MRCET CAMPUS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS INSTITUTION – UGC, GOVT. OF INDIA)

Department of CSE

(Emerging Technologies)

(CYBER SECURITY)

B.TECH(R-20 Regulation)

(III YEAR – I SEM)

(2023-24)

COMPILER DESIGN AND CASETOOLS LAB

(R20A0587)

LAB MANUAL

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12(B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE-Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad–500100, Telangana State, India

Department of Computer Science and Engineering

EMERGING TECHNOLOGIES

COMPILER DESIGN AND CASETOOLS LAB

(R20A0587)

LAB MANUAL

Prepared by

V.SUNEETHA,

ASSOCIATE PROFESSOR

On

30.07.2021

Department of Computer Science and Engineering

EMERGING TECHNOLOGIES

Vision

- "To be at the forefront of Emerging Technologies and to evolve as a Centre of Excellence in Research, Learning and Consultancy to foster the students into globally competent professionals useful to the Society."

Mission

The department of CSE (Emerging Technologies) is committed to:

- To offer highest Professional and Academic Standards in terms of Personal growth and satisfaction.
- Make the society as the hub of emerging technologies and thereby capture opportunities in new age technologies.
- To create a benchmark in the areas of Research, Education and Public Outreach.
- To provide students a platform where independent learning and scientific study are encouraged with emphasis on latest engineering techniques.

QUALITY POLICY

- To pursue continual improvement of teaching learning process of Undergraduate and Post Graduate programs in Engineering & Management vigorously.
- To provide state of art infrastructure and expertise to impart the quality education and research environment to students for a complete learning experiences.
- Developing students with a disciplined and integrated personality.
- To offer quality relevant and cost effective programmes to produce engineers as per requirements of the industry need.

For more information: www.mrcet.ac.in

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(UGC-Autonomous Institution , Govt. of India)
(Permanently Affiliated to JNTUH, Approved by AICTE-Accredited by NBA & NAAC- A-Grade; ISO 9001:2008 Certified)
Maisammaguda, Dhulapally Post, Via Hakimpet, Secunderabad – 500100
DEPARTMENT OF EMERGING TECHNOLOGIES

COMPILER DESIGN AND CASE TOOLS MANUAL (R20A0587)

TABLE OF CONTENTS

Sl. No	Description of the Item / Title	Page No.	Remarks
1.	General Laboratory Instructions	**i.**	
2.	Importance/Rationale behind the CD Lab		
3.	Objectives & Outcomes	**ii.**	
4.	Software / Hardware Requirements	**iii.**	
5.	**Case Study:** Description of the Syntax of the source Language(mini language) for which the compiler components are designed	**4**	
WEEK 1	Write a LEX Program to Scan and Count the number of characters, words, and lines in a file.	**7**	
WEEK 2	Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.	**11**	
WEEK 3	Write a C Program to implement DFAs that recognize identifiers, constants, and operators of the mini language.	**17**	
WEEK 4	Design a lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and new lines, comments etc.	**21**	
WEEK 5	Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.	**27**	
WEEK 6	Design Predictive Parser for the given language	**30**	
WEEK 7	Design a LALR bottom up parser for the given language	**35**	
WEEK 8	Convert the BNF rules into Yacc form and write code to generate abstract syntax tree.	**38**	
WEEK 9	A program to generate machine code from the Abstract syntax tree generated by the parser.	**44**	
WEEK 10	UML Diagram for ATM Transaction System	46	
WEEK 11	UML Diagram for Library Management System		
WEEK 12	UML Diagram for College Administration System		

Importance of Compiler Design &Case Tolls Manual

Every software has to be translated to its equivalent machine instruction form so that it will be executed by the underlying machine. There are many different types of system softwares that help during this translation process. Compiler is one such an important **System Software** that converts High level language programs to its equivalent machine (low level) language. It is impossible to learn and use machine language in software development for the users. Therefore we use high level computer languages to write programs and then convert such programs into machine understandable form with the help of mediator softwares such as compilers, interpreters, assemblers etc.Thus compiler bridges the gap between the user and the machine, i.e., computer.

It's a very complicated piece of software which took 18 man years to build first compiler .To build this software we must understand the principles, tools, and techniques used in its working. The compiler goes through the following sequence of steps called phases of a compiler.

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code Generation
5) Code Optimization
6) Code Generation.

In performing its role, compiler also uses the services of two essential components namely, Symbol Table and Error Handler

The objective of the Compiler Design Laboratory is to understand and implement the principles, techniques, and also available tools used in compiler construction process. This will enable the students to work in the development phase of new computer languages in industry.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
Maisammaguda, Dhulapally Post, Via Hakimpet, Secunderabad – 500100

DEPARTMENT OF EMERGING TECHNOLOGIES

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.
3. Student should enter into the laboratory with:
    a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
    b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.
    c. Proper Dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out ; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department **Principal**

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
Maisammaguda, Dhulapally Post, Via Hakimpet, Secunderabad – 500100

--

DEPARTMENT OF EMERGING TECHNOLOGIES

OBJECTIVES AND OUTCOMES

OBJECTIVES:

- To provide an Understanding of the language translation peculiarities by designing complete translator for mini language.

OUTCOMES : By the end of the semester, students will be able

- To understand the practical approach of how a compiler works.
- To Apply the techniques used in Compiler Construction
- To generate few phases of the compiler for the mini language using tools

RECOMMENDED SYSTEM / SOFTWARE REQUIREMENTS:

1.Intel based desktop PC with minimum of 166MHz or faster processor with at least 64 MB RAM and 100 MB free disk space.

2. C ++ Compiler and JDK kit, Lex or Flex and YACC tools (Unix/Linux utilities)

USEFUL TEXT BOOKS / REFERECES / WEBSITES :

1. Modern compiler implementation in C, Andrew w.Appel, Revised Edn, Cambridge University Press
2. Principles of Compiler Design. – A.V Aho, J.D Ullman ; Pearson Education.
3. **lex&yacc** , -John R Levine, Tony Mason, Doug Brown; O'reilly.
4. **Compiler Construction**,- LOUDEN, Thomson.
5. Engineering a compiler – Cooper& Linda, Elsevier
6. Modern Compiler Design – Dick Grune,Henry E.Bal, Cariel TH Jacobs, Wiley Dreatech

SOURCE LANGUAGE (A Case Study)

Consider the following mini language, a simple procedural High Level Language, operating on integer data with a syntax looking vaguely like a simple C crossed with Pascal. The syntax of the language is defined by the following BNF grammar:

<program> ::= <block>
<block> ::= { <variable definition> <slist> }
| { <slist> }
<variable definition> ::= int <vardeflist> ;
<vardeflist> ::= <vardec> | <vardec>, <vardeflist>
<vardec> ::= <identifier> | <identifier> [<constant>]
<slist> ::= <statement> |<statement> ; <slist>
<statement> ::= <assignment> | <ifstatement> | <whilestatement> | <block>
| <printstatement> | <empty>
<assignment> ::= < identifier> = <expression>
| <identifier> [<expression>] = [<expression>
<ifstatement> ::= if <bexpression> then <slist> else <slist> endif
| if <bexpression> then <slist> endif
<whilestatement> ::= while <bexpression> do <slist> enddo
<printstatement> ::= print{ <expression> }
<expression> ::= <expression> <addingop> <term> | <term> | <addingop> <term>
<bexpression> ::= <expression> <relop> <expression>
<relop> ::= < | <= | = = | >= | > |!=
<addingop> ::= + | -
<term> ::= <term> <multop> <factor> | <factor>
<multop> ::= * | /
<factor> ::= <constant> | <identifier> |<identifier> [<expression>
| (<expression>)
<constant> ::= <digit> | <digit> <constant>
<identifier> ::= <identifier> <letterordigit> | <letter>
<letterordigit> ::= a|b|c|….|y|z
<digit> ::= 0|1|2|3|…|8|9
<empty> ::= has the obvious meaning

Comments : zero or more characters enclosed between the standard C/Java style comment brockets /*…*/.
The language has the rudimentary support for 1-Dimensional arrays. Ex: int a[3] declares a as an array of 3 elements, referenced as a[0],a[1],a[2].

Sample Program written in this language is :

```
{
 int a[3],t1,t2;
t1=2;
a[0]=1; a[1]=2; a[t1]=3;
t2= -(a[2]+t1*6) / a[2]-t1);
if t2>5 then
print(t2);
```

```
else
 {
  int t3;
  t3=99;
  t2=25;
   print(-11+t2*t3);  /* this is not a comment on two lines */
}
 Endif
 }
```

1. Write a LEX Program to Scan and Count the number of characters, words, and lines in a file.
2. Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.
3. Write a C Program to implement DFAs that recognize identifiers, constants, and operators of the mini language.
4. Design a Lexical analyzer for the above language. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.
5. Implement the lexical analyzer using JLex, flex, flex or lex or other lexical analyzer generating tools.
6. Design Predictive parser for the given language
7. Design LALR bottom up parser for the above language.
8. Convert the BNF rules into Yacc form and write code to generate abstract syntax tree.
9. Write program to generate machine code from the abstract syntax tree generated by the parser. The following instruction set may be considered as target code.

The following is a simple register-based machine, supporting a total of 17 instructions. It has three distinct internal storage areas. The first is the set of 8 registers, used by the individual instructions as detailed below, the second is an area used for the storage of variables and the third is an area used for the storage of program. The instructions can be preceded by a label. This consists of an integer in the range 1 to 9999 and the label is followed by a colon to separate it from the rest of the instruction. The numerical label can be used as the argument to a jump instruction, as detailed below.

In the description of the individual instructions below, instruction argument types are specified as follows:

R specifies a register in the form R0, R1, R2, R3, R4, R5, R6 or R7 (or r0, r1, etc).
L specifies a numerical label (in the range 1 to 9999).
V specifies a "variable location" (a variable number, or a variable location pointed to by a register - see below).

A specifies a constant value, a variable location, a register or a variable location pointed to by a register (an indirect address). Constant values are specified as an integer value, optionally preceded by a minus sign, preceded by a # symbol. An indirect address is specified by an @ followed by a register.

So, for example an A-type argument could have the form 4 (variable number 4), #4 (the constant value 4), r4 (register 4) or @r4 (the contents of register 4 identifies the variable location to be accessed).

The instruction set is defined as follows:
LOAD A, R
loads the integer value specified by A into register R.
STORE R, V
stores the value in register R to variable V.
OUT R
outputs the value in register R.
NEG R
negates the value in register R.
ADD A, R
adds the value specified by A to register R, leaving the result in register R.
SUB A, R
subtracts the value specified by A from register R, leaving the result in register R.
MUL A, R
multiplies the value specified by A by register R, leaving the result in register R.
DIV A, R
divides register R by the value specified by A, leaving the result in register R.
JMP L
causes an unconditional jump to the instruction with the label L.
JEQ R, L
jumps to the instruction with the label L if the value in register R is zero.
JNE R, L
jumps to the instruction with the label L if the value in register R is not zero.
JGE R, L
jumps to the instruction with the label L if the value in register R is greater than or equal to zero.
JGT R, L
jumps to the instruction with the label L if the value in register R is greater than zero.
JLE R, L
jumps to the instruction with the label L if the value in register R is less than or equal to zero.
JLT R, L
jumps to the instruction with the label L if the value in register R is less than zero.
NOP
is an instruction with no effect. It can be tagged by a label.
STOP
stops execution of the machine.
All programs should terminate by executing a STOP instruction.

1.Problem Statement: Write a LEX Program to count the number of lines, spaces and tabs

AIM : Write a LEX Program to count the number of lines, spaces and tabs

ALGORITHM / PROCEDURE/PROGRAM:

1. Start
2. Read the input file/text
3. Initialize the counters for characters, words, lines to zero
4. Scan the characters, words, lines and
5. increment the respective counters
6. Display the counts
7. End

Program:

```
/*lex code to count the number of lines,
   tabs and spaces used in the input*/
%{
#include<stdio.h>
int lc=0, sc=0, tc=0, ch=0; /*Global variables*/
%}
/*Rule Section*/
%%
\n lc++; //line counter
([ ])+ sc++; //space counter
\t tc++; //tab counter
. ch++;     //characters counter
%%
int yywrap(){}
  int main()
{
   // The function that starts the analysis
   yylex();
   printf("\nNo. of lines=%d", lc);
   printf("\nNo. of spaces=%d", sc);
   printf("\nNo. of tabs=%d", tc);
   printf("\nNo. of other characters=%d", ch);
return 0;
```

}

Input:

Hello

How are you?

Output:

No. of lines=2

No. of spaces=4

No. of tabs=1

No. of other characters=15

[Viva Questions]

1. What is Compiler?
2. List various language Translators.
3. Is it necessary to translate a HLL program? Explain.
4. List out the phases of a compiler?
5. Which phase of the compiler is called an optional phase?why

Exercise program: Write a LEX Program to convert the substring “abc” to “ABC” from the given input string.

2. Problem Statement: Write a C Program to implement NFAs that recognize identifiers,constants, and operators of the mini language.

AIM: To Write a C Program to implement NFAs that recognize identifiers, constants, andoperators of the mini language.

ALGORITHM / PROCEDURE / PROGRAM:

1. Start
2. Design the NFA (N) to recognize Identifiers, Constants, and Operators
3. Read the input string **w** give it as input to the NFA
4. NFA processes the input and outputs “Yes” if *w* ∈ L(N), “No”, otherwise
5. Display the output
6. End

Program:

```
#include  <stdio.h>
#include <string.h>
#include <stdlib.h>
// Returns '1' if the character is a DELIMITER.
int isDelimiter(char ch)
{
if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
ch == '[' || ch == ']' || ch == '{' || ch == '}')
return (1);
return (0);
}
// Returns '1' if the character is an OPERATOR.
int isOperator(char ch)
{
if (ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == '>' || ch == '<' ||
ch == '=')
return (1);
return (0);
}
// Returns '1' if the string is a VALID IDENTIFIER.
int validIdentifier(char* str)
{
if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
str[0] == '3' || str[0] == '4' || str[0] == '5' ||
str[0] == '6' || str[0] == '7' || str[0] == '8' ||
str[0] == '9' || isDelimiter(str[0]) == 1)
```

```
        return (0);
        return (1);
        }
        // Returns '1' if the string is a KEYWORD.
        int isKeyword(char* str)

{
        if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int")
        || !strcmp(str, "double") || !strcmp(str, "float")
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (1);
        return (0);
        }

        // Returns '1' if the string is an INTEGER.
        int isInteger(char* str)
        {
        int i, len = strlen(str);
        if (len == 0)
        return (0);
        for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
        && str[i] != '3' && str[i] != '4' && str[i] != '5'
        && str[i] != '6' && str[i] != '7' && str[i] != '8'
        && str[i] != '9' || (str[i] == '-' && i > 0))
        return (0);
        }
        return (1);
        }
// Returns '1' if the string is a REAL NUMBER.

        int isRealNumber(char* str)
        {
        int i, len = strlen(str);
        int hasDecimal = 0;
        if (len == 0)
        return (0);
        for (i = 0; i < len; i++) {
```

```
if (str[i] != '0' && str[i] != '1' && str[i] != '2'
&& str[i] != '3' && str[i] != '4' && str[i] != '5'
&& str[i] != '6' && str[i] != '7' && str[i] != '8'
&& str[i] != '9' && str[i] != '.' ||
(str[i] == '-' && i > 0))
return (0);
if (str[i] == '.')
hasDecimal = 1;
}
return (hasDecimal);
```

```
}
// Extracts the SUBSTRING.
        char* subString(char* str, int left, int right)
        {
        int i;
        char* subStr = (char*)malloc(
        sizeof(char) * (right - left + 2));
        for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
        subStr[right - left + 1] = '\0';
        return (subStr);
        }
        // Parsing the input STRING.
        void parse(char* str)
        {
        int left = 0, right = 0;
        int len = strlen(str);
        while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == 0)
        right++;
        if (isDelimiter(str[right]) == 1 && left == right) {
        if (isOperator(str[right]) == 1)
        printf("'%c' IS AN OPERATOR\n", str[right]);
        right++;
        left = right;
        } else if (isDelimiter(str[right]) == 1 && left != right
        || (right == len && left != right)) {
        char* subStr = subString(str, left, right - 1);
        if (isKeyword(subStr) == 1)
        printf("'%s' IS A KEYWORD\n", subStr);
        else if (isInteger(subStr) == 1)
        printf("'%s' IS AN INTEGER\n", subStr);
        else if (isRealNumber(subStr) == 1)
        printf("'%s' IS A REAL NUMBER\n", subStr);
        else if (validIdentifier(subStr) == 1
        && isDelimiter(str[right - 1]) == 0)
        printf("'%s' IS A VALID IDENTIFIER\n", subStr);
        else if (validIdentifier(subStr) == 0
        && isDelimiter(str[right - 1]) == 0)
        printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
        left = right;
        }
        }
        return;
        }
        // DRIVER FUNCTION
        void main()
```

```
{
// maximum length of string is 100 here
char str[100] = "int a = c + y; ";
//clrscr();
parse(str); // calling the parse function
}
```

OUTPUT:

```
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'c' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'y' IS A VALID IDENTIFIER
```

EXERCISE:

1) Design an NFA to recognize the identifiers, keywords, constants, and comments of C language?
2) Write a C /Python C Program for the implementation of above NFA.

[Viva Questions]

1. What is a Preprocessor and what is its role in compilation?
2. Which language is both compiled and interpreted?
3. List out the languages that are interpreted?
4. Explain the working of a NFA?
5. When do you prefer to design an NFA to DFA?

Exercise Program: Write a Lex Program to count the number of w

3.Problem Statement: Write a C Program to implement DFAs that recognize identifiers, constants, and operators of the mini language.

AIM: To Write a C Program to implement DFAs that recognize identifiers, constants, andoperators of the mini language.

ALGORITHM / PROCEDURE:

1 Start
2 Design the DFAs (M) to recognize Identifiers, Constants, and Operators
3 Read the input string **w** give it as input to the DFA M
4 DFA processes the input and outputs “Yes” if w b Є L(M), “No” otherwise
5 Display the output
6 End

Program:

```
#include  <stdio.h>
#include <string.h>
#include <stdlib.h>
// Returns '1' if the character is a DELIMITER.
int isDelimiter(char ch)
{
if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
ch == '[' || ch == ']' || ch == '{' || ch == '}')
return (1);
return (0);
}
// Returns '1' if the character is an OPERATOR.
int isOperator(char ch)
{
if (ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == '>' || ch == '<' ||
ch == '=')
return (1);
return (0);
}
// Returns '1' if the string is a VALID IDENTIFIER.
int validIdentifier(char* str)
{
if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
str[0] == '3' || str[0] == '4' || str[0] == '5' ||
str[0] == '6' || str[0] == '7' || str[0] == '8' ||
str[0] == '9' || isDelimiter(str[0]) == 1)
return (0);
```

```
return (1);
```

```
}
        // Returns '1' if the string is a KEYWORD.
        int isKeyword(char* str)
        {
        if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int")
        || !strcmp(str, "double") || !strcmp(str, "float")
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (1);
        return (0);
        }
        // Returns '1' if the string is an INTEGER.
        int isInteger(char* str)
        {
        int i, len = strlen(str);
        if (len == 0)
        return (0);
        for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
        && str[i] != '3' && str[i] != '4' && str[i] != '5'
        && str[i] != '6' && str[i] != '7' && str[i] != '8'
        && str[i] != '9' || (str[i] == '-' && i > 0))
        return (0);
        }
        return (1);
        }
        // Returns '1' if the string is a REAL NUMBER.
        int isRealNumber(char* str)
        {
        int i, len = strlen(str);
        int hasDecimal = 0;
        if (len == 0)
        return (0);
        for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
        && str[i] != '3' && str[i] != '4' && str[i] != '5'
        && str[i] != '6' && str[i] != '7' && str[i] != '8'
        && str[i] != '9' && str[i] != '.' ||
```

```
(str[i] == '-' && i > 0))
return (0);
if (str[i] == '.')
```

```
hasDecimal = 1;
}
return (hasDecimal);
}
// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
int i;
char* subStr = (char*)malloc(
sizeof(char) * (right - left + 2));
for (i = left; i <= right; i++)
subStr[i - left] = str[i];
subStr[right - left + 1] = '\0';
return (subStr);
}
// Parsing the input STRING.
void parse(char* str)
{
int left = 0, right = 0;
int len = strlen(str);
while (right <= len && left <= right) {
if (isDelimiter(str[right]) == 0)
right++;
if (isDelimiter(str[right]) == 1 && left == right) {
if (isOperator(str[right]) == 1)
printf("'%c' IS AN OPERATOR\n", str[right]);
right++;
left = right;
} else if (isDelimiter(str[right]) == 1 && left != right
|| (right == len && left != right)) {
char* subStr = subString(str, left, right - 1);
if (isKeyword(subStr) == 1)
printf("'%s' IS A KEYWORD\n", subStr);
else if (isInteger(subStr) == 1)
printf("'%s' IS AN INTEGER\n", subStr);
else if (isRealNumber(subStr) == 1)
printf("'%s' IS A REAL NUMBER\n", subStr);
else if (validIdentifier(subStr) == 1
&& isDelimiter(str[right - 1]) == 0)
printf("'%s' IS A VALID IDENTIFIER\n", subStr);
else if (validIdentifier(subStr) == 0
&& isDelimiter(str[right - 1]) == 0)
printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
left = right;
}
}
```

```
return;
}
// DRIVER FUNCTION
```

```
void main()
{
// maximum length of string is 100 herechar
str[100] = "int a = c + y; ";
//clrscr();
parse(str); // calling the parse function
}
```

OUTPUT:

```
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'c' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'y' IS A VALID IDENTIFIER
```

[Viva Questions]

1. What is an Interpreter?
2. What are the other language processors you know?
3. What is the difference between DFA / minimum DFA?
4. Write the difference between the interpreter and Compiler

EXERCISE:

3) Design an DFA to recognize the identifiers, keywords, constants, and comments of C language?
4) Write a C Program to implement the above DFA.

