

INTRODUCTION TO OOAD & Argo UML

What is OOAD ?

Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state (data elements), and its behavior. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

Object-Oriented analysis:

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Implementation constraints are dealt during object-oriented design (OOD).

Object-oriented design:

Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.

ArgoUML:

ArgoUML is an UML diagramming application written in Java and released under the open source Eclipse Public License. By virtue of being a Java application, it is available on any platform supported by Java SE.

ArgoUML is different: i) it makes use of ideas from cognitive psychology, ii) it is based on open standards; iii) it is 100% pure Java; and iv) it is an open source project.

EXP NO: 1**DATE:** _____**AIM:**

To initial familiarization of ArgoUML

ARGO UML:

ArgoUML is an open source Unified Modeling Language (UML) modeling tool that includes support for all standard UML 1.4 diagrams. It runs on any Java platform and is available in ten languages.

FEATURES:

- Support open standards extensively: UML, XMI, SVG, OCL and others.
- 100% Platform independent thanks to the exclusive use of Java
- Open Source, which allows extending or customizing.
- Cognitive features like
 - reflection-in-action
 - Design Critics
 - Corrective Automations (partially implemented)
 - "To Do" List
 - User model (partially implemented)
 - opportunistic design
 - "To Do" List
 - Checklists
 - Comprehension and Problem Solving
 - Explorer Perspectives
 - Multiple, Overlapping Views
 - Alternative Design Representations: Graphs, Text, or Table

Supported Diagrams by ArgoUML:

- Use Case Diagrams
- Class Diagrams
- Behavior Diagrams
- State chart Diagrams
- Activity Diagrams
- Interaction Diagrams: Sequence Diagrams, Collaboration Diagrams
- Implementation Diagrams
- Component Diagrams
- Deployment Diagrams

USE CASE DIAGRAMS:

Present a high-level view of system usage

- These diagrams show the functionality of a system or a class and how the system interacts with the outside world.
- During analysis to capture the system requirements and to understand how the system should work.

- During the design phase, use-case diagrams specify the behavior of the system as implemented.

CLASS DIAGRAM:

- Helps you visualize the structural or static view of a system.
- Class diagrams show the relationships among class.
- Foundation for component and deployment diagrams.

SEQUENCE DIAGRAM

- Illustrates object interacts arranged in a time sequence
- This shows step-by-step what has to happen to accomplish something in the use case and emphasize the sequence of events.

COLLABORATION DIAGRAM

- Provides a view of the interactions or structured relationships between objects in current model.
- Emphasizes relation between objects.
- Used as the primary vehicle to describe interactions that express decision about system behavior.

STATE CHART DIAGRAM

- To model the dynamic behaviors of individual classes or objects
- Used to model the discrete stages of an objects lifetime.

ACTIVITY DIAGRAM

- Model the workflow of a business process and the sequence of activities in a process.
- Similar to a flowchart, it a workflow from activity to activity or from activity to state.
- It is help to understand the overall process.

COMPONENT DIAGRAM

A physical view of the current model and Show the organization and dependencies of software components, including source code, binary code, and executable components

DEPLOYMENT DIAGRAM

Each model contains a single deployment diagram that shows the mapping of processes to hardware.

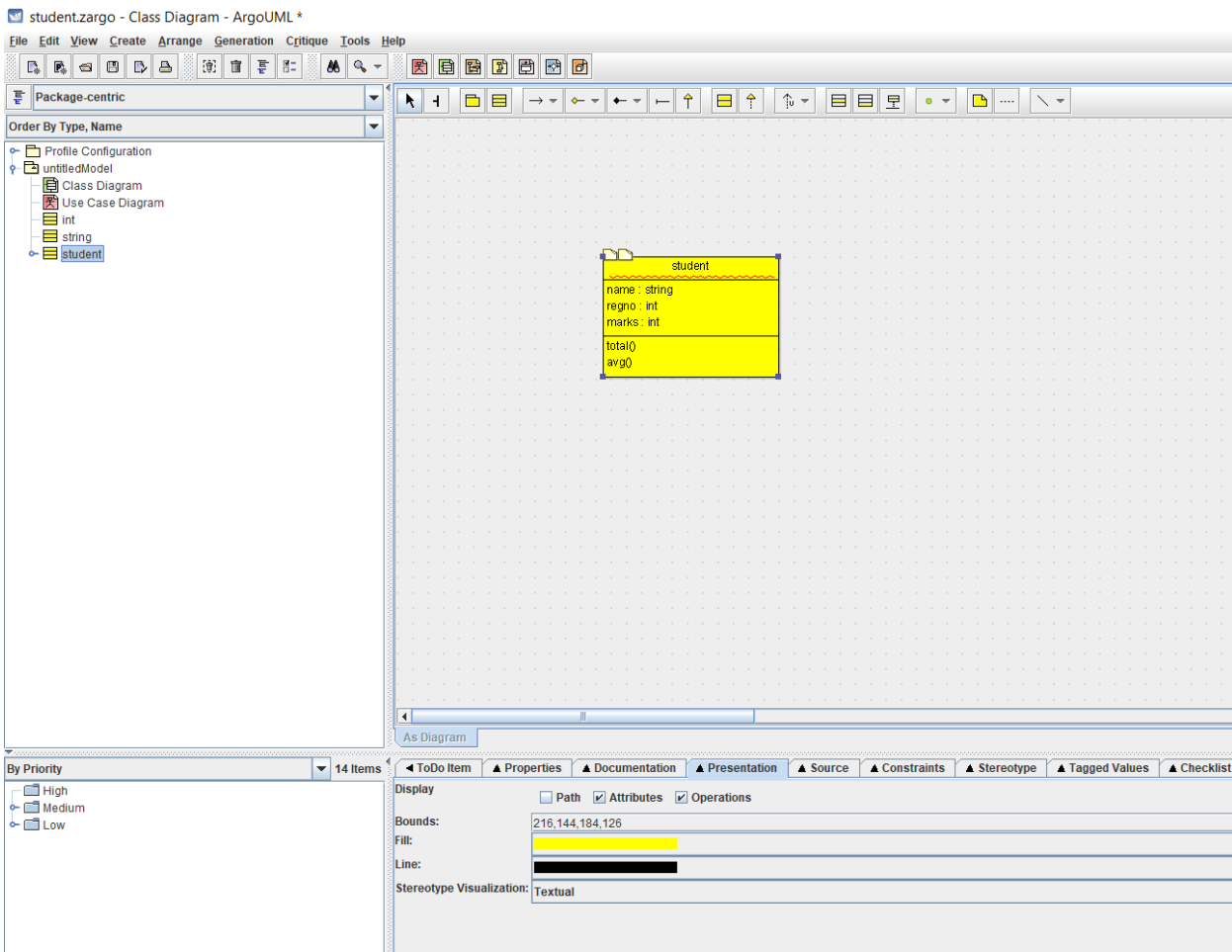
THE ARGOUML USER INTERFACE:

