

# **COMPILER DESIGN AND CASE TOOLS**

## **LABORATORY MANUAL [R20A0587]**

**B.TECH III YEAR – I SEM**

**[A.Y:2023-2024]**



## **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, and 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**

### **Vision**

- **To acknowledge quality education and instill high patterns of discipline making the students technologically superior and ethically strong which involves the improvement in the quality of life in human race.**

### **Mission**

- **To achieve and impart holistic technical education using the best of infrastructure, outstanding technical and teaching expertise to establish the Students into competent and confident engineers.**
  - **Evolving the center of excellence through creative and innovative teaching learning practices for promoting academic achievement to produce international accepted competitive and world class professionals.**
-

## **PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)**

### **PEO1–ANALYTICALSKILLS**

1. To facilitate the graduates with the ability to visualize, gather information, articulate, analyze, solve complex problems, and make decisions. Entail to address the challenges of complex and computation intensiveproblems increasing their productivity.

### **PEO2–TECHNICALSKILLS**

2. To facilitate the graduates with the technical skills that prepare them forimmediate employment and pursue certification providing a deeper understanding of the technology in advanced areas of computer science and related fields, thus encouraging to pursue higher education and research based on their interest.

### **PEO3–SOFTSKILLS**

3. To facilitate the graduates with the soft skills that include fulfilling themission, setting goals, showing self confidence by communicating effectively, having a positive attitude, getinvolved in team-work, being a leader, managing their career and their life.

### **PEO4–PROFESSIONALETHICS**

4. To facilitate the graduates with the knowledge of professional and ethical responsibilities by paying attention to grooming, being conservative with style, following dress codes, safety codes, and adapting themselves to technological advancement.

## **PROGRAM OUTCOMES (POs)**

By the end of the program in CSE, all graduates will be able to have the following measurable skills:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems searching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/ development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multi disciplinary environments.
12. **Life- Long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.



# MALLAREDDY COLLEGE OF ENGINEERING & TECHNOLOGY

Maisammaguda, Dhulapally Post, Via Hakimpet, Secunderabad-500100

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the 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 upto the last session experiments and other utensils (if any) needed in the lab.
  - c. Proper Dress code and Identity card.
  - d. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
4. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
5. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
6. Computer labs are established with sophisticated and high-end branded systems, which should be utilized properly.
7. 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.
8. 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.
9. 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**

## INDEX

S.No	Experiment/Topic	PageNo	Remarks
	Importance/Rational behind the CD Lab	<b>1</b>	
	Objectives & Outcomes	<b>2</b>	
1	<b>Case Study :</b> Description of the Syntax of the source Language(mini language) for which the compiler components are Designed	<b>5</b>	
2	Write a C Program to Scan and Count the number of characters, words and lines in a file.	<b>9</b>	
3	Write a C Program to implement NFAs that recognize identifiers, constants and operators of the mini language.	<b>16</b>	
4	Write a C Program to implement DFAs that recognize identifiers, constants and operators of the mini language.	<b>22</b>	
5	Design a lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and newlines, comments etc.	<b>28</b>	
6	Implement the lexical analyzer using JLex, flexor other lexical analyzer generating tools.	<b>33</b>	
7	Design Predictive Parser for the given language	<b>41</b>	
8	Design a LALR bottom up parser for the given language	<b>43</b>	
9	Convert the BNF rules into Yacc form and write code to generate Abstract syntax tree.	<b>53</b>	
10	A program to generate machine code from the abstract syntax tree generated by the parser.	<b>56</b>	
<b>Additional/ExtraPrograms[Optional]</b>			
1	Lex Program to convert abc to ABC		
2	Write a lex program to find out total number of vowels and consonants from the given input sting.		
3	Implementation of Predictive Parser		
4	Implementation of Recursive Descent Parser		
5	Implementation of SLR Parser		



## IMPORTANCE OF COMPILER DESIGN LAB

- Compiler is software which takes as input a program written in a High-Level language and translates it into its equivalent program in Low Level program.
- Compilers teach us how real- world applications are working and how to design them.
- Learning Compilers gives us with both theoretical and practical knowledge that is crucial in order to implement a programming language. It gives you an new level of understanding of a language in order to make better use of the language (optimization is just one example).
- Sometimes just using a compiler is not enough. You need to optimize the compiler itself for your application.
- Compilers have a general structure that can be applied in many other applications, from debuggers to simulators to 3D applications to a browser and even cmd/shell.
- Understanding compilers and how they work makes it super simple to understand all the rest. A bit like a deep understanding of math will help you to understand geometry or physics. We cannot do physics without the math not on the same level.
- Just using something (read: tool, device, software, programming language) is usually enough when everything goes as expected. But if something goes wrong, only a true understanding of the inner workings and details will help to fix it.
- Even more specifically, Compilers are super laborated/sophisticated systems (architecture ally speaking). If you will say that can or have written a compiler by yourself -there will be no doubt as to your capabilities as a programmer.
- There is nothing you cannot do in the Software realm. so, better be a pilot who has the knowledge and mechanics of an airplane than the one who just know how to fly.
- Every computer scientist can do much better if have knowledge of compilers a part from the domain and technical knowledge.
- Compiler design lab provides deep understanding of how programming language Syntax,
- Semantics are used in translation into machine equivalents apart from the knowledge of various compiler generation tools like LEX, YACC etc.

## 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 course students will be able to

- 1) Understand the practical approaches of how a compiler works.
- 2) Understand and analyze the role of syntax and semantics of Programming languages in compiler construction
- 3) Apply the techniques and algorithms used in Compiler Construction in compiler component design
- 4) To use different tools in construction of the phases of a compiler for the mini language

### RECOMMENDED SYSTEM/SOFTWARE REQUIREMENTS:

To execute the experiments, we should have the following hardware/software at minimum

1. Intel based desktop PC with minimum of 166MHz or faster processor with at least 64MB RAM and 100MB free disk space.
2. C++ Compiler and JDK kit, Lex or Flex and YACC tools (Unix/Linux utilities)

### USEFUL TEXTBOOKS/REFERENCES/WEBSITES:

1. Modern compiler implementation Inc, Andrew. Appel, Revised Edn, Cambridge University Press
2. Principles of Compiler Design.– Navaho, J. Dullman; Pearson Education.
3. **lex&yacc**, -John 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

## SOURCELANGUAGE ( ACaseStudy)

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>::={ <variabledefinition><slist> }
           |{<slist>}
<variabledefinition>::=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 brackets /\*...\*/. The language has the rudimentary support for 1-Dimensional arrays. Ex: int a[3] declares a as an array of 3elements,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);
el se
{
int t3;
t3= 99;
t2=25;
print(-11+t2*t3);/*this is not a comment on two lines*/
}
Endif }

```

**1. Problem Statement:**

Write a C Program to Scan and Count the number of characters, words, and lines in a file.

**AIM:** To Write a C Program to Scan and Count the number of characters, words, and lines in a file.

**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 their respective counters
6. Display the counts
7. End

**Program:**

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
FILE *fp;
char fname[10],ch;
int c_count=0,w_count=0,l_count=0;
/* Input Surce File name */
printf("\nEnter the file name: ");
scanf("%s",fname);
/* Open Source file in Read mode */
fp=fopen(fname,"r");
/* Check if the file existing */
if(fp==NULL)
{
printf(" Unable to Open %s file",fname);
printf("\n Please check if it exists!");
exit(EXIT_FAILURE);
}
while((ch=fgetc(fp))!=EOF)
{
c_count++;
/* Check for newline chars */ if(ch == '\n' || ch == '\0')
l_count++;
/* Word count */
if(ch == ' ' || ch == '\t' || ch == '\n' || ch == '\0')
w_count++;
}
if(c_count>0)
{ w_count++;
l_count++; }
printf("\n The file Statistics \n");
printf(" The No of Characters: %d", c_count);
```

```
printf("\n The No of Words: %d", w_count);  
printf(" \n The No of Lines: %d", l_count);  
/* Close file to release resource */  
fclose(fp);  
} //End of main()
```

**Input:** Enter the Identifier input string/file:

These are few sentences in  
mini Language

**Output:**

No of characters: 35

No of Words: 7

No of lines: 2

**[Viva Questions]**

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

**Signature of faculty**

**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 recognizes identifiers, constants, and operator soft the mini language.

**ALGORITHM/PROCEDURE/PROGRAM:**

1. Start
2. Design the NFA(N) to recognize Identifiers ,Constants ,and Operators
3. Read the input string give it as input to the NFA
4. NFA processes the input and outputs “Yes” if  $w \in L(N)$ , “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 == '}')
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
```

**[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:**

- 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.

Signature of faculty

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

**AIM:** To Write a C Program to implement DFAs that recognize identifiers, constants, and operator soft he 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 DFAM
- 4 DFA processes the input and outputs “Yes” if  $w \in L(M)$ , “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 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

```

**[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.

**Signature of faculty**

**4. Problem Statement:** Design a Lexical analyzer. The lexical analyzer should ignore redundant blanks, tabs and new lines. It should also ignore comments. Although the syntax specification s those identifiers canbe 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. lookup()–it takes string as argument and check sits presence in the symbol table. If the string is found the n 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 in to 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/SOURCECODE:**

```
#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||strcmp("swi
tch",str)==0||strcmp("case",str)==0)
printf("\n%s is a keyword",str);
else
printf("\n%s is an identifier",str);
}
```

```

int 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("\nEnter 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("\nThe 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 identifiersare:");
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("\nSpecial characters are");
while((c=getc(f3))!=EOF)
    printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
return 0;
```

**Output:**

Enter the C Program:

a+b\*c

Ctrl-D

The no's in the program  
are:

The key words 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?
5. Which Finite state machines are used in lexical analyzer design?
6. What is the role of regular expressions, grammars in Lexical Analyzer?

**EXERCISE:**

- 1) Design a lexical analyzer to generate tokens for the identifiers, constants, keywords, and operators of language.
- 2) Write a C implementation of the above Lexical Analyzer.

**Signature of faculty**



**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) % {  
     Definitionofconstant/headerfiles  
  
     % }
  - b) RegularExpressions  
     %%  
     Transitionrules  
     %%
  - c) AuxiliaryProcedure(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

**LEXSPECIFICATIONPROGRAM/SOURCECODE(lexprog.l):**

```
// ***** LEX Program to identify Mini language Tokens
*****//DIGIT [0-
9]LETTER    [A-Za-z]
DELIM      [\t\n]
WS         { DELIM }+
ID         {(LETTER)[LETTER/DIGIT]}+
INTEGER    {DIGIT}+
%%
{WS}       {printf("\n WSspecialcharacters \n");}
{ID}       {printf("\nIdentifiers\n");}
{DIGIT}    {printf("\nIntgers\n");}
if         {printf("\nKeywords\n");}
else       {printf("\nkeywords\n");}
">"       { printf("\n Relational Operators\n");}
}"<"
           {printf("\nRelationalOperators\n");}"<
="
           {printf("\nRelationalOperators\n");}"=
>"
           {printf("\nRelationalOperators\n");}"=
"
           {printf("\nRelationalOperators\n");}"!
="
           { printf("\n Logical Operators \n");}
}"&&"     { printf("\n Logical Operators \n");}
}"||"     {printf("\nLogicalOperators\n");}"!"
           {printf("\n Logical Operators \n");}
}"+"      { printf("\n Arithmetic Operator\n");}
}"-"      { printf("\n Arithmetic Operator\n");}
}"*"      { printf("\n Arithmetic Operator\n");}
}"/"      { printf("\n Arithmetic Operator\n");}
}"%"      {printf("\nArithmeticOperator\n");}

%%
intyywrap(){
}
int main()
{
Printf(" Enger the text :");
yylex();
return0;
}
```

**OUTPUT:**

```
[root@localhost]#lex
lexprog.l[root@localhost]#cclex.y
y.c[root@localhost]#./a.outlexprog
```

**TEST CASES:**

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

**[Viva Questions]**

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

**EXERCISE:**

- 1) Write a LEX specification program for the tokens of C language
- 2) Execute the above LEX file using any LEX tools
- 3) Generate tokens of a simple C program

**Signature of faculty**

**6. Problem Statement: Design a Predictive Parser for the following grammar: {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 LeftMostDerivation of the input string if it is valid, error otherwise.

Step1: Start

Step2: Declare a character array w[10] and

Zasanarray 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 the n increment and check for (D(w[z]))

Step5:

if step4 is satisfied then the string is accepted Else string is not

Step6: Exit

**SOURCECODE:**

```
/**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 col7
#define size 50
int token;
charlexbuff[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;

}

else

F()

{

token=lexer();

if(token==id)

return 1;

else

{

if(token==4)

{

if(E())

{

if(token==5)

return 1;

return 0;

}

else

return 0;

}

else

return 0;

}

}

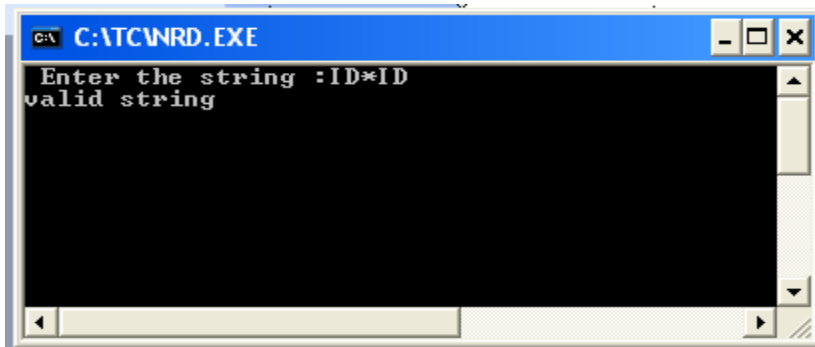
lexer()

{
```

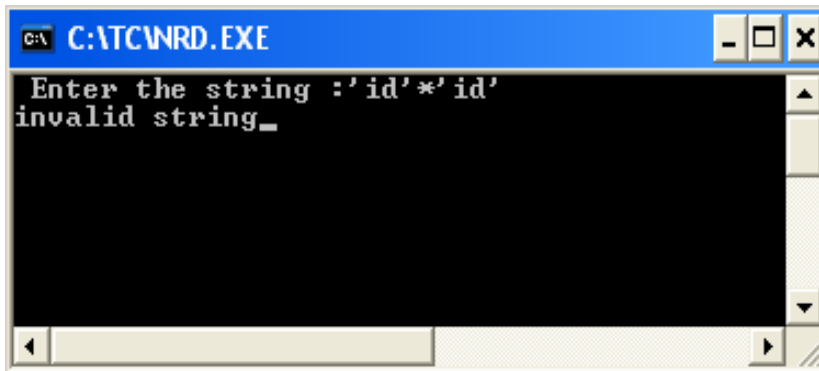


```
if(lexbuff[lookahead]!='\n')
{
while(lexbuff[lookahead]=='\t') lookahead++;
if(isalpha(lexbuff[lookahead]))
{
while(isalnum(lexbuff[lookahead])) lookahead++;
return(id);
}
else
{
if(isdigit(lexbuff[lookahead]))
{
while(isdigit(lexbuff[lookahead])) lookahead++;
return CONST;
}
else
{
else

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

**OUTPUT:**

```
C:\TC\NRD.EXE
Enter the string :ID*ID
valid string
```



```
C:\TC\NRD.EXE
Enter the string :\'id\'*\'id\'
invalid string_
```

**Viva Questions:**

1. What is a parse 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:**

1. Write a CFG to express the syntax of arithmetic Expressions of C language.
2. Design a Predictive Parsing table to recognize the arithmetic expressions of the C language.

Signature of faculty

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

**AIM:** To Design and implement an LALR bottomup Parser for checking the syntax of the statements in the given language.

### **ALGORITHM/PROCEDURE/CODE:**

1. Writing augmented grammar
2. LR(1) collection of items to be found
3. Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the LALR parsing table

### **LALR BottomUp Parser**

```

<parser.l>
% {
#include<stdio.h>
#include"y.tab.h"
% }
%%
[0-9]+

{yylval.dval=atof(yytext);return
DIGIT;
}
\n|.returnyytext[0];
%%
<parser.y>
% {
/*This YACCspecification file generates the LALR parser for the program considered
in experim ent4.*/
#include<stdio.h>
% }
%union
{
doubledval;
}
%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
;
%%
intmain()
{
Print("EnterAE:");
yyparse();
}
yyerror(char*s)
{
printf("%s",s);
}
```

**Output:**

```
$lex parser.l
$yacc-dparser.y
$cclex.yy.cy.tab.c -ll-lm
$./a.out
2+3
5.0000
```

**Viva Questions?**

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

**EXERCISE:**

- 1) Construct LALR parsing table to accept the arithmetic expressions of the C language.

**8. Problem statement:** Convert the BNF rules into YACC form and write code to generate abstractsyntax tree.

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

**ALGORITHM/PROCEDURE:**

ALGORITHM:

1. Start
2. Include the header file.
3. In int code.l, declare the variable line no as integer and assign it to be equal to '1'.
4. Start the int code. l with declarative section.
5. In translation rules section define keywords, data types and integer along with their actions .
6. Start the main block. In main block check the statement

1. declarative
2. assignment
3. conditional
4. if and else
5. While assignment.
8. Perform the actions of that particular block.
9. In main program declare the parameters arg c as int end \*argv[] as char.
10. In main program open file in read mode.
11. Print the output in a file.
12. End

**PROGRAM:****<int.l>**

```
% {
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
>intLineNo=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);returnRELOP;}
[\t];
\nLineNo++;
.returnyytext[0];
%%
```

**<int.y>**

```
% {
#include<string.h>
#include<stdio.h>
structquad
{
char op[5];char
arg1[10];char
arg2[10];
charresult[10];
}QUAD[30];
structstack
```

```
    {

}stk;

int items[100];
inttop;

int Index=0,tIndex=0,StNo,Ind,tInd;externint;
    }
    %token<var>NUMVARRELOP
    %tokenMAINIFELSEWHILE TYPE
    %type<var>EXPRASSIGNMENTCONDITIONIFSTELSESTWHILELOOP
    % left '-' '+'
    % left '*' '/'
    %% PROGRAM:MAINBLOCK
    ; BLOCK: '{' CODE '}'
    ; CODE:BLOCK
    |STATEMENTCODE
    |STATEMENT
    ; STATEMENT:DESCT','
    |ASSIGNMENT','
    |CONDST
    |WHILEST
    ; DESCCT:TYPEVARLIST
    ; 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:IF
ST{
Ind=pop();printf(QUAD[Ind].result,"%d",Index);
Ind=pop();printf(QUAD[Ind].result,"%d",Index);
}
|FSTELSEST
; IFST:IF('CONDITION'){
strcpy(QUAD[Index].op,"==");

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"FALSE");

strcpy( QUAD[Index].result,"- 1");

push(Index);

ndex++;

}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);
printf(QUAD[Ind].result,"%d",Index);
}BLOCK
{
Ind=pop();printf(QUAD[Ind].result,"%d"
,Index);
};
CONDITION:VARRELOPVAR
{
AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
|VAR

```



```

|NUM
;
WHILEST:WHILEL
OOP{
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(intargc,char*argv[])
{ FILE *fp;
int i;
if(argc >1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\nFilenotfound");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----- ""\n\t\tPosOperatorArg1Arg2Result""\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;
}
intpop()
{
int data;
if(stk.top ==-1)
{
printf("\nStackunderflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(charop[5],chararg1[10],chararg2[10],charresult[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("\nErroronlineno:%d",LineNo);
}

```

Input:

**\$vi**

test.cmai

n()

{

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:**

\$lexint.l

\$ yacc-dint.y

\$ gcclex.yy.cy.tab.c-ll-lm

\$/a.outtest.c

**OUTPUT:**

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	to
1	==	to	FALSE	5
2	+	a	b	t1
3	=	t1		a
4	GOTO			5
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t3
16	=	t6		c

**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?

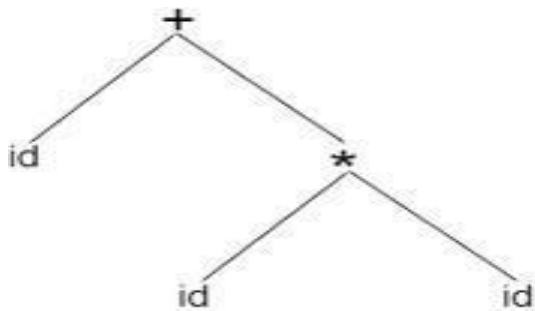
**Signature of faculty**

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

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

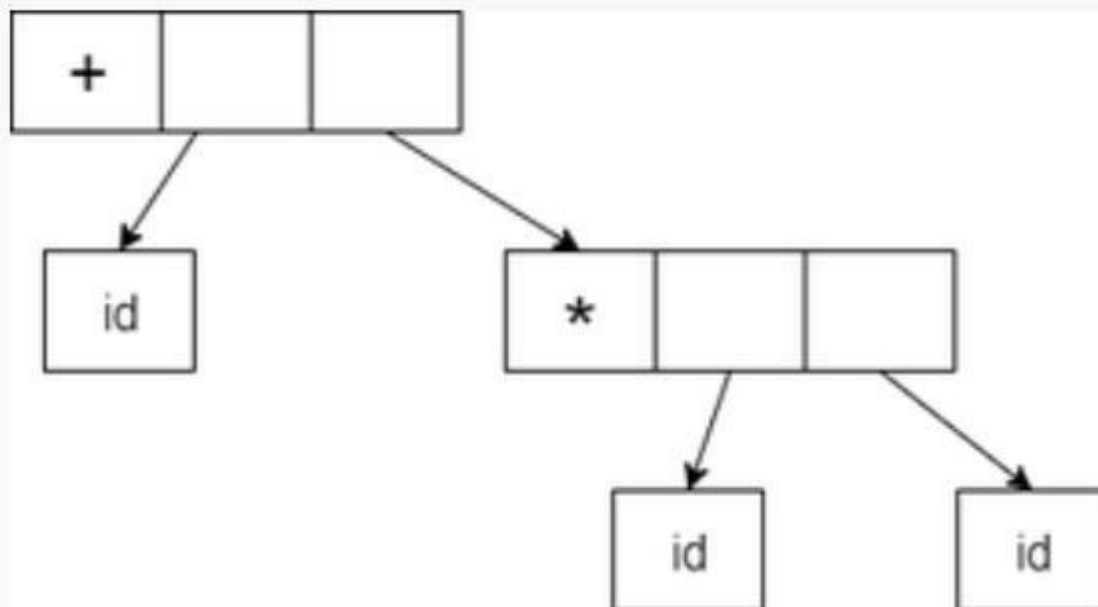
**ALGORITHM/PROCEDURE/SOURCECODE:**

*Flow Chart:*



*id + id \* id would have the following syntax tree which is as follows:*

*Abstract syntax tree will be as follows:*



**PROGRAM:**