Exercise Program: Write a LEX program to count the number of vowels and consonants in a given string

4. Problem Statement: Design a Lexical analyzer. The lexical analyzer should ignore redundant s tabs and new lines. It should also ignore comments. Although the syntax specification s those identifiers can be arbitrarily long, you may restrict the length to some reasonable Value.

AIM: Write a C/C++ program to implement the design of a Lexical analyzer to recognize the tokens defined by the given grammar.

ALGORITHM / PROCEDURE:

We make use of the following two functions in the process.
look up() – it takes string as argument and checks its presence in the symbol table. If the string is found then returns the address else it returns NULL.
insert() – it takes string as its argument and the same is inserted into the symbol table and the corresponding address is returned.

1. Start
2. Declare an array of characters, an input file to store the input;
3. Read the character from the input file and put it into character type of variable, say 'c'.
4. If 'c' is blank then do nothing.
5. If 'c' is new line character line=line+1.
6. If 'c' is digit, set token Val, the value assigned for a digit and return the 'NUMBER'.
7. If 'c' is proper token then assign the token value.
8. Print the complete table with
   Token entered by the user,
   Associated token value.
9. Stop

PROGRAM / SOURCE CODE :

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
    if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str
    )==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("static",str)==0||strc
    mp("switch",str)==0||strcmp("case",str)==0)
        printf("\n%s is a keyword",str);
    else
        printf("\n%s is an identifier",str);
```

```
}
main()
{
	FILE *f1,*f2,*f3;
	char c, str[10], st1[10];
	int num[100], lineno=0, tokenvalue=0,i=0,j=0,k=0;
	printf("\n Enter the c program : ");/*gets(st1);*/
	f1=fopen("input","w");
	while((c=getchar())!=EOF)
	putc(c,f1);
	fclose(f1);
	f1=fopen("input","r");
	f2=fopen("identifier","w");
	f3=fopen("specialchar","w");
	while((c=getc(f1))!=EOF)
	{
		if(isdigit(c))
		{
			tokenvalue=c-'0';
			c=getc(f1);
			while(isdigit(c))
			{
				tokenvalue*=10+c-'0';
				c=getc(f1);
			}
			num[i++]=tokenvalue;
			ungetc(c,f1);
		}
		else
		if(isalpha(c))
		{
			putc(c,f2);
			c=getc(f1);
			while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
```

```
                {
                        putc(c,f2);
                        c=getc(f1);
                }
                putc(' ',f2);
                ungetc(c,f1);
        }
        else
        if(c==' '||c=='\t')
                printf(" ");
        else
        if(c=='\n')
                lineno++;
        else
                putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\n The no's in the program are :");
for(j=0; j<i; j++)
printf("%d", num[j]);
printf("\n");
f2=fopen("identifier", "r");
k=0;
printf("The keywords and identifiers are:");
while((c=getc(f2))!=EOF)
{
        if(c!=' ')
                str[k++]=c;
        else
        {
                str[k]='\0';
                keyword(str);
```

```
                    k=0;
                }
        }
        fclose(f2);
        f3=fopen("specialchar","r");
        printf("\n Special characters are : ");
        while((c=getc(f3))!=EOF)
                printf("%c",c);
        printf("\n");
        fclose(f3);
        printf("Total no. of lines are:%d", lineno);
}
```

Output :

Enter the C program: a+b*c

Ctrl-D

The no's in the program are:

The keywords and identifiers are:

a is an identifier and terminal

b is an identifier and terminal

c is an identifier and terminal

Special characters are:

+ *

Total no. of lines are: 1

[Viva Questions]

1. What is lexical analyzer?
2. Which compiler is used for lexical analysis?
3. What is the output of Lexical analyzer?
4. Which Finite state machines are used in lexical analyzer design?
5. What is the role of regular expressions, grammars in Lexical Analizer?

Exercise Program:Write a Lex program to find the Length of a String

5. Problem Statement: Implement the lexical analyzer using JLex, flex or other lexical Analyzer generating tools.

AIM: To Implement the lexical analyzer using JLex, flex or lex other lexical analyzer generating Tools.

ALGORITHM / PROCEDURE:

Input : LEX specification files for the token
Output : Produces the source code for the Lexical Analyzer with the name lex.yy.c and displays the tokens from an input file.

1. Start
2. Open a file in text editor

3. Create a Lex specifications file to accept keywords, identifiers, constants, operators and relational operators in the following format.
   a) %{
      Definition of constant /header files
      %}
   b) Regular Expressions
      %%
      Transition rules
      %%
   c) Auxiliary Procedure (main( ) function)

4. Save file with **.l** extension e.g. **mylex.l**
5. Call lex tool on the terminal e.g. [root@localhost]# lex mylex.l. This lex tool will convert ".l" file into ".c" language code file i.e., **lex.yy.c**
6. Compile the file lex.yy.c using C / C++ compiler. e.g. **gcc lex.yy.c**. After compilation the file lex.yy.c, the output file is in **a.out**
7. Run the file a.out giving an input(text/file) e.g. **./a.out**.
8. Upon processing, the sequence of tokens will be displayed as output.
9. Stop

LEX SPECDIFICATION PROGRAM / SOURCE CODE (lexprog.l) :

```
// ********* LEX Tool program to identify C Tokens *****************//
DIGIT          [0-9]
LETTER         [A-Z a-z]
DELIM          [ \t\n]
WS             {DELIM}+
ID             {(LETTER)[LETTER/DIGIT]}+
INTEGER        {DIGIT}+
%%
  {WS}         { printf("\n special characters\n"); }
  {ID}         { printf("\n Identifiers\n"); }
  {DIGIT}      {printf("\n Intgers\n"); }
  If           { printf("\n Keywords\n"); }
```

```
 else          { printf("\n keywords\n"); }
 ">"           { printf("\n Logical Operators\n"); }
 "<"           { printf("\n Logical Operators\n"); }
 "<="          { printf("\n Logical Operators\n"); }
 "=>"          { printf("\n Logical Operators\n"); }
 "="           { printf("\n Logical Operators\n"); }
 "!="          { printf("\n Logical Operators\n"); }
 "&&"          { printf("\n Relational \n"); }
 "||"          { printf("\n Relational  Operatos\n"); }
 "!"           { printf("\n Relational Operators\n"); }
 "+"           { printf("\n Arthmetic Operator\n"); }
 "-"           { printf("\n Arthmetic Operator\n"); }
 "*"           { printf("\n Arthmetic Operator\n"); }
 "/"           { printf("\n Arthmetic Operator\n"); }
 "%"           { printf("\n Arthmetic Operator\n"); }
%%
main()
{
  yylex();
}
```

OUTPUT :

```
[root@localhost]#  lex lexprog.l
[root@localhost]#  cc lex.yy.c
[root@localhost]#  ./a.out lexprog
```

TEST CASES:

INPUT	OUTPUT
If	Keyword
%	Arithmetic Operator
>=	Relational Operator
&&	Logical Operator

[Viva Questions]

1. What is are the functions of a Scanner?
2. What is Token?
3. What is lexeme, Pattern?
4. What is purpose of Lex?
5. What are the other tools used in Lexical Analysis?

Exercise Program: Write a Lex Program to accept string starting with vowel

6. Problem Statement : Design a Predictive Parser for the following grammar

G: { E-> TE' , E' -> +TE' | 0, T-> FT' , T'-> *FT'|0 , F-> (E) | id }

AIM: To write a 'C' Program to implement for the Predictive Parser (Non Recursive Descent parser) for the given grammar ,

Given the parse Table:

	id	**+**	*****	**(**	**)**	**$**
E	**E-> TE'**			**E-> TE'**		
E'		**E' -> +TE'**			**E'->0**	**E'->0**
T	**T-> FT'**			**T-> FT'**		
T'		**T'->0**	**T'-> *FT'**		**T'->0**	**T'->0**
F	**F-> id**			**F->(E)**		

ALGORITHM / PROCEDURE :

Input : string w$, Predictive Parsing table M
Output : A Left Most Derivation of the input string if it is valid , error otherwise.

Step1: Start
Step2: Declare a character array w[10] and Z as an array
Step3: Enter the string with $ at the end
Step4: if (A(w[z]) then increment z and check for (B(w[z])) and if satisfies increment z and check for 'd' if d is present then increment and check for (D(w[z]))
Step5: if step 4 is satisfied then the string is accepted
Else string is not
Step 6: Exit

SOURCE CODE :

```
// ***IMPLEMENTATION OF PREDICTIVE / NON-RECURSIVE DESCENT PARSING *****//
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
char ch;
#define id 0
#define CONST 1
#define mulop 2
#define addop 3
#define op 4
#define cp 5
```

```
#define err 6
#define col 7
#define  size 50
int token;
char lexbuff[size];
int lookahead=0;
int main()

{
  clrscr();
  printf(" Enter the string :");
  gets(lexbuff);
  parser();
  return 0;
}
parser()
{
 if(E())
 printf("valid string");
 else
 printf("invalid string");
 getch();
 return 0;
}
E()
{
 if(T())
 {
   if(EPRIME())
    return 1;
    else
    return 0;
   }
  else
  return 0;
}
T()
{
  if(F())
  {
    if(TPRIME())
     return 1;
     else
     return 0;
    }
  else
   return 0;
}
EPRIME()
{
  token=lexer();
```

```
  if(token==addop)
  {
   lookahead++;
   if(T())
   {
     if(EPRIME())
      return 1;
      else
      return 0;
   }
   else
   return 0;
  }

else
return 1;
}
TPRIME()
{
  token=lexer();
  if(token==mulop)
  {
    lookahead++;
    if(F())
    {
      if(TPRIME())
       return 1;
       else
       return 0;
      }
    else return 0;
  }
  else return 1;
}
F()
{
  token=lexer();
  if(token==id)
  return 1;
  else
  {
    if(token==4)
    {
      if(E())
      {
            if(token==5)
            return 1;
            else return 0;
      }
      else return 0;
    }
```

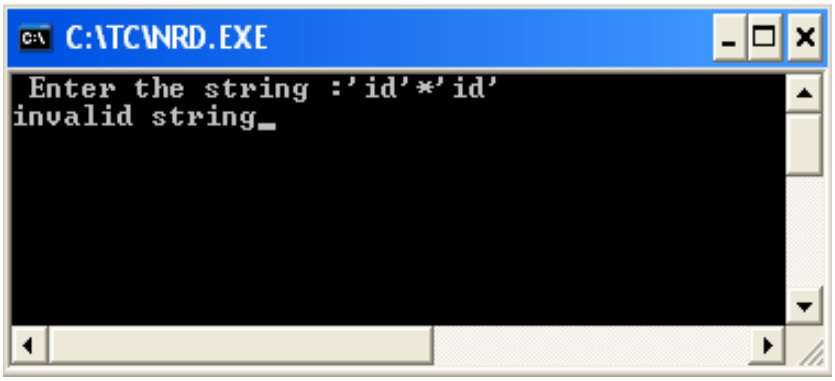
```
  else return 0;
}
}
lexer()
{
  if(lexbuff[lookahead]!='\n')
  {
    while(lexbuff[lookahead]=='\t')
    lookahead++;
    if(isalpha(lexbuff[lookahead]))
    {
      while(isalnum(lexbuff[lookahead]))
      lookahead++;
      return(id);
    }
    else
    {
       if(isdigit(lexbuff[lookahead]))
       {

while(isdigit(lexbuff[lookahead]))
            lookahead++;
            return CONST;
       }
     else
     {
       if(lexbuff[lookahead]=='+')
       {
           return(addop);
       }
       else
       {
           if(lexbuff[lookahead]=='*')
           {
             return(mulop);
           }
           else
           {
             if(lexbuff[lookahead]=='(')
             {
                lookahead++;
              return(op);
             }
             else
             {
               if(lexbuff[lookahead]==')')
               {
                 return(op);
               }
               else
                {


DEPARTMENT OF EMERGING TECHNOLOGIES
```

```
                return(err);
              }
            }
            }
            }
          }
      }
     }
     else
    return (col);
  }
```

OUTPUT :



Viva Questions:

1. **What is a parser and state the Role of it?**
2. **Types of parsers? Examples to each**
3. **What are the Tools available for implementation?**
4. **How do you calculate FIRST(),FOLLOW() sets used in Parsing Table construction?**

Exercise Program: **Write a lex program can be used for any string which is consisting of small letters as well as capital letter...**

7. Problem Statement: Design a LALR Bottom Up Parser for the given grammar

AIM: To Design and implement an LALR bottom up Parser for checking the syntax of the Statements in the given language.

ALGORITHM/PROCEDURE/CODE:

LALR Bottom Up Parser

```
<parser.l>
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.dval=atof(yytext);
return DIGIT;
}
\n|. return yytext[0];
%%
<parser.y>
%{
/*This YACC specification file generates the LALR parser for the program
considered in experiment 4.*/
#include<stdio.h>
%}
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
;
printf("%g\n",$1);
}
expr: expr '+' term {$$=$1 + $3 ;}
| term
;
term: term '*' factor {$$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' {$$=$2 ;}
| DIGIT
;
%%
int main()
```

```
{
yyparse();
}
yyerror(char *s)
{

printf("%s",s);
}
```

Output:

```
$lex parser.l
$yacc –d parser.y
$cc lex.yy.c  y.tab.c –ll –lm
$./a.out
2+3
5.0000
```

Viva Questions?

1. What is yacc? Are there any other tools available for parser generation?
2. How do you use it?
3. Structure of parser specification program
4. How many ways we can generate the Parser

8. Problem statement: Convert The BNF rules into YACC form and write code to generate abstract syntax tree.

AIM: To Implement the process of conversion from BNF rules to Yacc form and generate Abstract Syntax Tree.

ALGORITHM/PROCEDURE

<int.l>

```
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
% }
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
    return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
```

<**int.y**>

```
%{
#include<string.h>
#include<stdio.h>
struct quad
{
    char op[5];
    char arg1[10];
    char arg2[10];
    char result[10];
}QUAD[30];
```

```
struct stack
{
    int items[100];
    int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;

extern int LineNo;
%}
%union
{
    char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR :    EXPR '+' EXPR { AddQuadruple("+",$1,$3,$$);}
 |        EXPR '-' EXPR  { AddQuadruple("-",$1,$3,$$);}
 |        EXPR '*' EXPR { AddQuadruple("*",$1,$3,$$);}
 |        EXPR '/' EXPR  { AddQuadruple("/",$1,$3,$$);}
 |         '-' EXPR          { AddQuadruple("UMIN",$2,"",$$);}
 |        '(' EXPR ')'       { strcpy($$,$2); }
```

```
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {

strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
```

```
WHILELOOP: WHILE '(' CONDITION ')'
 {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;

%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------------------------""\n\t\t Pos Operator Arg1 Arg2 Result" "\n\t\t
--------------------");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t %s\t
%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t ------------------------");
printf("\n\n");
return 0;
}
void push(int data)
{
stk.top++;
if(stk.top==100)
```

```
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{
int data;
if(stk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

Input:

```
$vi test.c
main()
{
int a,b,c;
if(a<b)
{
a=a+b;
}
while(a<b)
{
a=a+b;
}
if(a<=b)
{
c=a-b;
}
else
{
c=a+b;
}
}
```

Output:

$ lex int.l

$ yacc –d int.y

$ gcc lex.yy.c y.tab.c –ll –lm

$ ./a.out test.c

OUTPUT:

Viva Questions:

1. What is abstract syntax tree?
2. What is quadruple?
3. What is the difference between Triples and Indirect Triple?
4. State different forms of Three address statements.
5. What are different intermediate code forms?

9. Problem Statement: A program to generate machine code from the abstract syntax tree generated by the parser.

Aim: To write a C Program to Generate Machine Code from the Abstract Syntax Tree using the specified machine instruction formats.