Overview of the Window :

The title bar of the window shows the following 4 parts of information, separated from each other by a dash.

- The current filename.
- The name of the currently active diagram.
- The name “ArgoUML”.

- An asterisk (*). This item is only present if the current project file is “dirty”, i.e. it is altered, but not yet saved. In other words, if the asterisk is absent, then the current file has not been altered.



The window comprises four sub-windows or panes namely,

- The Explorer pane,
- The Editing pane
- The Details Pane and
- The To-Do Pane.

The editing pane:

-The Toolbar





The toolbar at the top of the editing pane provides the main functions of the pane.

- File operations
- Edit operations
- View operations
- Create operations


-File operations

-  New
-  Open Project...
-  Save Project
-  Project Properties
-  Print









-Edit operations

-  Remove From Diagram
gram".
-  Delete From Model
-  Configure Perspectives
-  Settings

-View operations

-  Find...

-Create operations

-  Zoom
-  New Use Case Diagram
-  New Class Diagram
-  New Sequence Diagram
-  New Collaboration Diagram
-  New Statechart Diagram
-  New Activity Diagram
-  New Deployment Diagram

The tools fall into four categories.


- **Layout tools.** Provide assistance in laying out model elements on the diagram.
- **Annotation tools.** Used to annotate model elements on the diagram.
- **Drawing tools.** Used to add general graphic objects to diagrams.
- **Diagram specific tools.** Used to add UML model elements specific to a particular diagram type to the diagram.

-Layout Tools

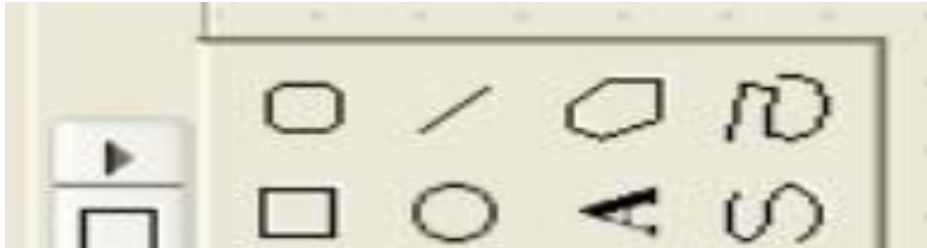
↑ **Select** - This tool provides for general selection of model elements on the diagram.









└ **Broom Tool** – This tool is used to sweep all model elements along.

-Annotation Tools

 **Annotation tool** - It is used to add a comment to a selected UML model element.

-Drawing Tools



-  **Rectangle.** Provides a rectangle.
-  **Rounded Rectangle.** Provides a rectangle with rounded corners. There is no control over the degree of rounding.
-  **Circle.** Provides a circle.
-  **Line.** Provides a line.
-  **Text.** Provides a text box. The text is entered by selecting the box and typing. Text is centered horizontally and after typing, the box will shrink to the size of the text. However it can be re-sized by dragging on the corners.
-  **Polygon.** Provides a polygon. The points of the polygon are selected by button 1 click and the polygon closed with button 1 double click (which will link the final point to the first point).
-  **Spline.** Provide an open spline. The control points of the spline are selected with button 1 and the last point selected with button 1 double click.
-  **Ink.** Provide a polyline. The points are provided by button 1 motion.

