the human maintainers and peer reviewers of a software project. Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual. Coding conventions are not enforced by compilers. As a result, not following some or all of the rules has no impact on the executable programs created from the source code.

1. Naming Convention

Use full English descriptors that accurately describe the variable/field/class/interface
For example, use names like **firstName**, **grandTotal**, or **CorporateCustomer**.

Use terminology applicable to the domain
If the users of the system refer to their clients as Customer, then use the term Customer for the class, not client.

Use mixed case to make names readable

Use abbreviations sparingly, but if you do so then use then intelligently and document it
For example, to use a short form for the word "number", choose one of **nbr**, **no** or **num**.

Avoid long names (<15 characters is a good tradeoff)

Avoid names that are similar or differ only in case

2. Documentation

Comments should add to the clarity of code.

Avoid decoration, i.e., do not use banner-like comments
Document why something is being done, not just what.

Java Comments

Comment Type	**Usage**	**Example**
Documentation Starts with /** and ends with */	Used before declarations of interfaces, classes, member functions, and fields to document them.	/** * Customer – a person or * organization */
C style Starts with /* and ends with */	Used to document out lines of code that are no longer applicable. It is helpful in debugging.	/* This code was commented out by Ashish Sarin */
Single line Starts with // and go until the end of the line	Used internally within member functions to document business logic, sections of code, and declarations of temporary variables.	// If the amount is greater // than 10 multiply by 100

3. Standards For Member Functions

3. 1 Naming member functions

Member functions should be named using a full English description, using mixed case with the first letter of any non-initial word capitalized. The first word of the member function should be a verb.

Examples

openAccount()
printMailingList()
save()
delete()

This results in member functions whose purpose can be determined just by looking at its name.

Naming Accessor Member Functions

Getters: member functions that return the value of a field / attribute / property of an object.

Use prefix “get” to the name of the field / attribute / property if the field in not boolean
Use prefix “is” to the name of the field / attribute / property if the field is Boolean
A viable alternative is to use the prefix ‘has’ or ‘can’ instead of ‘is’ for boolean getters.

Examples

getFirstName()
isPersistent()

Setters: member functions that modify the values of a field.

Use prefix 'set' to the name of the field.

Examples
setFirstName()

Constructors: member functions that perform any necessary initialization when an object is created. Constructors are always given the same name as their class.

Examples
Customer()
SavingsAccount()

Member Function Visibility

A good design requires minimum coupling between the classes. The general rule is to be as restrictive as possible when setting the visibility of a member function. If member function doesn't have to be public then make it protected, and if it doesn't have to be protected than make it private.

Documenting Member Functions

Member Function Header

Member function documentation should include the following:

- What and why the member function does what it does
- What member function must be passed as parameters
- What a member function returns
- Known bugs
- Any exception that a member function throws
- Visibility decisions (if questionable by other developers)
- How a member function changes the object – it is to helps a developer to understand how a member function invocation will affect the target object.
- Include a history of any code changes
- Examples of how to invoke the member function if appropriate.
- Applicable pre conditions and post conditions under which the function will work properly. These are the assumptions made during writing of the function.
- All concurrency issues should be addressed.
- Explanation of why keeping a function synchronized must be documented.
- When a member function updates a field/attribute/property, of a class that implements the Runnable interface, is not synchronized then it should be documented why it is unsynchronized.

- If a member function is overloaded or overridden or synchronization changed, it should also be documented.

Note: It's not necessary to document all the factors described above for each and every member function because not all factors are applicable to every member function.

Internal Documentation: Comments within the member functionsUse

C style comments to document out lines of unneeded code.
Use single-line comments for business logic.

Internally following should be documented:

Control Structures This includes comparison statements and loops
Why, as well as what, the code does
Local variables
Difficult or complex code
The processing order If there are statements in the code that must be executed in a defined order

Document the closing braces If there are many control structures one inside another

4.0 Techniques for Writing Clean Code:

Document the code Already discussed above
Paragraph/Indent the code: Any code between the { and } should be properly indented
Paragraph and punctuate multi-line statements
Example

```
Line 1   BankAccount newPersonalAccount = AccountFactory
Line 2          createBankAccountFor(currentCustomer, startDate,
Line 3          initialDeposit, branch)
```

Lines 2 & 3 have been indented by one unit (horizontal tab)

Use white space

A few blank lines or spaces can help make the code more readable.

Single blank lines to separate logical groups of code, such as control structures
Two blank lines to separate member function definitions

Specify the order of Operations: Use extra parenthesis to increase the readability of the code using AND and OR comparisons. This facilitates in identifying the exact order of operations in the code

Write short, single command lines Code should do one operation per line So only one statement should be there per line

Standards for Fields (Attributes / Properties)

Naming Fields

Use a Full English Descriptor for Field Names
Fields that are collections, such as arrays or vectors, should be given names that are plural to indicate that they represent multiple values.

Examples
firstName
orderItems

If the name of the field begins with an acronym then the acronym should be completely in lower case

Example
sqlDatabase

Naming Components

Use full English descriptor postfixed by the widget type. This makes it easy for a developer to identify the purpose of the components as well as its type.

Example

okButton
customerList
fileMenu
newFileMenuItem

Naming Constants

In Java, constants, values that do not change, are typically implemented as *static final* fields of classes. The convention is to use full English words, all in upper case, with underscores between the words

Example
MINIMUM_BALANCE
MAX_VALUE
DEFAULT_START_DATE[4]

Field Visibility

Fields should not be declared public for reasons of encapsulation. All fields should be declared private and accessor methods should be used to access / modify the field value. This results in less coupling between classes as the protected / public / package access of field can result in direct access of the field from other classes

Documenting a Field

Document the following:

It's description
Document all applicable invariants Invariants of a field are the conditions that are always true about it. By documenting the restrictions on the values of a field one can understand important business rules, making it easier to understand how the code works / how the code is supposed to work
Examples For fields that have complex business rules associated with them one should provide several example values so as to make them easier to understand
Concurrency issues
Visibility decisions If a field is declared anything but private then it should be documented why it has not been declared private.

Usage of Accesors Accessors can be used for more than just getting and setting the values of instance fields. Accesors should be used for following purpose also:

Initialize the values of fields Use lazy initialization where fields are initialized by their getter member functions.

Example

```
/**
  * Answer the branch number, which is the leftmost four digits of the full account
  * number. Account numbers are in the format BBBBAAAAAA.
  */
protected int getBranchNumber()
{
        if(branchNumber == 0)
        {
                                        h number is 1000, which is the
                                       owntown Bedrock
                setBranchNumber(1000);
        }
        return branchNumber;
}
```

Note:
This approach is advantageous for objects that have fields that aren't regularly accessed
Whenever lazy initialization is used in a getter function the programmer should document what is the type of default value, what the default value as in the example above.

Access constant values Commonly constant values are declared as *static final* fields. This approach makes sense for "constants" that are stable.

If the constants can change because of some changes in the business rules as the business matures then it is better to use getter member functions for constants.

By using accesors for constants programmer can decrease the chance of bugs and at the same time increase the maintainability of the system.