ALGORITHM / PROCEDURE / SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int label[20];
int no=0;
int main()
{
FILE *fp1,*fp2;
char fname[10],op[10],ch;
char operand1[8],operand2[8],result[8];
int i=0,j=0;
printf("\n Enter filename of the intermediate code");
scanf("%s",&fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL || fp2==NULL)
{
printf("\n Error opening the file");
exit(0);
}
while(!feof(fp1))
{
fprintf(fp2,"\n");
fscanf(fp1,"%s",op);
i++;
if(check_label(i))
fprintf(fp2,"\nlabel#%d",i);
if(strcmp(op,"print")==0)
{
fscanf(fp1,"%s",result);
fprintf(fp2,"\n\t OUT %s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s %s",operand1,operand2);
fprintf(fp2,"\n\t JMP %s,label#%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]=")==0)
{
fscanf(fp1,"%s %s %s",operand1,operand2,result);
```

```
fprintf(fp2,"\n\t STORE %s[%s],%s",operand1,operand2,result);
}
if(strcmp(op,"uminus")==0)
{
fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t LOAD -%s,R1",operand1);
fprintf(fp2,"\n\t STORE R1,%s",result);
}
switch(op[0])
{
case '*': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t MUL R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '+': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t ADD R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '-': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t SUB R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '/': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t DIV R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '%': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t DIV R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '=': fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t STORE %s %s",operand1,result);
break;
case '>': j++;
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t JGT %s,label#%s",operand2,result);
label[no++]=atoi(result);
break;
case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1); fprintf(fp2,"\n\t
```

```
JLT %s,label#%d",operand2,result);
label[no++]=atoi(result);
break;
}
}
fclose(fp2); fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening the file\n");
exit(0);
}
do
{
ch=fgetc(fp2);
printf("%c",ch);
}while(ch!=EOF);
fclose(fp1);
return 0;
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
return 0;
}
```

Input :