USE CASE DIAGRAM SPECIFIC TOOLS



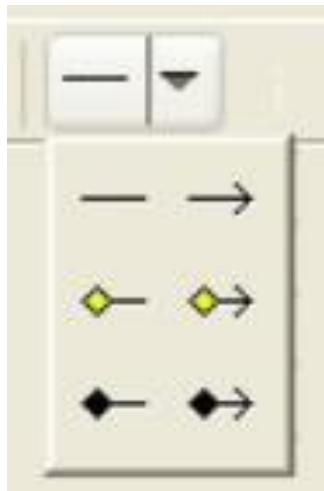
Actor - Add an actor to the diagram.








Use Case – Add a use case to the diagram.

Association - Add an association between two model elements selected using button 1 motion (from the first model element to the second).









The association tool selector:










-  Dependency. Add a dependency between two model elements selected using button 1 motion (from the dependent model element).
-  Generalization. Add a generalization between two model elements selected using button 1 motion (from the child to the parent).
-  Extend. Add an extend relationship between two model elements selected using button 1 motion (from the extended to the extending use case).
-  Include. Add an include relationship between two model elements selected using button 1 motion (from the including to the included use case).
-  Add Extension Point. Add an extension point to a selected use case.

CLASS DIAGRAM SPECIFIC TOOLS :







Class diagrams are used for only one of the UML structure diagrams, the class diagram itself. Object diagrams are represented on the argo UML deployment diagram. Argo UML uses the class diagram to shoe model structure through the use of packages.

-  Association-end. Add another end to an already existing association using button 1 (from the association middle to a class, or vice versa). This is the way to create so-called N-ary associations.
-  Generalization. Add a generalization between two model elements selected using button 1 (from the child to the parent).
-  Interface. Add an interface to the diagram. For convenience, when the mouse is over a selected interface it displays a handle at the bottom which may be dragged to form a realization relationship (the target being the realizing class).
-  Realization. Add a realization between a class and an interface selected using button 1 motion (from the realizing class to the realized interface).
-  Dependency. Add a dependency between two model elements selected using button 1 motion (from the dependent model element). There are also 2 special types of dependency offered here, Permission () and Usage (). A Permission is created by default with stereotype Import, and is used to import elements from one package into another.
-  Attribute. Add a new attribute to the currently selected class. The attribute is given the default name newAttr of type int and may be edited by button 1 double click and using the keyboard, or by selecting with button 1 click (after the class has been selected) and using the property tab.











SEQUENCE DIAGRAM SPECIFIC TOOLS :

-  ClassifierRole. Add a classifierrole to the diagram.
-  Message with Call Action. Add a call message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  Message with Return Action. Add a return message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  Message with Create Action. Add a create message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  Message with Destroy Action. Add a destroy message between two classifierroles selected using button 1 motion (from the originating classifierrole to the receiving classifierrole).
-  Add Vertical Space to Diagram. Add vertical space to a diagram by moving all messages below this down. Click the mouse at the point where you want the space to be added and drag down the screen vertically the distance which matches the height of the space you'd like to have added.
-  Remove Vertical Space in Diagram. Remove vertical space from diagram and move all elements below up vertically. Click and drag the mouse vertically over the space that you want deleted.




COLLABORATION DIAGRAM SPECIFIC TOOLS :

-  Classifier Role. Add a classifier role to the diagram.
-  Simple State. Add a simple state to the diagram.
-  Composite State. Add a composite state to the diagram. There are 6 association, aggregation and composition, and all these three can be bidirectional or unidirectional.
-  Generalization. Add a generalization between two model elements selected using button 1 (from the child to the parent).
-  Dependency. Add a dependency between two model elements selected using button 1 motion (from the dependent model element).
-  Add Message. Add a message to the selected association role.











STATE CHART DIAGRAM SPECIFIC TOOLS

-  Transition. Add a transition between two states selected using button 1 motion
-  Synch State. Add a synchstate to the diagram.
-  Submachine State. Add a submachinestate to the diagram.
-  Stub State. Add a stubstate to the diagram.
-  Initial. Add an initial pseudostate to the diagram.
-  Shallow History. Add a shallow history pseudostate to the diagram.
-  Deep History. Add a deep history pseudostate to the diagram.
-  Call Event. Add a Call Event as trigger to a transition. There are 4 types of events offered here: Call Event, Change Event, Signal Event and Time Event.
-  Guard. Add a guard to a transition.
-  Call Action. Add a call action (i.e. the effect) to a transition.

ACTIVITY DIAGRAM SPECIFIC TOOLS :

-  Action State. Add an action state to the diagram.
-  Transition. Add a transition between two action states selected using button 1 motion (from the originating action state to the receiving action state).
-  Initial. Add an initial pseudostate to the diagram.