```

#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("\nEnter file name of the intermediate code");
scanf("%s",&fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL||fp2==NULL)
{
printf("\nError 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\tOUT%s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s%s",operand1,operand2);
fprintf(fp2,"\n\tJMP%s,label#%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]")==0)
{
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\tSTORE%s[%s],%s",operand1,operand2,result);
}
if(strcmp(op,"uminus")==0)
{

```

```

    fscanf(fp1,"%s%s",operand1,result);
    fprintf(fp2,"\n\tLOAD-%s,R1",operand1);
    fprintf(fp2,"\n\tSTORER1,%s",result);
}
switch(op[0])
{
case '*':fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\tLOAD",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n\tMULR1,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\tADDR1,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\tSUBR1,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\tDIVR1,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\tDIVR1,R0");
    fprintf(fp2,"\n \t STORER0,%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 \tLOAD%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("Erroropeningthefile\n");
    exit(0);
}
do
{
    ch=fgetc(fp2);
    printf("%c",ch);
}while(ch!=EOF);
fclose(fp1);
return0;
}
int check_label(intk)
{
    int i;
    for(i=0;i<no;i++)
    {
        if(k==label[i])
            return 1;
    }
    return0;
}

```

**Input :**

```

$ vi int.txt
= t1 2
[]=a01
[]=a 1 2
[]=a 2 3
*t16 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
```

**Signature of faculty**

**INDEX**

<b>S. No</b>	<b>Experiment/Topic</b>	<b>Page No.</b>	<b>Remarks</b>
	Outcomes	<b>2</b>	
	Software / Hardware Requirements	<b>2</b>	
	Introduction about UML	<b>1-32</b>	
1	UML Diagram for ATM Transaction System	<b>33-49</b>	
2	UML Diagram for Library Management System	<b>50-57</b>	
3	UML Diagram for College administration System	<b>58</b>	

**OUTCOMES :**

At the end of the Course, the Student will be able to:

- Identify the requirements specification for an intended software system.
- Model the Use case and Class diagrams for the given application or Case study
- Develop the sequence and collaboration diagrams for the given application.
- Build Activity diagram and State Chart diagrams for the given application
- Design Component and Deployment diagrams for the given application.

**RECOMMENDED SYSTEM / SOFTWARE REQUIREMENTS:**

To execute the experiments, we should have the following hardware /softwares at minimum

1. IBM Rational Rose Star UML

**TEXT BOOKS:**

1. Grady Booch, James Rumbaugh, Ivar Jacobson : “The Unified Modeling Language User Guide”, 2nd Edition, Pearson Education, Reprint 2017.
2. Object Oriented Analysis & Design by Prof .Partha Pratim Das IIT Kharagpur.
3. Erich Gamma, “Design Patterns By Elements of Reusable Object-Oriented Software”, Pearson Education, 2015.
4. Pascal Roques, “Modeling Software Systems Using UML2”, 1st Edition, WILEY Dreamtech, 2007.

## What is the UML?

- The Unified Modeling Language is a family of graphical notations, backed by a single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented style.
- UML first appeared in 1997
- UML is standardized. Its content is controlled by the Object Management Group (OMG), a consortium of companies.
- Unified
  - UML combined the best from object-oriented software modeling methodologies that were in existence during the early 1990's.
  - Grady Booch, James Rumbaugh, and Ivor Jacobson are the primary contributors to UML.
- Modeling
  - Used to present a simplified view of reality in order to facilitate the design and implementation of object-oriented software systems.
  - All creative disciplines use some form of modeling as part of the creative process.
  - UML is a language for documenting design
  - Provides a record of what has been built.
  - Useful for bringing new programmers up to speed.
- Language
  - UML is primarily a graphical language that follows a precise syntax.
  - UML 2 is the most recent version
  - UML is standardized. Its content is controlled by the Object Management Group (OMG), a consortium of companies.

## Why We Model

- The importance of modeling
- Four principles of modeling
- Object-oriented modeling

## The Importance of Modeling

- A successful software organization is one that consistently deploys quality software that meets the needs of its users.
- An organization that can develop such software in a timely and predictable fashion, with an efficient and effective use of resources, both human and material, is one that has a sustainable business

## What, then, is a model? Simply put,

- *A model is a simplification of reality.*
- A model provides the blueprints of a system.
- A good model includes those elements that have broad effect and omits those minor elements that are not relevant to the given level of abstraction.

## Why do we model? There is one fundamental reason.

*We build models so that we can better understand the system we are developing.*

Through modeling, we achieve four aims

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

Modeling is not just for big systems. Even the software equivalent of a dog house can benefit from some modeling.

*We build models of complex systems because we cannot comprehend such a system in its entirety.*

## Principles of Modeling

Four principles of modeling:

1. The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.
4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

## An Overview of the UML

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

## The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include (but are not limited to)

- Requirements
- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