Access Collections The main purpose of accesors is to encapsulate the access to fields so as to reduce the coupling within the code. Collections, such as arrays and vectors, being more complex than single value fields have more than just standard getter and setter member function implemented for them. Because the business rule may require to add and remove to and from collections, accessor member functions need to be included to do so.

Example

Member function type	*Naming Convention*	*Example*
Getter for the collection	getCollection()	getOrderItems()
Setter for the collection	setCollection()	setOrderItems()
Insert an object into the collection	insertObject()	insertOrderItems()
Delete an object from the collection	deleteObject()	deleteOrderItems()
Create and add a new object into the collection	newObject()	newOrderItem()

Note
The advantage of this approach is that the collection is fully encapsulated, allowing programmer to later replace it with another structure
It is common to that the getter member functions be *public* and the setter be *protected*

Always Initialize Static Fields because one can't assume that instances of a class will be created before a static field is accessed

Standards for Local Variables

Naming Local Variables
Use full English descriptors with the first letter of any non-initial word in uppercase.

Naming Streams
When there is a single input and/or output stream being opened, used, and then closed within a member function the convention is to use **in** and **out** for the names of these streams, respectively.

Naming Loop Counters

A common way is to use words like **loopCounters** or simply **counter** because it helps facilitate the search for the counters in the program.
i, j, k can also be used as loop counters but the disadvantage is that search for i ,j and k in the code will result in many hits.

Naming Exception Objects

The use of letter **e** for a generic exception

Declaring and Documenting Local Variables

Declare one local variable per line of code
Document local variable with an endline comment
Declare local variables immediately before their use
Use local variable for one operation only. Whenever a local variable is used for more than one reason, it effectively decreases its cohesion, making it difficult to understand. It also increases the chances of introducing bugs into the code from unexpected side effects of previous values of a local variable from earlier in the code.

Note

Reusing local variables is more efficient because less memory needs to be allocated, but reusing local variables decreases the maintainability of code and makes it more fragile

Standards for Parameters (Arguments) to Member Functions

Naming Parameters

Parameters should be named following the exact same conventions as for local variable

Name parameters the same as their corresponding fields (if any)

Example

If **Account** has an attribute called **balance** and you needed to pass a parameter representing a new value for it the parameter would be called **balance** The field would be referred to as **this.balance** in the code and the parameter would be referred as **balance**

Documenting Parameters

Parameters to a member function are documented in the header documentation for the member function using the *javadoc @param* tag. It should describe:
What it should be used for
Any restrictions or preconditions
Examples If it is not completely obvious what a parameter should be, then it should provide one or more examples in the documentation

Note

Use interface as a parameter to the member function then the object itself.

Standards for Classes

Class Visibility

Use package visibility for classes internal to a component
Use public visibility for the façade of components

Naming classes

Use full English descriptor starting with the first letter capitalized using mixed case for the rest of the name

Documenting a Class

The purpose of the class
Known bugs
The development/maintenance history of the class
Document applicable variants
The concurrency strategy Any class that implements the interface **Runnable** should have its concurrency strategy fully described

Ordering Member Functions and Fields

The order should be:
Constructors
private fields
public member functions
protected member functions
private member functions
finalize()

Standards for Interfaces

Naming Interfaces

Name interfaces using mixed case with the first letter of each word capitalized.
Prefix the letter "I" or "Ifc" to the interface name

Documenting Interfaces

The Purpose
How it should and shouldn't be used

Standards for Packages

Local packages names begin with an identifier that is not all upper case
Global package names begin with the reversed Internet domain name for the organization
Package names should be singular

Documenting a Package

The rationale for the package
The classes in the packages

Standards for Compilation Unit (Source code file)

Naming a Compilation Unit

A compilation unit should be given the name of the primary class or interface that is declared within it. Use the same name of the class for the file name, using the same case.

Beginning Comments

```
/**
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Declaration

Class/interface documentation comment (/**...*/)	See Documentation standard for class / interfaces
Class or interface statement	
Class/interface implementation comment (/*...*/), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
Class (static) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private.
Instance variables	First public, then protected, then package level (no access modifier), and then private.
Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length-generally no more than 70 characters.

Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

Break after a comma.
Break before an operator.
Prefer higher-level breaks to lower-level breaks.
Align the new line with the beginning of the expression at the same level on the previous line.
If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some **examples** of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);
                var = someMethod1(longExpression1,
                someMethod2(longExpression2,
                        longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
                + 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
                - longName5) + 4 * longname6; // AVOID
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
```

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();          //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

Declaration

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //WRONG!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int       level;          // indentation level
int       size;            // size of table
Object currentEntry;    // currently selected table entry
```

Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void myMethod() {
int int1 = 0;       // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
}
```

Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

No space between a method name and the parenthesis "(" starting its parameter list
Open brace "{" appears at the end of the same line as the declaration statement

Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
            }

    int emptyMethod() {}

    ...
}
```

A blank line separates methods

Statements

Simple Statements

Each line should contain at most one statement.

Example:

```
argv++;                 // Correct
argc--;                 // Correct
argv++; argc--;         // AVOID!
```

Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "{ statements }". See the following sections for examples.

The enclosed statements should be indented one more level than the compound statement.

The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way.

Example:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

```
if (condition) {
   statements;
}

if (condition) {
   statements;
} else {
   statements;
}

if (condition) {
   statements;
} else if (condition) {
   statements;
} else {
   statements;
}
```

Note: if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
   statement;
```

for Statements

A for statement should have the following form:

```
for (initialization; condition; update) {
   statements;
}
```

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

while Statements

A while statement should have the following form:

```
while (condition) {
   statements;
}
```

An empty while statement should have the following form:

```
while (condition);
```

do-while Statements

A do-while statement should have the following form:

```
do {
   statements;
} while (condition);
```

switch Statements

A switch statement should have the following form:

```
switch (condition) {
case ABC:
   statements;
   /* falls through */

case DEF:
   statements;
   break;

case XYZ:
   statements;
   break;

default:
   statements;
   break;
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

try-catch Statements

A try-catch statement should have the following format:

```
try {
   statements;
} catch (ExceptionClass e) {
   statements;
}
```

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

```
try {
   statements;
} catch (ExceptionClass e) {
   statements;
} finally {
   statements;
}
```

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:
Between sections of a source file
Between class and interface definitions

One blank line should always be used in the following circumstances:
Between methods
Between the local variables in a method and its first statement
Before a block or single-line comment
Between logical sections inside a method to improve readability

Blank Spaces

Blank spaces should be used in the following circumstances:

A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {
   ...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

A blank space should appear after commas in argument lists.

All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.

Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

The expressions in a for statement should be separated by blank spaces. Example:

```
        for (expr1; expr2; expr3)
```

Casts should be followed by a blank space. Examples:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3))
                              + 1);
```

Naming Conventions Summary

Identifier Type	**Rules for Naming**	**Examples**
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project,	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

	machine, or login names.	

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class Raster;
		class ImageSprite;

Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate;
		interface Storing;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though [5]both are allowed.
	Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except
		int i;
		char c;
		float myWidth;

	for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

Experiment No.9

AIM- Use of testing tool such as JUnit

Objective:-

To show how unit testing is carried out in java using Junit.