DEPLOYMENT DIAGRAM SPECIFIC TOOLS :

-  Join. Add a join pseudostate to the diagram.
-  CallState. Add a callstate to the diagram. A call state is an action state that calls a single operation. Hence, the name of the operation being called is put in the symbol, along with the name of the classifier that hosts the operation in parentheses under it.
-  Component. Add a component to the diagram. For convenience, when the mouse is over a selected component it displays four handles to left, right, top and bottom which may be dragged to form dependency relationships.
-  Component Instance. Add a component instance to the diagram. For convenience, when the mouse is over a selected component instance it displays four handles to left, right, top and bottom which may be dragged to form dependency relationships.
-  Generalization. Add a generalization between two model elements selected using button 1 (from the child to the parent).
-  Realization. Add a realization between a class and an interface selected using button 1 motion (from the realizing class to the realized interface).
-  Dependency. Add a dependency between two model elements selected using button 1 motion (from the dependent model element).
-  Association. Add an association between two model elements
-  Node. Add a node to the diagram. For convenience, when the mouse is over a selected node it displays four handles to left, right, top and bottom which may be dragged to form association relationships.
-  Node Instance. Add a node instance to the diagram. For convenience, when the mouse is over a selected node instance it displays four handles to left, right, top and bottom which may be dragged to form link relationships.

RESULT:

The Argo UML is successfully installed and familiarization of argo uml is done successfully.

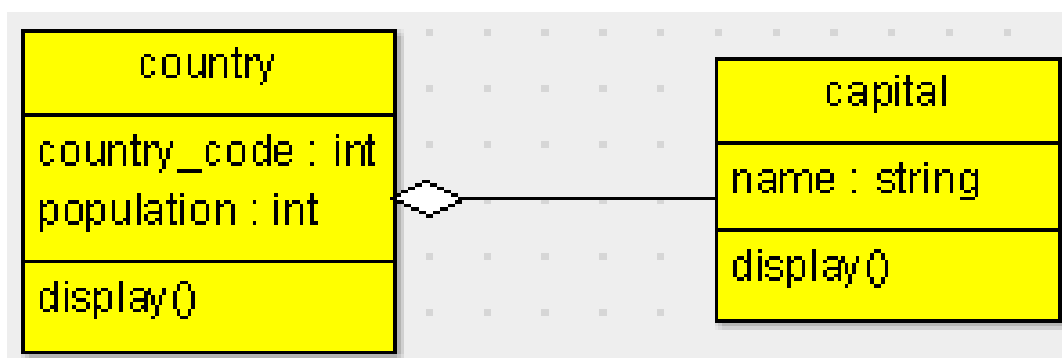
EXP NO: 2**DATE: _____****AIM:**

To draw a class diagram for a very simple problem such as following in argo uml lab and generating skeleton code in java and c++.

- A country has a capital city
- A dining philosopher uses a fork
- A file is an ordinary file or a directory file
- Files contain records
- A class can have several attributes
- A relation can be association or generalization
- A polygon is composed of an ordered set of points
- A person uses a computer language on a project
- A drawing object is text, a geometrical object or a group
- A person plays for a team in a certain year
- A route connects two cities
- A student takes a course from a professor
- Modems and keyboards are input/output devices

DESCRIPTION:**Modeling steps for Class Diagrams:**

- Identity the things that are interacting with class diagram.
- Set the attributes and operations.
- Set the responsibilities.
- Identify the generalization and specification classes.
- Set the relationship among all the things.
- Adron with tagged values, constraints and notes.

A country has a capital city :

Skeleton code:**Java:**

```
import java.util.List;

public class country {

    public int country_code;

    public int population;

    public List<capital> has a;

    public capital capital;

    public List<capital> capital;

    public void display() {

    }

}

import java.util.List;

public class capital {

    public string name;

    public List<country> has a;

    public country country;

    public List<country> country;

    public void display() {

    }

}
```

C++:

```
#ifndef country_h

#define country_h

#include "int.h"

class capital;

class country {

public:

    virtual void display();
```

```
public:

    int country_code;

    int population;

public:

    /**

    * @element-type capital

    */

    capital *has a;

    capital *mycapital;

    /**

    * @element-type capital

    */

    capital *mycapital;

};

#endif // country_h

#ifndef capital_h

#define capital_h

#include "string.h"

class country;

class capital {

public:

    virtual void display();

public:

    string name;

public:

    /**

    * @element-type country

    */

    country *has a;

    country *mycountry;
```

```

/**
 * @element-type country
 */

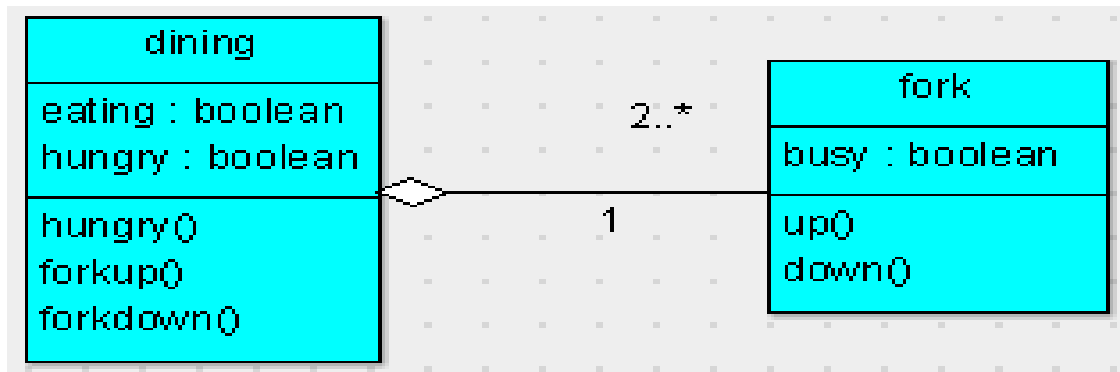
country *mycountry;

};

#endif // capital_h