## Where Can the UML Be Used?

The UML is intended primarily for software-intensive systems. It has been used effectively for such domains as

- Enterprise information systems
- Banking and financial services
- Telecommunications
- Transportation
- Defense/aerospace
- Retail
- Medical electronics
- Scientific
- Distributed Web-based services

## A Conceptual Model of the UML

- A conceptual model needs to be formed by an individual to understand UML.
- UML contains three types of building blocks: things, relationships, and diagrams.
- Things
  - Structural things
    - Classes, interfaces, collaborations, use cases, components, and nodes.
  - Behavioral things
    - Messages and states.
  - Grouping things
    - Packages
  - Annotational things
    - Notes
- Relationships: Dependency, Association, Generalization and Realization.
- Diagrams: class, object, use case, sequence, collaboration, statechart, activity, component and deployment.

## Building Blocks of the UML:

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things
2. Relationships
3. Diagrams

## Things in the UML

There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

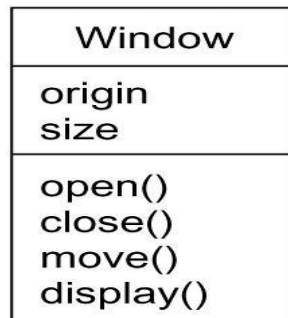
## Structural Things

- *Structural things* are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.
- Classes
- Interface
- Cases
- Active Classes
- Components
- Nodes
- Collaborations

### Classes:

a *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations

**Figure : Classes**



### Interfaces

- an *interface* is a collection of operations that specify a service of a class or component. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface

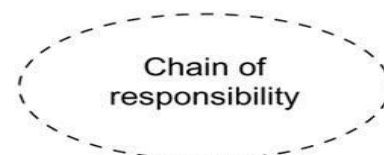
**Figure : Interfaces**



### Collaborations:

- A *collaboration* defines an interaction. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name

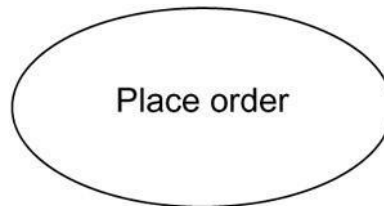
**Figure:  
Collaborations**



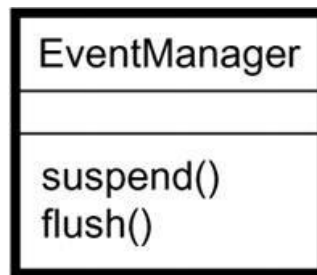


**Use Cases:**

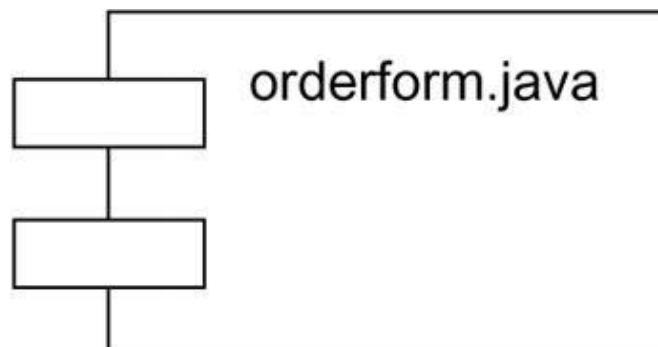
- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name

**Figure :Use Cases****Active Classes:**

- An active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations

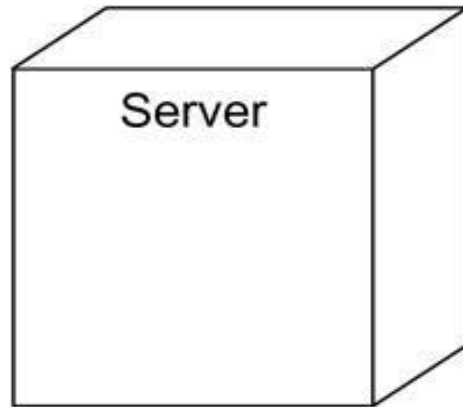
**Figure :Active Classes****Components:**

- A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name.

**Figure :Components**

**Nodes:**

- 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.
- A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name.

**Figure :Nodes****Behavioral Things:**

*Behavioral things* are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

1. Messages
2. States

**Messages:**

- An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. Graphically, a message is rendered as a directed line, almost always including the name of its operation.

**States:**

- A *state machine* is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.



### Grouping Things:

- *Grouping things* are the organizational parts of UML models. These are the boxes into which a model can be decomposed. There is one primary kind of grouping thing, namely, packages.

### Packages:

- A *package* is a general-purpose mechanism for organizing elements into groups. Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents

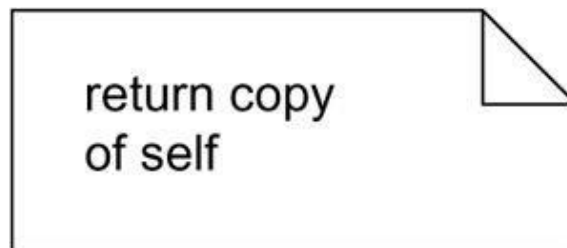
**Figure: Packages**



### Annotational Things:

- *Annotational things* are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model.
- There is one primary kind of annotation thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

**Figure: Notes**



### Relationships in the UML

There are four kinds of relationships in the UML:

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

**Dependency** is a semantic relationship between two model elements in which a change to one element (the independent one) may affect the semantics of the other element (the dependent one). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.



**Association** is a structural relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes.

Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and end names



**Generalization** is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). The child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.



**Realization** is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. generalization and a dependency relationship.



## UML Diagrams

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships).
- A diagram represents an elided view of the elements that make up a system.
- In theory, a diagram may contain any combination of things and relationships.
- In practice, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software intensive system

### The UML includes Nine kinds of diagrams:

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

1. Class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

2. Object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams.
3. Use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system.
4. Sequence diagram is an interaction diagram that emphasizes the time-ordering of messages;
5. Collaboration diagram a communication diagram is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages.
6. Statechart diagram shows a state machine, consisting of states, transitions, events, and activities. A state diagrams shows the dynamic view of an object.
7. Activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system.
8. Component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system.
9. Deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture

## Class Diagram

Class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams address the static design view or the static process view of the system.

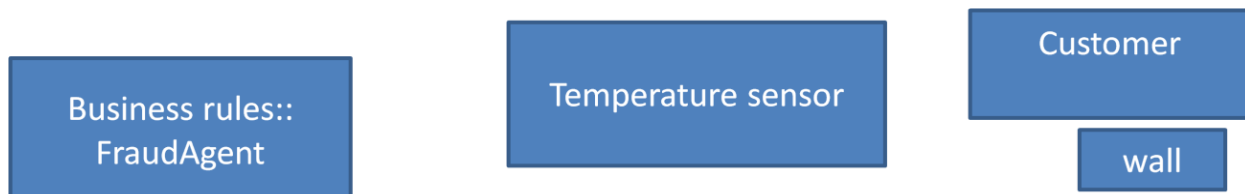