Software Required:-

Eclipse IDE and Junit

OVERALL DESCRIPTION :

JUnit is an open source Java testing framework used to write and run repeatable tests which is wildly used now days**.**

Unit Test

Many different types of tests are implemented and executed. In this report we only talk about JUnit with Unit Test Unit test is a method of testing that verifies the individual units of source code are working properly. A unit is the smallest testable part of an application. In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class. A unit test examines the behavior of a distinct task that is not directly dependent on the completion of any other task.

What is JUnit

JUnit is a simple, open source framework to write and run repeatable tests for Java Programming Language. It is an instance of the xUnit architecture for unit testing frameworks. JUnit features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test suites for easily organizing and running tests
- Graphical and textual test runners

Method Design

Here creation of unit test will be described. There are few rules how to write the JUnit method. First of all we have to create the test class in which all test method will be. It's good to name the test case class after the class under test.

Criteria for test methods

For JUnit to recognize a method as test method , it must meet the following criteria:

- The method must be declared public.
- The method must return nothing (void).
- The name of the method must start with the word test.
- The method can't take any arguments.

Assertion methods

An assertion is a function or macro that verifies the behavior of the unit under test. Failure of an assertion typically throws an exception, aborting the execution of the current test.

The TestCase class extends a utility class name Assert in the JUnit framework. The Assert class are included methods which are used to make assertions about the state of testing object. Some assert methods are as follows.

assertEquals([String message], Object expected, Object actual)
This method checks that two values are equal. If they are not, the method throws an AssertionFailedError with the given message (if any).

assertFalse([String message], boolean condition)
This method checks that a condition is false. If it isn't, the method throws an AssertionFailedError with the given message (if any).

assertNotNull([String message], Object object)
This method checks that the object is not null. If it is, the method throws an AssertionFailedError with the given message (if any).

assertNotSame([String message], Object expected, Object actual)
This method checks that two objects not refer to the same object using == operator. If they do, the method throws an AssertionFailedError with the given message (if any).

assertNull([String message], Object object)
This method checks that the object is null. If it isn't, the method throws an AssertionFailedError with the given message (if any).

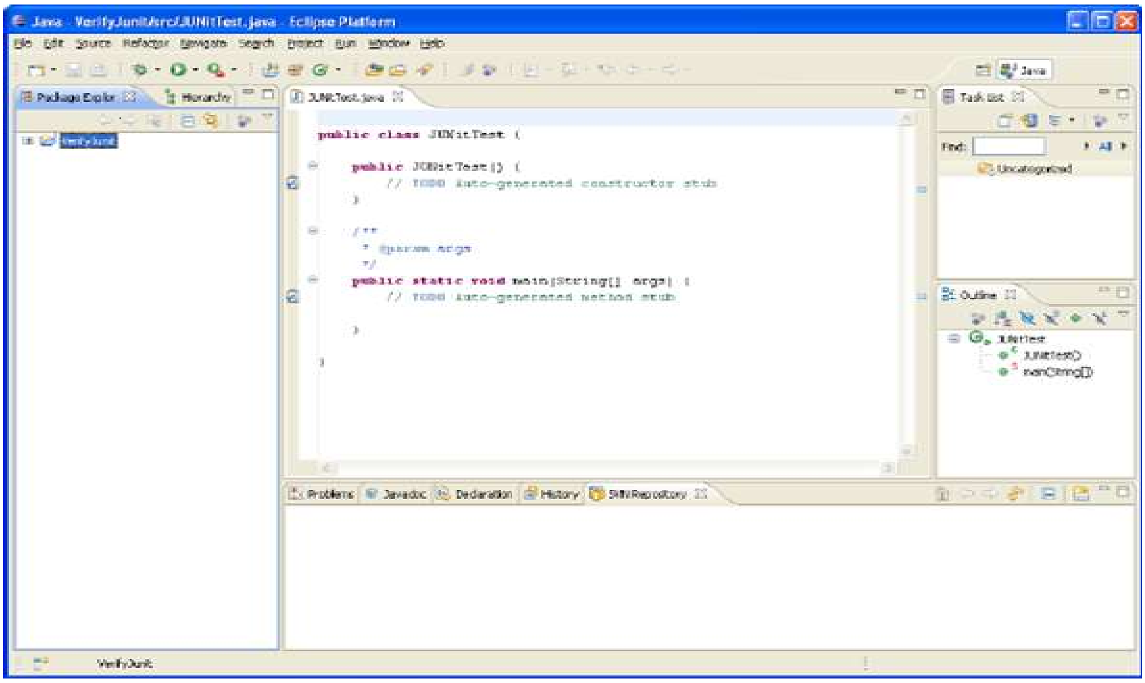
assertSame([String message], Object expected, Object actual)
This method checks that two objects refer to the same object using == operator. If they do not, the method throws an AssertionFailedError with the given message (if any).

assertTrue([String message], boolean condition)
This method checks that a condition is true. If it isn't, the method throws an AssertionFailedError with the given message (if any).

fail(String message)
This method fails a test with the given message.

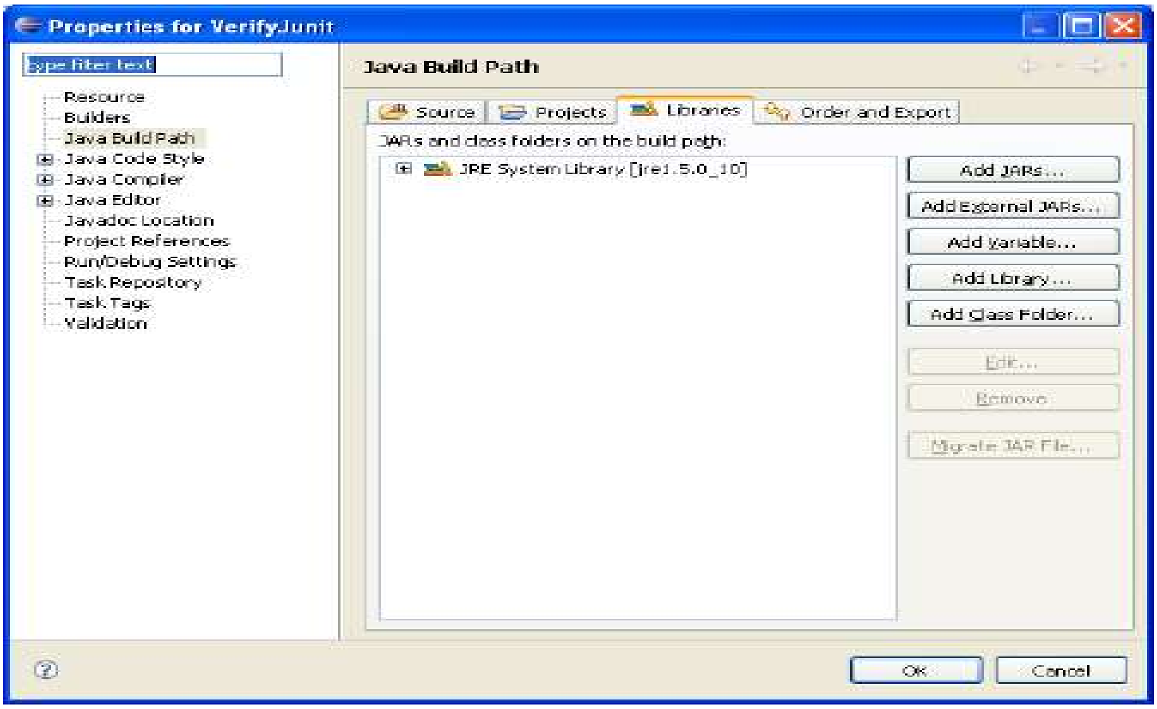
Configure Junit in Eclipse

Step 1: In the Package Explorer view, select your project and right click go to properties, the last item in the Menu. Your Project Name will be different.