```

A dining philosopher uses a fork :



Skeleton code:

Java:

```

public class dining {

    public boolean eating;

    public boolean hungry;

    public fork 2- - *;

    public fork 2..*;

    public void hungry() {

    }

    public void forkup() {

    }

    public void forkdown() {

    }

}

import java.util.List;

public class fork {

    public boolean busy;

```

```
public dining dining;

public List<dining> dining;

public void up() {

}

public void down() {

}

}
```

C++:

```
#ifndef dining_h

#define dining_h

#include "boolean.h"

class fork;

class dining {

public:

    virtual void hungry();

    virtual void forkup();

    virtual void forkdown();

public:

    boolean eating;

    boolean hungry;

public:

    fork *2- - *;

    fork *2..*;

};

#endif // dining_h

#ifndef fork_h

#define fork_h

#include "boolean.h"

class dining;

class fork {
```



```

public:

    virtual void up();

    virtual void down();

public:

    boolean busy;

public:

    dining *mydining;

    /**
     * @element-type dining
     */

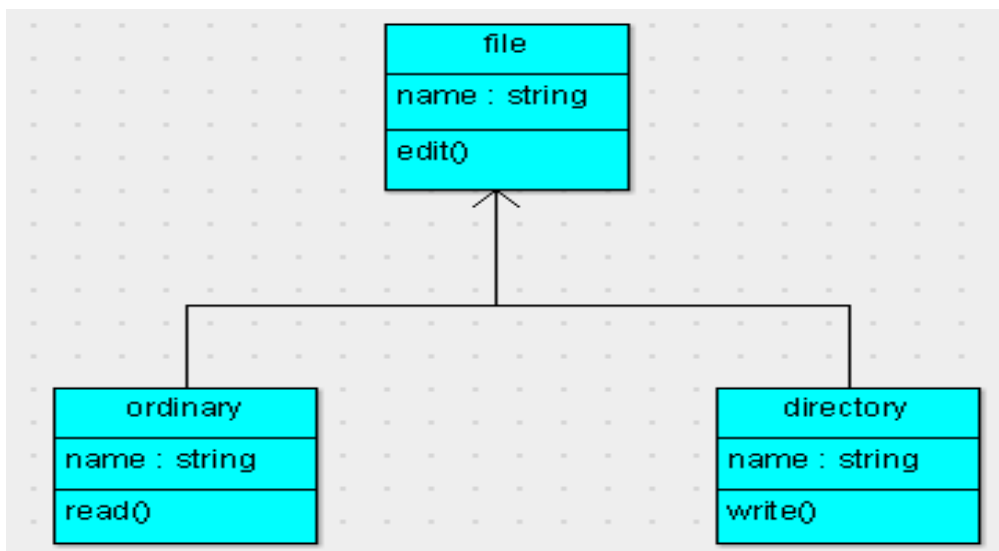
    dining *mydining;

};

#endif // fork_h

```

A file is an ordinary file or a directory file :



Skeleton code:

Java:

```

public class file {

    public string name;

    public void edit() {

    }

}

```

```
}

import java.util.List;

public class ordinary {

    public string name;

    public List<directory> directory;

    public void read() {

    }

}

import java.util.List;

public class directory {

    public string name;

    public List<ordinary> ordinary;

    public void write() {

    }

}
```

C++:

```
#ifndef file_h

#define file_h

#include "string.h"

class file {

public:

    virtual void edit();

public:

    string name;

};

#endif // file_h

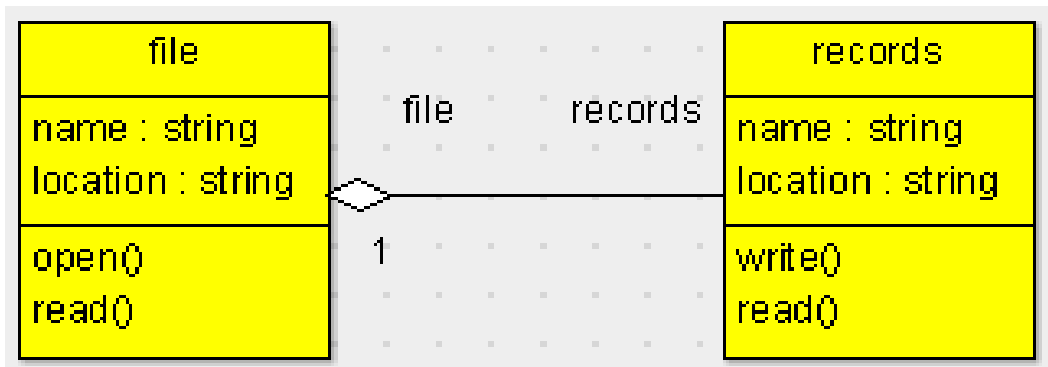
#ifndef ordinary_h

#define ordinary_h

#include "string.h"

class directory;
```

```
class ordinary {  
  
    public:  
  
        virtual void read();  
  
    public:  
  
        string name;  
  
    public:  
  
        /**  
  
        * @element-type directory  
  
        */  
  
        directory *mydirectory;  
  
};  
  
#endif // ordinary_h  
  
#ifndef directory_h  
  
#define directory_h  
  
#include "string.h"  
  
class ordinary;  
  
class directory {  
  
    public:  
  
        virtual void write();  
  
    public:  
  
        string name;  
  
    public:  
  
        /**  
  
        * @element-type ordinary  
  
        */  
  
        ordinary *myordinary;  
  
};  
  
#endif // directory_h
```