Graphically it is represented as follows.

### Classes

- A Class is a description of set of objects that share same attributes, operations, relationships and semantics .
- Graphically, a class is rendered as a rectangle

### Name

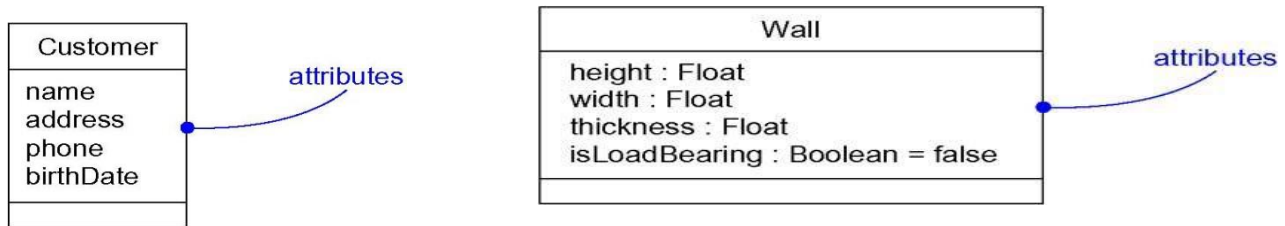
- Every class must have a name that distinguishes it from other classes. A *name* is a textual string.
- That name alone is known as a *simple name*;
- *path name* is the class name prefixed by the name of the package in which that class lives.



### Attributes

- An attribute is a named property of a class that describes range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all. An attribute
- represents some property of the thing you are modeling that is shared by all objects of that class.
- Graphically, attributes are listed in a compartment just below the class name.
- Attributes may be drawn showing only their names,

## Attributes

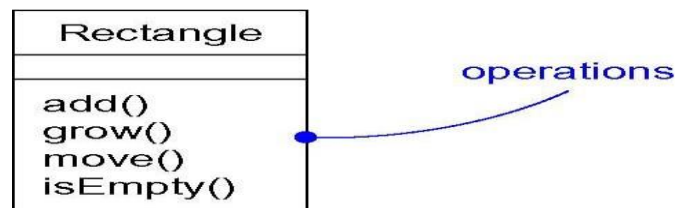


further specify an attribute by stating its class and possibly a default initial value

## Attributes and Their Class

### Operations

- An operation is the implementation of a service that can be requested from any object of the class to affect behavior.
- An operation is an abstraction of something you can do to an object and that is shared by all objects of that class.
- A class may have any number of operations or no operations at all.
- Operations may be drawn showing only their names.



Example :

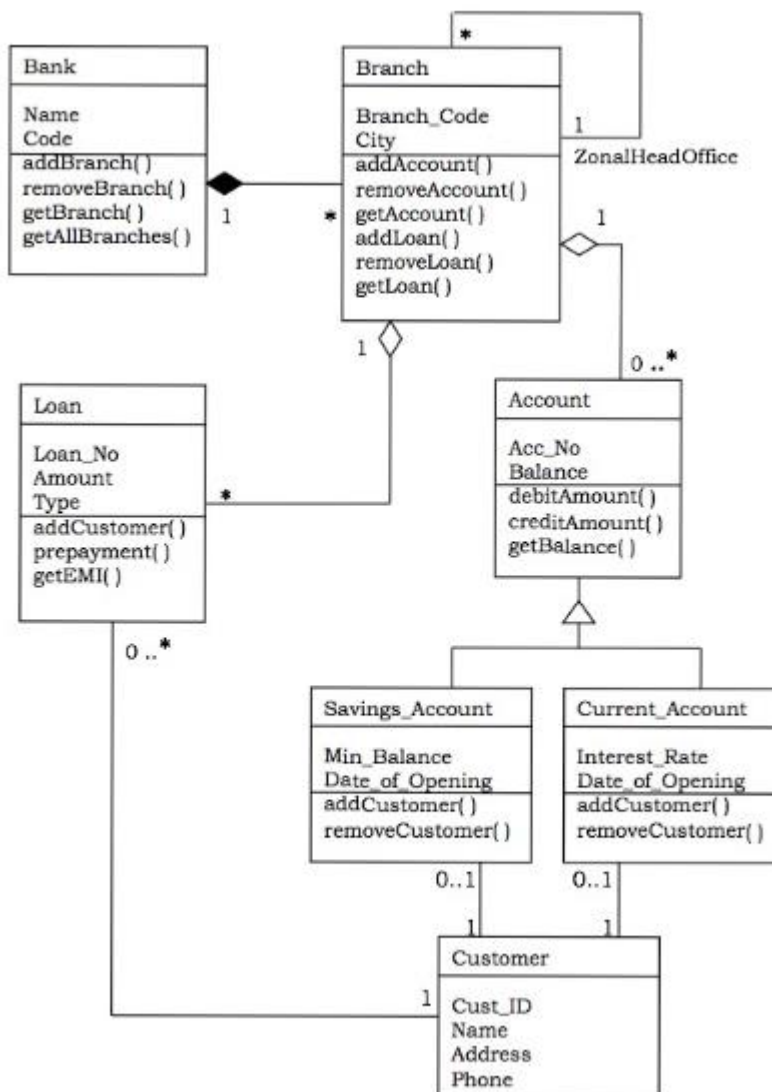


Fig: Class diagram for Simple Banking System

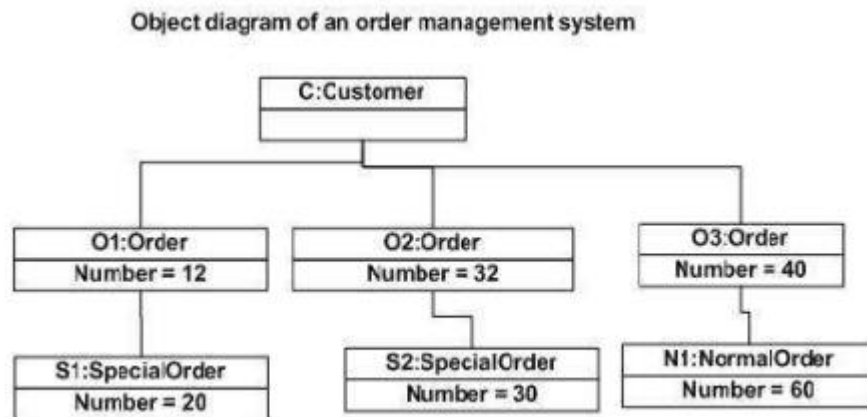
### Object Diagram

- Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.
- Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.
- Object diagrams are used to render a set of objects and their relationships as an instance.
- Object Diagrams use real world examples to depict the nature and structure of the system at a particular point in time.

The following diagram is an example of an object diagram. It represents the Order management system which we have discussed in the chapter Class Diagram. The following diagram is an instance of the system at a particular time of purchase. It has the following objects.

- Customer
- Order
- SpecialOrder
- NormalOrder

Now the customer object (C) is associated with three order objects (O1, O2, and O3). These order objects are associated with special order and normal order objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.



### Class vs. Object diagram

Serial No.	Class Diagram	Object Diagram
1.	It depicts the static view of a system.	It portrays the real-time behavior of a system.
2.	Dynamic changes are not included in the class diagram.	Dynamic changes are captured in the object diagram.
3.	The data values and attributes of an instance are not involved here.	It incorporates data values and attributes of an entity.
4.	The object behavior is manipulated in the class diagram.	Objects are the instances of class

### 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

It consists of

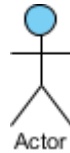
- Usecases
- Actors
- Communication Links of a particular system.

#### Actor

- Someone interacts with use case (system function).
- Named by noun.
- Actor plays a role in the business
- Similar to the concept of user, but a user can play different roles
- For example:



- A prof. can be instructor and also researcher
- plays 2 roles with two systems
- Actor triggers use case(s).
- Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs).



### Use Case

- System function (process - automated or manual)
- Named by verb + Noun (or Noun Phrase).
- i.e. Do something
- Each Actor must be linked to a use case, while some use cases may not be linked to actors.



### Communication Link

- The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link.
- Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.

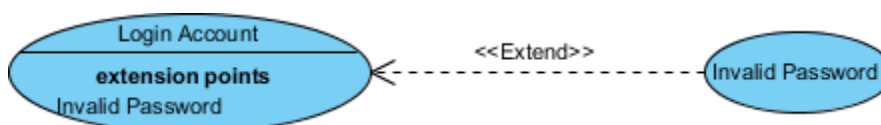
---

### Structuring Use Case Diagram with Relationships

Use cases share different kinds of relationships. Defining the relationship between two use cases is the decision of the software analysts of the use case diagram. A relationship between two use cases is basically modeling the dependency between the two use cases. The reuse of an existing use case by using different types of relationships reduces the overall effort required in developing a system. Use case relationships are listed as the following:

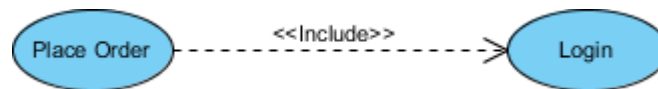
#### Extends

- Indicates that an "Invalid Password" use case may include (subject to specified in the extension) the behavior specified by base use case "Login Account".
- Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow.
- The stereotype "<<extends>>" identifies as an extend relationship



### Include

- When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as include or uses relationship.
- A use case includes the functionality described in another use case as a part of its business process flow.
- A uses relationship from base use case to child use case indicates that an instance of the base use case will include the behavior as specified in the child use case.
- An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at the base of the arrow.
- The stereotype "<<include>>" identifies the relationship as an include relationship.

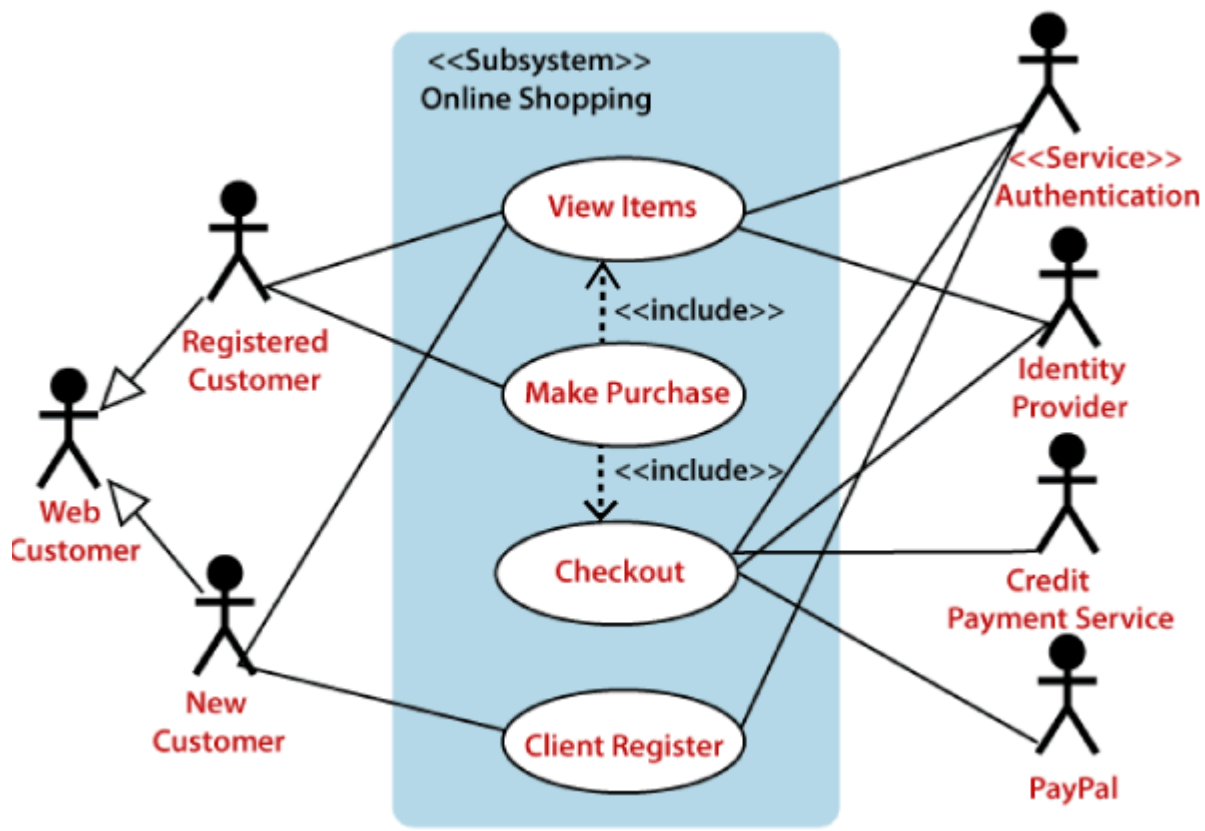


### Generalization

- A generalization relationship is a parent-child relationship between use cases.
- The child use case is an enhancement of the parent use case.
- Generalization is shown as a directed arrow with a triangle arrowhead.
- The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.



### Example of Use case diagram for Online Shopping



## Sequence Diagram

Sequence diagram are interaction diagrams. This diagram emphasizes the time- ordering of messages. These diagrams address the dynamic view of a system.