```
$ vi int.txt
= t1 2
[]= a 0 1
[]=a 1 2
[]=a 2 3
*t1 6 t2
+a[2] t2 t3
-a[2] t1 t2
/t3 t2 t2
uminus t2 t2
print t2
goto t2 t3
=t3 99
uminus 25 t2
*t2 t3 t3
uminus t1 t1
+t1 t3 t4
print t4
```

Output :

Enter filename of the intermediate code: int.txt

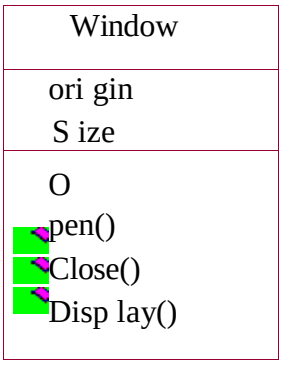
```
STORE  t1, 2
STORE  a[0], 1
STORE  a[1], 2
STORE  a[2], 3
LOAD  t1, R0
LOAD  6, R1
ADD  R1, R0
STORE  R0, t3
LOAD  a[2], R0
LOAD t2, R1
ADD  R1,R0
STORE  R0,t3
LOAD  a[t2],R0
LOAD  t1,R1
SUB  R1,R0
STORE  R0,t2
LOAD  t3,R0
LOAD t2,R1
DIV  R1,R0
STORE  R0,t2
LOAD  t2,R1
STORE  R1,t2
LOAD  t2,R0
JGT  5,label#11
Label#11: OUT t2
JMP  t2,label#13
Label#13: STORE t3,99
LOAD  25,R1
STORE  R1,t2
LOAD  t2,R0
LOAD  t3,R1
MUL  R1,R0
STORE  R0,t3
LOAD  t1,R1
STORE  R1,t1
LOAD  t1,R0
LOAD  t3,R1
ADD  R1,R0
STORE  R0,t4
OUT  t4
```

CASE TOOLS LAB MANUAL

DEPARTMENT OF EMERGING TECHNOLOGIES

CLASS: A class is a description of a set of objects that shares the common attributes, operations,relationships, and semantics. A class implements one or more interfaces.

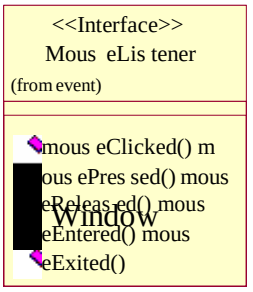
Graphically, a class is represented as a rectangle, usually including its name, attributes andoperations, as shown below.



Interface:

An interface is a collection of operations that specify a service of a class or component. An interface describes the externally visible behavior of that element.

Graphically the interface is rendered as a circle together with its name.



Collaboration:

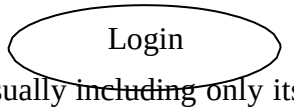
Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that‟s bigger than the sum of all the elements.

Graphically, collaboration is rendered as an ellipse with dashed lines, usually including only itsname as shown below.

Chain of Responsibility

UseCase:

Use case is a description of a set of sequence of actions performed by a system for a specific goal for the system.

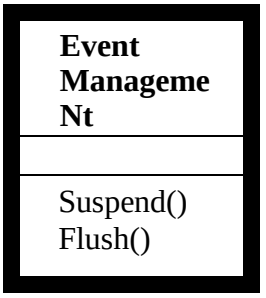


Graphically, Use Case is rendered as an ellipse with dashed lines, usually including only its name as shown below.

ActiveClass:

An active class is a class whose objects own one or more processes or threads and therefore can initiate control activity.

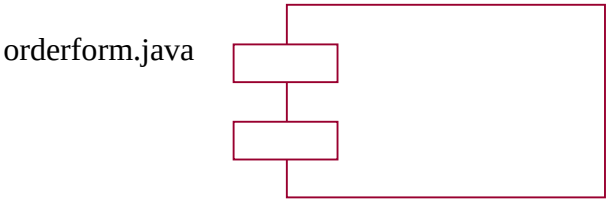
Graphically, an active class is rendered just like a class, but with heavy lines usually includingits name, attributes and operations as shown below.



Component:

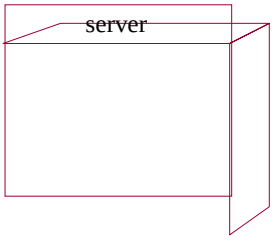
Component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

Graphically, a component is rendered as a rectangle with tabs, usually including only its name,as shown below.



Node: A Node is a physical element that exists at run time and represents a computational resource,generally having at least some memory and often, processing capability.

Graphically, a node is rendered as a cube, usually including only its name, as shown below.



1) BEHAVIORAL THINGS:

Behavioral things are the dynamic parts of UML models.

These are the verbs of a model, representing behavior over time and space.

1) Interaction:

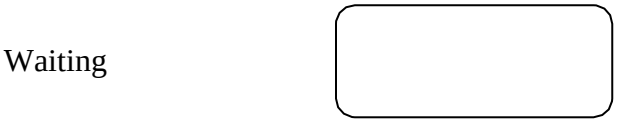
An interaction is a behavior that consists of a set of messages exchanged among a set of objects(elements) within a particular context to accomplish a specific task.

Graphically, a message is rendered as a direct line, almost always including the name if its operation, as shown below.

2) State Machine:

A state machine is a behavior that specifies the sequence of states of an object in its life cycle.It defines the sequence of states an object goes through in response to events.

Graphically, a state is rendered as a rounded rectangle usually including its name and its sub-states, if any, as shown below.



3) GROUPING THINGS:

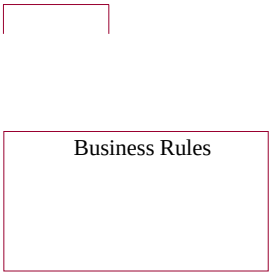
Grouping things are the organizational parts of the UML models. These are the boxes into which a model can be decomposed.

There is one primary kind of grouping thing with “package”.

Package:

A package is a general-purpose mechanism for organizing elements into groups.

Package is the only one grouping thing available for gathering structural and behavioral things.



Package

4) ANNOTATIONAL THINGS:

Annotational things are the explanatory parts of the UML models.

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.

Note:

note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

Graphically a note is represented as a rectangle with dog-eared corner together, with a textualor graphical comment, as shown below.

Note:

RELATIONSHIPSINTHEUML:

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application. There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

Dependency

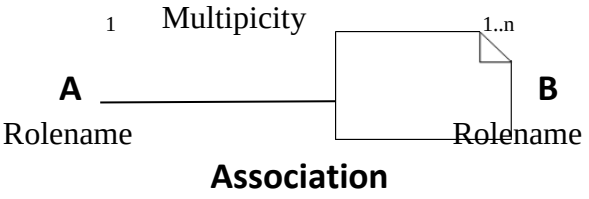
Dependency is a relationship between two things in which change in one element also affects the other one.

Dependency

Ex:

Association:

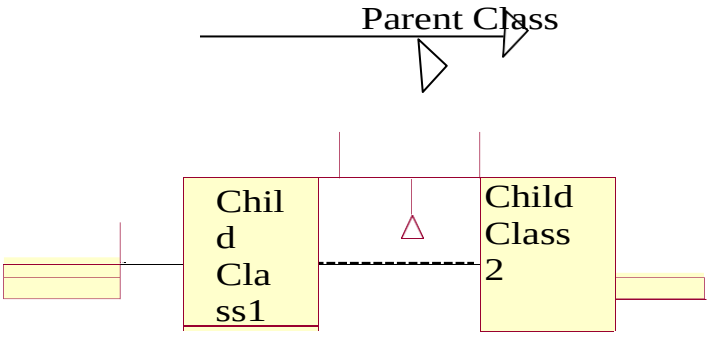
Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.



Generalization:

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.

Ex:



Realization:

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.

Ex:

DIAGRAMSINUML:

All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process.

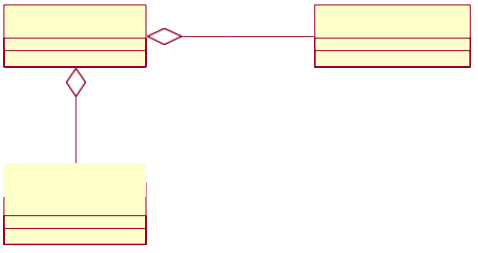
Each UML diagram is designed to let developers and customers view a software system from adifferent perspective and in varying degrees of abstraction.

UML diagrams are the ultimate output of the entire system.

A diagram is the graphical presentation of a set of elements ,most often rendered as aconnected graph of vertices(things) arcs (relationships).

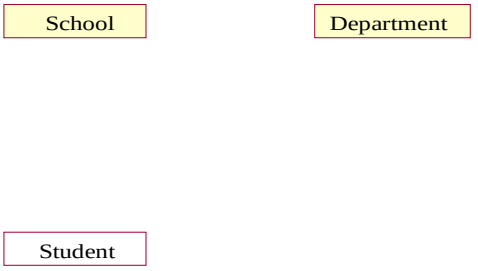
UML includes the following nine diagrams:

1) Class diagram
2) Object diagram
3) Use case diagram
4) Sequence diagram
5) Collaboration diagram
6) Activity diagram
7) State chart diagram
8) Deployment diagram
9) Component diagram



1. ClassDiagram

Class diagram is a diagram that shows a set of classes, interfaces, and collaborationsand their relationships. Class diagrams address the static design view or the static process view of the system. Graphically it is represented as follows:-



2. ObjectDiagram

Object diagram shows a set of objects and their relationships. These diagram the static design view or static process view of a system.

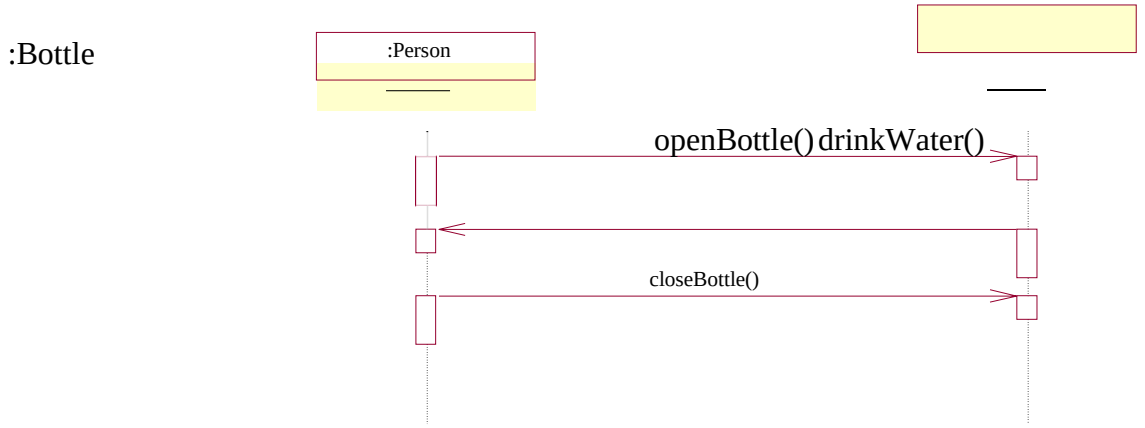
3. UsecaseDiagram

Use Case diagram shows a set of use cases and actors (a special kind of class) and their relationships.

These diagrams address the static use case view of a system. Graphically it is represented as follows:-

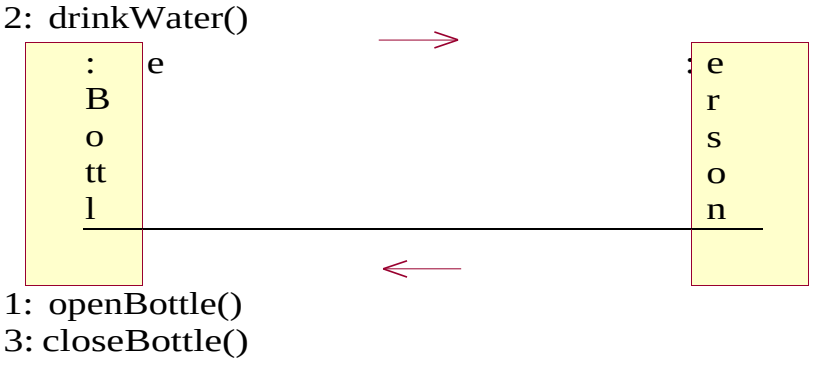
1. SequenceDiagram

Sequence diagram are interaction diagrams. This diagram emphasizes the time- ordering of messages. These diagrams address the dynamic view of a system. SequenceDiagram displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).Graphically it is represented as follows:-



5 Collaboration Diagram

Collaboration diagram are also interaction diagrams. These diagrams emphasizes the structural organization of the objects that send and receive messages. These diagrams address the dynamicview of a system. Collaboration Diagram displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.Graphically it is represented as follows:-



6. Statechart Diagram

State chart diagram shows a state machine, consisting of states, transitions, events and activities. These diagrams address the dynamic view of the system. State Chart diagram displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli,

together with its responses and actions.

7 .Activity Diagram

Activity diagram is a special kind of a state chart diagram that shows the flow from activity to activity within a system. These diagrams address dynamic view of a system. Activity Diagram displays a special

state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. Graphically it is represented as follows:-

8. ComponentDiagram

Component diagram shows the organizations and dependencies among a set of components. These diagrams address the static implementation of view of a system. Component Diagram displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more

than one time.Graphically it is represented as follows:

fraudagent.exe

fraudagent.dll

9. Deployment Diagram

Deployment diagram shows the configuration of run-time processing nodes and the components that live on them. These diagrams address the static deployment view of architecture. Deployment Diagram displays the configuration of run-time processing elements and the software components, processes,and objects that live on them. Software component instances represent run-time manifestations of code.

Graphically it is represented as follows:-

Admin Client
admin.exe

<<10-T Ethernet>>

Automatic Teller Machine (ATM)

Description of ATM System

The software to be designed will control a simulated automated teller machine (ATM) having a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash, a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank"s computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.)

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) – both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned – except as noted below.

The ATM must be able to provide the following services to the customer:

1. A customer must be able to make a cash withdrawal from any suitable account linked to the card. Approval must be obtained from the bank before cash is dispensed.
2. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.
3. A customer must be able to make a transfer of money between anytwo accounts linked to the card.
4. A customer must be able to make a balance inquiry of any account linked to thecard.
5. A customer must be able to abort a transaction in progress bypressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it was allowed

by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. (If the customer fails to deposit the envelope within the timeout period, or presses cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.)