Files contain records:**DATE:** _____**Skeleton code:****Java:**

```

import java.util.List;
public class file {
    public String name;
    public String location;
    public List<records> records;
    public void open() {
    }
    public void read() {
    }
}
public class records {
    public String name;
    public String location;
    public file file;
    public void write() {
    }
    public void read() {
    }
}
  
```

C++:

```

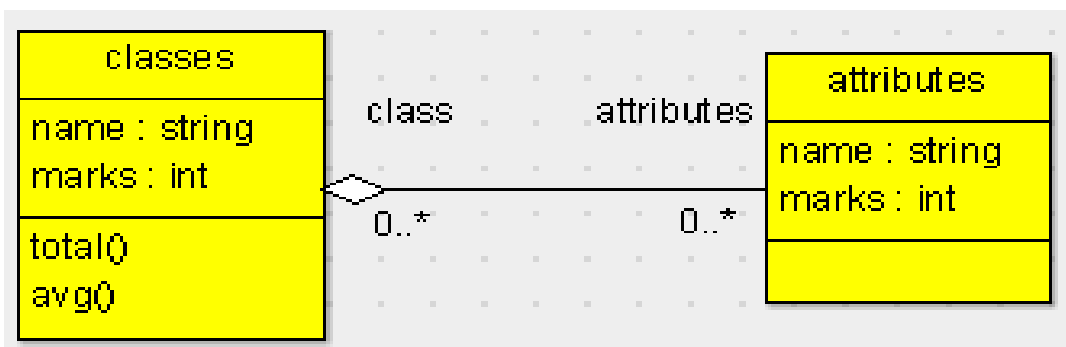
#ifndef file_h
#define file_h
#include "string.h"
class records;
class file {
public:
    virtual void open();
    virtual void read();
public:
    string name;
    string location;
public:
    /**
  
```

```

* @element-type records
*/
records *records;
};
#endif // file_h
#ifndef records_h
#define records_h
#include "string.h"
class file;
class records {
public:
    virtual void write();
    virtual void read();
public:
    string name;
    string location;
public:
    file *file;
};
#endif // records_h

```

A class can have several attributes:



Skeleton code:

Java:

```

import java.util.List;
public class classes {
    public string name;
    public int marks;
    /**
     *
     */
    public List<attributes> attributes;
    public void total() {
    }
    public void avg() {
    }
}

```

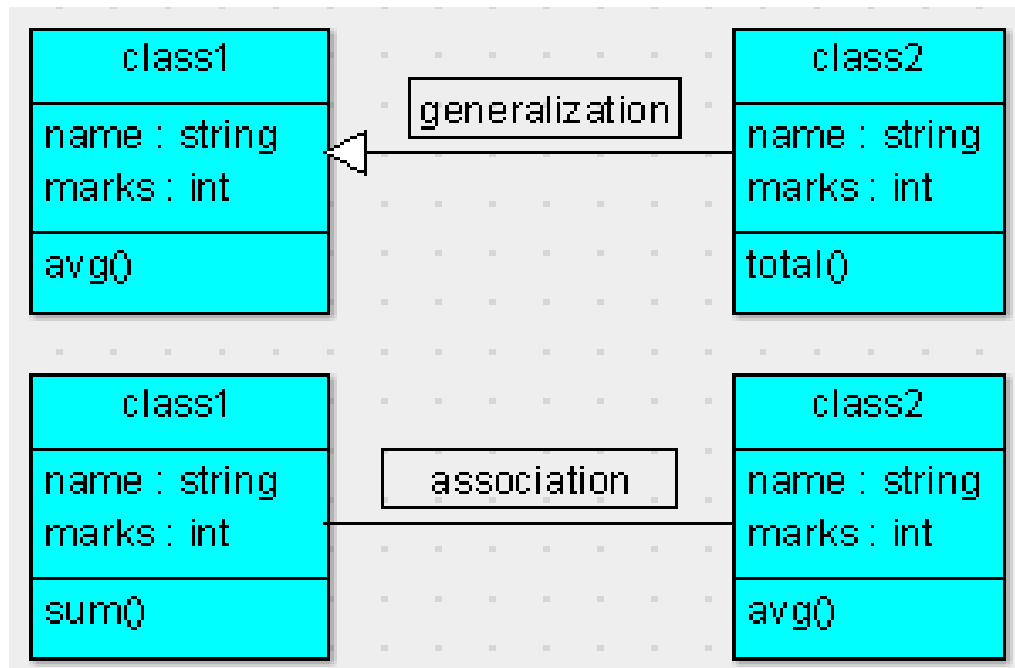
```
import java.util.List;

public class attributes {
    public string name;
    public int marks;
    /**
     *
     */
    public List<classes> class;
```