Sequence Diagram displays the timesequence 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.

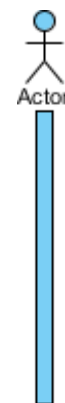
- the interaction that takes place in a collaboration that either realizes a use case or an operation (instance diagrams or generic diagrams)
- high-level interactions between user of the system and the system, between the system and other systems, or between subsystems (sometimes known as system sequence diagrams)

### Actor

- a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data)
- external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject).
- represent roles played by human users, external hardware, or other subjects.

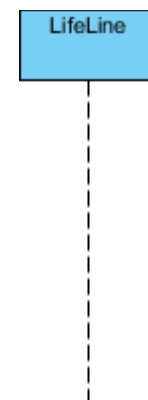
### Note that:

- An actor does not necessarily represent a specific physical entity but merely a particular role of some entity
- A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person.



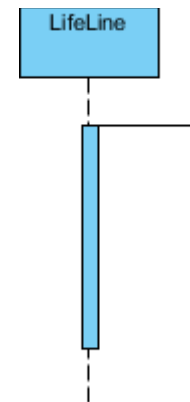
### Lifeline

- A lifeline represents an individual participant in the Interaction.



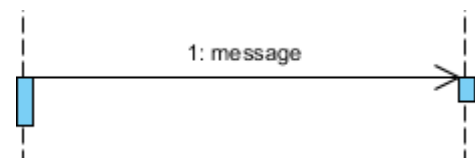
### Activations

- A thin rectangle on a lifeline) represents the period during which an element is performing an operation.
- The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively



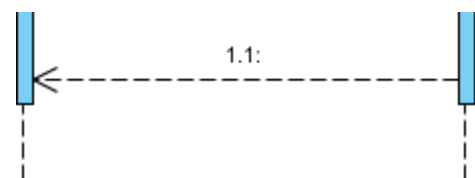
### Call Message

- A message defines a particular communication between Lifelines of an Interaction.
- Call message is a kind of message that represents an invocation of operation of target lifeline.



### Return Message

- A message defines a particular communication between Lifelines of an Interaction.
- Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.



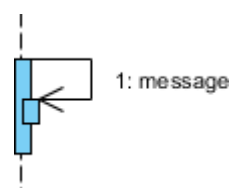
### Self Message

- A message defines a particular communication between Lifelines of an Interaction.
- Self message is a kind of message that represents the invocation of message of the same lifeline.



### Recursive Message

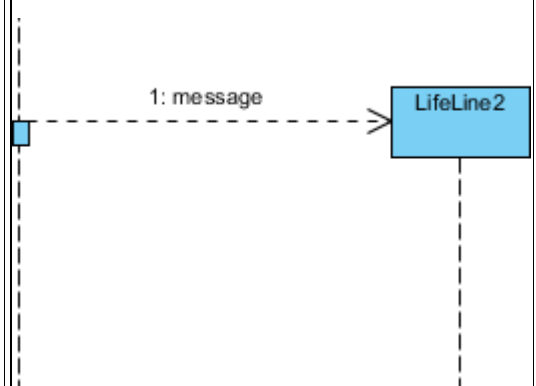
- A message defines a particular communication between Lifelines of an Interaction.
- Recursive message is a kind of message that represents the invocation of message of the same lifeline. It's target points to an activation on top



of the activation where the message was invoked from.

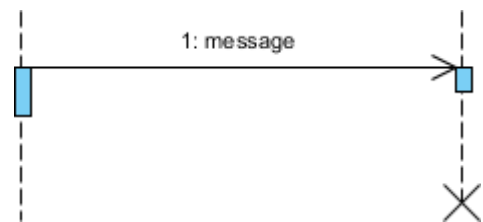
#### Create Message

- A message defines a particular communication between Lifelines of an Interaction.
- Create message is a kind of message that represents the instantiation of (target) lifeline.



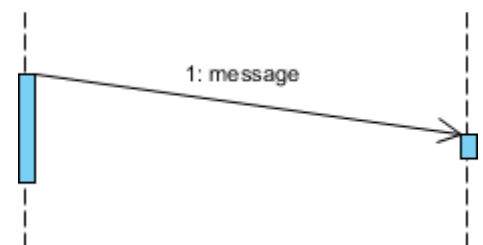
#### Destroy Message

- A message defines a particular communication between Lifelines of an Interaction.
- Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.

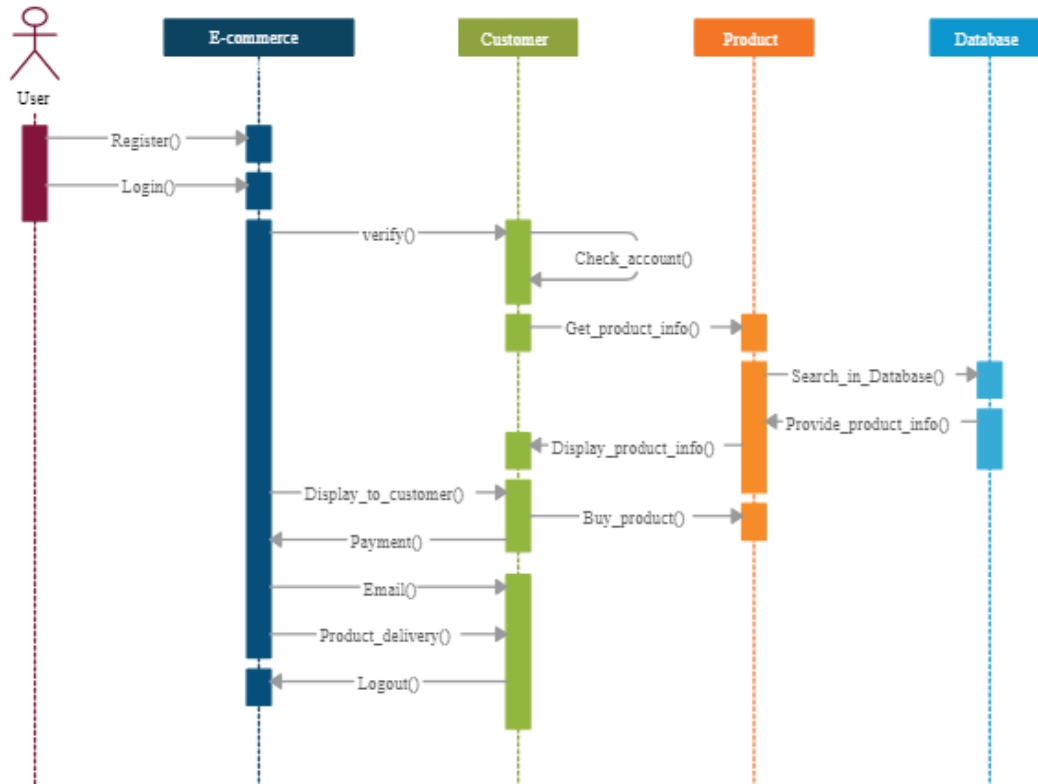


#### Duration Message

- A message defines a particular communication between Lifelines of an Interaction.
- Duration message shows the distance between two time instants for a message invocation.



### Example: Sequence diagram for Buying a product use case in a E-commerce website



## 5 Collaboration Diagram

Collaboration diagrams are also interaction diagrams. These diagrams emphasize the structural organization of the objects that send and receive messages. These diagrams address the dynamic view 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:-

### Notations of Collaboration Diagram Objects

An object is represented by an object symbol showing the name of the object and its class underlined, separated by a colon:

Object\_name : class\_name

You can use objects in collaboration diagrams in the following ways:

- Each object in the collaboration is named and has its class specified
- Not all classes need to appear
- There may be more than one object of a class

- An object's class can be unspecified. Normally you create a collaboration diagram with objects first and specify their classes later.
- The objects can be unnamed, but you should name them if you want to discriminate different objects of the same class.

### Actors

Normally an actor instance occurs in the collaboration diagram, as the invoker of the interaction. If you have several actor instances in the same diagram, try keeping them in the periphery of the diagram.

- Each Actor is named and has a role
- One actor will be the initiator of the use case

### Links

Links connect objects and actors and are instances of associations and each link corresponds to an association in the class diagram

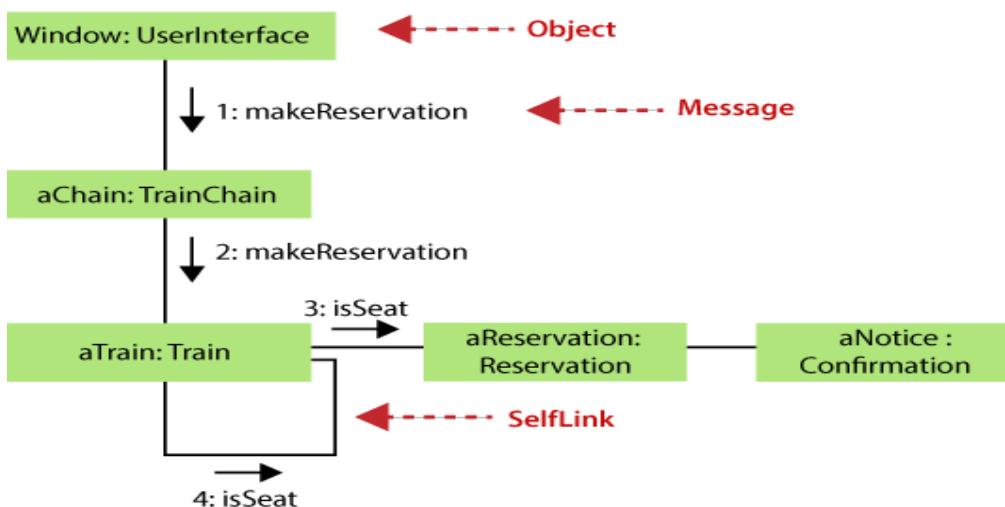
Links are defined as follows:

- A link is a relationship among objects across which messages can be sent. In collaboration diagrams, a link is shown as a solid line between two objects.
- An object interacts with, or navigates to, other objects through its links to these objects.
- A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.
- Message flows are attached to links, see Messages.

### Messages

A message is a communication between objects that conveys information with the expectation that activity will ensue. In collaboration diagrams, a message is shown as a labeled arrow placed near a link.

- The message is directed from sender to receiver
- The receiver must understand the message
- The association must be navigable in that direction



## 6. State chart 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.

- A state diagram is used to represent the condition of the system or part of the system at finite instances of time.
- It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as State machines and State-chart Diagrams.
- These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli. We can say that each and every class has a state but we don't model every class using State diagrams. We prefer to model the states with three or more states.