If the bank determines that the customer‟s PIN is invalid, the customer will be required to re-enterthePIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have tocontactthe bank to get it back.

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation ofthe problem, and will then ask the customer whether he/she wants to do another transaction.

The ATM will provide the customer with a printed receipt for each successful transaction, showing thedate, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account (“to” account for transfers).

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing ofcustomers. After turning the switch to the “on” position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the “off” position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

Name of the experiment: Class diagram for ATM System

1. **AIM**: To design and implement ATM system through Class Diagram

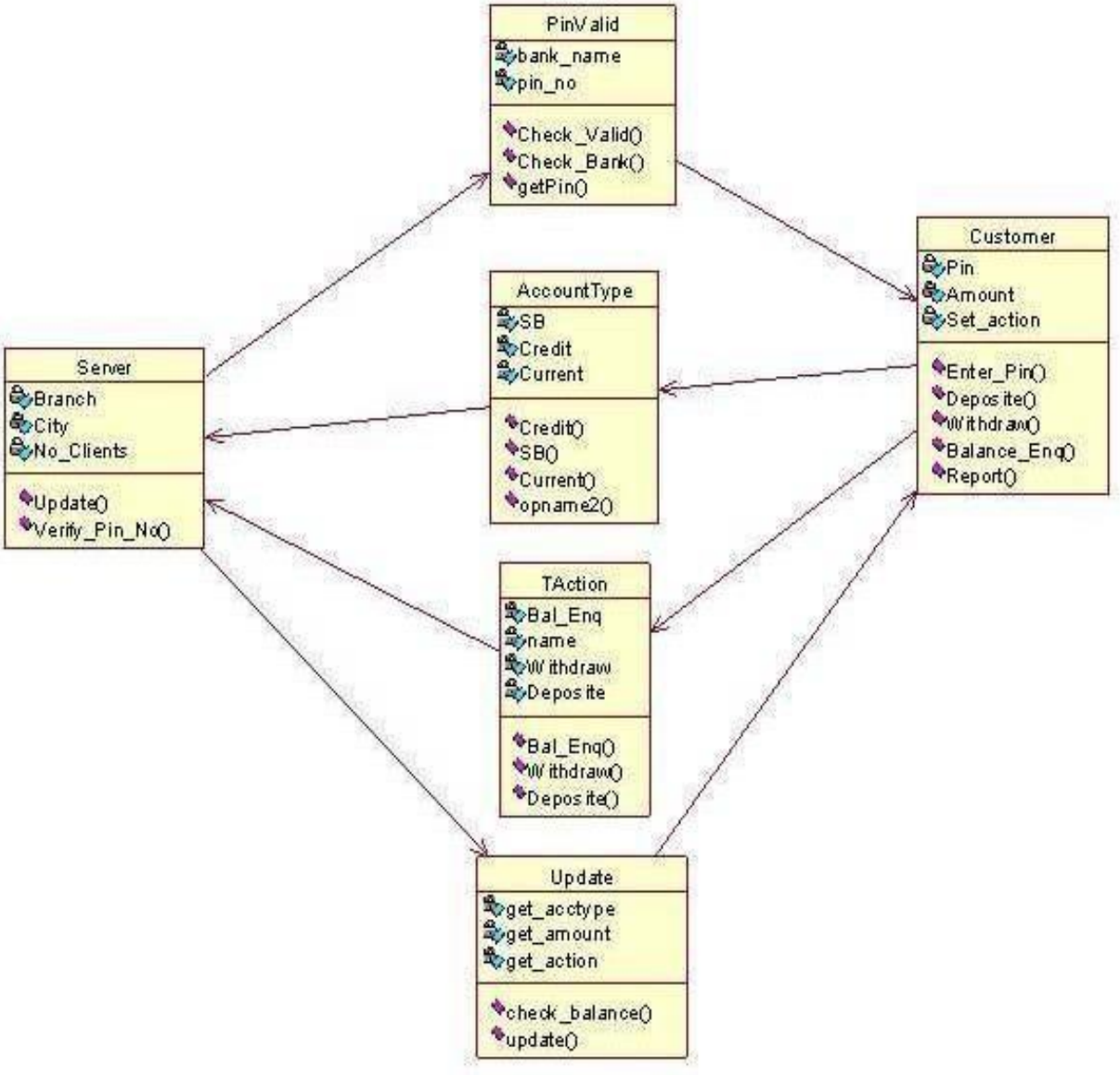
Procedure:-

Step1: First Classes are created.

Step2: Named as PinValid, Account Type, Transaction, Update, Server, Customer classes are created.

Step3: Appropriate relationships are provided between them as association.

DIAGRAM:



Inferences:

1. understand the concept of classes
2. identify classes and attributes and operations for a class
3. model the class diagram for the system

Applications:

Online transactionOnline banking

A) NAME OF EXPERIMENT: Use case diagram for ATMSystem.

AIM: To design and implement ATM System through Use case Diagram.

Purpose:

The purpose of use case diagram is to capture the dynamic aspect of a system. Because other four diagrams (activity, sequence, collaboration and State chart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams. Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

Procedure:

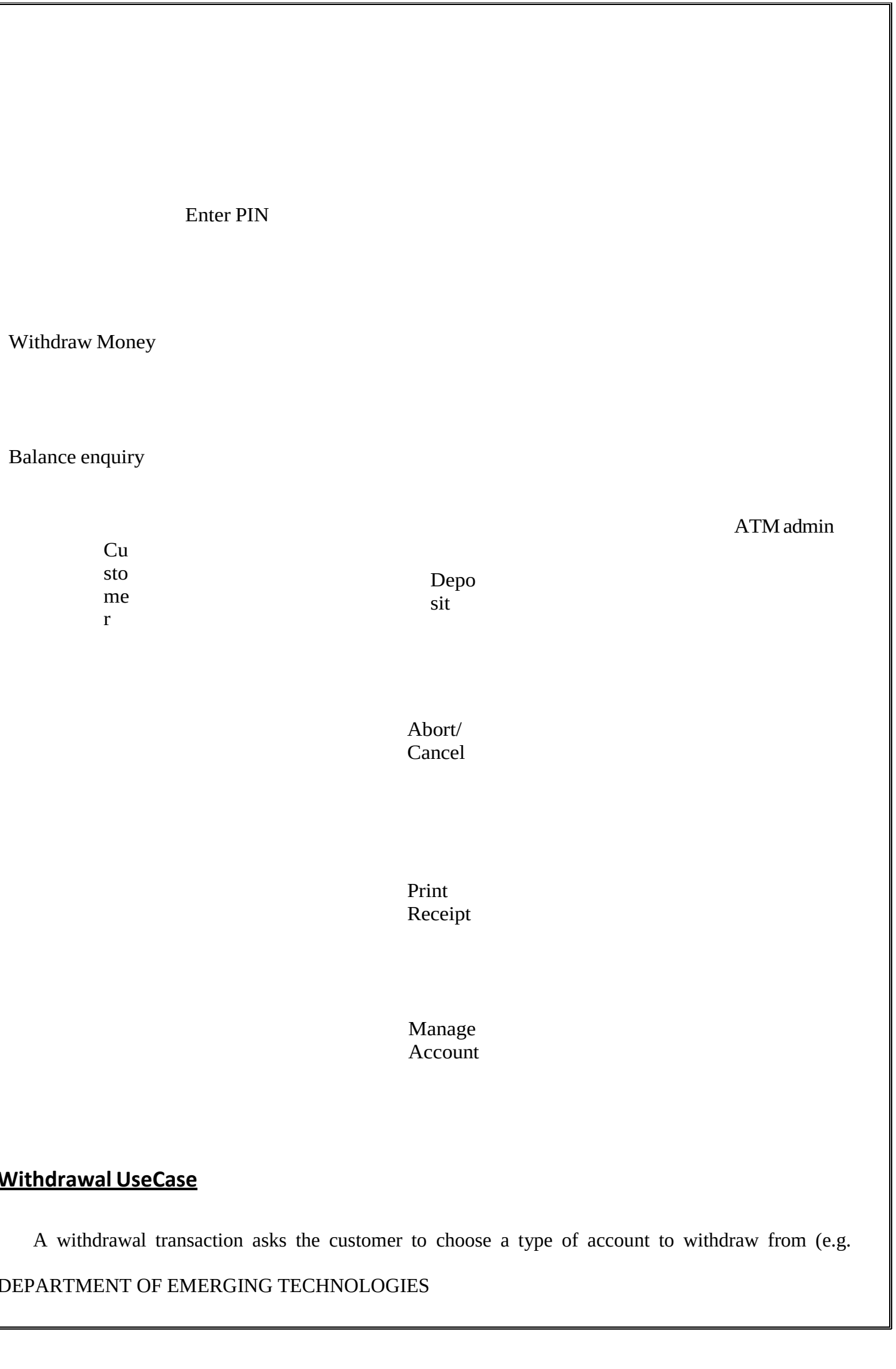
Step1: First an Actor is Created and named as User/Customer.

Step2: Secondly a system is created for ATM.

Step3: A use case Enter PIN, Withdraw money is created and connected with user as association relationship.

Step4: Similarly various use cases like Deposit money, Balance Enquiry, Manage Account etc are created and appropriate relationships are associated with each of them.

USECASE DIAGRAM:



Withdrawal UseCase

A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g.

checking) from a menu of possible accounts, and to choose an amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before

sendingthe transaction to the bank. (If not, the customer is informed and asked to enter a different amount.) If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the amount.

DepositUseCase

A deposit transaction asks the customer to choose a type of account to deposit to (e.g. checking) from a menu of possible accounts, and to type in amount on the keyboard. The transaction is initially sent tothe bank to verify that the ATM can accept a deposit from this customer to this account. If the

transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks before it issues a receipt. Once the envelope has been received, a second message is sent to the bank, to confirm that the bank can credit the customer‟s account – contingent on manual verification of the deposit envelope contents by an operator later.

A deposit transaction can be cancelled by the customer pressing the Cancel key any time prior to inserting the envelope containing the deposit. The transaction is automatically cancelled if the customer fails to insert the envelope containing the deposit within a reasonable period of time after being asked to do so.

InquiryUseCase

An inquiry transaction asks the customer to choose a type of account to inquire about from a menu of possible accounts. No further action is required once the transaction is approved by the bank before printing the receipt. An inquiry transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the account to inquire about.

ValidateUserUsecase:

This use case is for validate the user i.e. check the pin number, when the bank reports that the customer‟s transaction is disapproved due to an invalid PIN. The customer is required to re-enter the PIN and the original request is sent to the bank again. If the bank now approves the transaction, or disapproves it for some other reason, the original use case is continued; otherwise the process of re-entering the PIN is repeated. Once the PIN is successfully re-entered

If the customer fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed informing the customer of this and suggesting he/she contact the bank, and the entire customer session is aborted.

PrintBillusecase

This usecase is for printing corresponding bill after transactions (withdraw or deposit, or balance enquiry, transfer) are completed.

ManageAccount

This use case is for updating corresponding user accounts after transactions (withdraw or deposit ortransfer) are completed. DEPARTMENT OF EMERGING TECHNOLOGIES

RESULT:

Inferences:
1. Identification of use cases.
2. Identification of actors.

INTERACTIONDIAGRAMS