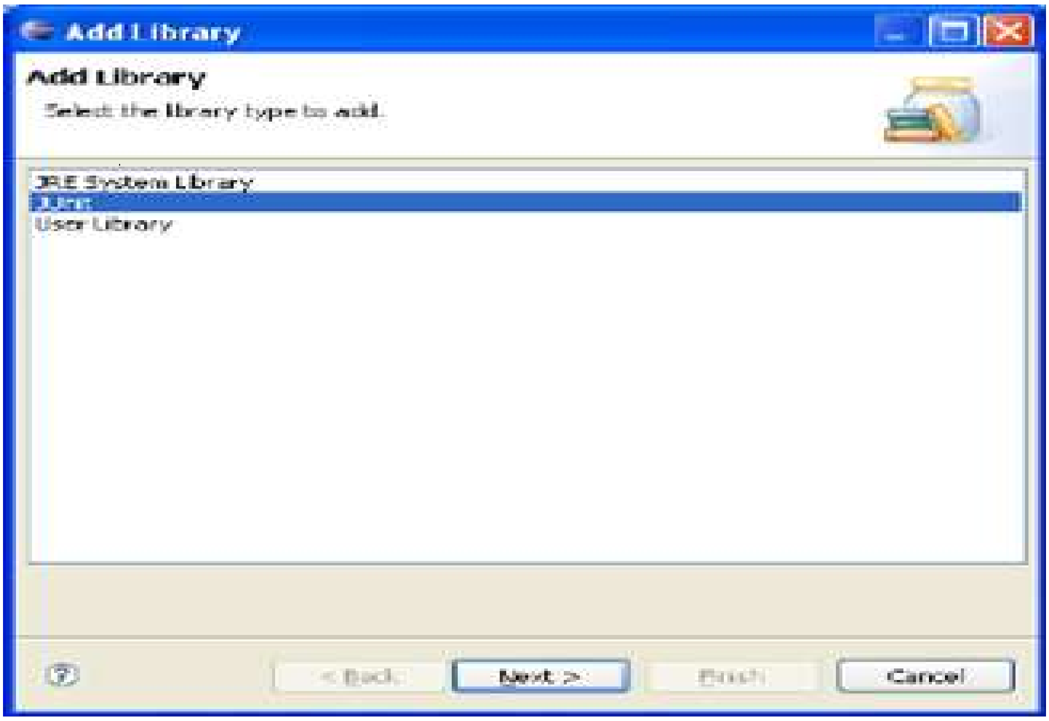


```
public class JUNitTest {

    public JUNitTest() {
        // TODO Auto-generated constructor stub
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

Step 2: Select the Java Build Path and the Libraries tab. The window should look like this.



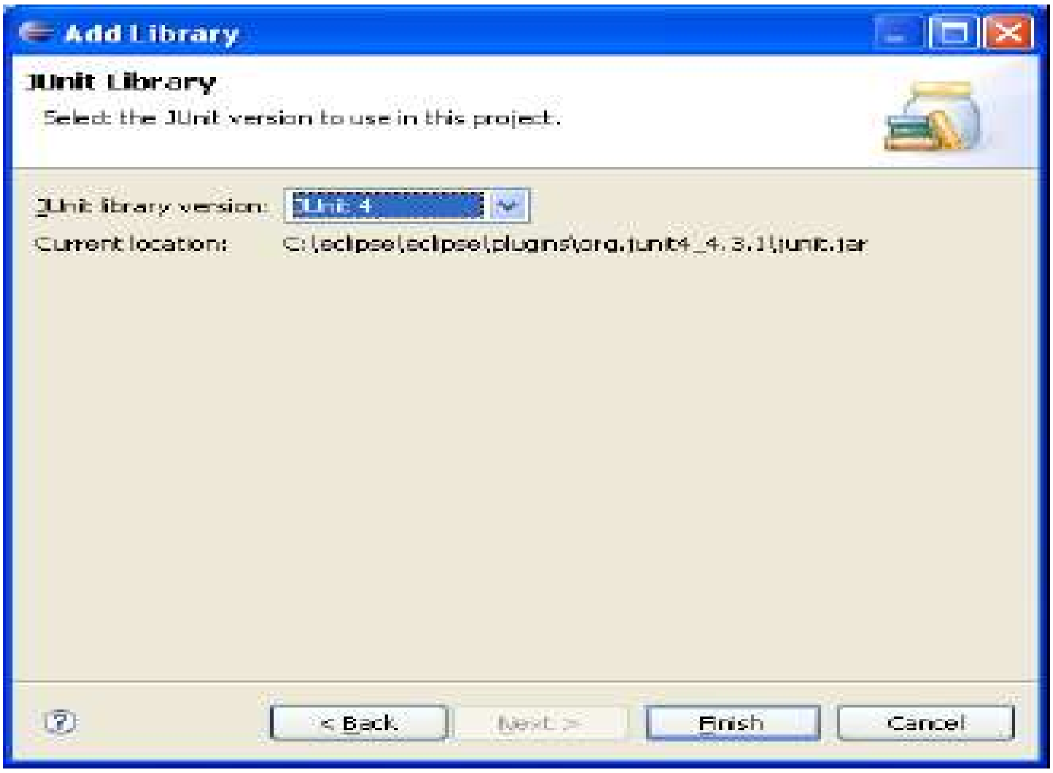
Step 3: Select Add Library and select Junit



Add Library

Select the library type to add.

Step 4: Select JUnit 4, then Finish.



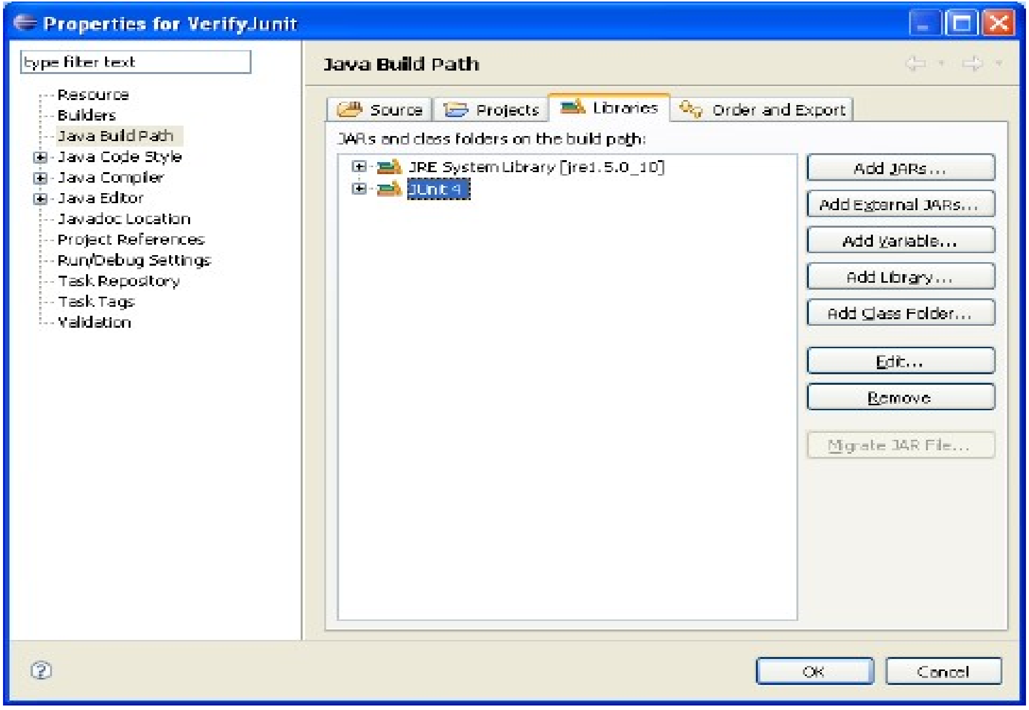
JUnit Library

Select the JUnit version to use in this project.

JUnit library version: JUnit 4

Current location: C:\eclipse\eclipse\plugins\org.junit4_4.3.1\junit.jar

Step 5: Your Window should look like this.



JARs and class folders on the build path:

- JRE System Library [jre1.5.0_10]
- JUnit 4

Experiment No.10

AIM- Project management using GanttProject

Objective: To show how Project management is carried out in GanttProject.

Software Required:-

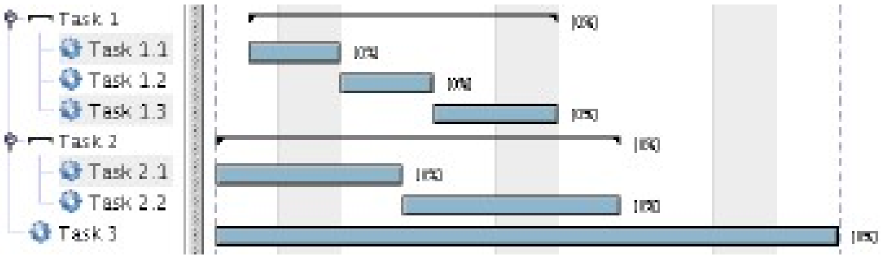
GanttProject

OVERALL DESCRIPTION :

GanttProject is an open source framework used to perform planning, scheduling and resource allocation activities**.**

TASK CREATION

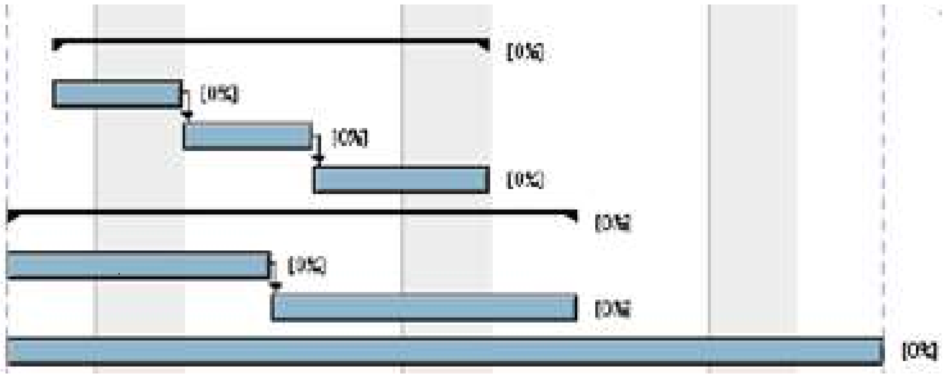
First, you create some tasks by using the New Task button or directly from the Tasks menu choose New Task. The tasks appear on the tree on the left pane; you can directly change their name here. Next, you can organize tasks by indenting them forming groups or categories. So you could have a hierarchy like this:



Tasks can also be re-organized by using the up and down functions. These functions move the selected task up or down in its hierarchy reordering it.

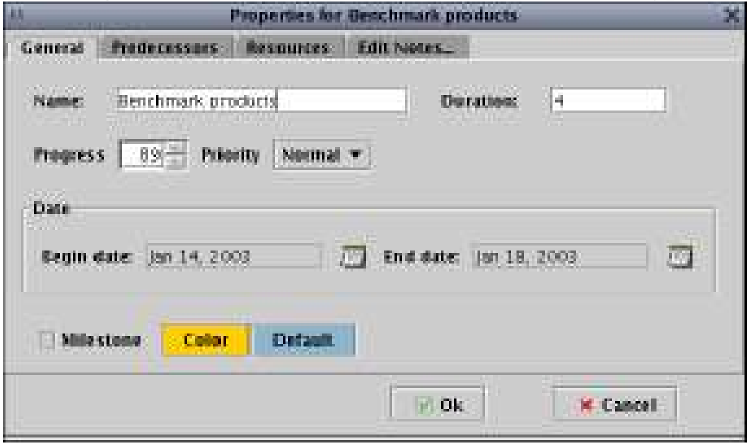
RELATIONSHIPS

Ganttproject allows you to specify a relationship between two tasks. You can set them by dragging directly on the chart. Click and hold on the first task and moving the cursor to the second task. An arrow will appear, following the mouse. Drag the arrowhead to the second task and release the mouse. The second task will be dependent on the first one. You will have a chart like this:



EDITING PROPERTIES

For each task you can edit the properties in a dialog box, by using the Properties menu, or by double-clicking on either the task's name, or it's Gantt bar. The properties box allows you to edit the name of the task, the duration, the percent complete, the start and end dates, the color on the chart, the priority, and the explanatory notes. You can also define the relationship between tasks by choosing different predecessors. You do this by selecting the second panel of the box and choosing the name of the predecessor task, and the type of relationship.