### Uses of state chart diagram –

- We use it to state the events responsible for change in state (we do not show what processes cause those events).
- We use it to model the dynamic behavior of the system .
- To understand the reaction of objects/classes to internal or external stimuli.

Firstly let us understand what are Behavior diagrams?

### There are two types of diagrams in UML :

1.**Structure Diagrams** – Used to model the static structure of a system, for example- class diagram, package diagram, object diagram, deployment diagram etc.

2.**Behavior diagram** – Used to model the dynamic change in the system over time. They are used to model and construct the functionality of a system. So, a behavior diagram simply guides us through the functionality of the system using Use case diagrams, Interaction diagrams, Activity diagrams and State diagrams.

### Difference between state diagram and flowchart :

The basic purpose of a state diagram is to portray various changes in state of the class and not the processes or commands causing the changes. However, a flowchart on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.

### Basic components of a statechart diagram –

1. **Initial state** – We use a black filled circle represent the initial state of a System or a class.



Figure – initial state notation

2. **Transition** – We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.

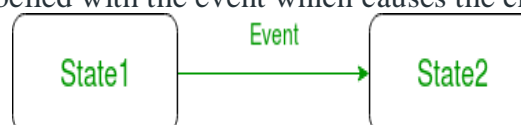


Figure – transition

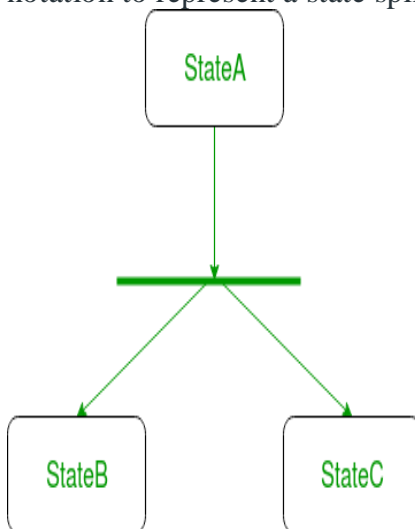


3. **State** – We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.



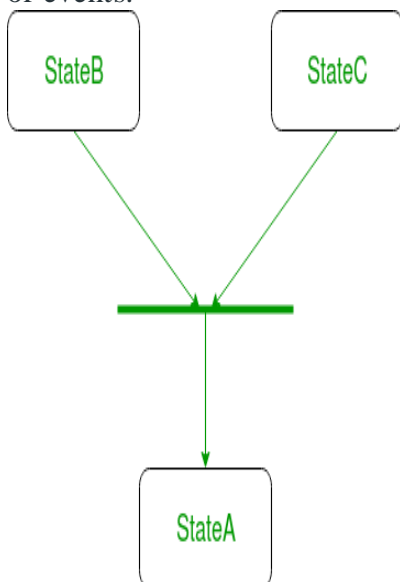
**Figure** – state notation

4. **Fork** – We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



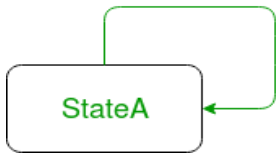
**Figure** – a diagram using the fork notation

5. **Join** – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.



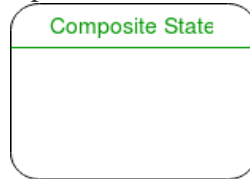
**Figure** – a diagram using join notation

6. **Self transition** – We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.



**Figure** – self transition notation

7. **Composite state** – We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.



**Figure** – a state with internal activities

8. **Final state** – We use a filled circle within a circle notation to represent the final state in a state machine diagram.

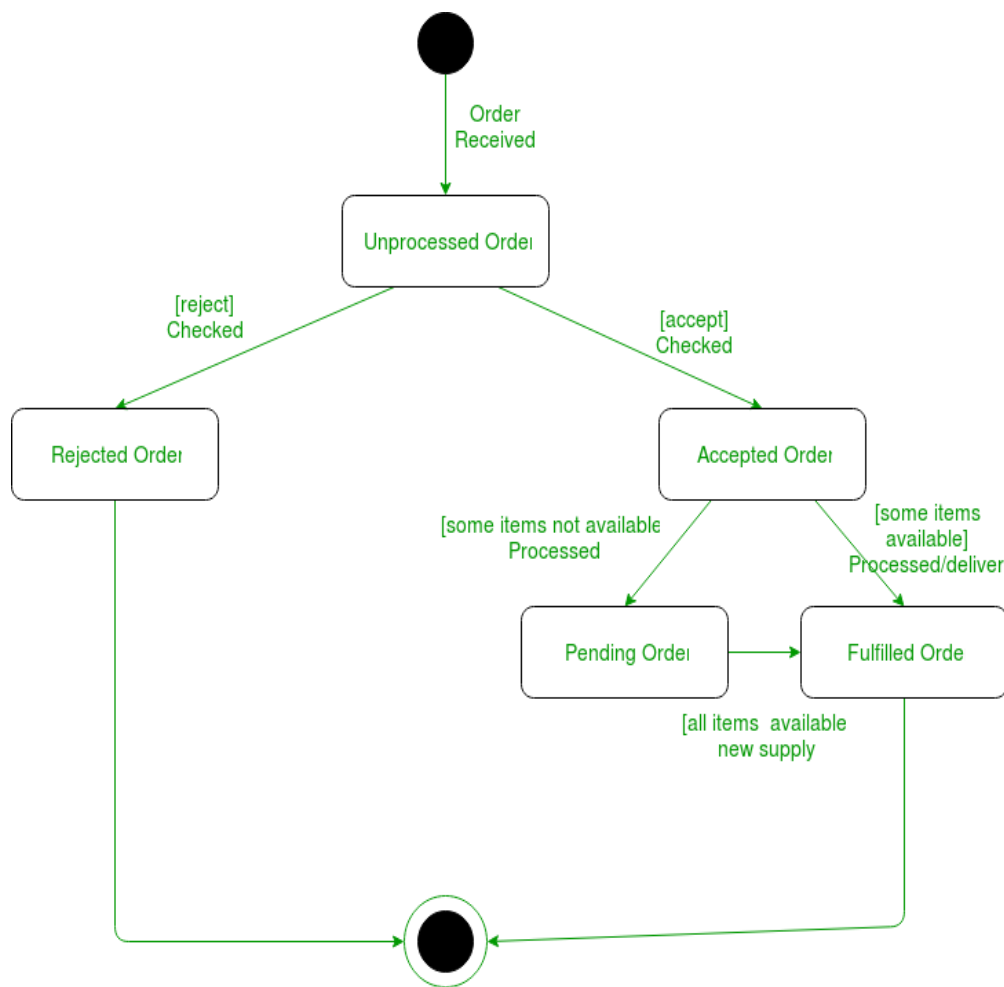


**Figure** – final state notation

#### Steps to draw a state chart diagram –

1. Identify the initial state and the final terminating states.
2. Identify the possible states in which the object can exist (boundary values corresponding to different attributes guide us in identifying different states).
3. Label the events which trigger these transitions.

### Example – state chart diagram for an online ordering system



### 7. Activity Diagram

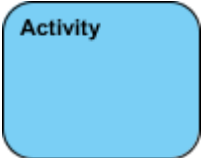


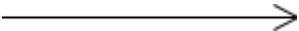


Activity diagram is a special kind of a state chart diagram that shows the flow from activity to activity within a system. These diagrams address dynamic view of a system. Activity Diagram displays a special

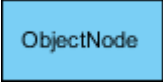


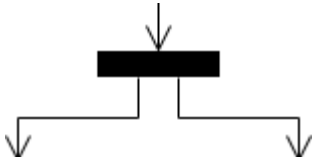
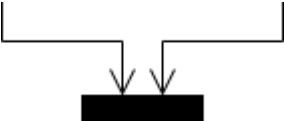
state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. Graphically it is represented as follows:-

The purpose of an activity diagram can be described as –

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

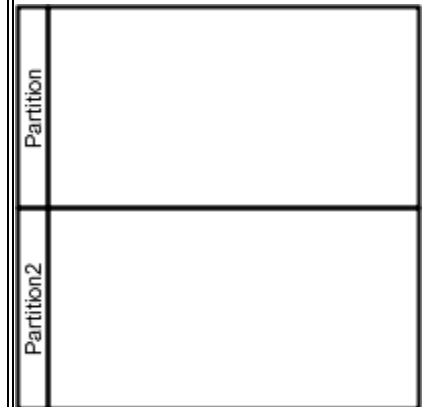
## Activity Diagram Notation Summary

Notation Description	UML Notation
<p><b>Activity</b> Is used to represent a set of actions</p>	
<p><b>Action</b> A task to be performed</p>	
<p><b>Control Flow</b> Shows the sequence of execution</p>	
<p><b>Object Flow</b> Show the flow of an object from one activity (or action) to another activity (or action).</p>	
<p><b>Initial Node</b> Portrays the beginning of a set of actions or activities</p>	
<p><b>Activity Final Node</b> Stop all control flows and object flows in an activity (or action)</p>	

<p><b>Object Node</b></p> <p>Represent an object that is connected to a set of Object Flows</p>	
<p><b>Decision Node</b></p> <p>Represent a test condition to ensure that the control flow or object flow only goes down one path</p>	
<p><b>Merge Node</b></p> <p>Bring back together different decision paths that were created using a decision-node.</p>	
<p><b>Fork Node</b></p> <p>Split behavior into a set of parallel or concurrent flows of activities (or actions)</p>	
<p><b>Join Node</b></p> <p>Bring back together a set of parallel or concurrent flows of activities (or actions).</p>	

### Swimlane and Partition



A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread



## 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

Symbol	Name	Description
	Component symbol	An entity required to execute a stereotype function. A component provides and consumes behavior through interfaces, as well as through other components. Think of components as a <b>type of class</b> . In UML 1.0, a component is modeled as a rectangular block with two smaller rectangles protruding from the side. In UML 2.0, a component is modeled as a rectangular block with a small image of the old component diagram shape.
		



Interface symbol

Shows input or materials that a component either receives or provides. Interfaces can be represented with textual notes or symbols, such as the lollipop, socket, and ball-and-socket shapes.



Port symbol

Specifies a separate interaction point between the component and the environment. Ports are symbolized with a small square.



Package symbol