C++:

```
#ifndef classes_h
#define classes_h
#include <vector>
#include "int.h"
#include "string.h"
class attributes;
class classes {
public:
    virtual void total();
    virtual void avg();
public:
    string name;
    int marks;
public:
    /**
     * @element-type attributes
     */
    std::vector< attributes* > attributes;
};
#endif // classes_h
#ifndef attributes_h
#define attributes_h
#include <vector>
#include "int.h"
#include "string.h"
class classes;
class attributes {
public:
    string name;
    int marks;
public:
    /**
     * @element-type classes
     */
    std::vector< classes* > class;
};
#endif // attributes_h
```

- A relation can be association or generalization:



Skeleton code:

Java:

Generalization:

```
public class class1 {
    public string name;
    public int marks;
    public void avg() {
    }
}

public class class2 extends class1 {
    public string name;
    public int marks;
    public void total() {
    }
}

import java.util.List;
public class class1 {
    public string name;
    public int marks;
    public List<class2> class2;
    public void sum() {
    }
}
```

Association:

```
import java.util.List;
public class class1 {
    public string name;
```



```
public int marks;
    public List<class2> class2;
    public void sum() {
    }
}
import java.util.List;
public class class2 {
    public string name;
    public int marks;
    public List<class1> class1;
    public void avg() {
    }
}
```

C++:

Generalization:

```
#ifndef class1_h
#define class1_h
#include "int.h"
#include "string.h"
class class1 {
public:
    virtual void avg();
public:
    string name;
    int marks;
};
#endif // class1_h
#ifndef class2_h
#define class2_h
#include "class1.h"
#include "int.h"
#include "string.h"
class class2 : public class1 {
public:
    virtual void total();
public:
    string name;
    int marks;
};
#endif // class2_h
```

Association:

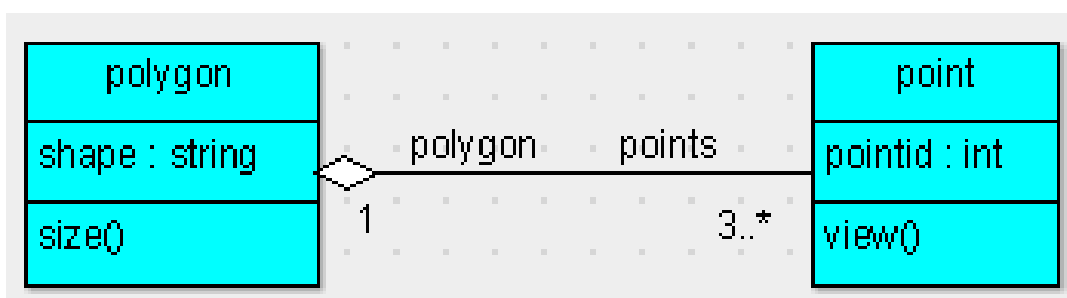
```
#ifndef class1_h
#define class1_h
#include "int.h"
#include "string.h"
class class2;
class class1 {
public:
```

```

    virtual void sum();
public:
    string name;
    int marks;
public:
    /**
     * @element-type class2
     */
    class2 *myclass2;
};
#endif // class1_h
#ifndef class2_h
#define class2_h
#include "int.h"
#include "string.h"
class class1;
class class2 {
public:
    virtual void avg();
public:
    string name;
    int marks;
public:
    /**
     * @element-type class1
     */
    class1 *myclass1;
};
#endif // class2_h

```

- **A polygon is composed of an ordered set of points:**



Skeleton code:

Java:

```

import java.util.List;
public class polygon {
    public string shape;
    /**
     *
     *
     */
}

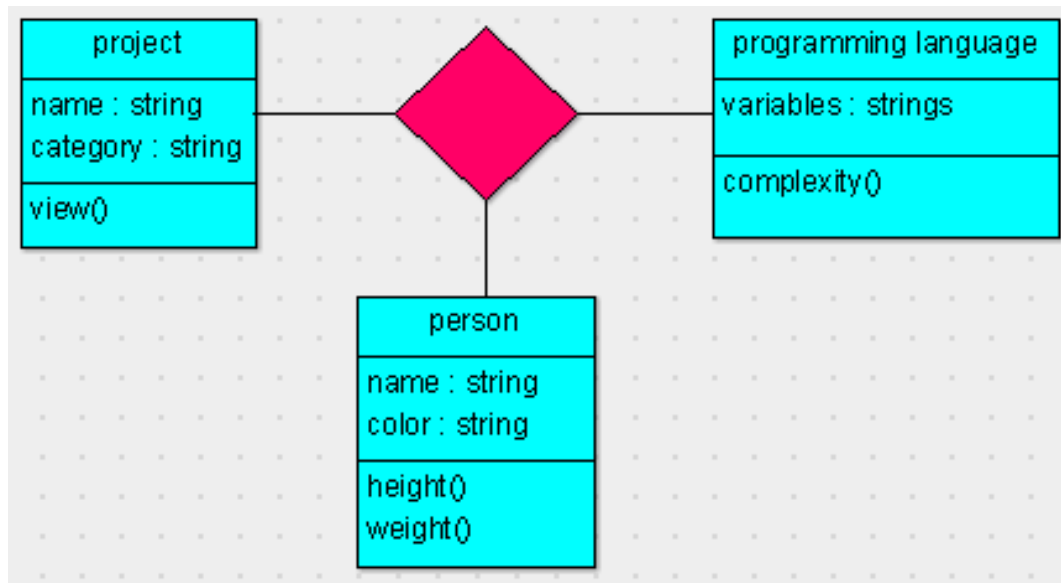
```

```
public List<point> points;
public void size() {
}
}
public class point {
public int pointid;
public polygon polygon;
public void view() {
}
}
```