We have two types of interaction diagrams in UML. One is sequence diagram and the other isa collaboration diagram. The sequence diagram captures the time sequence of message flow from oneobject to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

So the following things are to be identified clearly before drawing the interaction diagram:

1. Objects taking part in the interaction.
2. Message flows among the objects.
3. The sequence in which the messages are flowing.
4. Object organization.

Purpose:

1. To capture dynamic behavior of a system.
2. To describe the message flow in the system.
3. To describe structural organization of the objects.
4. To describe interaction among objects.

Contents of a Sequence Diagram

Objects
Focus of controlMessages
Life line

Contents of a Collaboration Diagram

Objects Links Messages

B) NAME OF EXPERIMENT: Sequence diagram for ATMSystem.

AIM: To design and implement ATM System through Sequence Diagram.

Procedure:-

Step1: First An actor is created and named as user.

Step2: Secondly an object is created for Atm.

Step3: Timelines and lifelines are created automatically for them.

Step4: In sequence diagram interaction is done through time ordering of messages. So appropriate messages are passed between user and ATM is as shown in the figure.

DIAGRAM:

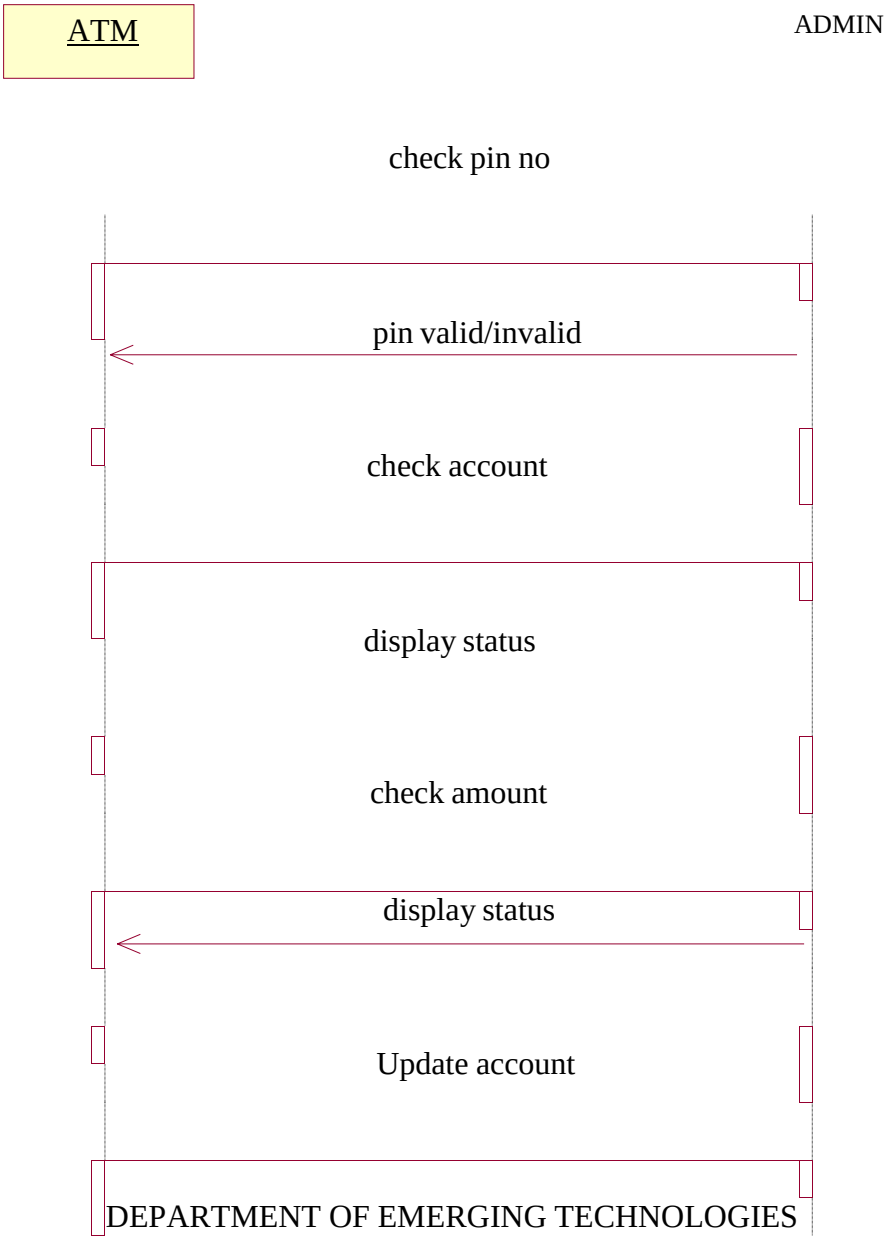
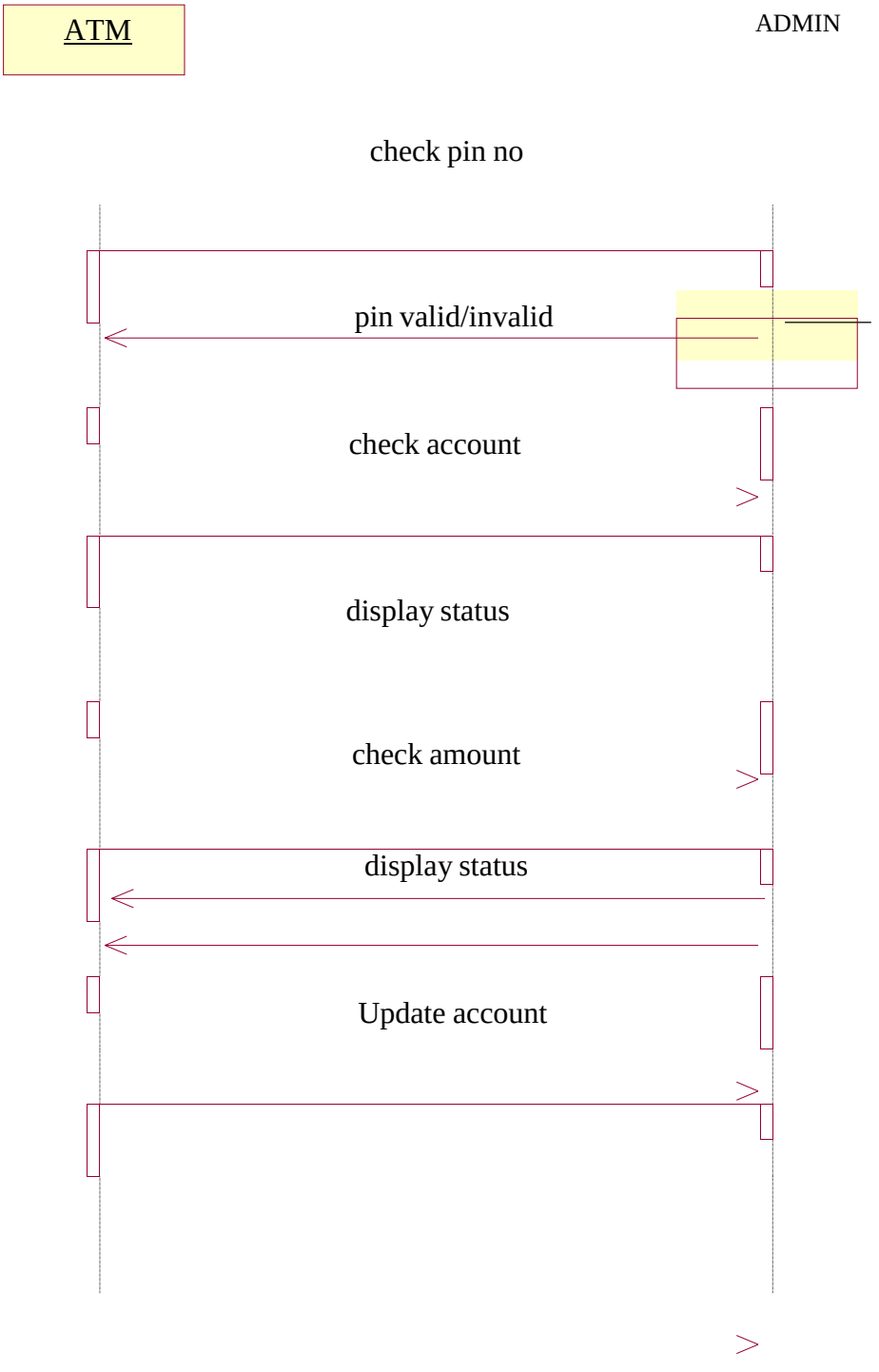


DIAGRAM:


ATM
ADMIN
check pin no
pin valid/invalid
check account
display status
check amount
display status
Update account

C) NAME OF EXPERIMENT: collaboration for ATMSystem.

AIM: To design and implement ATM System through Collaboration diagram.

Procedure:-

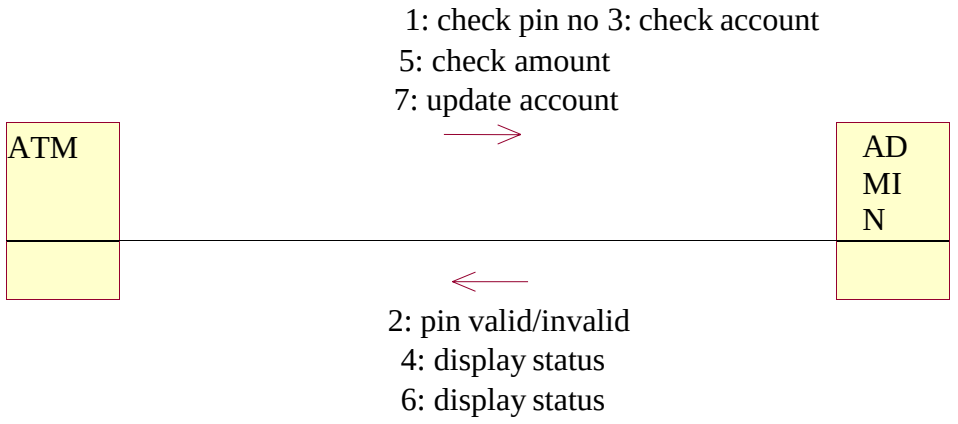
Step1: First an actor is created and named as user.

Step2: Secondly an object is created for ATM.

Step3: In collaboration diagram interaction is done through organization.

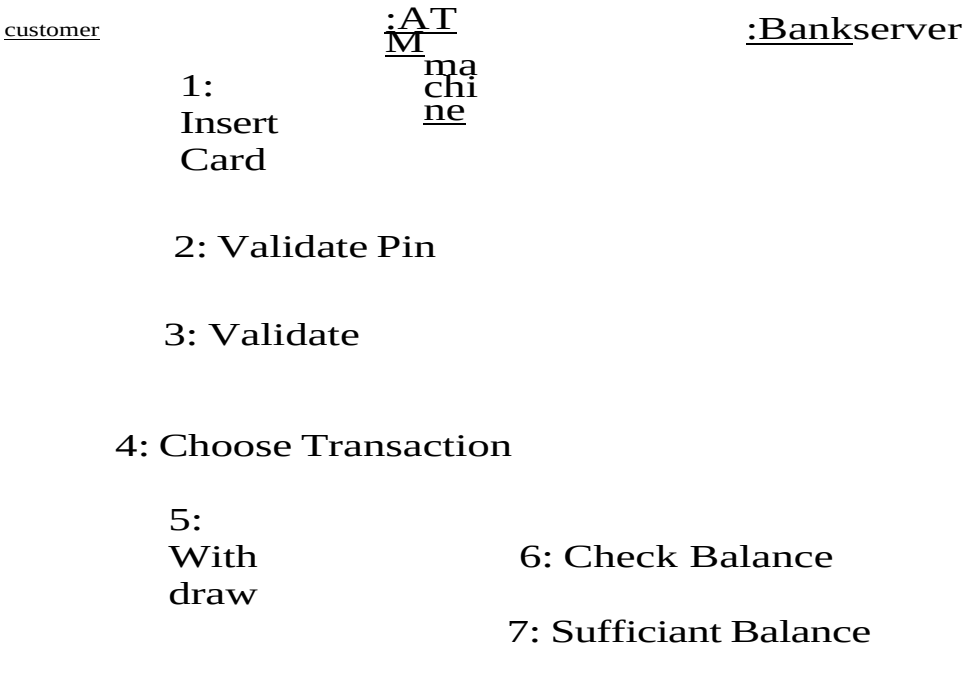
Step4: So appropriate messages are passed between user and ATM as shown in the figure.

DIAGRAM:

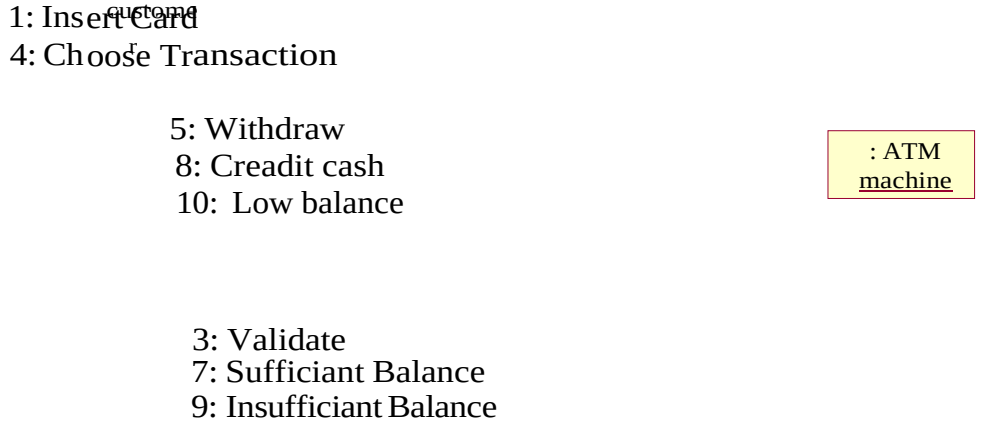


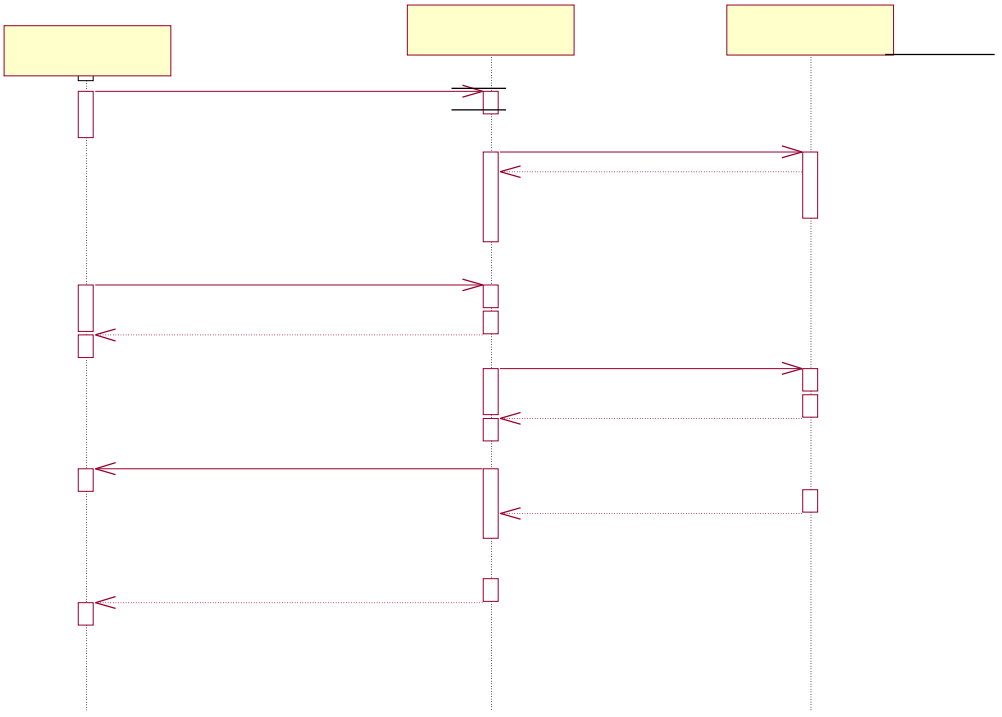
WITHDRAW UseCase:

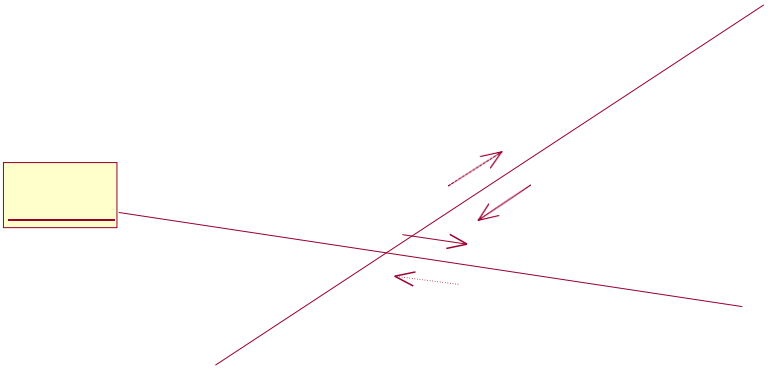
SEQUENCE DIAGRAM



COLLABORATION DIAGRAM





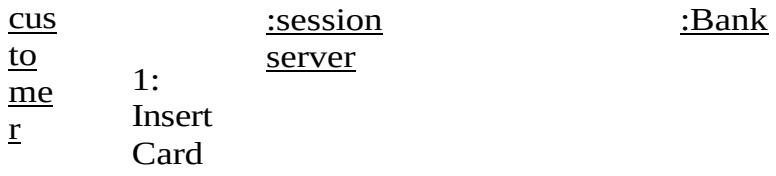


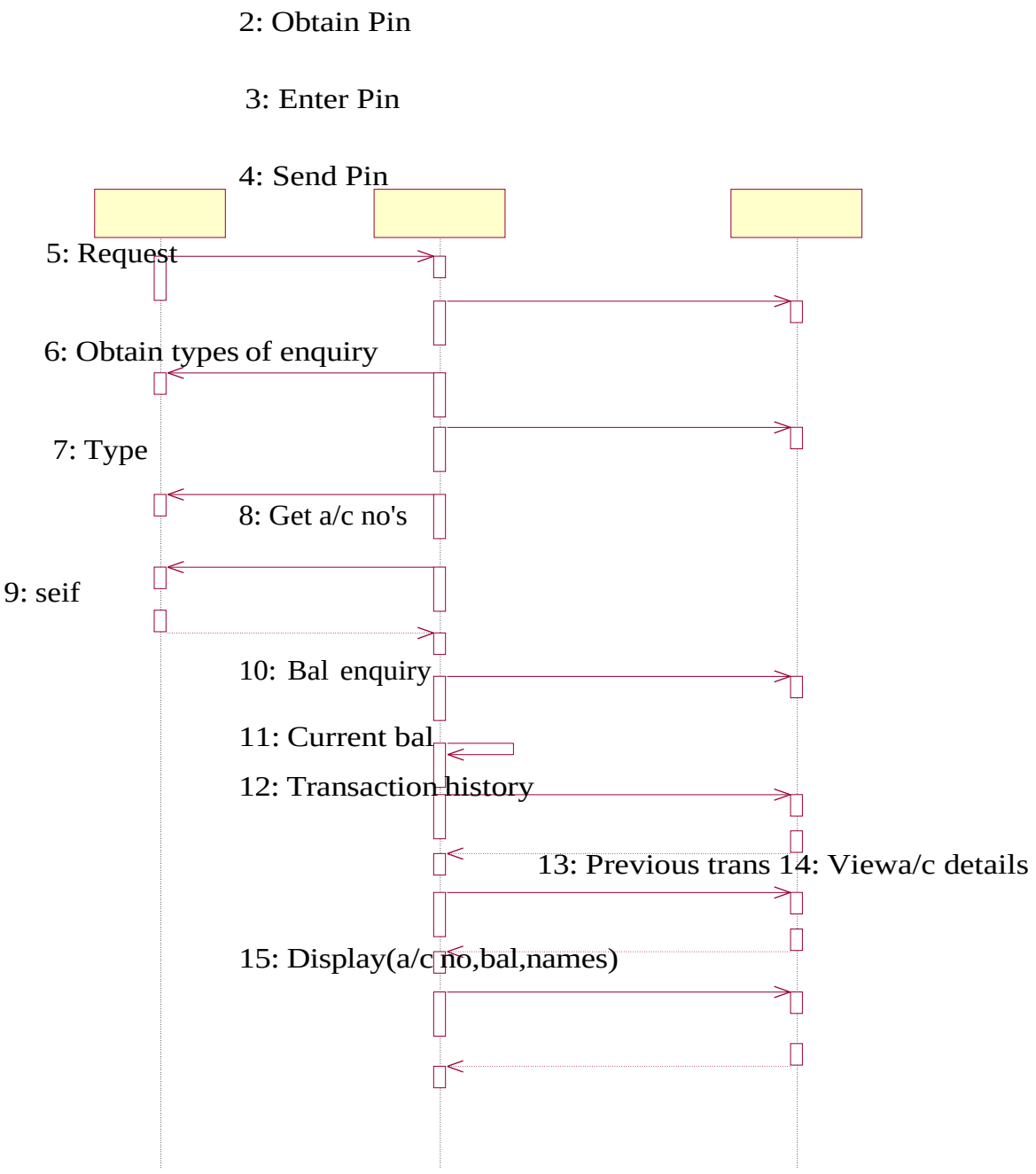
2: Validate Pin
6: Check Balance

: Bank server

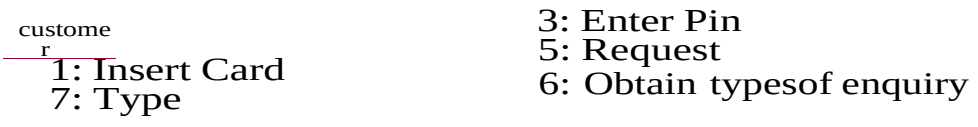
ENQUIRY UseCase:

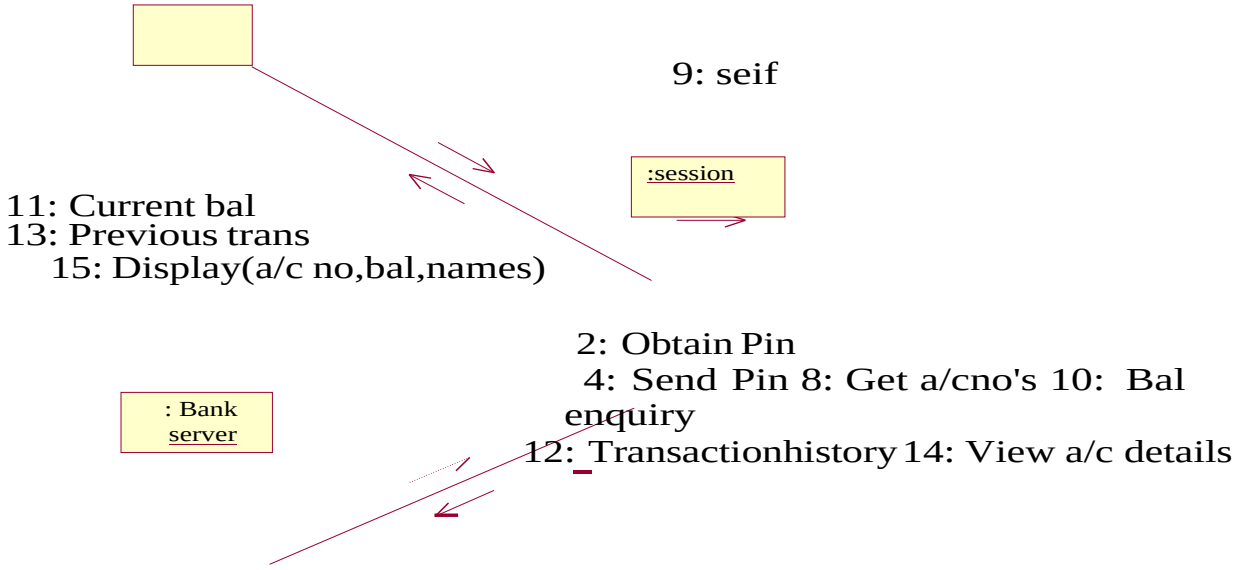
SEQUENCE DIAGRAM:





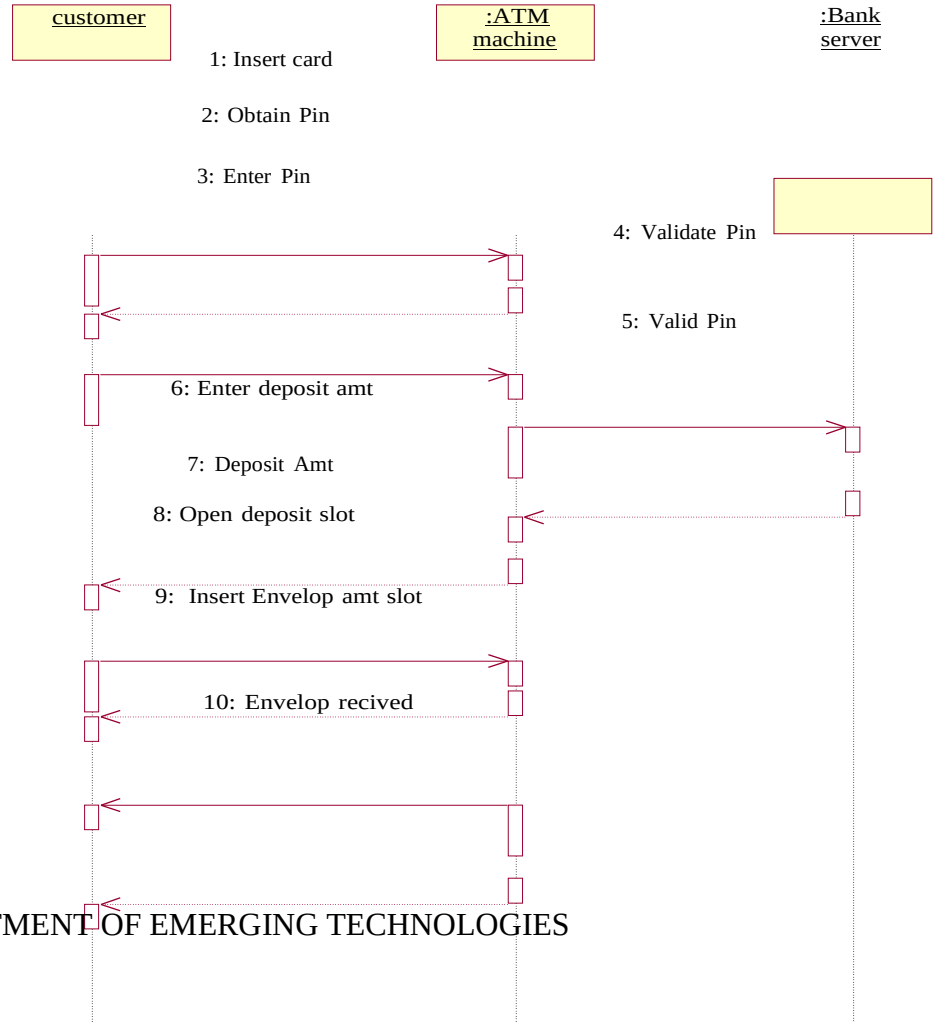
COLLABARATION DIAGRAM:



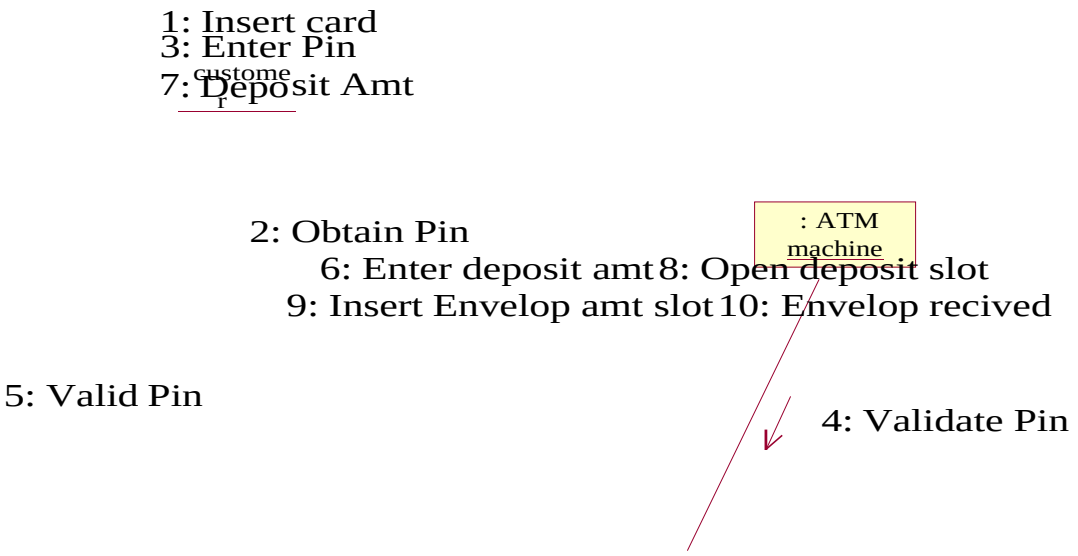


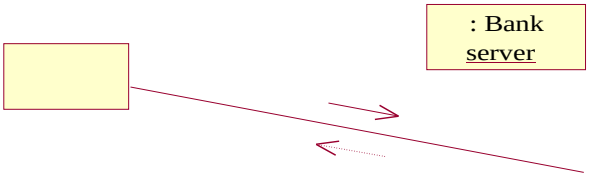
DEPOSIT UseCase:

SEQUENCE DIAGRAM:



COLLABARATION DIAGRAM:

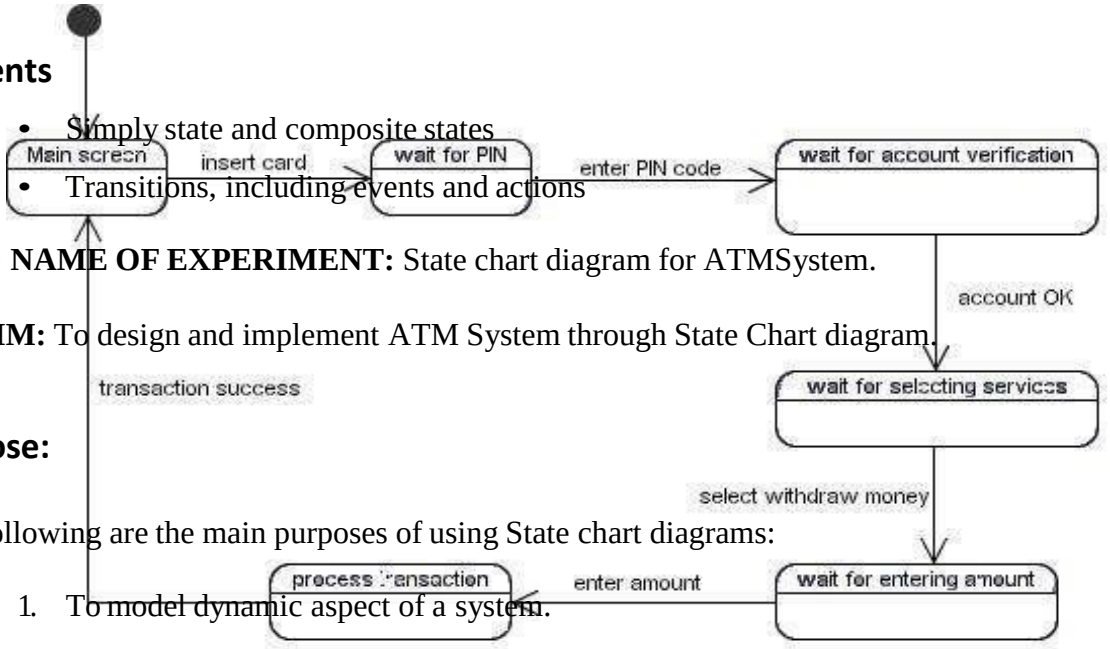




STATECHARTDiagram

State Chart diagram is used to model dynamic nature of a system. They define different states of an object during its lifetime. And these states are changed by events. State chart diagram describes the flow of control from one state to another state.

States are defined as a condition in which an object exists and it changes when some event is triggered. But the main purpose is to model reactive system.



Contents

- Simply state and composite states
- Transitions, including events and actions

D) NAME OF EXPERIMENT: State chart diagram for ATMSystem.

AIM: To design and implement ATM System through State Chart diagram.

Purpose:

Following are the main purposes of using State chart diagrams:

1. To model dynamic aspect of a system.
2. To model life time of a reactive system.
3. To describe different states of an object during its life time.
4. Defines a state machine to model states of an object.

Procedure:-

Step1: First after initial state control undergoes transition to ATM screen.

Step2: After inserting card it goes to the state wait for pin.

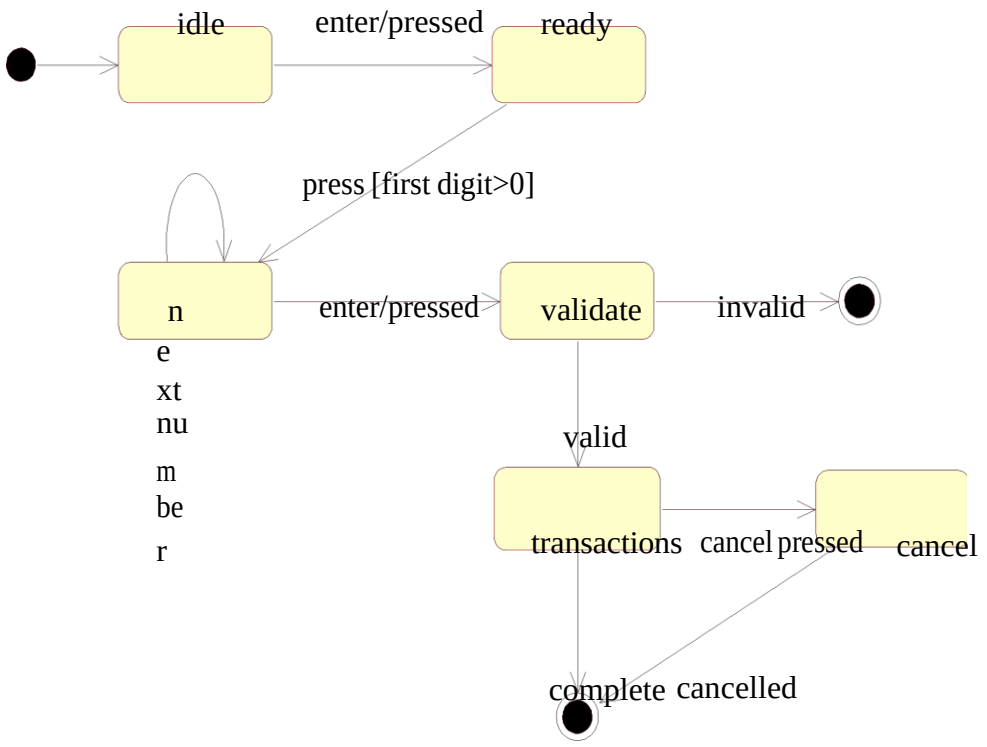
Step3: After entering pin it goes to the state account verification.

Step4:. In this way it undergoes transitions to various states and finally reaches the ATM screen state

as shown in the fig.

DIAGRAM:

STATE CHART FOR ATM



E) NAME OF EXPERIMENT: Activity diagram for ATM System.

AIM: To design and implement ATM System through Activity Diagram**.**

THEORY: An activity diagram shows the flow from activity to activity .An activity is an ongoing nonatomic execution within a state machine .Activities ultimately results in some action, which is made up of executable atomic computations. We can use these diagrams to model the dynamic aspects of a system.

Activity diagram is basically a flow chart to represent the flow form one activity to another . The activity can be described as an operation of the system. So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow by using elements like fork, join etc.

Contents

Initial/Final State, Activity, Fork & Join, Branch, Swim lanes

Fork

A fork represents the splitting of a single flow of control into two or more concurrent Flow of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below fork the activities associated with each of these pathcontinues in parallel.

Join