Properties for Benchmark products

General | Predecessors | Resources | Edit Notes...

Name: Benchmark products
Duration: 4
Progress 83
Priority Normal ▼

Date
Begin date: Jan 14, 2003
End date: Jan 18, 2003

□ Milestone

CREATING RESOURCES

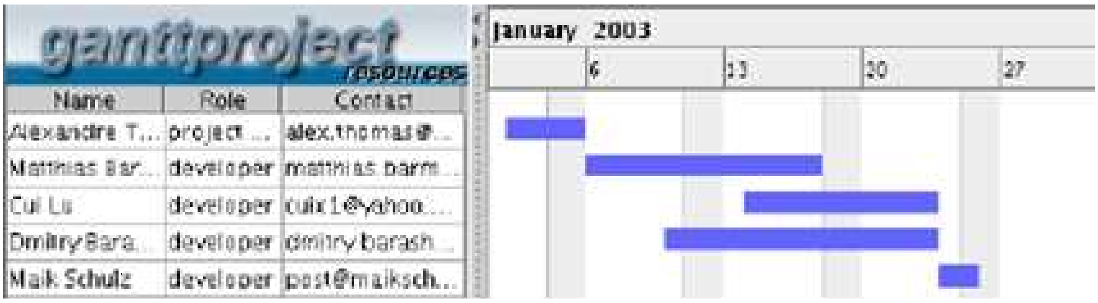
A project is composed of tasks and people (or resources) who are assigned to each task. You can create resources on the Resources panel by specifying the name, the function and contact information (mail or phone by example).

ASSIGN TO TASKS

A resource can be assigned to a task directly on the properties dialog box of the task. Select the third tabbed panel and choose the name of the resource you want to assign. Then, specify a unit for the resources

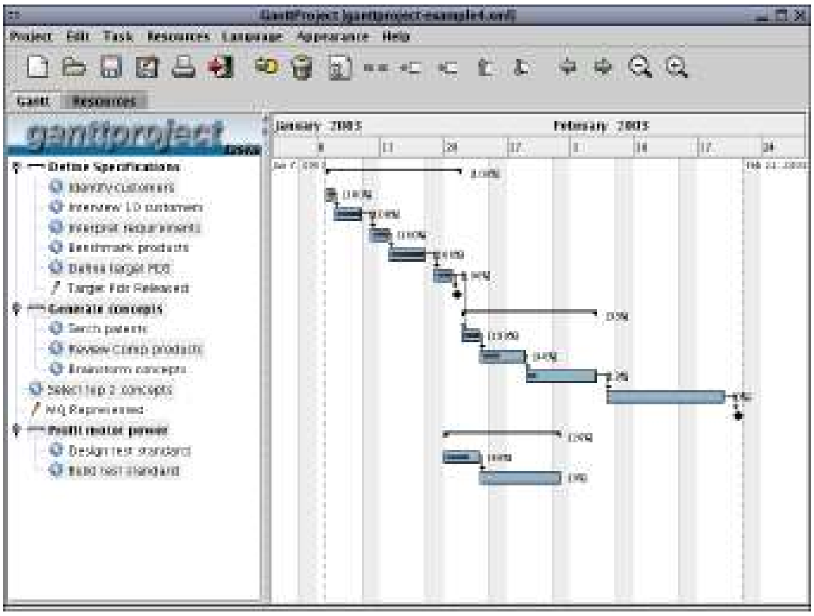
RESOURCES CHART

A special chart is available for all resources on the panel. It shows the resource time allocation and is similar to the Gantt Chart. An example is giving here :

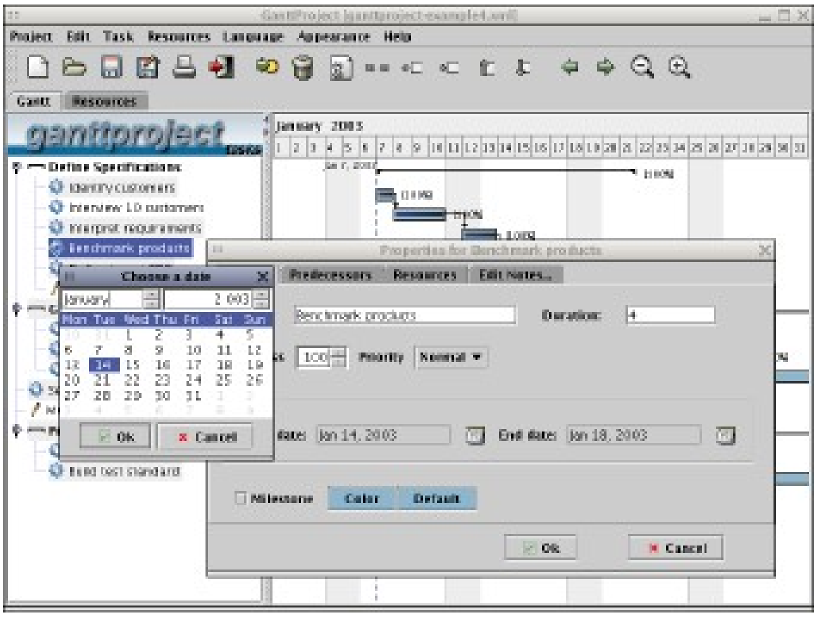


Name	Role	Contact
Alexandre T...	project ...	alex.thomas@...
Matthias Bar...	developer	matthias.barm...
Cui Lu	developer	cuix1@yahoo....
Dmitry Bara...	developer	dmitry.barash...
Maik Schulz	developer	post@maiksch...

Here are some snapshorts of GanttProject:



- Define Specifications
  - Identify customers
  - Interview 10 customers
  - Interpret requirements
  - Benchmark products
  - Define target PDS
  - Target Pds Released
- Generate concepts
  - Serch patents
  - Review Comp products
  - Brainstorm concepts
- Select top 2 concepts
- MQ Represented
- Profil moteur present
  - Design test standard
  - Build test standard



Define Specifications
Identify customers
Interview 10 customers
Interpret requirements
Benchmark products
Build test standard

Jan 7, 2003

100%

January 2003

Mon	Tue	Wed	Thu	Fri	Sat	Sun
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Benchmark products Duration: 4

100 Priority Normal

date: Jan 14, 2003 End date: Jan 18, 2003

□ Milestone

Experiment No.11

AIM- Version control using Subversion

Objective: To familiar with configuration management using TortoiseSVN

Software Required: TortoiseSVN

Description:

In software engineering, configuration management deals with the control and management of the actual software product. The key aspects include using version (or revision) control for source code and other important software artifacts, recording and tracking issues with the software, and ensuring backups are made. This tutorial will focus on the first of these: software version control. While some software engineering practices are critical only for large software development efforts, every software project, regardless of how large or small, should use a version control system for the source code.