Groups together multiple elements of the system and is represented by file folders in Lucidchart. Just as file folders group together multiple sheets, packages can be drawn around several components.



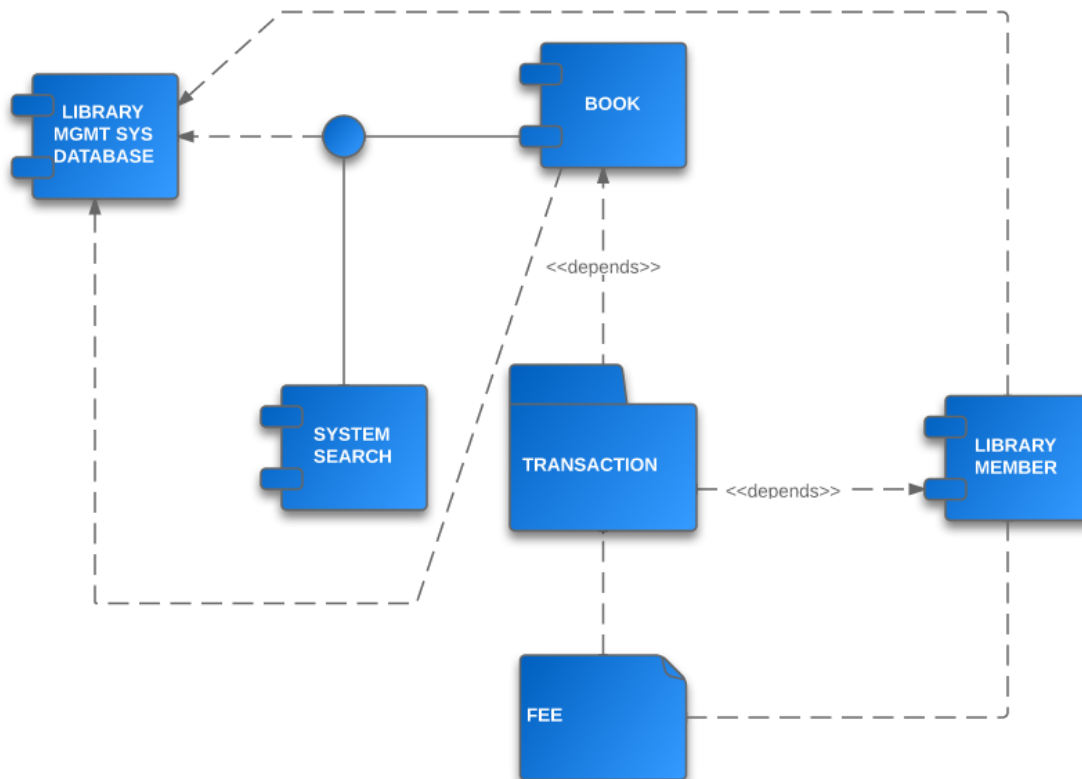
Note symbol

Allows developers to affix a meta-analysis to the component diagram.



Dependency symbol

Shows that one part of your system depends on another. Dependencies are represented by dashed lines linking one component (or element) to another.



Example: Component Diagram of Library Management System.

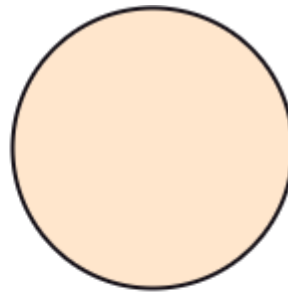
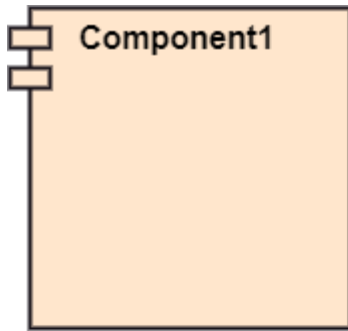
## 8. Deployment Diagram

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.

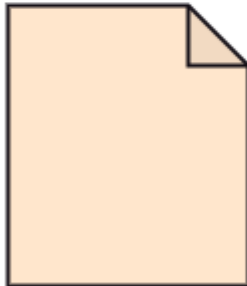
The deployment diagram consist of the following notations:

1. A component
2. An artifact
3. An interface
4. A node

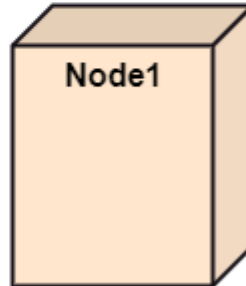




Interface1



Artifact1



Node1

### How to draw a Deployment Diagram?

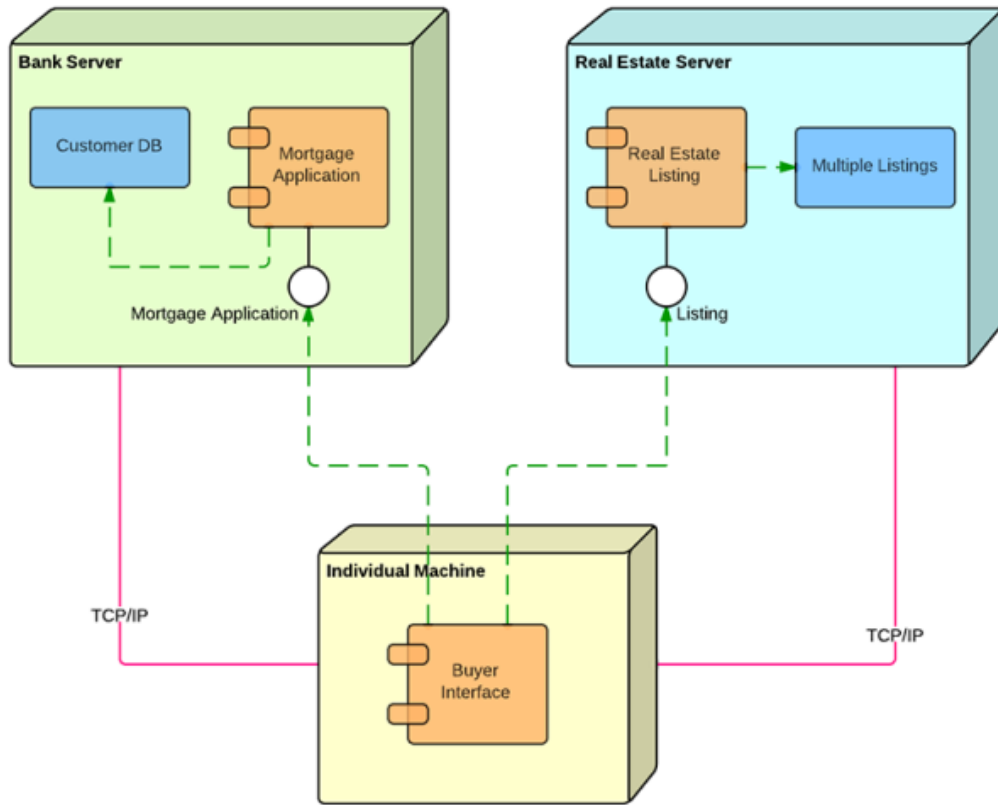
The deployment diagram portrays the deployment view of the system. It helps in visualizing the topological view of a system. It incorporates nodes, which are physical hardware. The nodes are used to execute the artifacts. The instances of artifacts can be deployed on the instances of nodes.

Since it plays a critical role during the administrative process, it involves the following parameters:

1. High performance
2. Scalability
3. Maintainability
4. Portability
5. Easily understandable

One of the essential elements of the deployment diagram is the nodes and artifacts. So it is necessary to identify all of the nodes and the relationship between them. It becomes easier to develop a deployment diagram if all of the nodes, artifacts, and their relationship is already known.

- Dependency
- Association relationships.
- May also contain notes and constraints.



**Deployment Diagram for banking system**

# Case Study 1: 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 any two accounts linked to the card.
4. A customer must be able to make a balance inquiry of any account linked to the card.
5. A customer must be able to abort a transaction in progress by pressing 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-enter the PIN 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 to contact the bank to get it back.

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the 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 the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

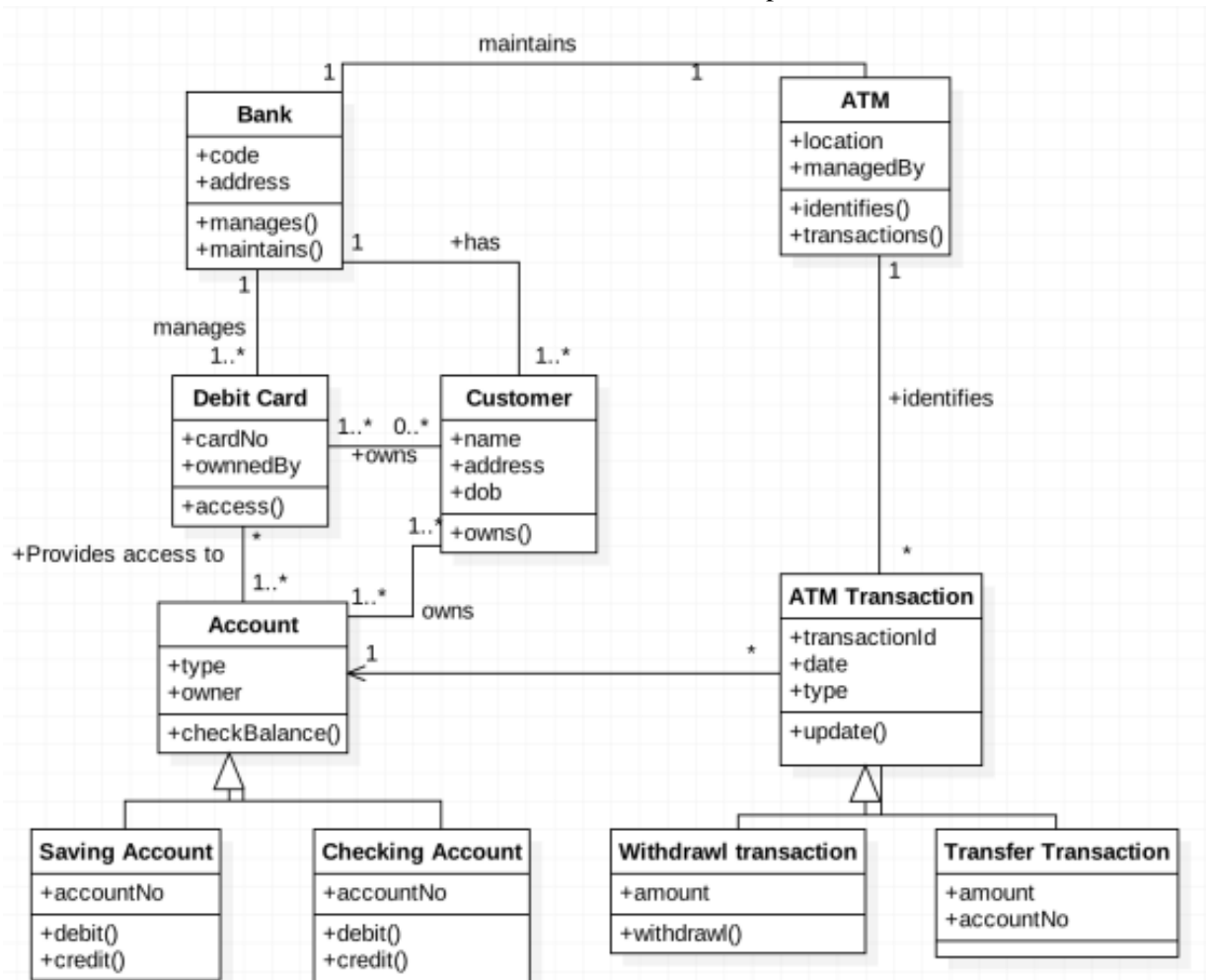
The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. 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

# CLASS DIAGRAM

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

Class Diagram:- Class diagrams describe the static structure of a system, or how it is structured rather than how it behaves. These diagrams contain the following elements:

1. Classes , which represent entities with common characteristics or features. These features include attributes, operations, and associations.
2. Associations , which represent relationships that relate two or more other classes where the relationships have common characteristics or features. These features include attributes and operations.



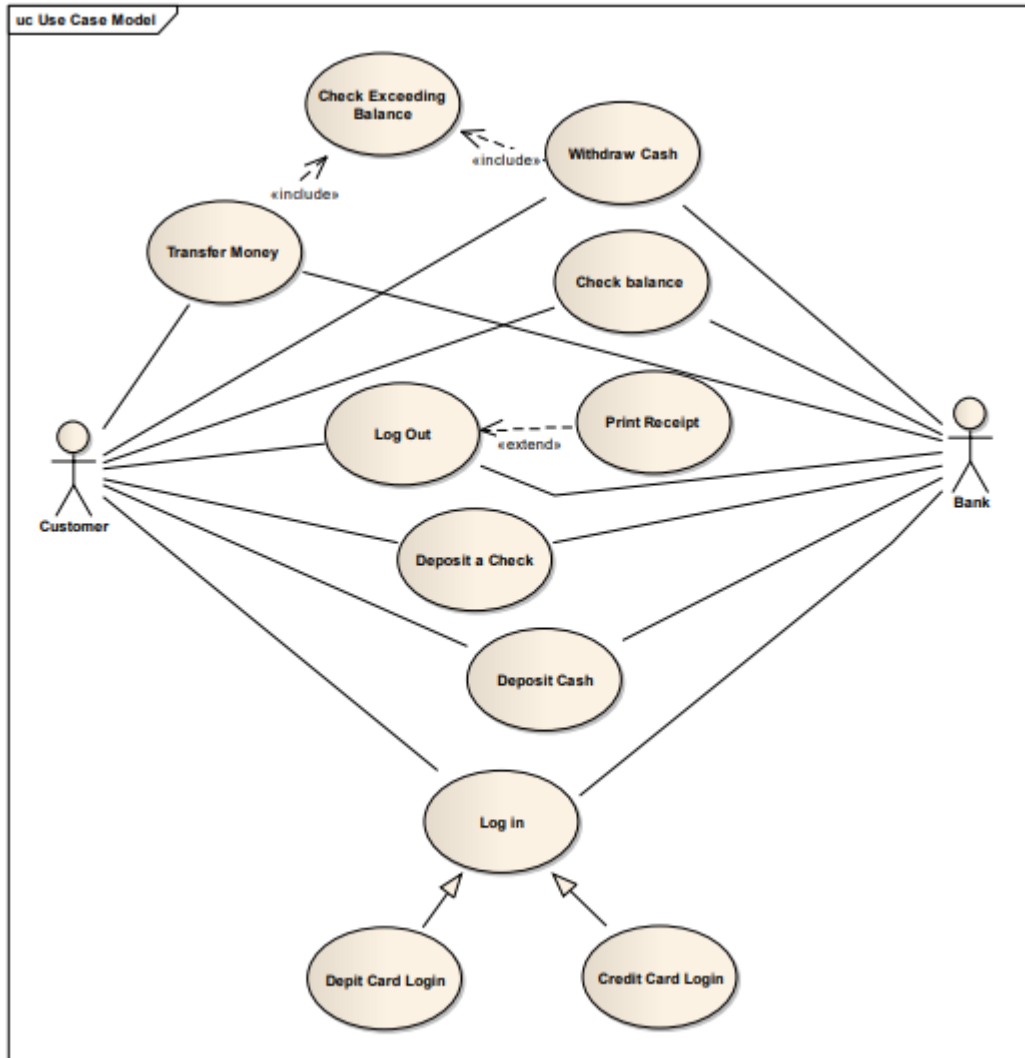
### USE CASE DIAGRAM

For Function: Use case, Sequence, Collaboration/Communcation

Use Case Diagram: Use case diagrams describe the functionality of a system and users of the system.

They contain the following elements:

1. Actors , which represent users of a system, including human users and other systems
2. Use cases , which represent functionality or services provided by a system to users Here, is a use case diagram for the ATM System.



## **INTERACTIONDIAGRAMS**

We have two types of interaction diagrams in UML. One is sequence diagram and the other is a collaboration diagram. The sequence diagram captures the time sequence of message flow from one object 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 control  
Messages  
Life line

### **Contents of a Collaboration Diagram**

Objects Links Messages

## Sequence Diagram:

Sequence diagrams typically show the flow of functionality through a use case, and consist of the following components:

1. Actors , involved in the functionality
2. Objects , that a system needs to provide the functionality
3. Messages , which represent communication between objects Here, is an example of Sequence diagram for withdrawing amount from ATM.

### Procedure for drawing Sequence Diagram

**Step1:** First An actor is created and named as user or Customer

**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.

### Withdrawal UseCase

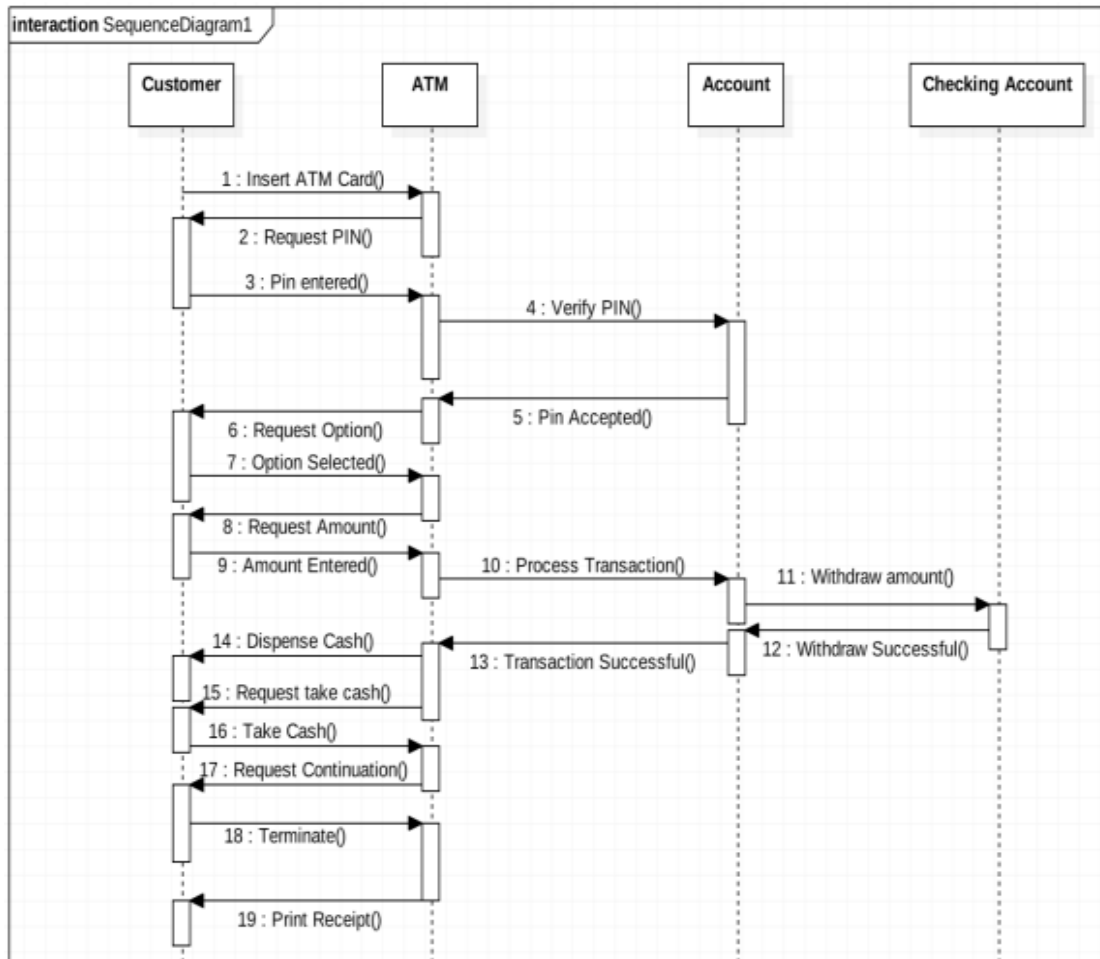
A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose an amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction 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.

## SEQUENCE DIAGRAM:

Sequence diagrams typically show the flow of functionality through a use case, and consist of the following components:

1. Actors , involved in the functionality
2. Objects , that a system needs to provide the functionality
3. Messages , which represent communication between objects Here, is an example of Sequence diagram for withdrawing amount from ATM





**Fig: Sequence diagram for Withdrawal UseCase**

### Deposit UseCase

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 to the 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.

## **Inquiry Use Case**

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.

## **Validate User Usecase:**

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.

## **Print Bill usecase**

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

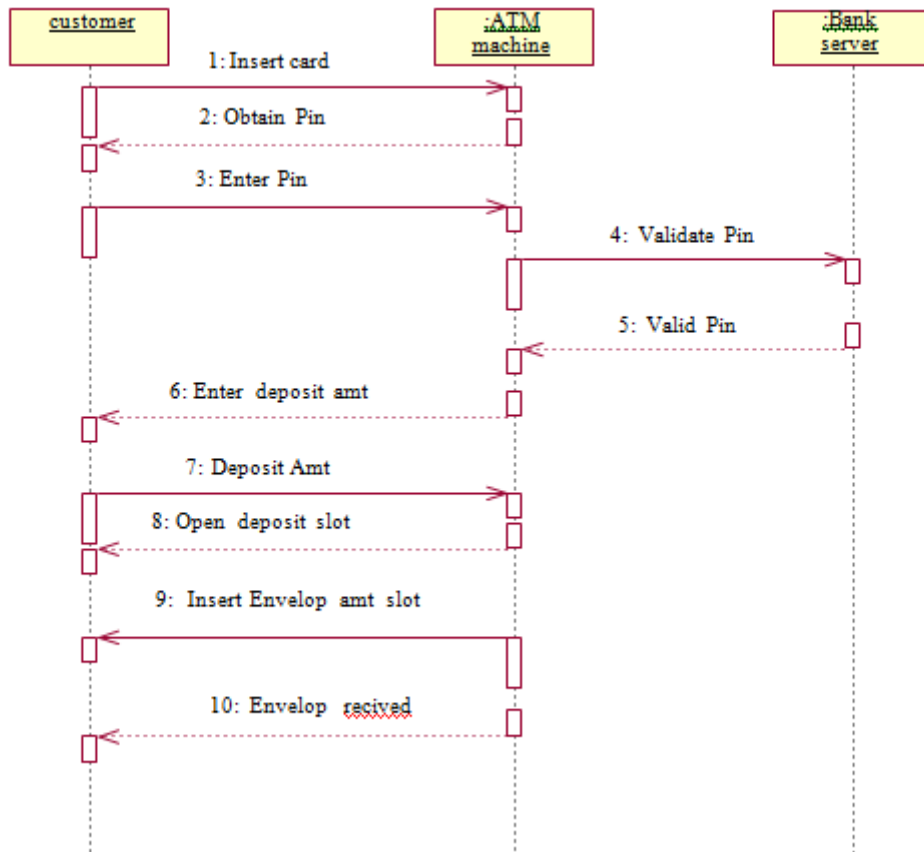
## **Manage Account**

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