A join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transition and one outgoing transition. Above the join the activities associatedwith each of these paths continues in parallel.

Branching

A branch specifies alternate paths takes based on some Boolean expression Branch is represented by diamond Branch may have one incoming transition and two or more outgoing one on each outgoing transition, you place a Boolean expression shouldn‟t overlap but they should cover all possibilities.

Swimlane:

Swimlanes are useful when we model workflows of business processes to partition the activity states on an activity diagram into groups. Each group representing the business organization responsible for those activities, these groups are called Swimlanes .

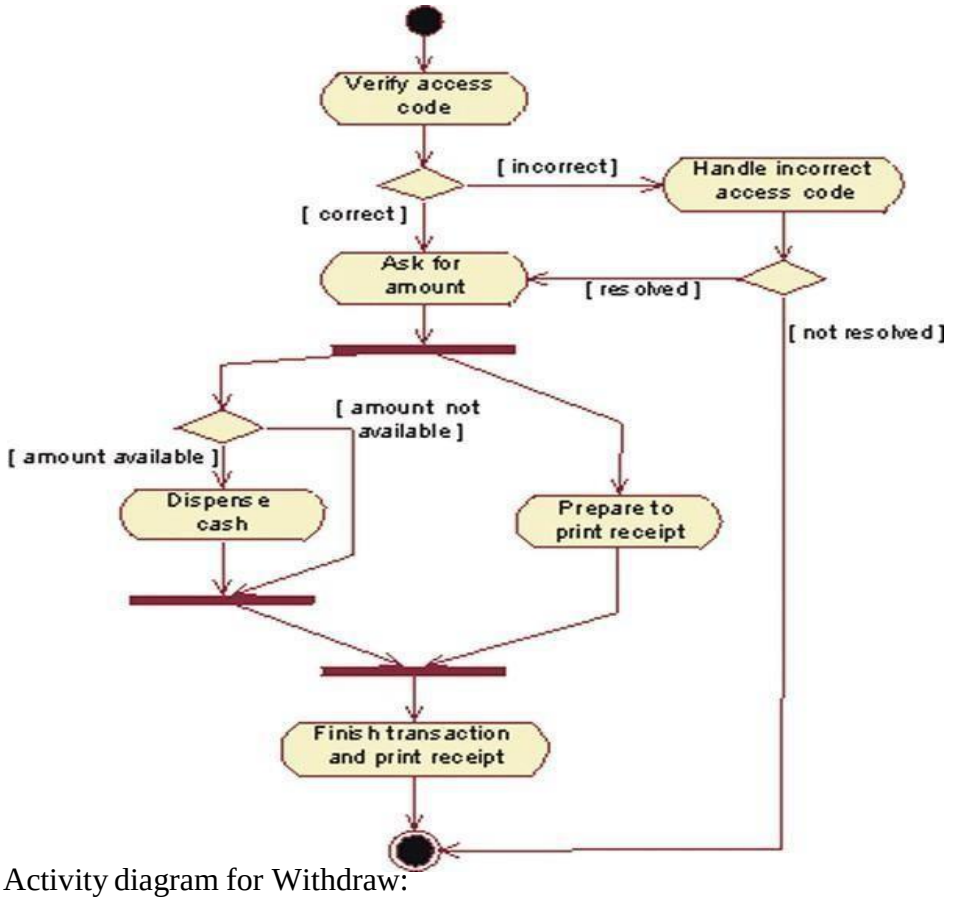
Procedure:-

Step1: First initial state is created.

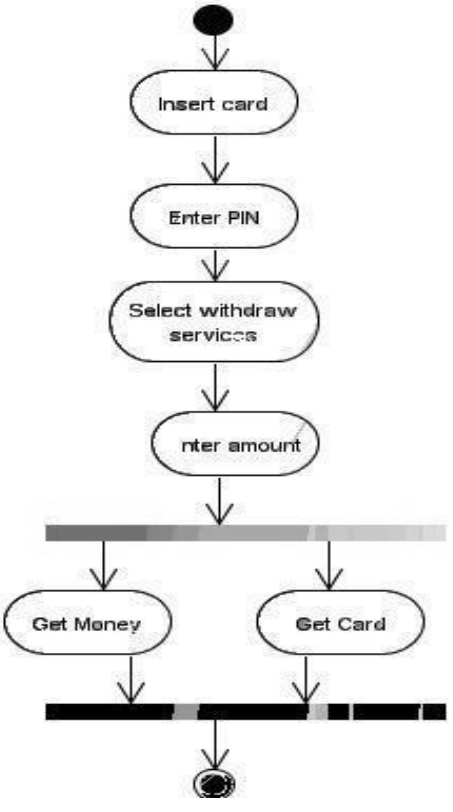
Step2: After that it goes to the action state insert card.

Step3: Next it undergoes transition to the state enter pin **Step4**: In this way it undergoes transitionsto the various states.**Step5:** Use forking and joining wherever necessary

DIAGRAM: Activity diagram for Transactions:



Activity diagram for Withdraw:



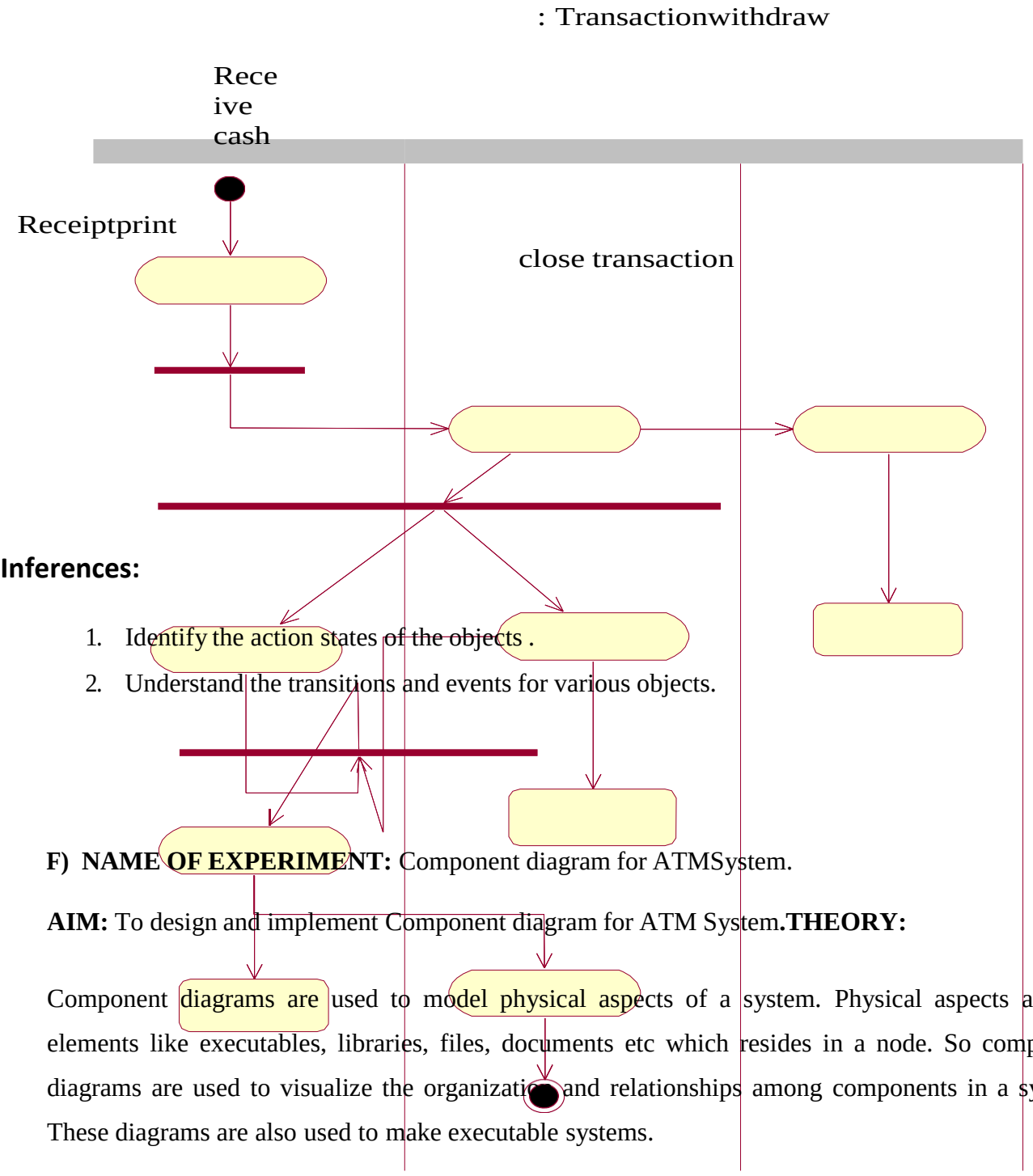
Customer **ATM** **bank serv er**

insert card

enter PIN validate

select transaction : Validation[success]

start transact ion



Inferences:

1. Identify the action states of the objects .
2. Understand the transitions and events for various objects.

F) NAME OF EXPERIMENT: Component diagram for ATMSystem.

AIM: To design and implement Component diagram for ATM System.**THEORY:**

Component diagrams are used to model physical aspects of a system. Physical aspects are the elements like executables, libraries, files, documents etc which resides in a node. So component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose:

Component diagrams can be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment. A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.

Before drawing a component diagram the following artifacts are to be identified clearly:

- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.
- Now after identifying the artifacts the following points needs to be followed:
- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing using tools.
- Use notes for clarifying important points.

Contents

Components, Interfaces, Relationships

Procedure:-

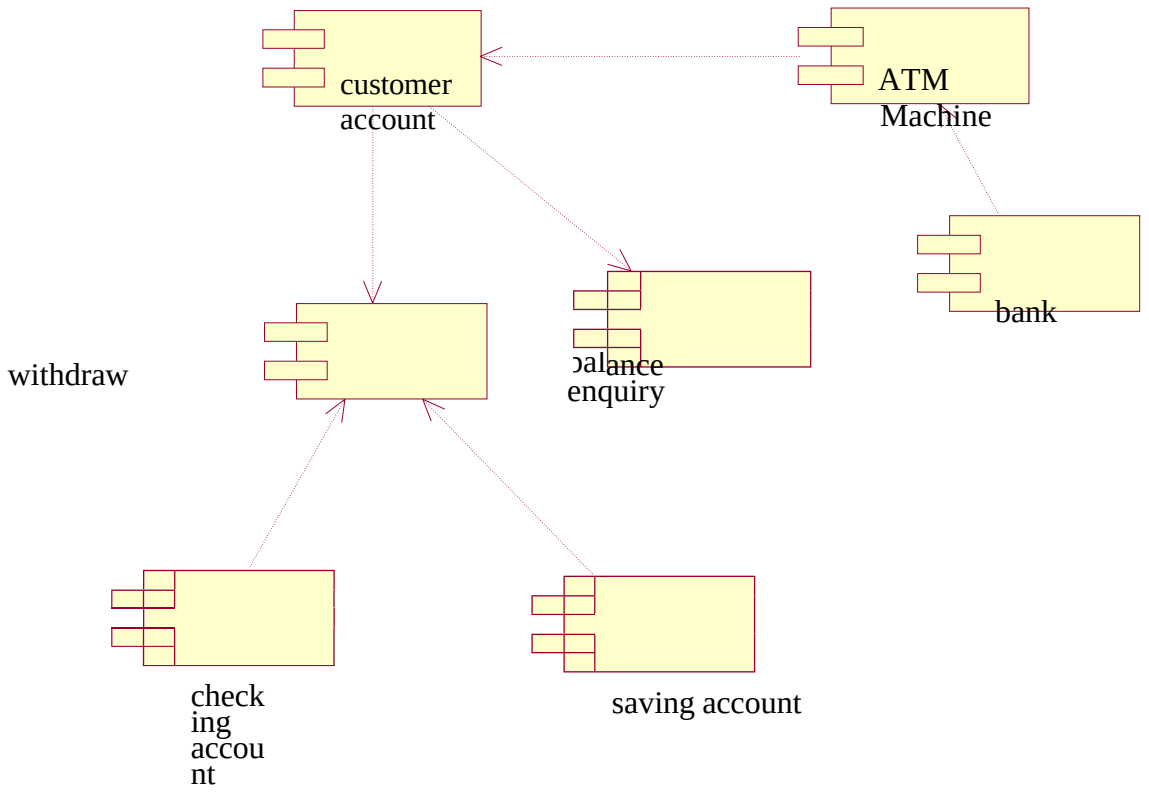
Step1: First user component is created.

Step2: ATM system package is created.

Step3: In it various components such as withdraw money, deposit money, check balance, transfer money etc. are created.

Step4: Association relationship is established between user and other components.

DIAGRAM:



G) NAME OF EXPERIMENT: Deployment diagram for ATMSystem**.**

AIM: To design and implement ATM System through Deployment diagram.

Purpose:

Deployment diagrams are used to visualize the topology of the physical components of a system where the software components are deployed. So deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams are used for describing the hardware components where software components are deployed. Component diagrams and deployment diagrams are closely related. Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

Contents: Nodes, Dependency & Association relationships

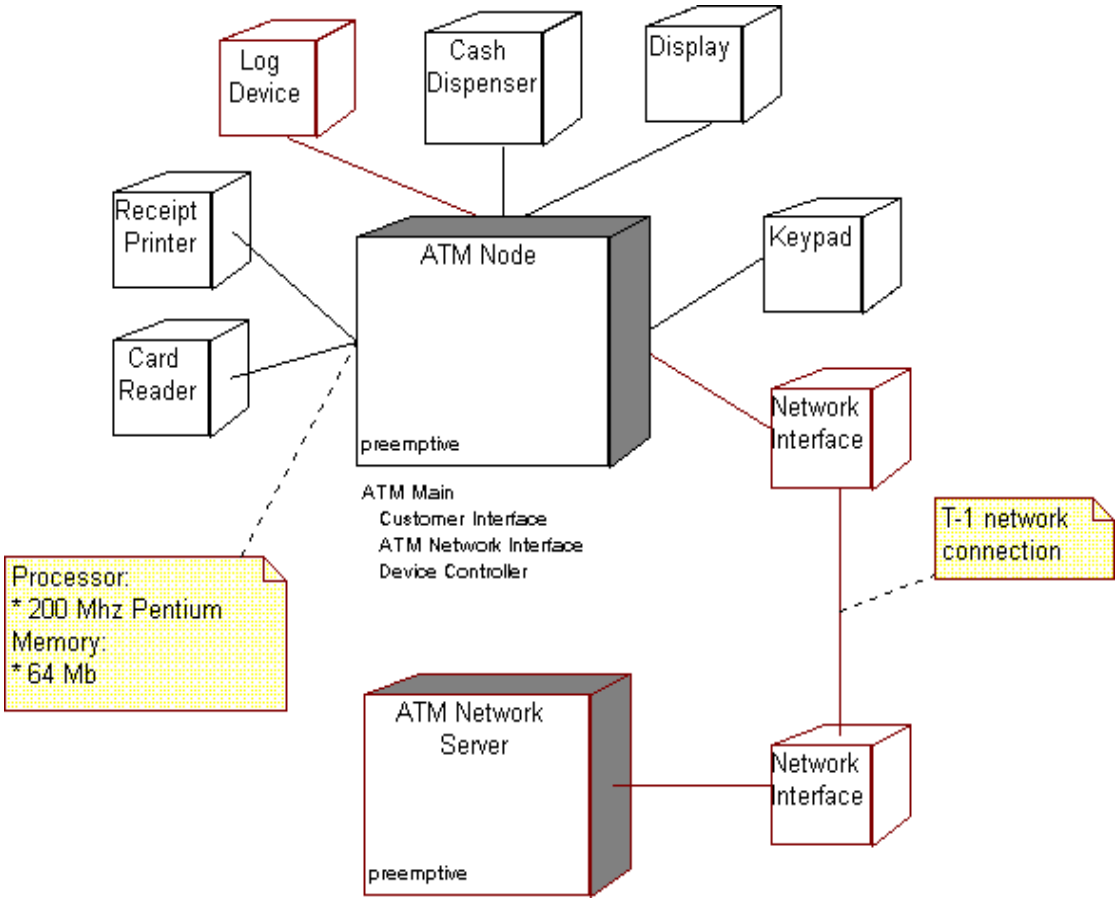
Procedure:-

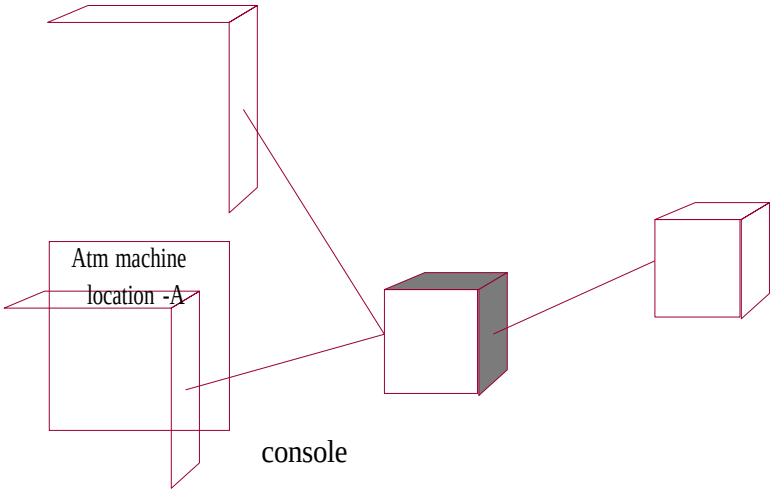
Step1: First user node is created

Step2: various nodes withdraw money, deposit money, and check balance, transfer money etc. are created.

Step4: Association relationship is established between user and other nodes.

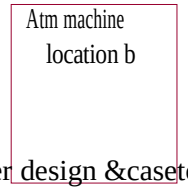
Step5: Dependency is established between deposit money and check balance.





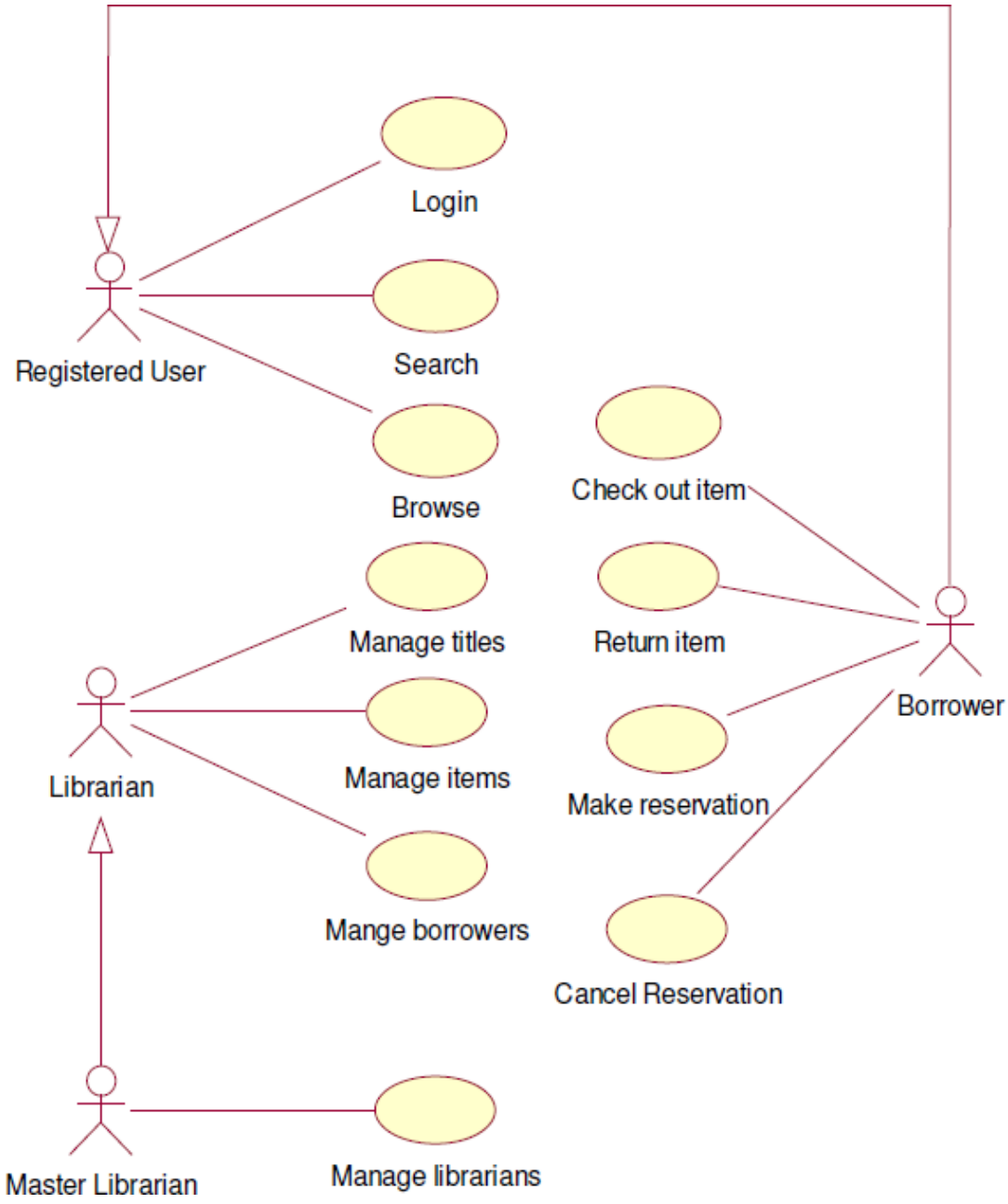
console

bankserver

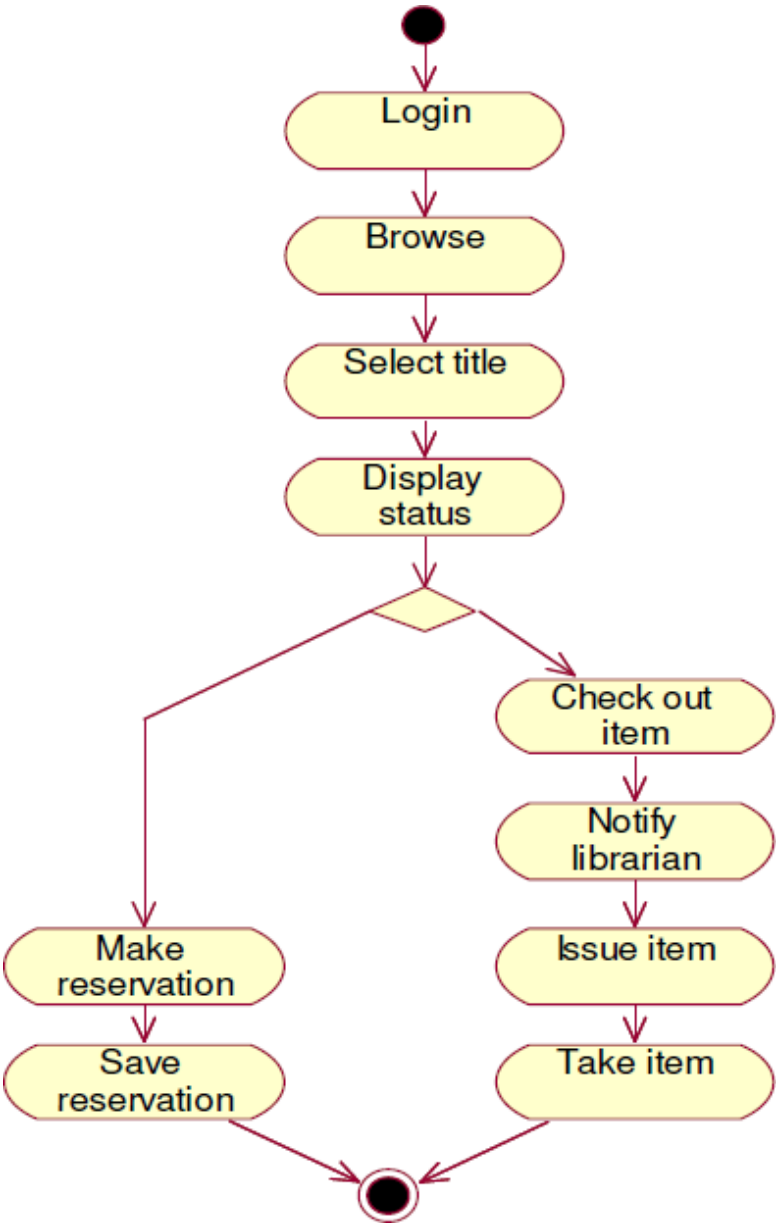


Library Management System (LMS)

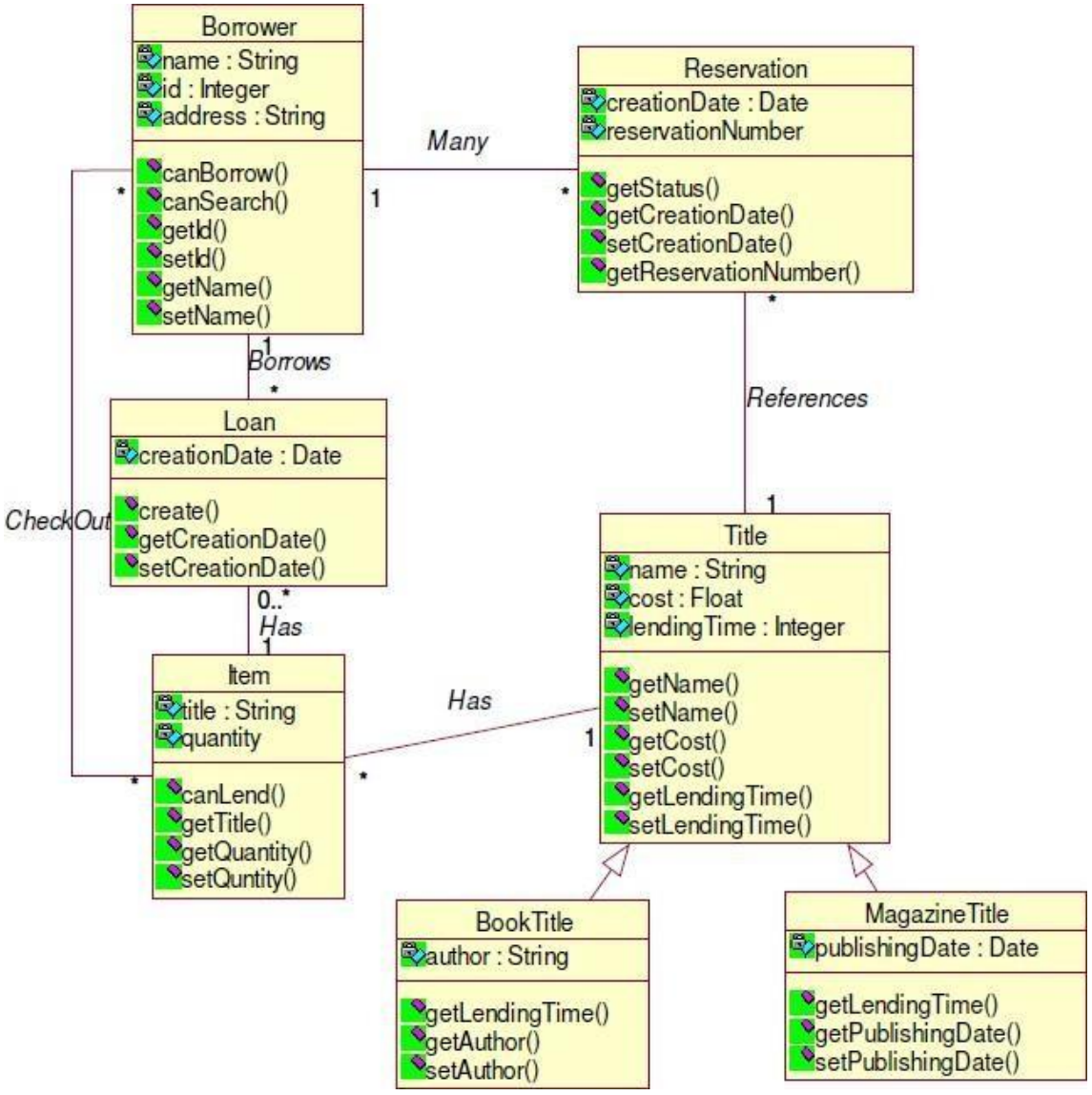
LMS Login Use Case Diagram:



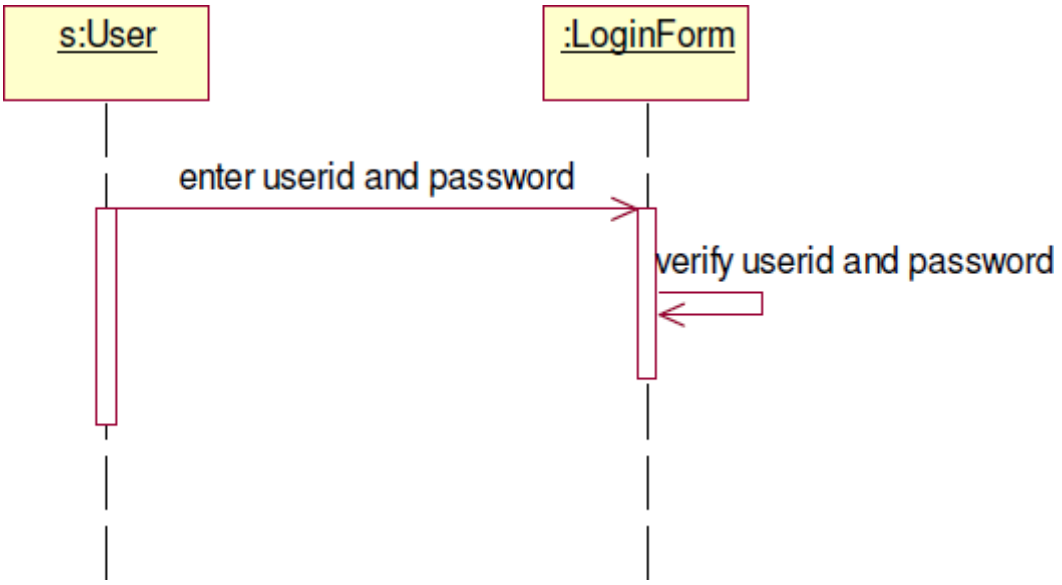
LMS Activity Diagram:



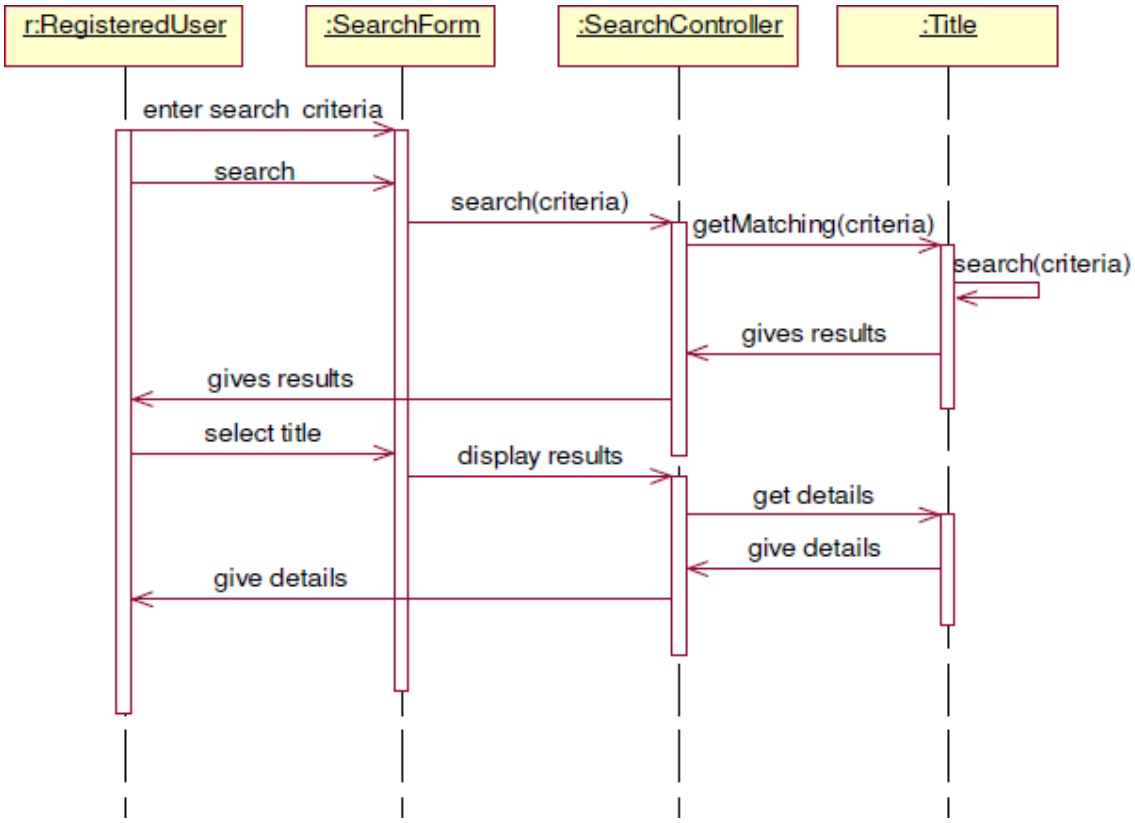
LMS Class Diagram:



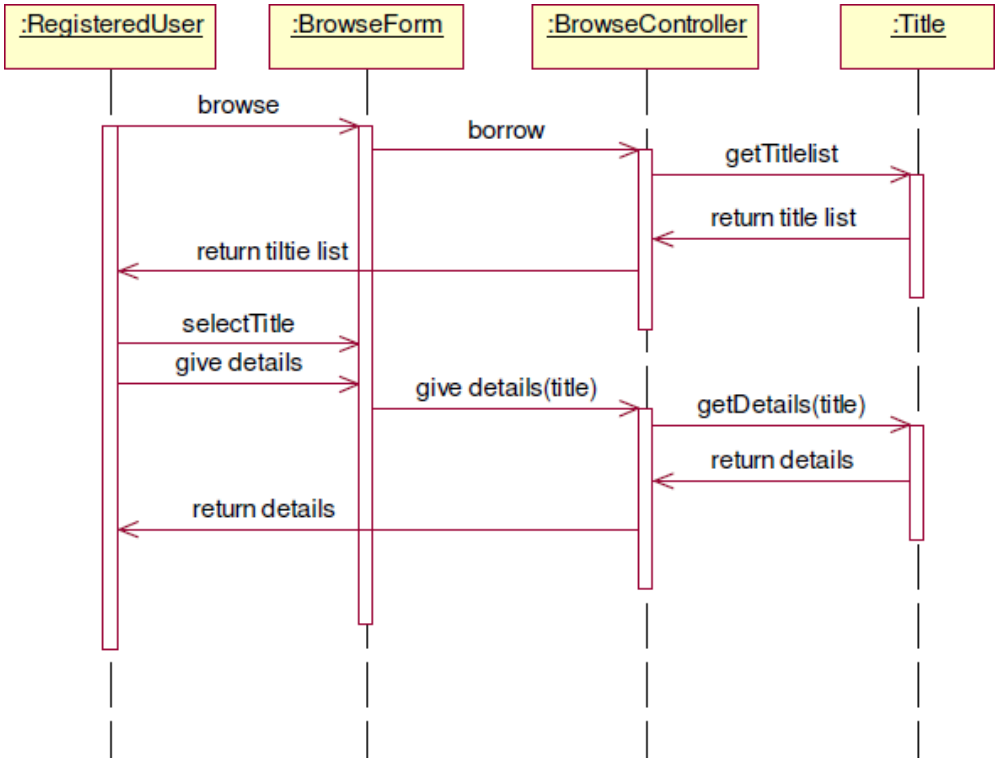
LMS Login Sequence Diagram:



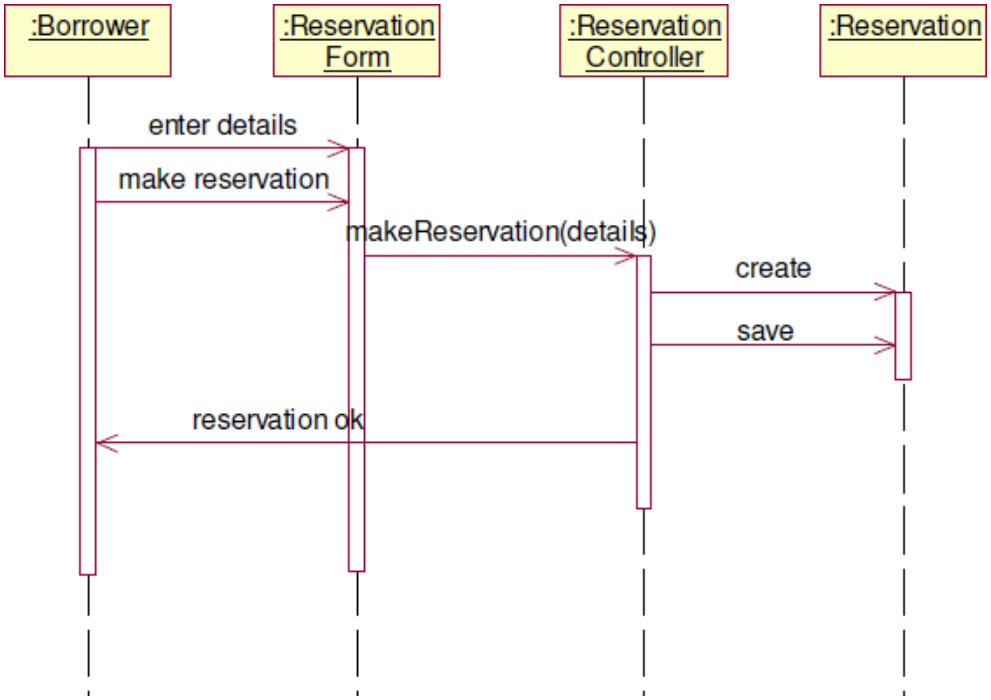
LMS Search Sequence Diagram:



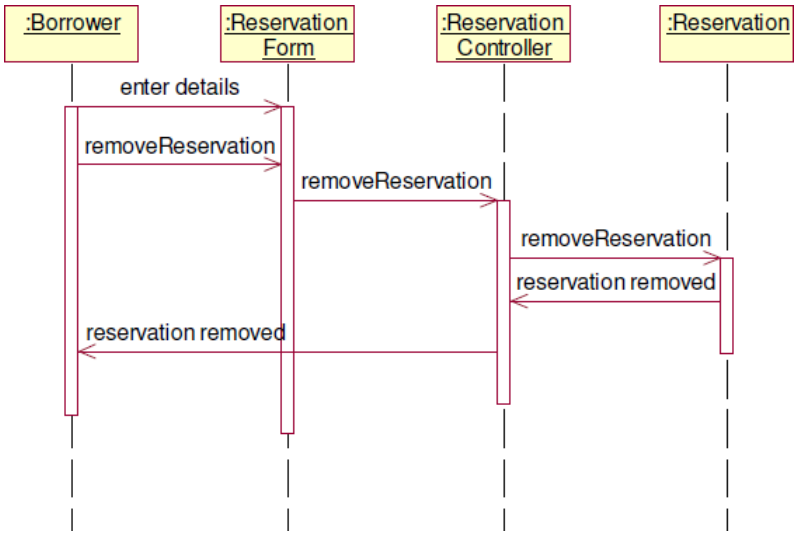
LMS Browse Sequence Diagram:



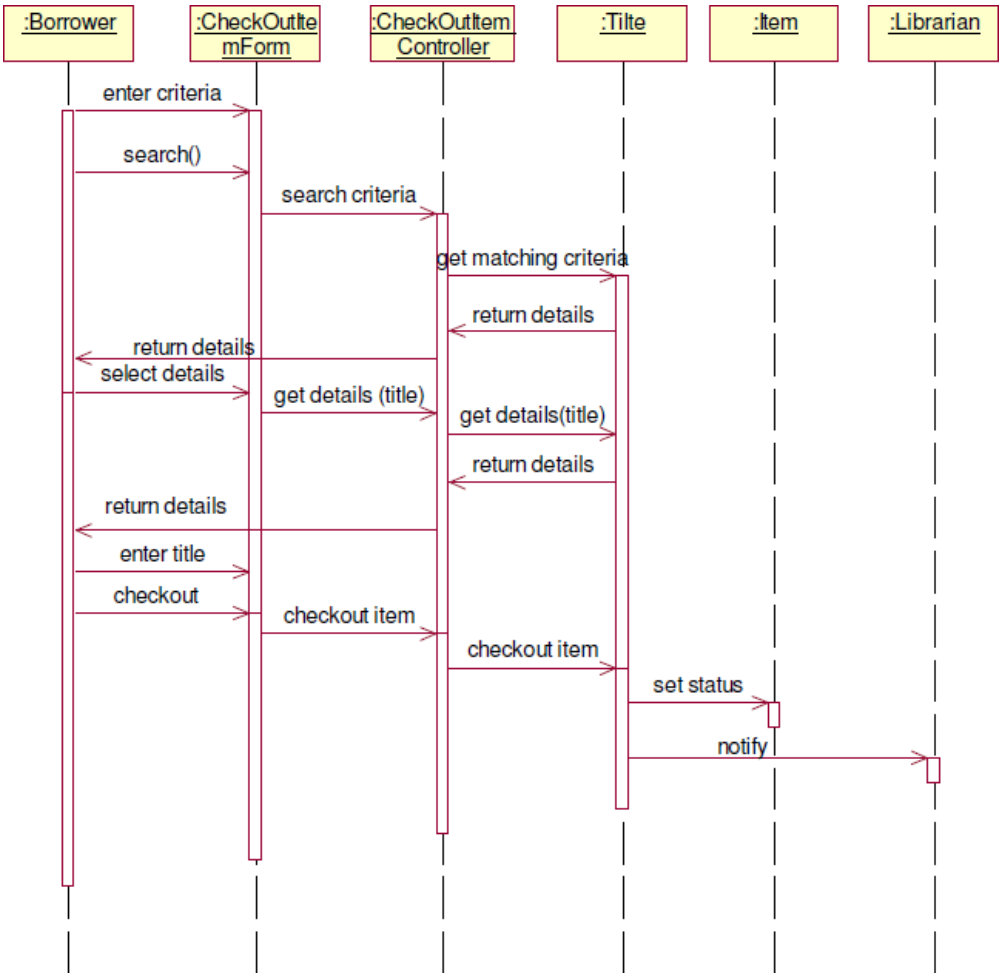
LMS Reservation Sequence Diagram:



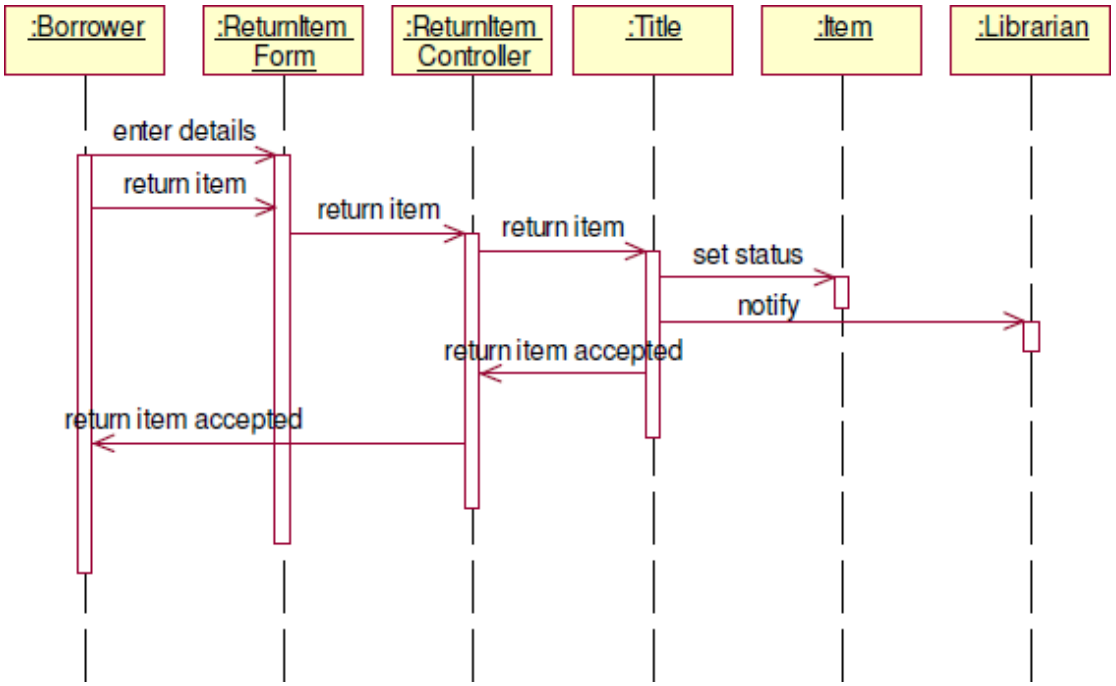
LMS Remove Reservation Sequence Diagram:



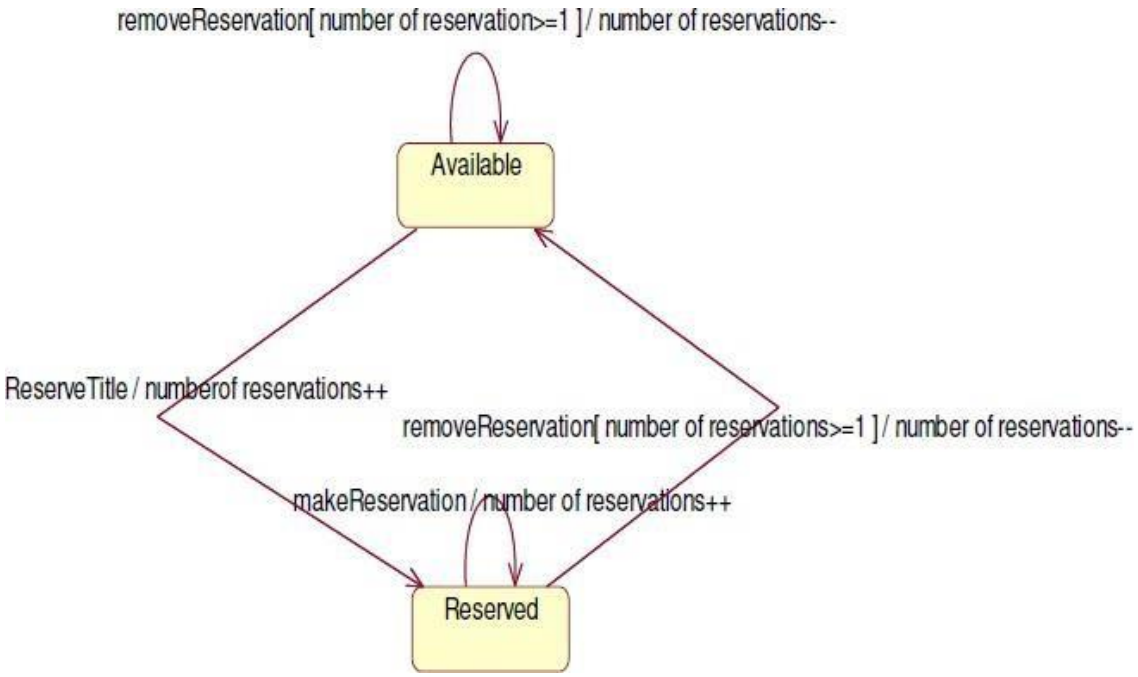
LMS Check Out Item Sequence Diagram:



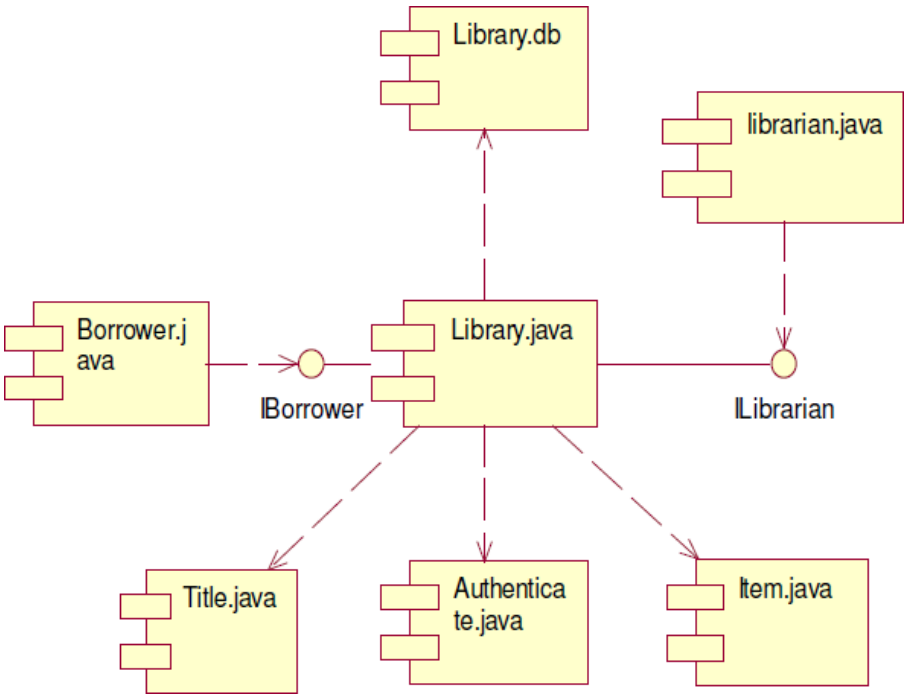
LMS Return Item Sequence Diagram:



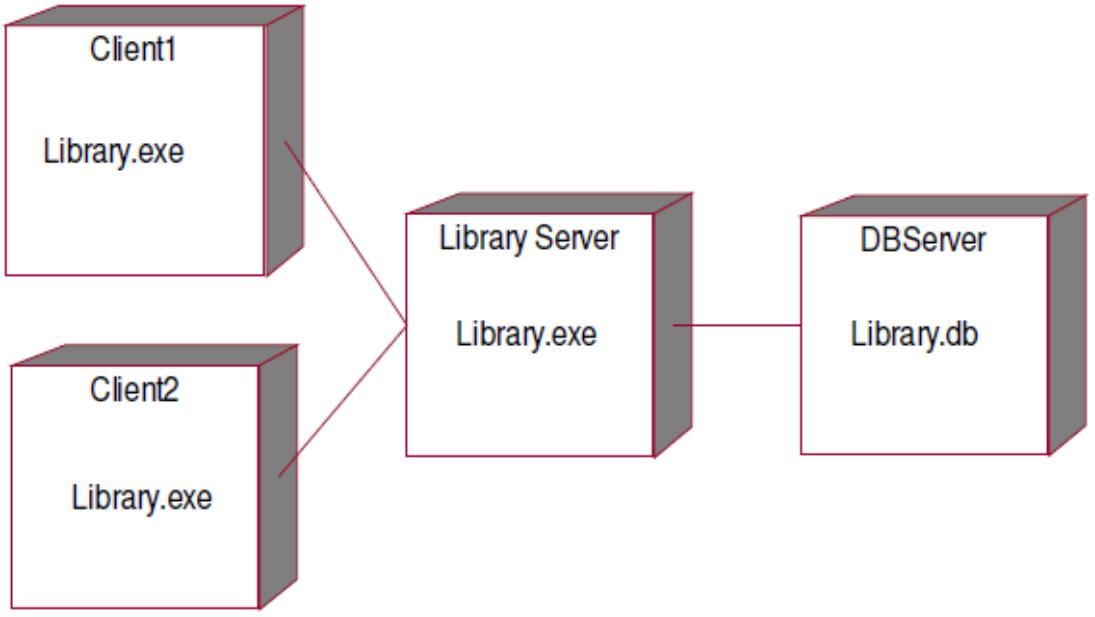
LMS State Chart Diagram for Title Class:



LMS Component Diagram:



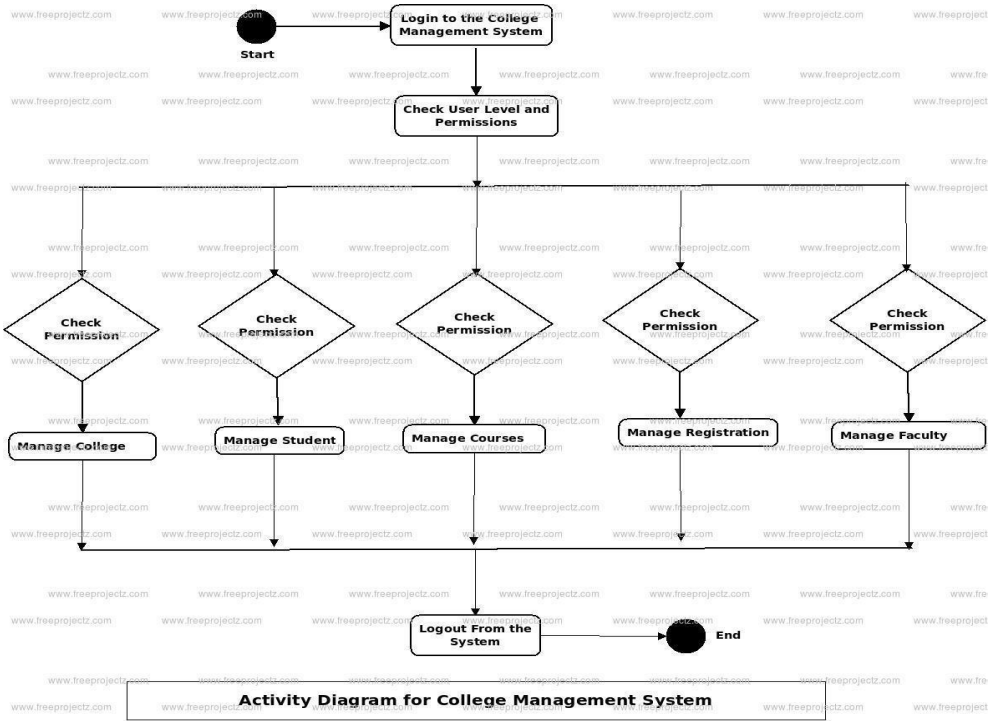
LMS Deployment Diagram:



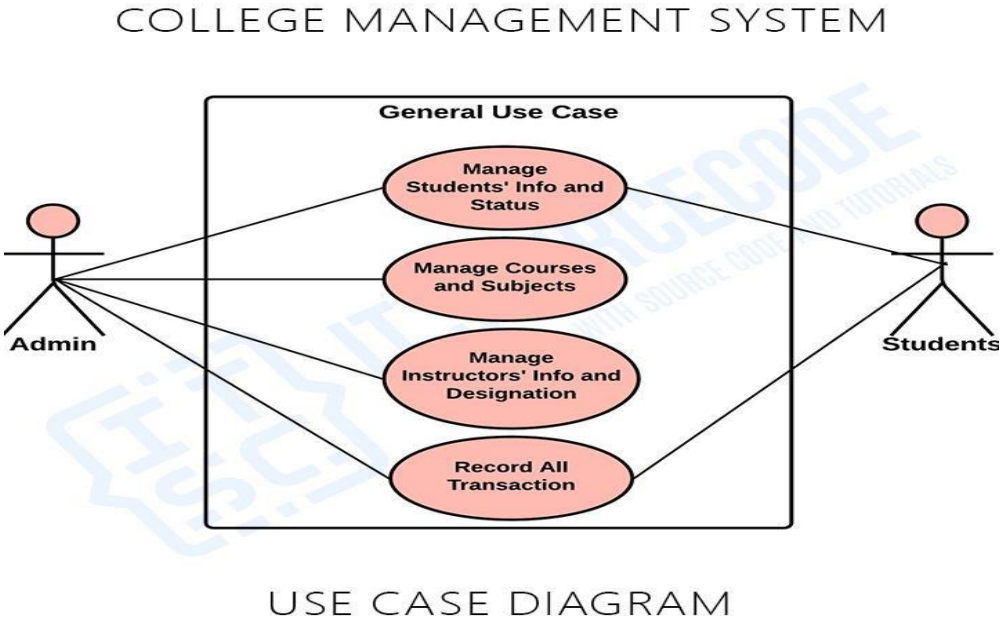
DEPARTMENT OF EMERGING TECHNOLOGIES

College Management

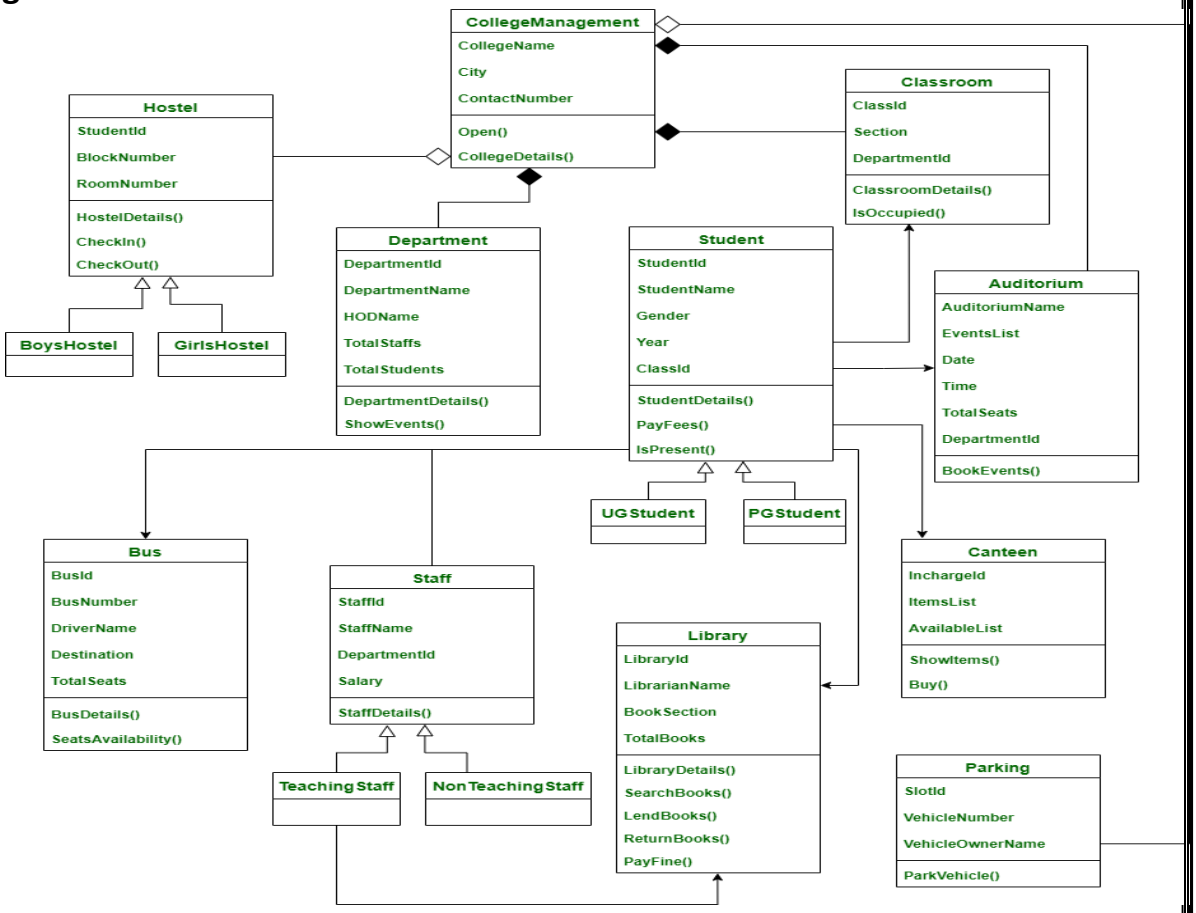
SystemActivity Diagram



Use case diagram



Class diagram



Sequence diagram