Version Control

Version control tracks changes to source code or any other files. A good version control system can tell you what was changed, who changed it, and when it was changed. It allows a software developer to undo any changes to the code, going back to any prior version, release, or date. This can be particularly helpful when a researcher is trying to reproduce results from an earlier paper or report and merely requires documentation of the version number. Version control also provides a mechanism for incorporating changes from multiple developers, an essential feature for large software projects or any projects with geographically remote developers. Some key concepts pertaining to version control are discussed below. Note that the generic descriptor "file" is used and could represent not only source code, but also user's manuals, software tests, design documents, web pages, or any other item produced during software development.

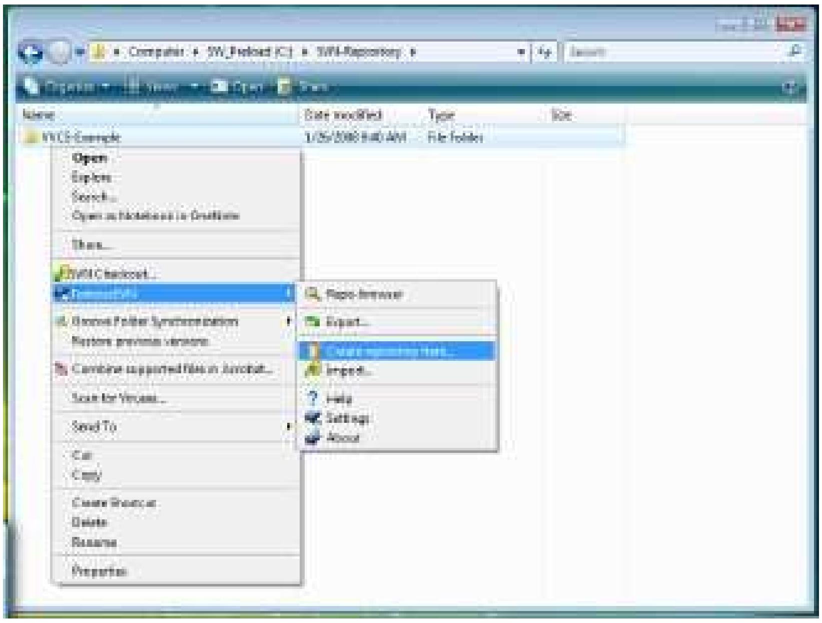
repository – single location where the current and all prior versions of the files are stored
working copy – the local copy of a file from the repository which can be modified and then checked in or "committed" to the repository
check-out – the process of creating a working copy from the repository (either the current version or an earlier version)
check-in – a check-in or commit occurs when changes made to a working copy are merged into the repository
diff – a summary of the differences between a working copy and a file in the repository, often taking the form of the two files side-by-side with differences highlighted
conflict – a conflict occurs when two or more developers attempt to make changes to the same file and the system is unable to reconcile the changes (note: conflicts generally must be resolved by either choosing one version over the other or by integrating the changes from both into the repository by hand)
update – merges recent changes to the repository into a working copy

The basic steps that one would use to get started with a version control tool are as follows:

1. Create a repository
2. Import a directory structure and/or files into the repository
3. Check-out the repository version as a working copy
4. Edit/modify the files in the working copy and examine the differences between the working copy and the repository (i.e., diff)