## **RESULT:**

### **Inferences:**

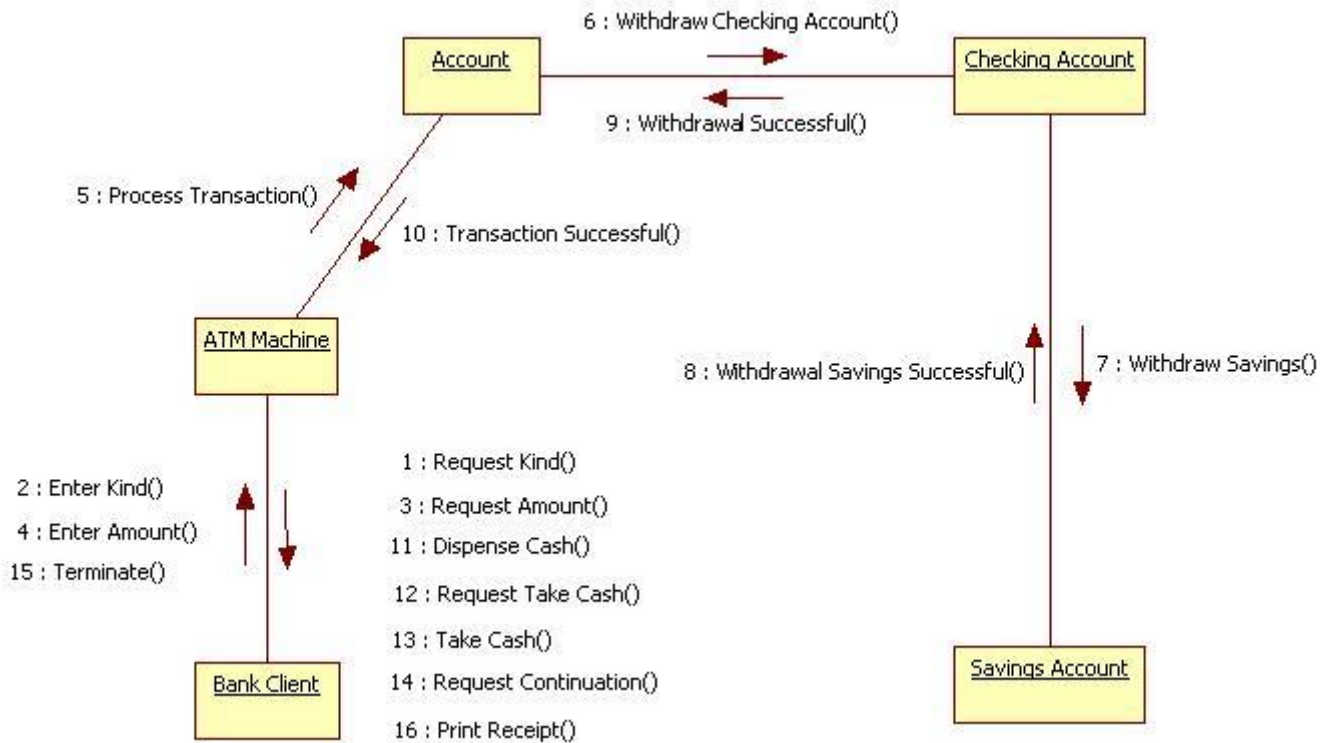
2. Identification of use cases.
3. Identification of actors.



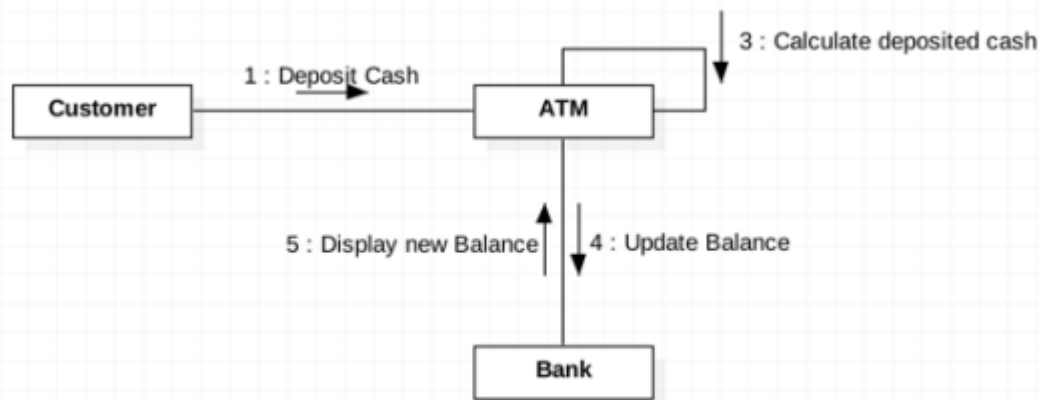
Sequence Diagram for Deposit Money or Cash Use Case

**Collaboration Diagram for ATM**

A Communication or Collaboration diagram, as shown is a directed graph that uses objects and actors as graph nodes. The focus of the collaboration diagram is on the roles of the objects as they interact to realize a system function. Directional links are used to indicate communication between objects. These links are labeled using appropriate messages. Each message is prefixed with a sequence number indicating the time ordering needed to realize the system function

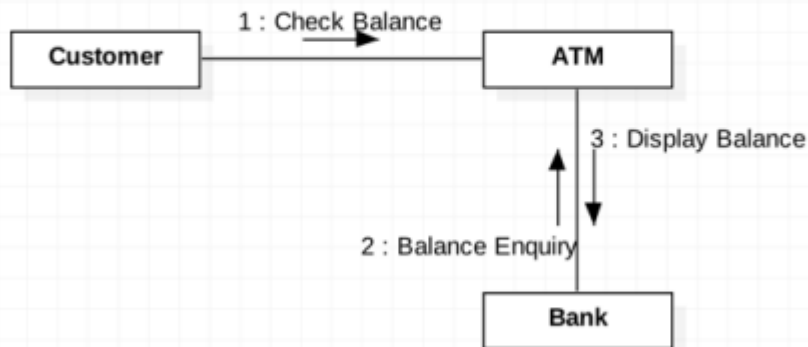


**Collaboration Diagram for Withdraw Money**



**Example for Cash Deposit or Deposit Money**

Here is an example of the Check Balance communication diagram:



### State Chart and Activity Diagram

For behavior: State, Activity Diagram State Diagram:- State transition diagrams provide a way to model the various states in which an object can exist. While the class diagram show a static picture of the classes and their relationships, state transition diagrams model the dynamic behavior of a system in response to external events (stimuli).

State transition diagrams consist of the following:

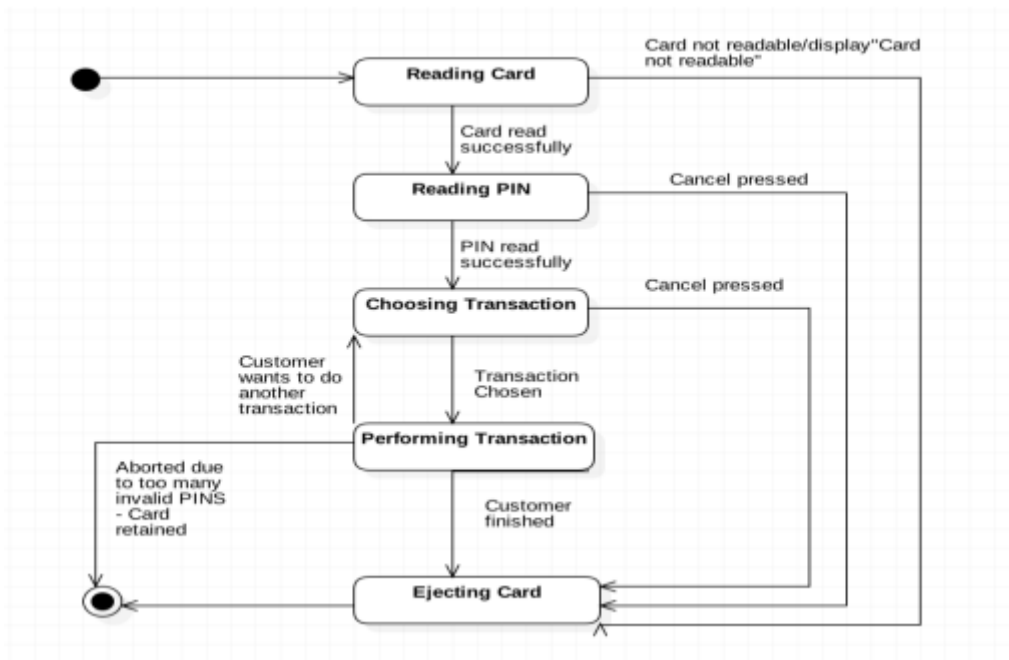
1. States , which show the possible situations in which an object can find itself
2. Transitions , which show the different events which cause a change in the state of an object.

Here, is an example of the state diagram for the session of ATM

### 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.



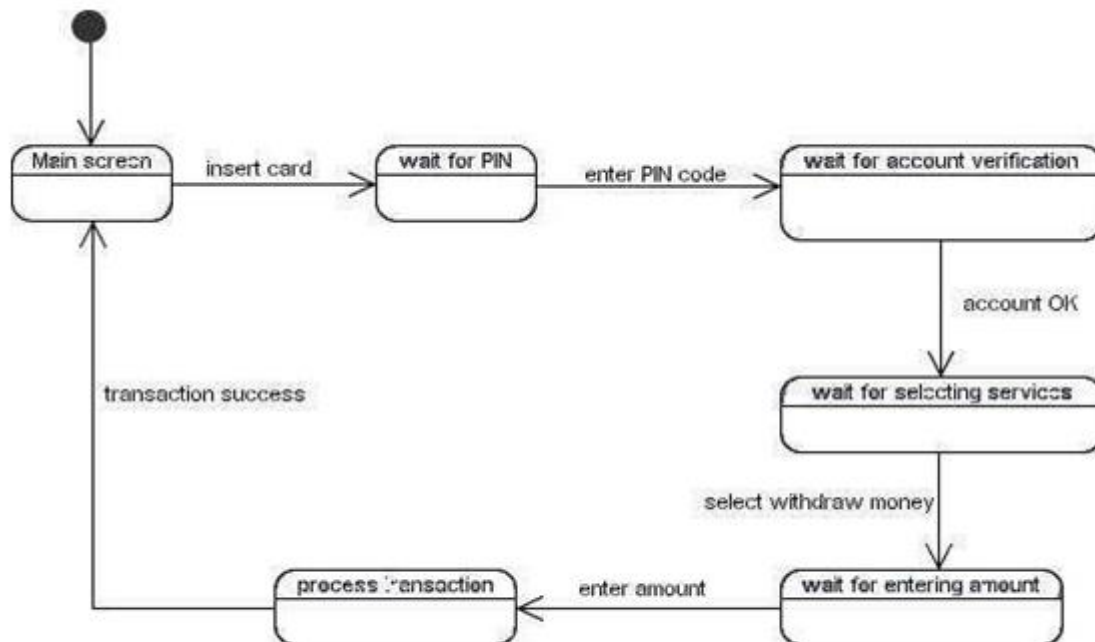
**Procedure:-**

**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 states as shown in the fig.



# Activity Diagram

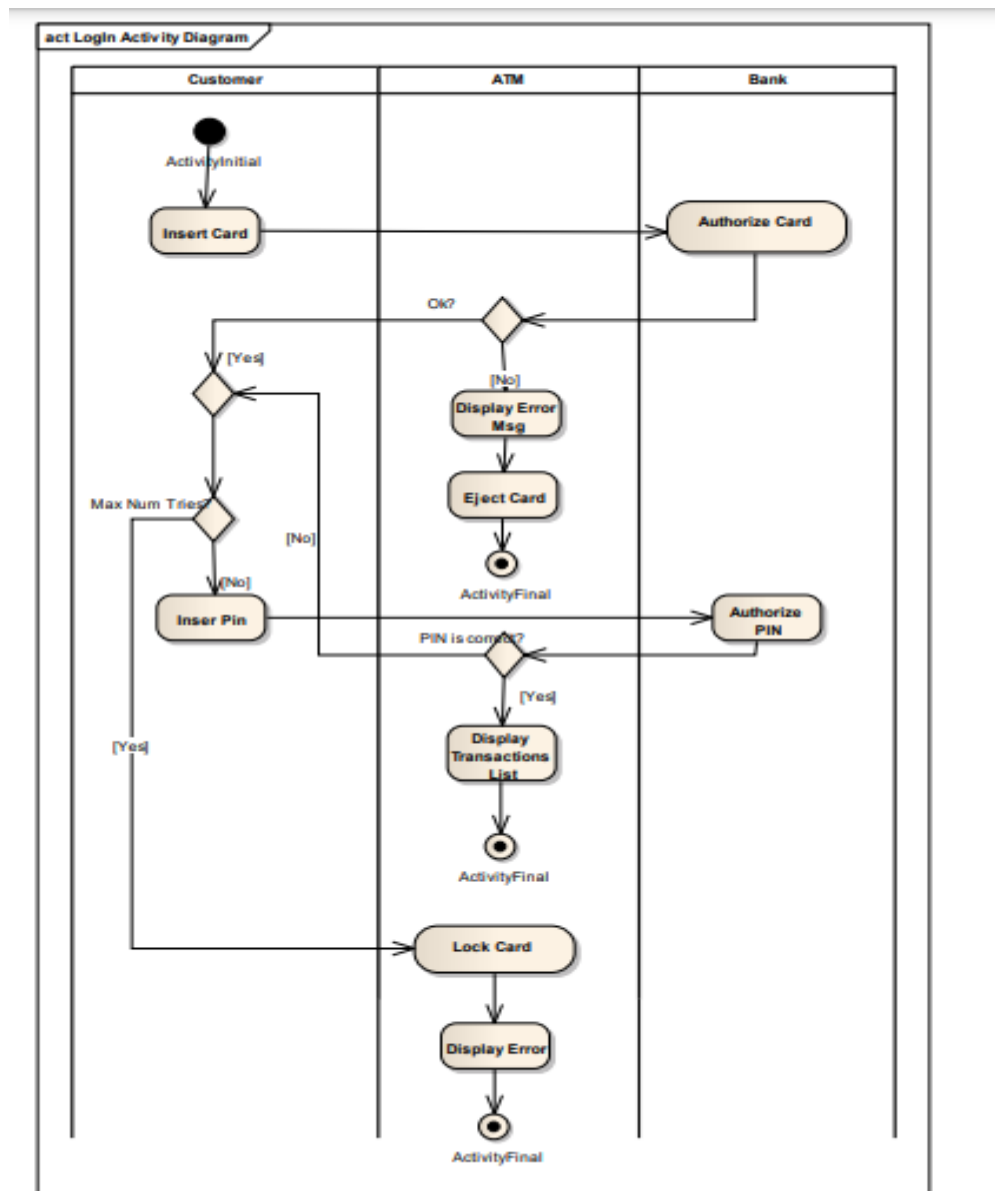
Activity Diagram:- Activity diagrams describe the activities of a class. They are similar to state transition diagrams and use similar conventions, but activity diagrams describe the behavior/states of a class in response to internal processing rather than external events.

They contain the following elements:

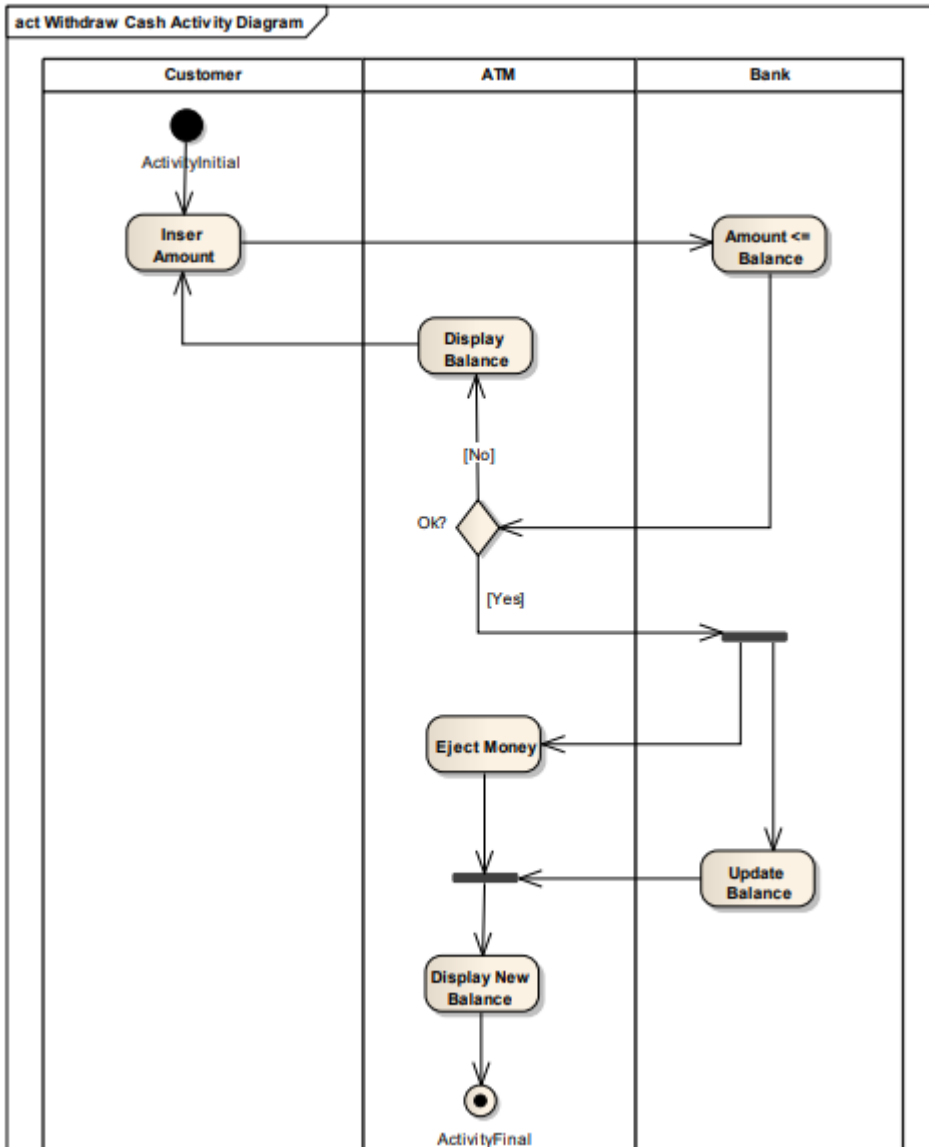
1. Swimlanes , which delegate specific actions to objects within an overall activity
2. Action States , which represent uninterruptible actions of entities, or steps in the execution of an algorithm
3. Action Flows , which represent relationships between the different action states on an entity
4. Object Flows , which represent utilization of objects by action states, or influence of action states on objects.

Following are the examples of Login, Withdraw Activity Diagrams.

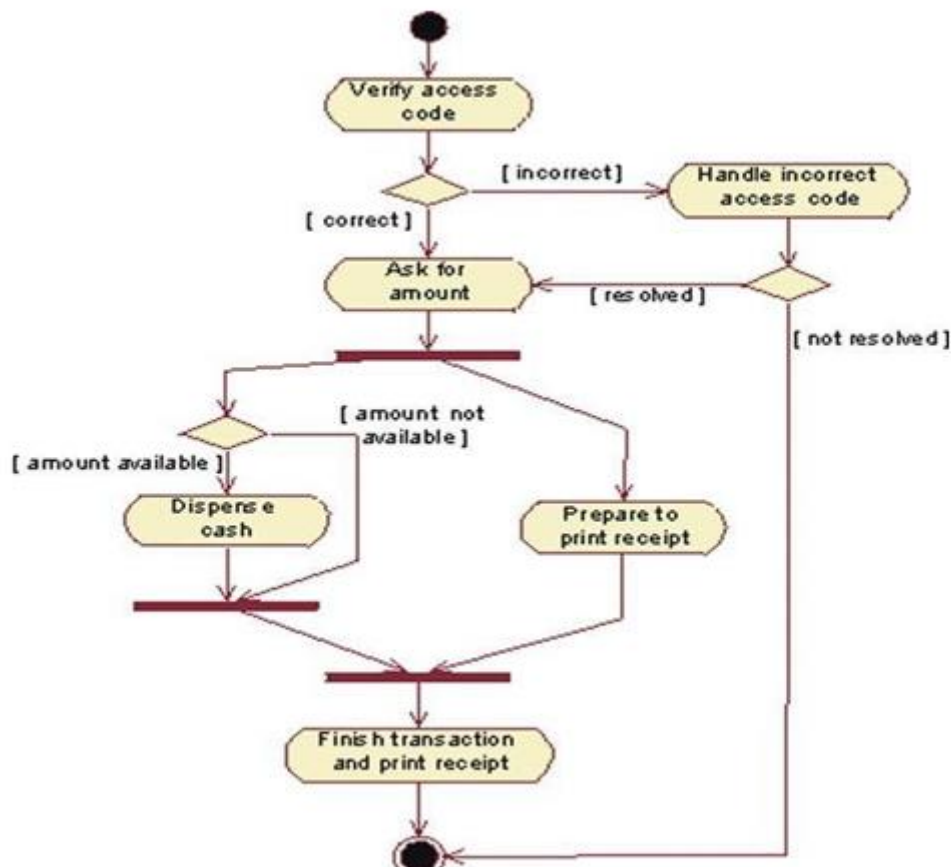
## Activity Diagram for Login into ATM using card



Activity Diagram for withdraw money from ATM Machine







Activity diagram for Withdraw:

## COMPONENT DIAGRAM for ATM System.

To design and implement Component diagram for ATM System.

### Component Diagram

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, transfermoney etc. are created.

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

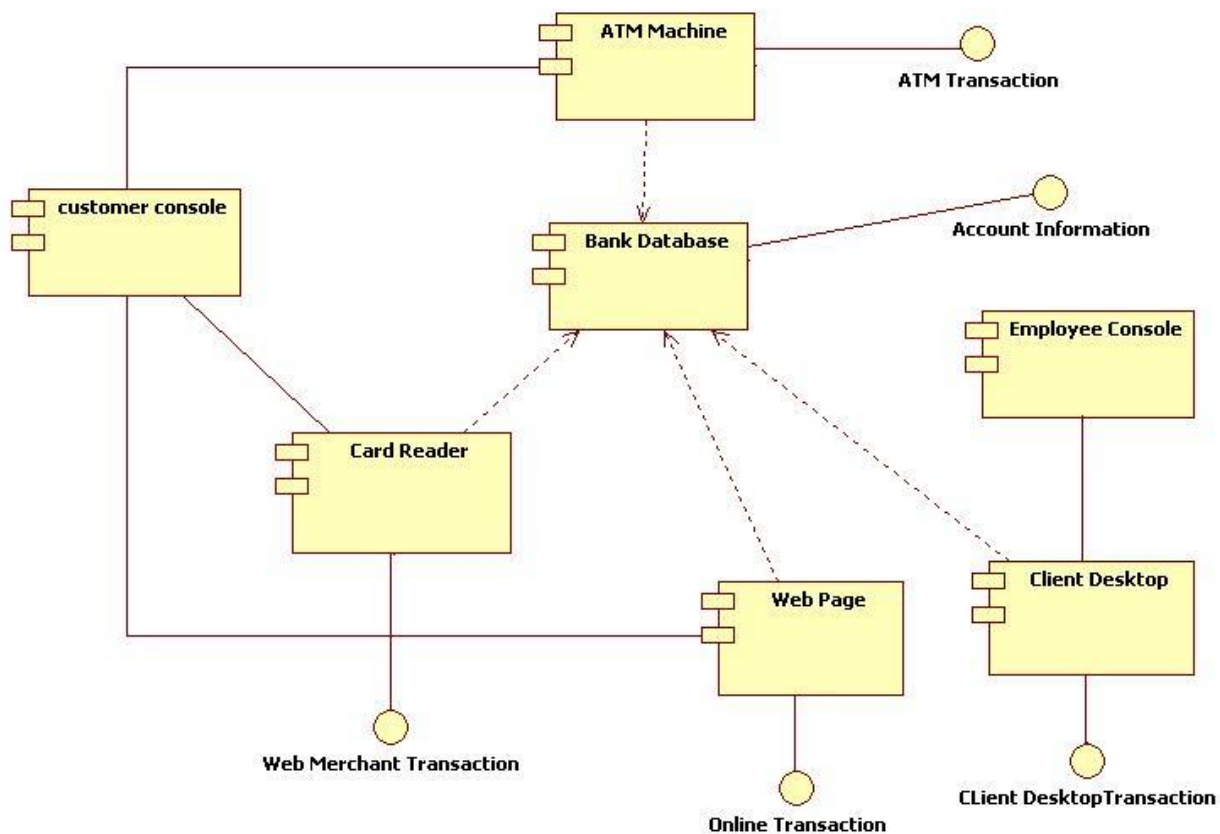


Fig: Component Diagram for ATM

# DEPLOYMENT DIAGRAM

**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

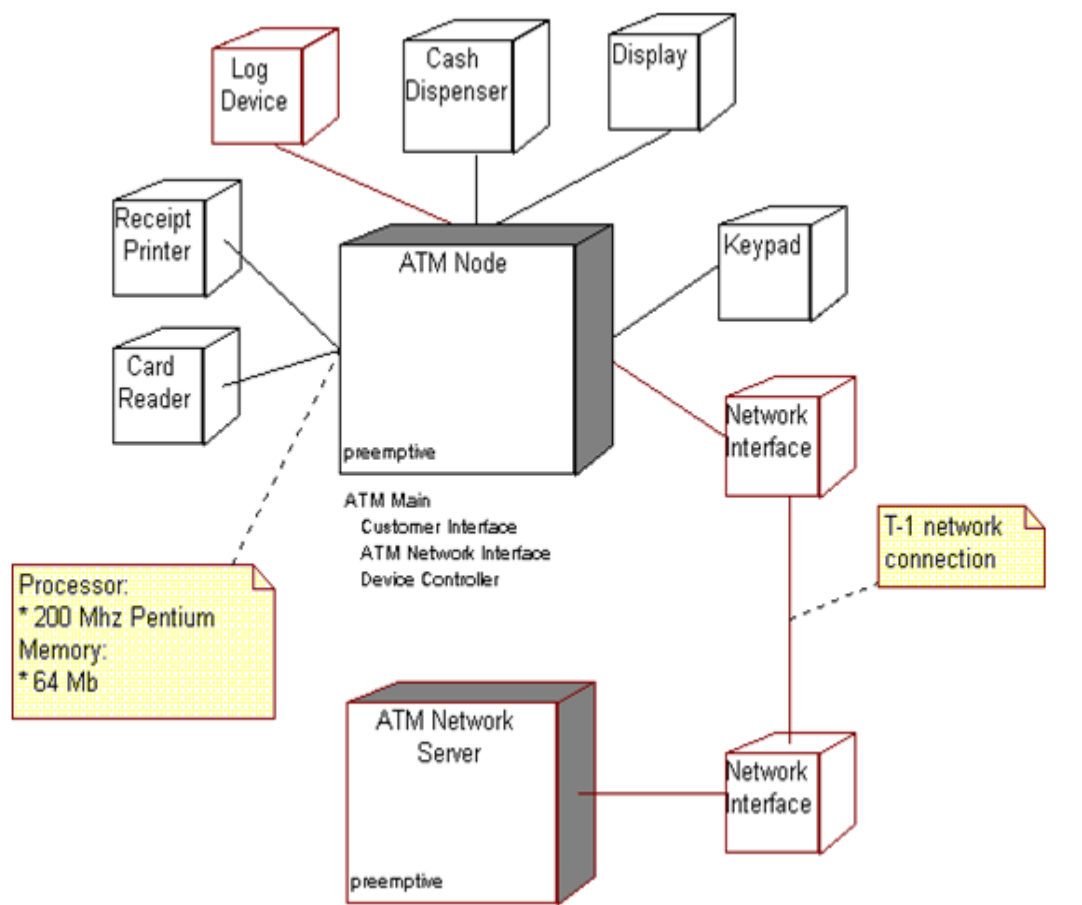
## Procedure:-

**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.



Deployment Diagram for ATM system

## CASE STUDY 2: LIBRARY MANAGEMENT SYSTEM

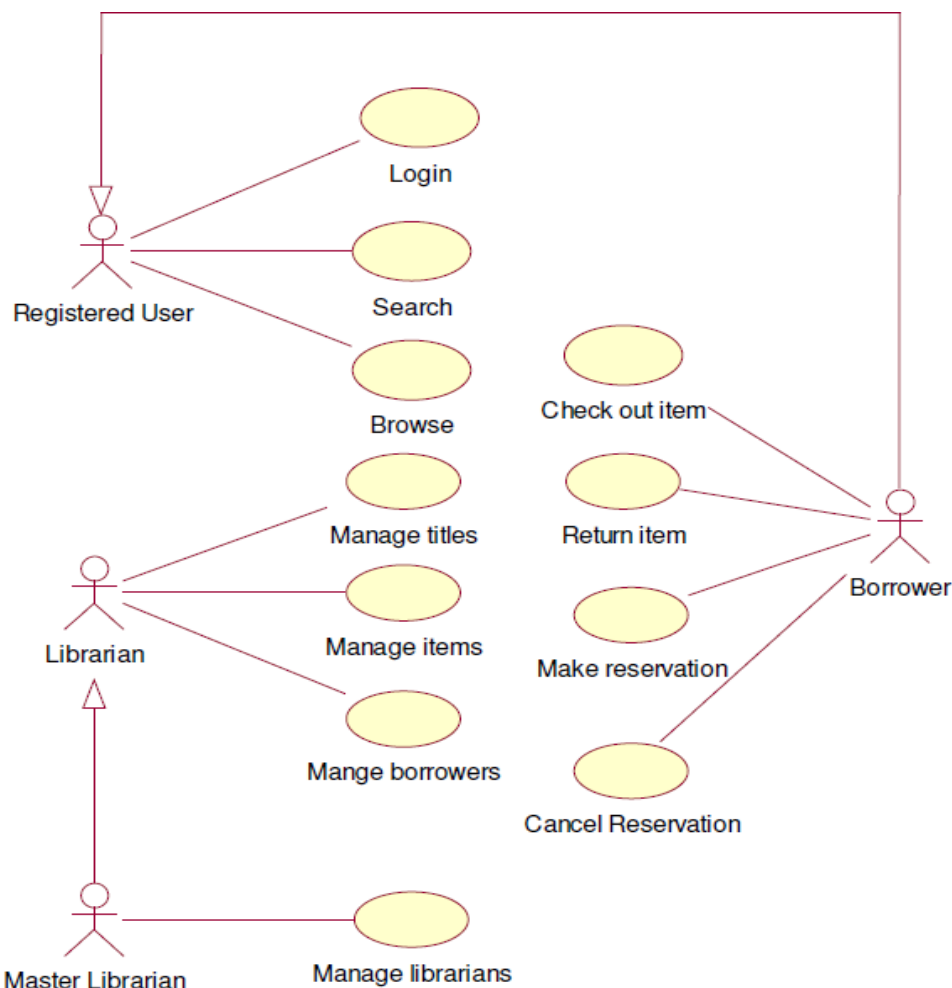
The Library Management application will help to manage the issuing of books, students, etc. It allows the library owner to manage the day-to-day process of a book issuing conveniently.

### Users of Library Management System

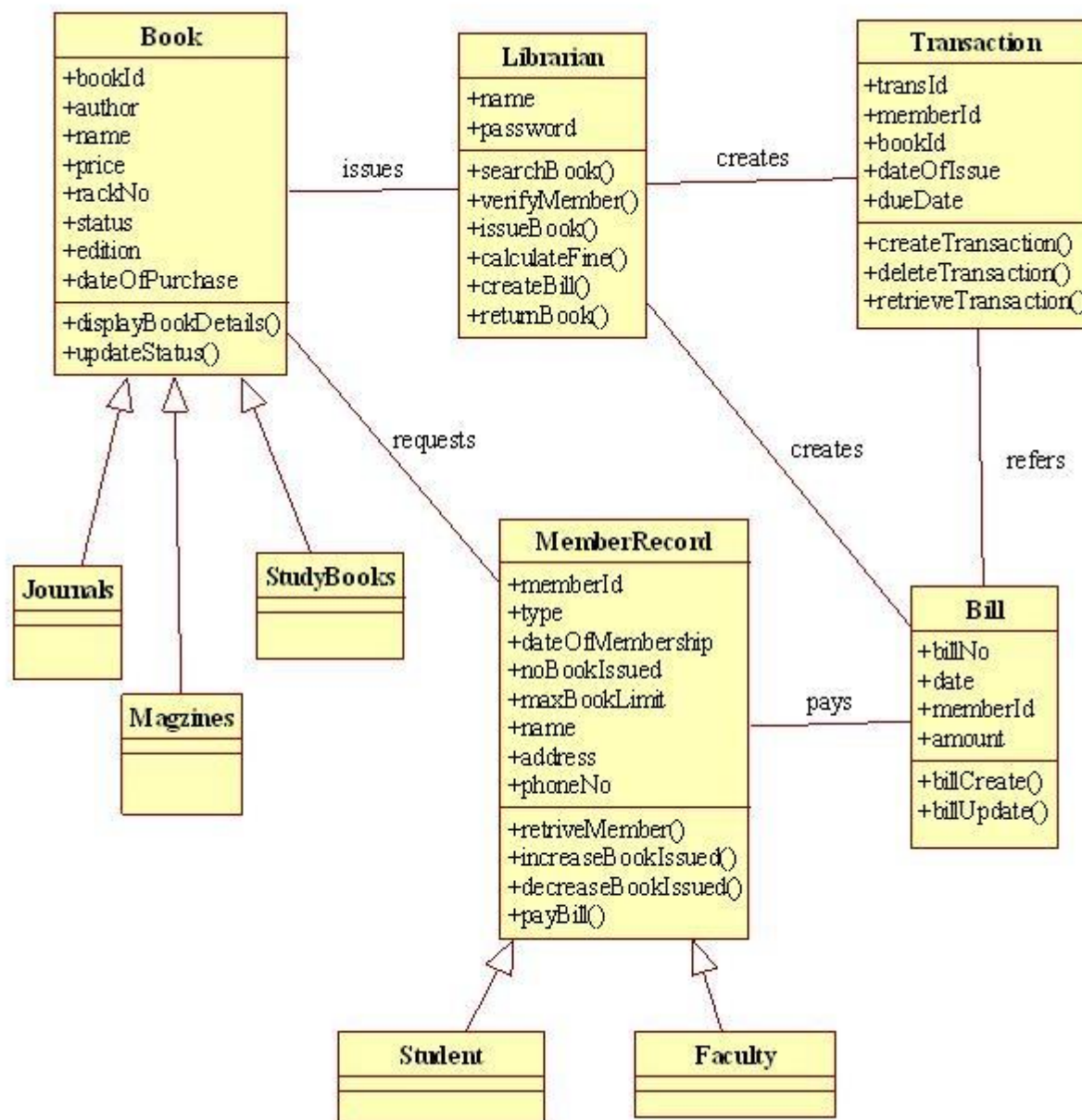
The users of the College Library Management System Sequence Diagram are the following:

- **Librarian:** The school librarians will be the ones to use the system most of the time. They will monitor the books from time to time and will cater to the borrowing and returning of books. They were also responsible for all the activities related to the library.
- **Book Borrowers:** Book borrowers were not just the students but also the professors or instructors. They will also have access to the system and to do that, they will have to log into the system. This will help the librarian and the admin monitor the book borrowers.
- **Admin:** The Library Management System can be a stand-alone project or a part of a bigger project. Nevertheless, it always has the admin which can access all of the library information. This is done when there are serious scenarios or problems.

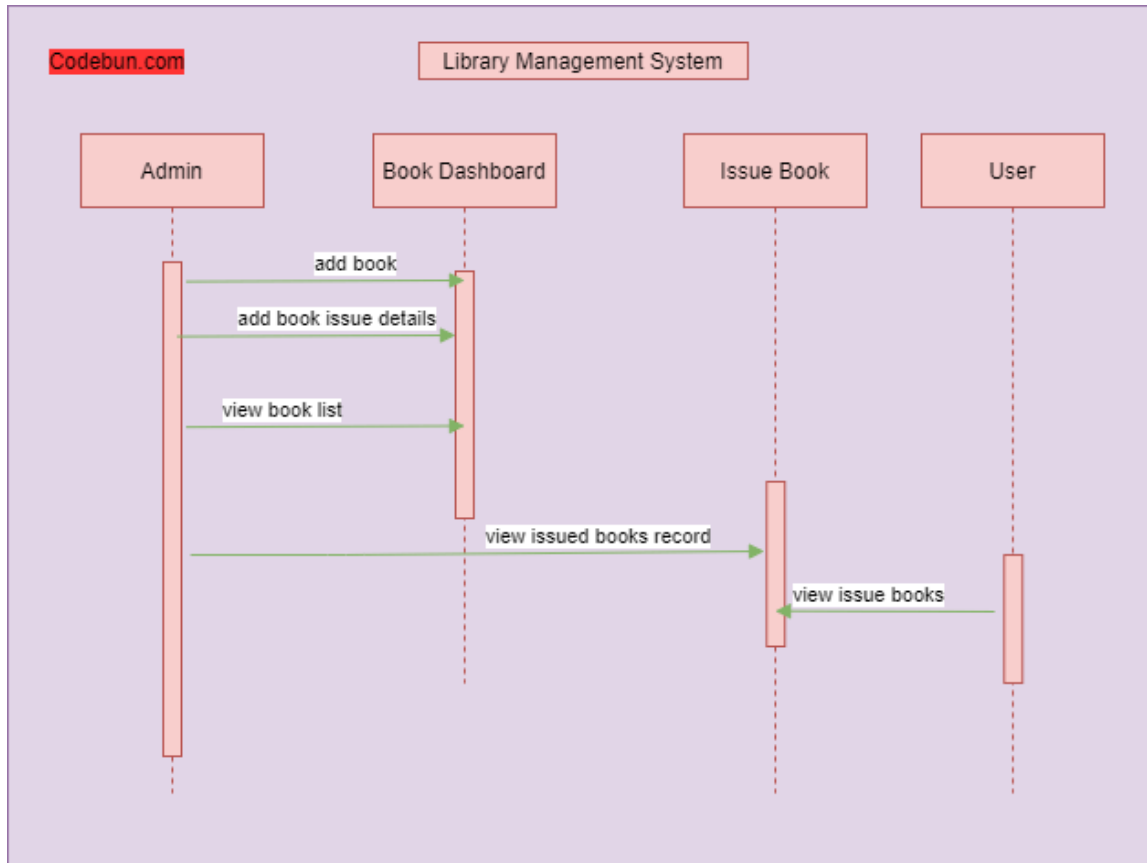
### Use Case Diagram



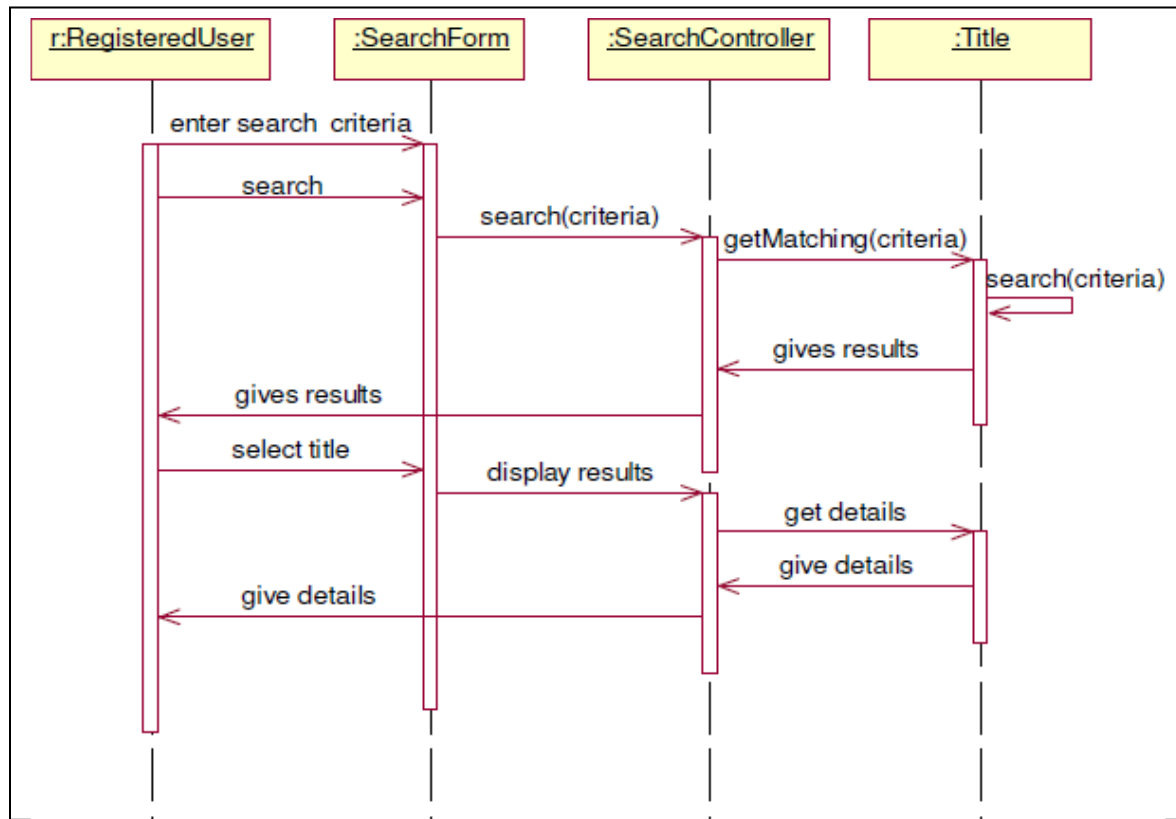
Class Diagram



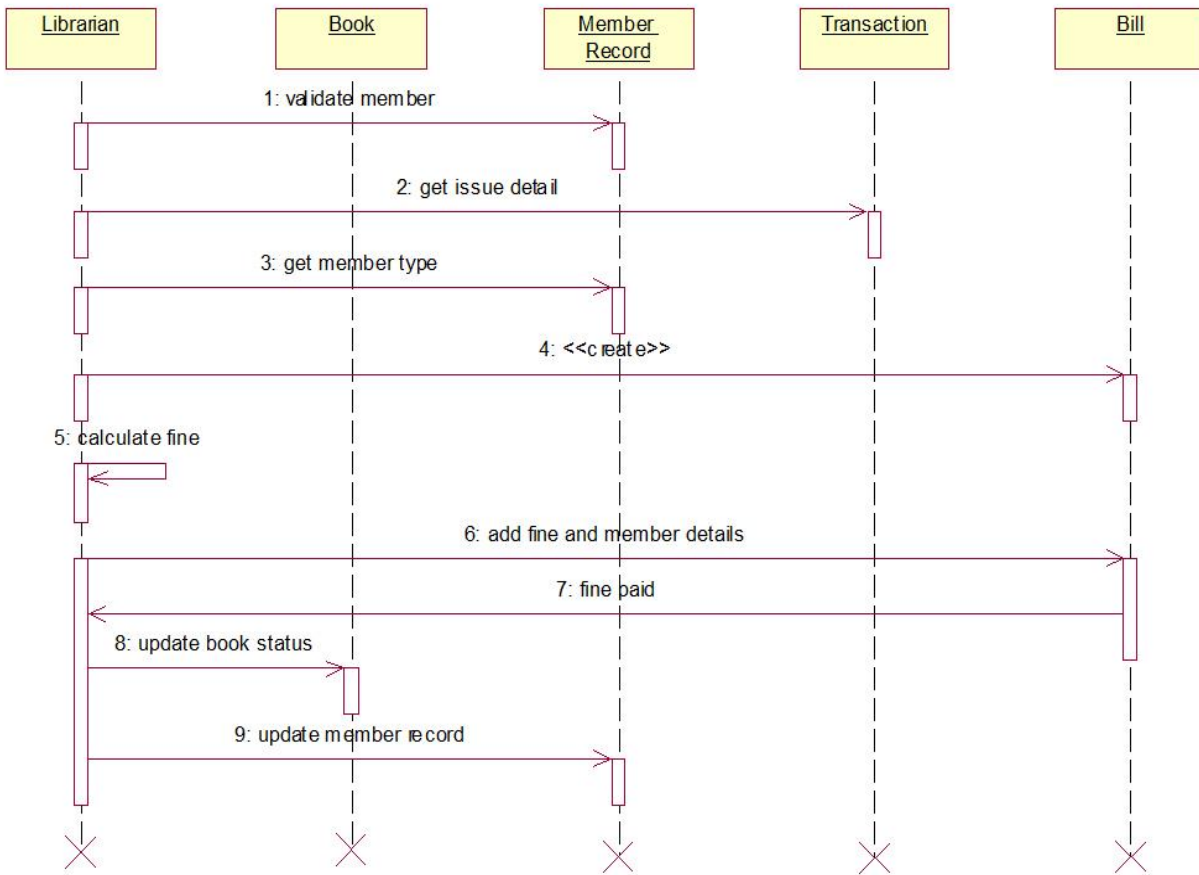
### Sequence Diagram for adding Book



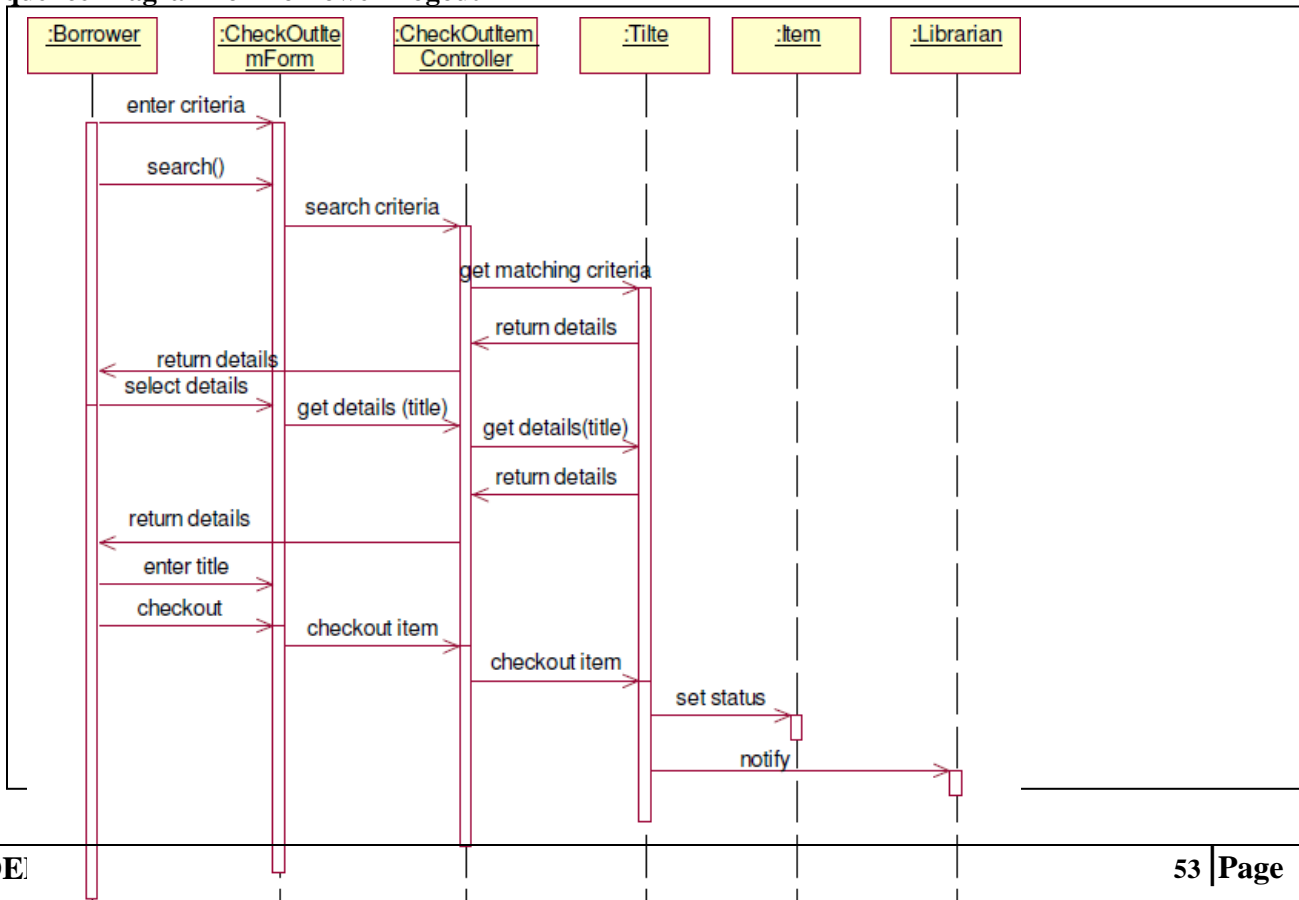
### Sequence diagram for checking or availability of books



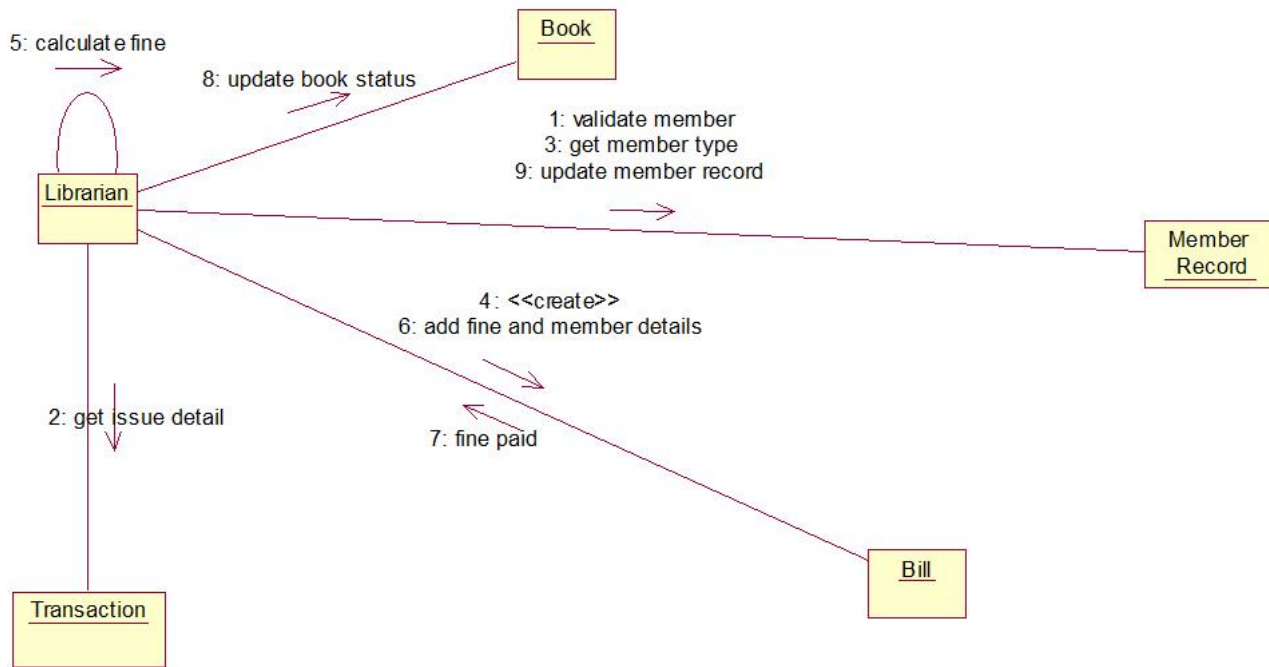
### Sequence diagram for calculating fine for the book returned after due date



### Sequence Diagram for Borrower Logout

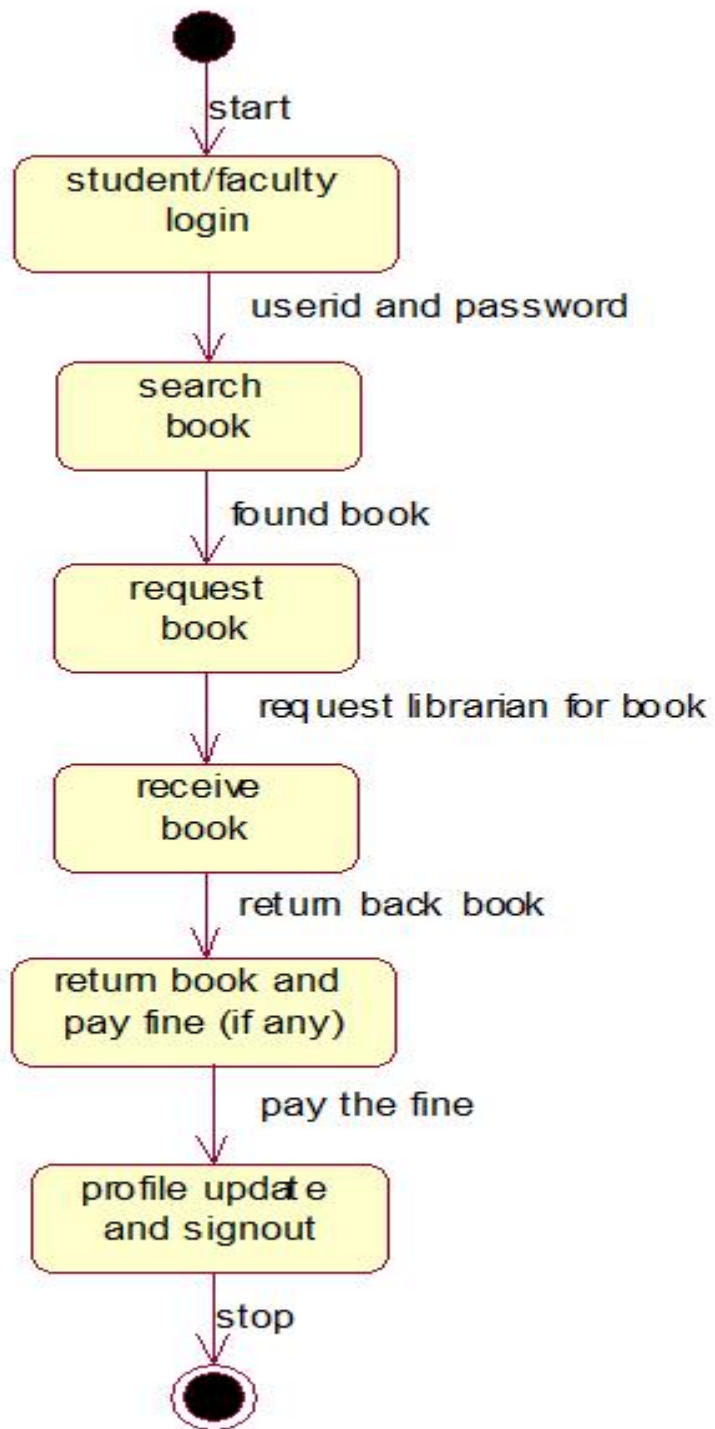


# Collaboration Diagram

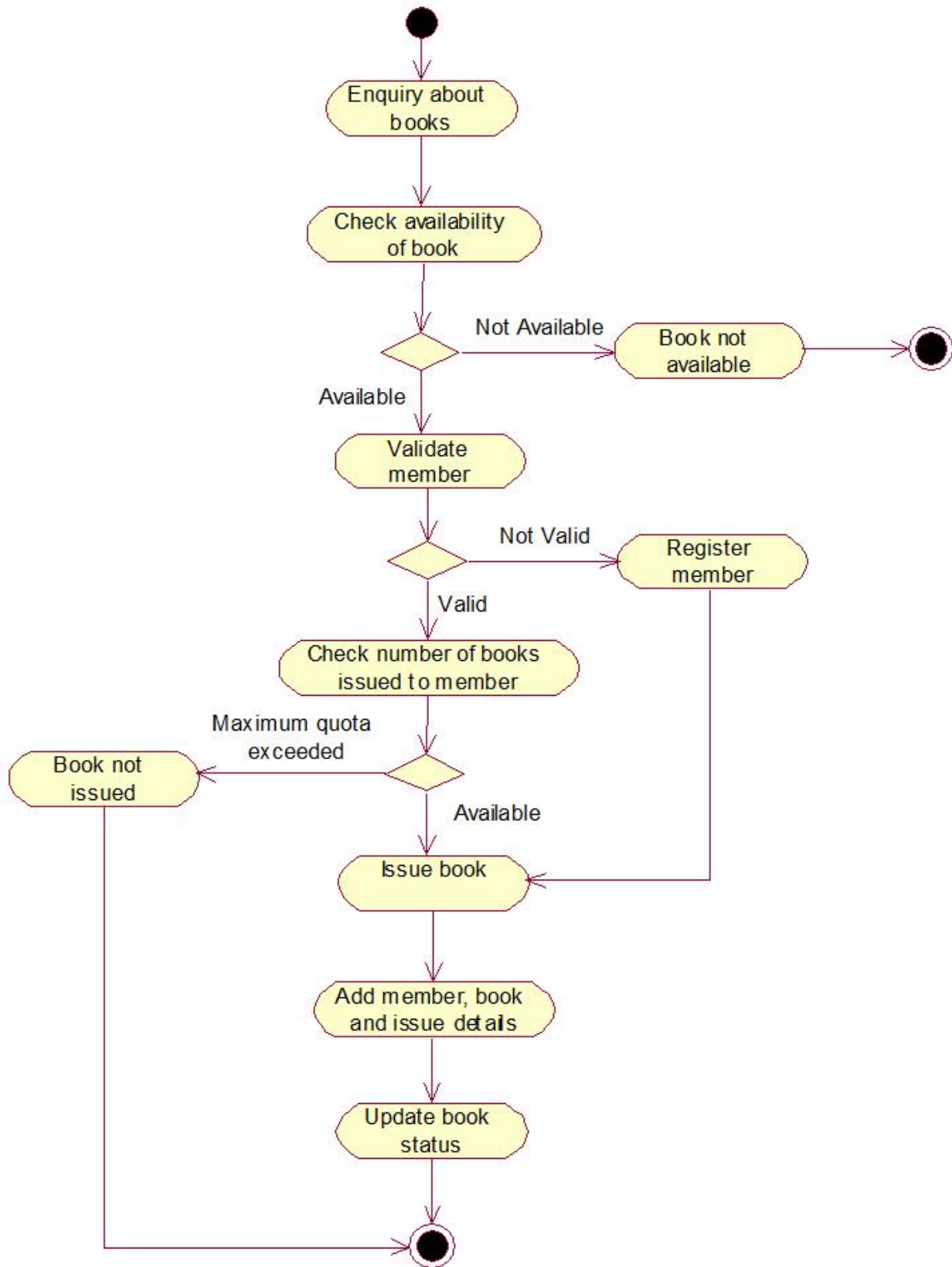




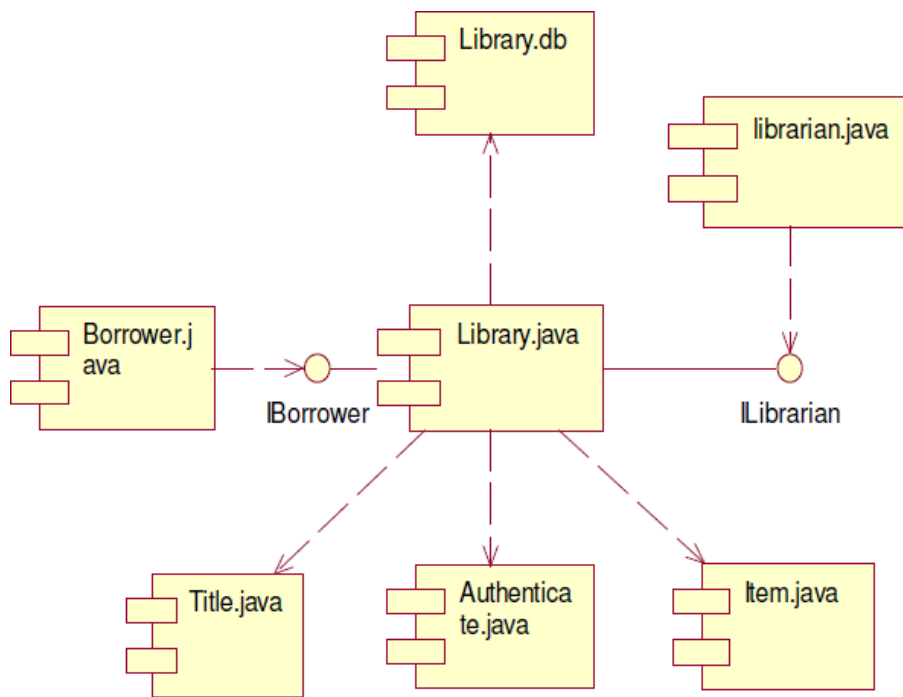
## State Chart Diagram



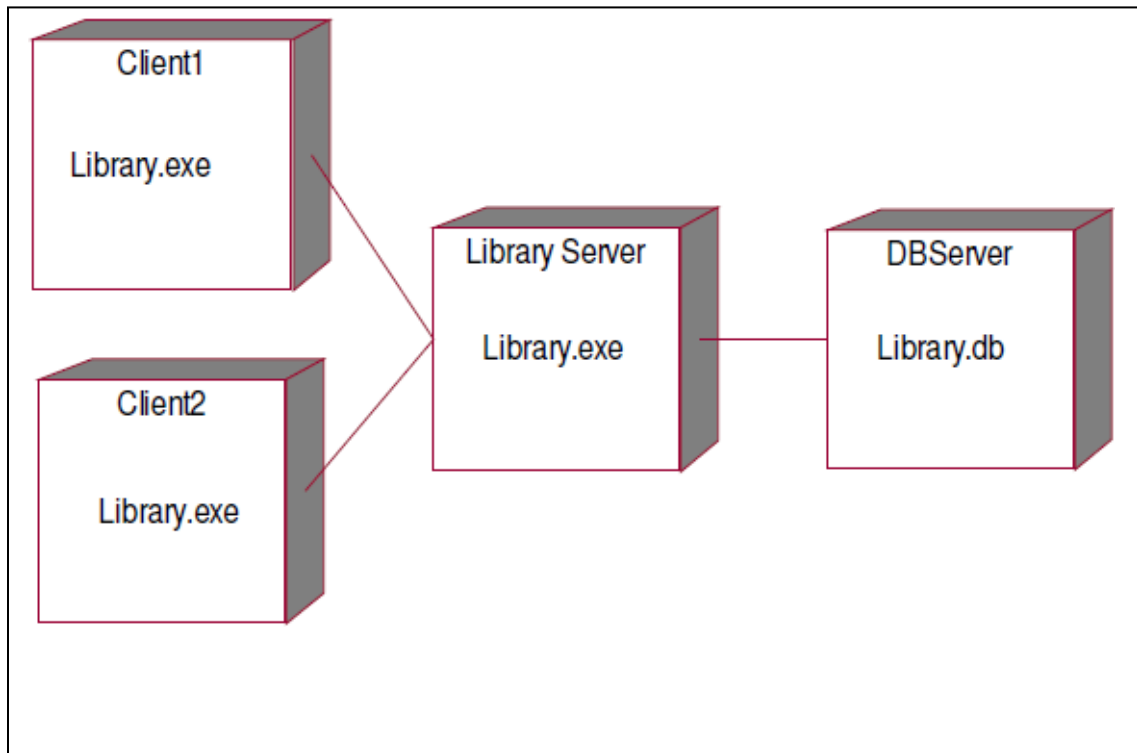
### Activity Diagram for checking and Issuing the book



### Component Diagram

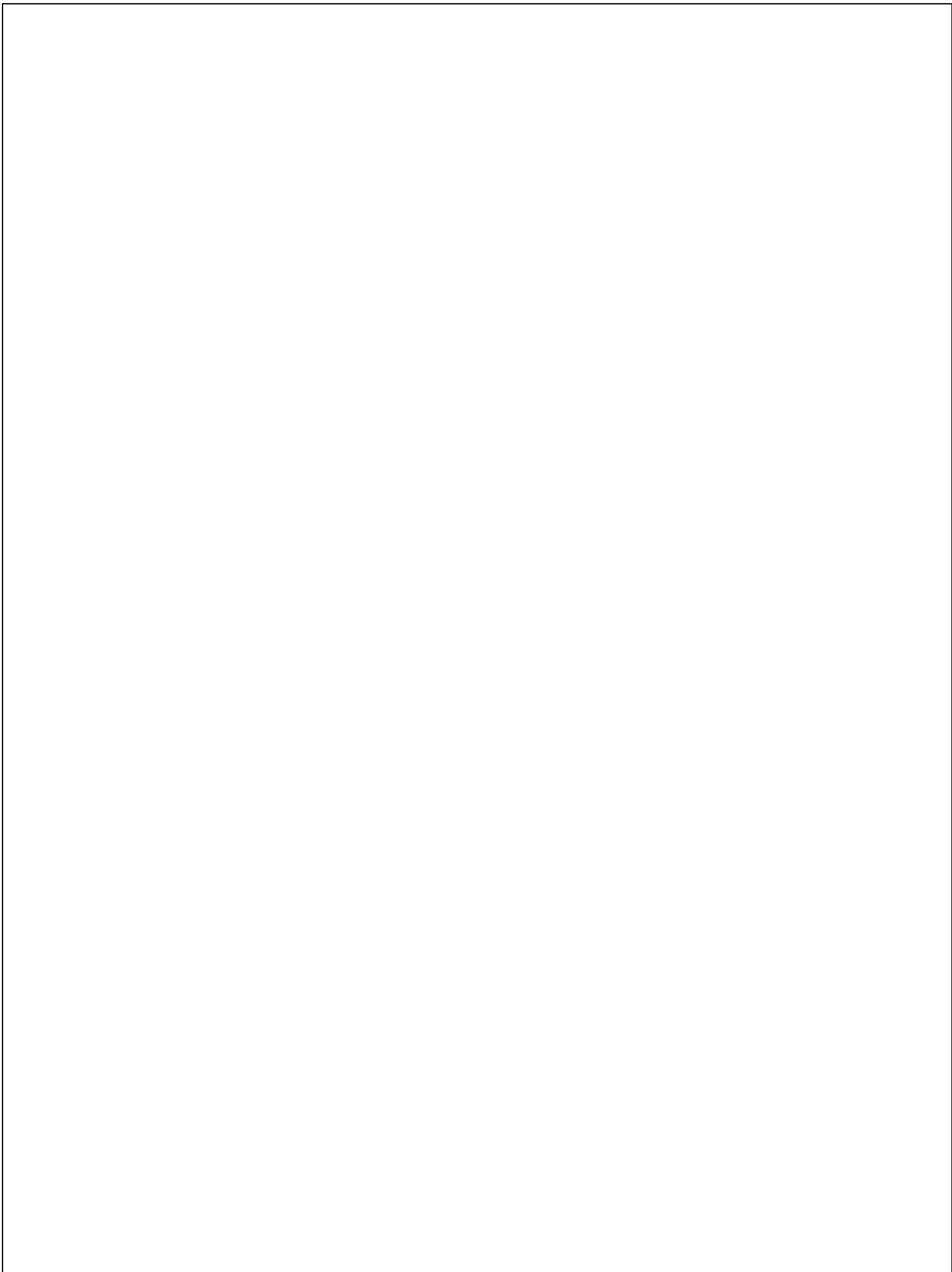


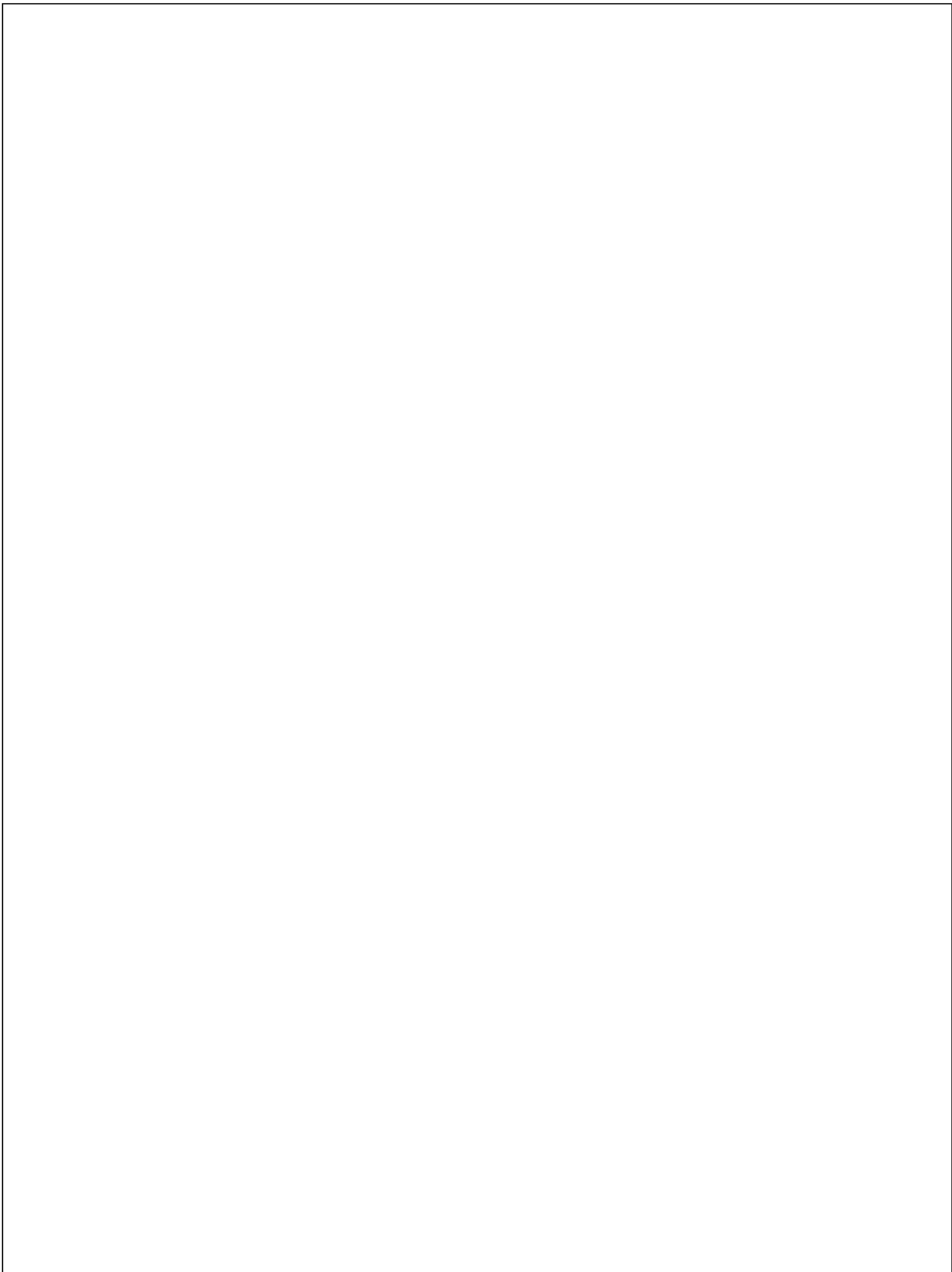
### Deployment Diagram

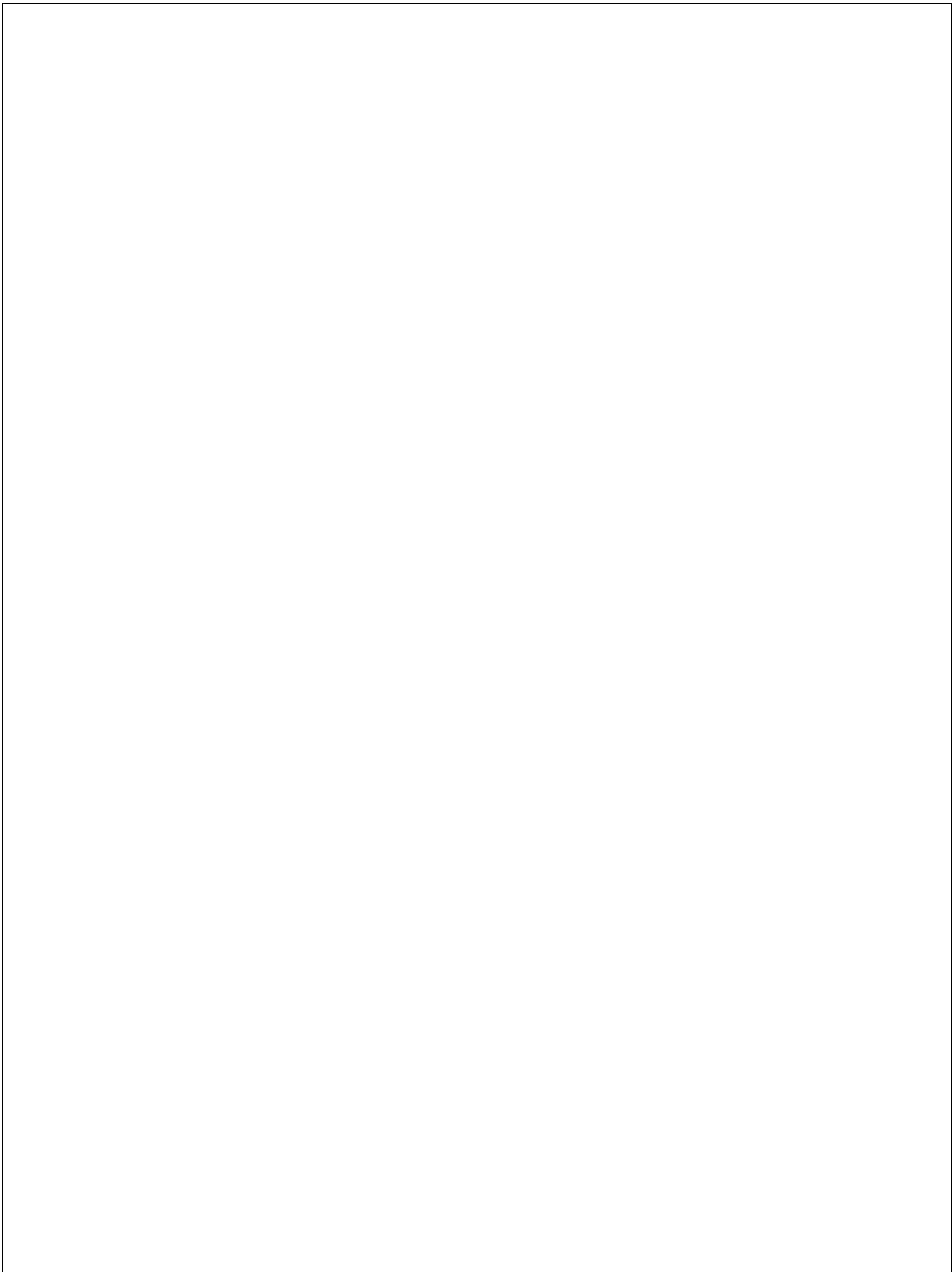


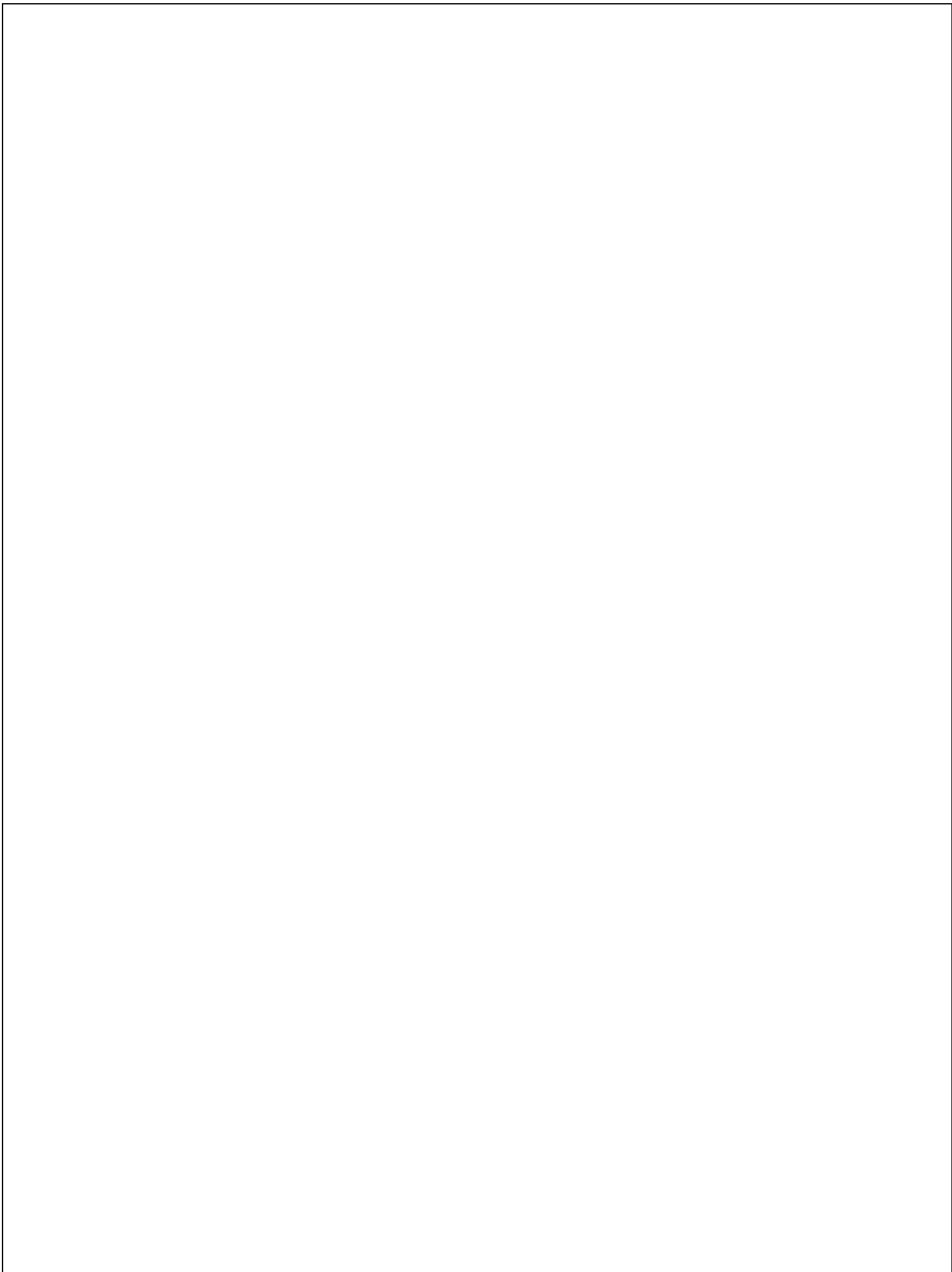
# **Case Study 3:**

## **College Administration System**











**Signature of the Faculty**