5. Check-in (or commit) the changes to the repository

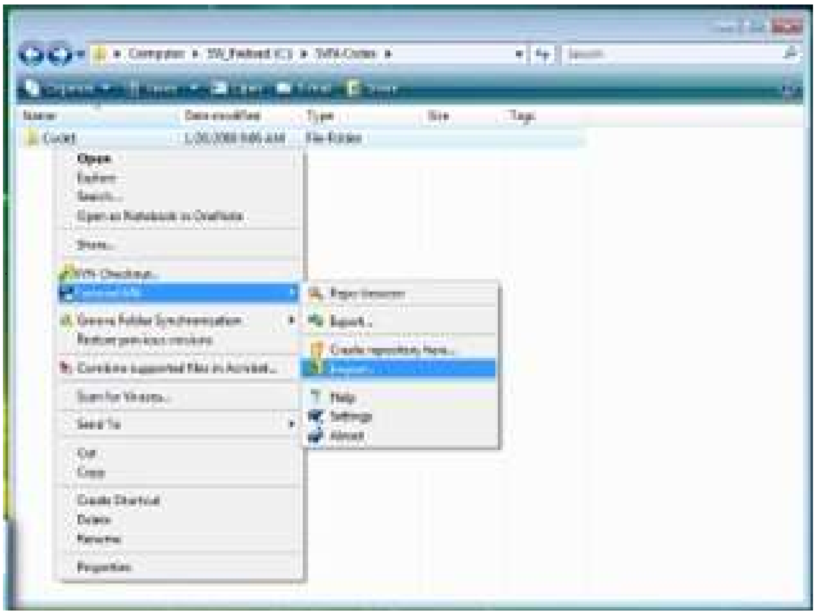
1. Creating a Repository

Determine a location for the repository, ideally on a server which is automatically backed up. Create a folder with the name of the repository; in this example the repository is called “VVCS-Example.” Right click on the folder name, choose “TortoiseSVN” (which is integrated into the Microsoft Windows Explorer menu), then “Create Repository Here.” Choose the Native Filesystem, then you should see the message “Repository Successfully Created.”



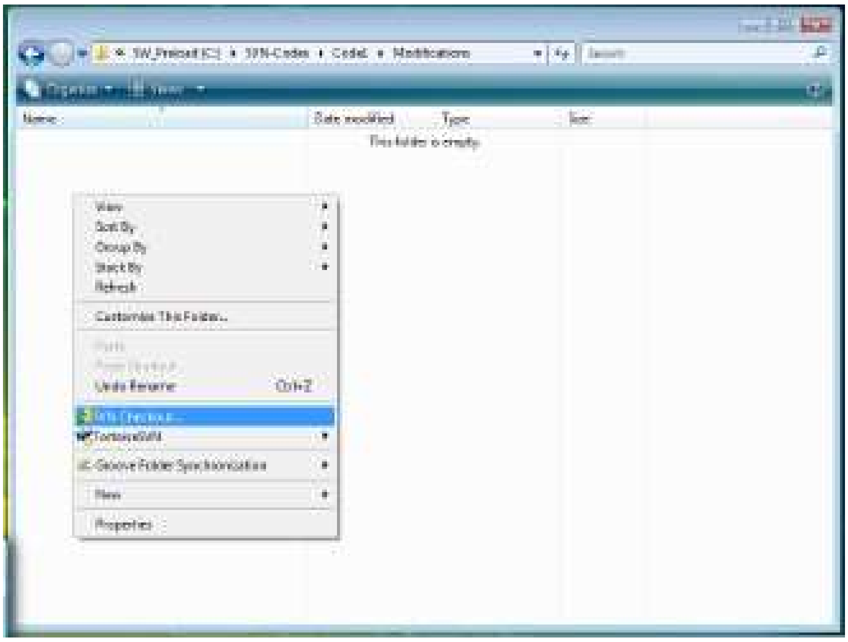
2. Importing a File into the Repository

Right click on the directory containing the file(s) and/or directory structure you wish to import to the repository (note, the directory that you click on will not be imported). Here we will simply be importing the file “code1.f” from directory “Code1.” This code creates a 17×17 two-dimensional Cartesian grid for x and y between 0 and 1. Browse until you find the location of the repository “VVCS-Example” and select that directory name. This version of the code will be Revision 1.



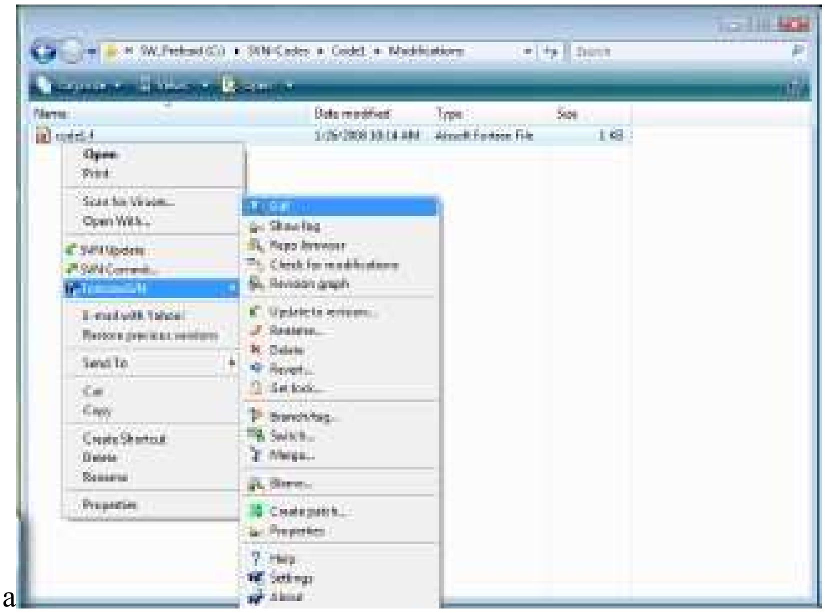
3. Checking the Code out from the Repository

You now have the code “code1.f” safely placed in the repository. To modify this code and create a new revision, you will need to check out a working copy of the code. Go to the directory where you will be modifying the code, in this example, the directory “Modifications.” Right click in Windows Explorer, and select “SVN Checkout…” Select the name of the repository you just created, then click “OK.” You will now get a window telling you that you are at Revision 1. Notice the green check mark on the “code1.f” icon. This indicates that this working copy is up to date with the version in the repository.

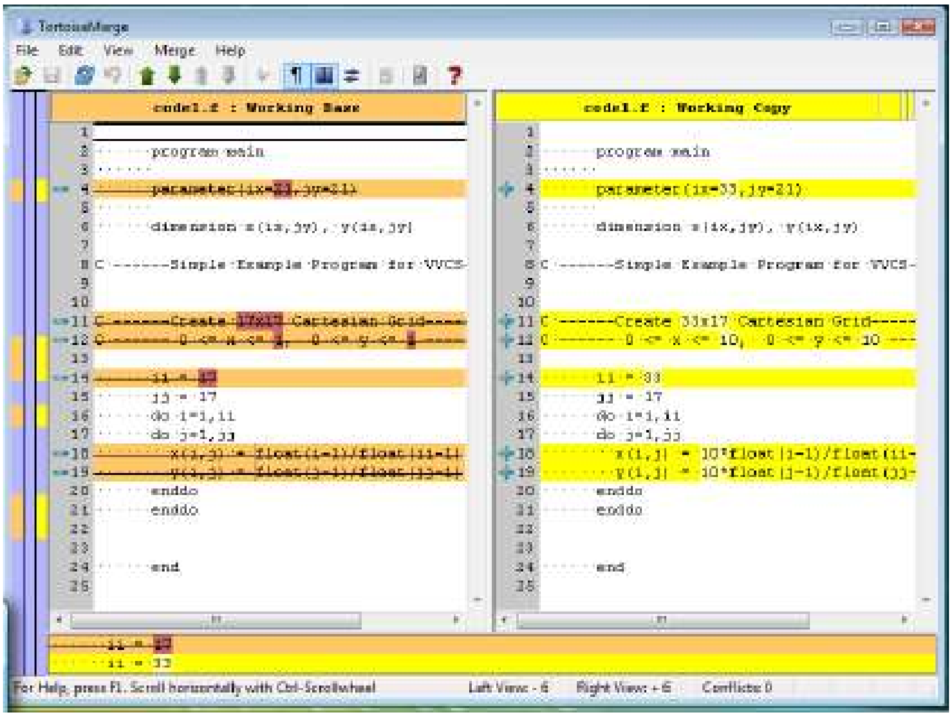


4. Modify the Code and Compare to the Repository Version

The code “code1.f” can now be modified. Here we will change the code to allow the Cartesian grid to contain 33x17 points between the values of zero and ten. Once the code has been modified, you will notice that the green check mark has been replaced by a red exclamation point, indicating that the current working copy has been modified from the version in the repository. To examine these differences, right click on the “code1.f” file, select “TortoiseSVN,” then “Diff.” This opens the “TortoiseMerge” tool which clearly shows the modifications to the repository version (Working Base) that were made in the Working Copy.



a



code1.f : Working Base

```

      program main

      parameter(ix=17,jy=21)

      dimension x(ix,jy), y(ix,jy)

C ------Simple Example Program for VVCS-


C ------Create 17x17 Cartesian Grid----
C ------- 0 <= x <= 1,  0 <= y <= 1 ----

      ii = 17
      jj = 17
      do i=1,ii
      do j=1,jj
         x(i,j) = float(i-1)/float(ii-1)
         y(i,j) = float(j-1)/float(jj-1)
      enddo
      enddo


      end

```

code1.f : Working Copy

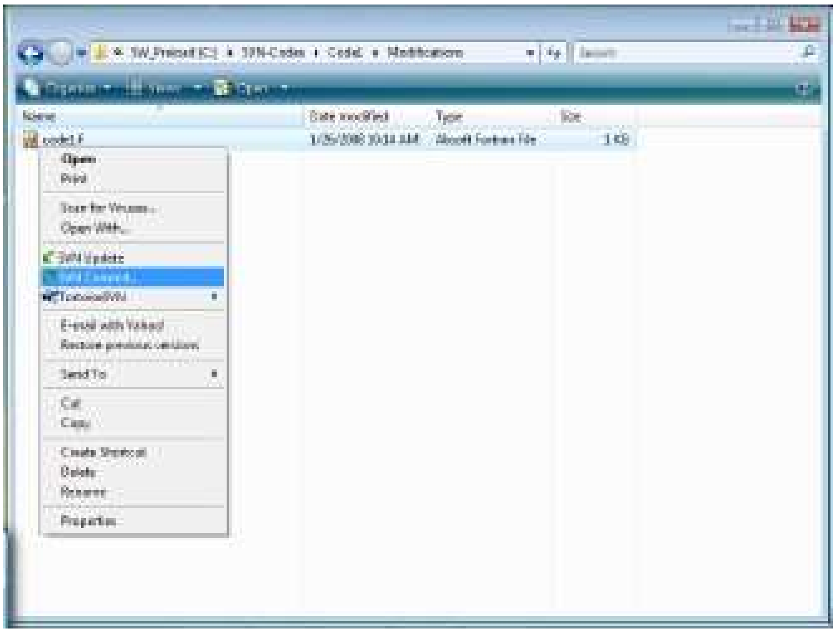
```

      program main

      parameter(ix=33,jy=21)

      dimension x(ix,jy), y(ix,jy)

C ------Simple Example Program for VVCS-


C ------Create 33x17 Cartesian Grid-----
C ------- 0 <= x <= 10,  0 <= y <= 10 ---

      ii = 33
      jj = 17
      do i=1,ii
      do j=1,jj
         x(i,j) = 10*float(i-1)/float(ii-
         y(i,j) = 10*float(j-1)/float(jj-
      enddo
      enddo


      end

```

```
      ii = 17
      ii = 33
```

For Help, press F1. Scroll horizontally with Ctrl-Scrollwheel   Left View: -6   Right View: +6   Conflicts: 0

5. Check the Code back into the Repository

When the changes are complete, the repository can be updated with your modified Working Copy by performing a checkin. Just right click on the file (or directory) and select “SVN Commit…” Enter a message describing the changes that were made, then select “OK.” You are now at Revision 2.


SW_Preload (C:) ▸ SVN-Codes ▸ Code1 ▸ Modifications
Search
Organize ▾
Views ▾
Open ▾
Name
Date modified
Type
Size
code1.f
1/26/2008 10:14 AM
Absoft Fortran File
3 KB
Open
Print
Scan for Viruses...
Open With...
SVN Update
SVN Commit...
TortoiseSVN
E-mail with Yahoo!
Restore previous versions
Send To
Cut
Copy
Create Shortcut
Delete
Rename
Properties