C++:

```
#ifndef polygon_h
#define polygon_h
#include <vector>
#include "string.h"
class point;
class polygon {
public:
virtual void size();
public:
string shape;
public:
/**
 * @element-type point
 */
std::vector< point* > points;
};
#endif // polygon_h
#ifndef polygon_h
#define polygon_h
#include <vector>
#include "string.h"
class point;
class polygon {
public:
virtual void size();
public:
string shape;
public:
/**
 * @element-type point
 */
std::vector< point* > points;
};
#endif // polygon_h
```

DATE: _____

A person uses a computer language on a project:**Skeleton code:****Java:**

```

import java.util.List;
public class project {
    public string name;
    public string category;
    public List<programming language> programming language;
    public List<programming language> programming language;
    public List<person> person;
    public void view() {
    }
}

import java.util.List;
public class programming language {
    public strings variables;
    public List<project> project;
    public List<project> project;
    public List<person> person;
    public void complexity() {
    }
}

import java.util.List;
public class person {
    public string name;
    public string color;
    public List<project> project;
    public List<programming language> programming language;
    public void height() {
    }
}
  
```

```
public void weight() {  
}  
}
```

C++:

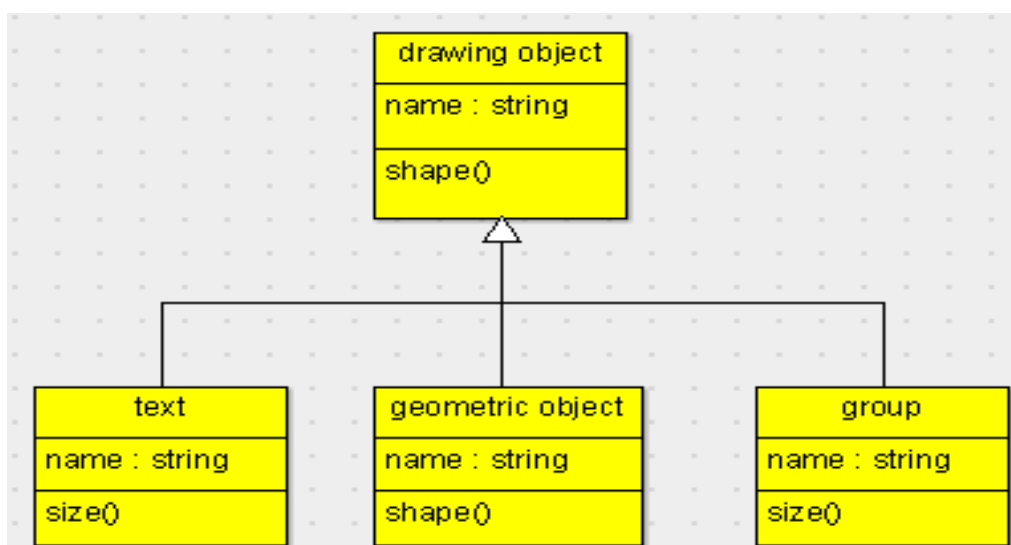
```
#ifndef project_h  
#define project_h  
#include "string.h"  
class programming language;  
class person;  
class project {  
public:  
    virtual void view();  
public:  
    string name;  
    string category;  
public:  
    /**  
     * @element-type programming language  
     */  
    programming language *myprogramming language;  
    /**  
     * @element-type programming language  
     */  
    programming language *myprogramming language;  
    /**  
     * @element-type person  
     */  
    person *myperson;  
};  
#endif // project_h  
#ifndef programming language_h  
#define programming language_h  
#include "strings.h"  
class project;  
class person;  
class programming language {  
public:  
    virtual void complexity();  
public:  
    strings variables;  
public:  
    /**  
     * @element-type project  
     */  
    project *myproject;  
    /**  
     * @element-type project  
     */
```

```

project *myproject;
/**
 * @element-type person
 */
person *myperson;
};
#endif // programming language_h
#ifndef person_
#define person_h
#include "string.h"
class project;
class programming language;
class person {
public:
    virtual void height();
    virtual void weight();
public:
    string name;
    string color;
public:
    /**
     * @element-type project
     */
    project *myproject;
    /**
     * @element-type programming language
     */
    programming language *myprogramming language;
};
#endif // person_h

```

• **A drawing object is text, a geometrical object or a group:**



Skeleton code:

Java:

```

public class drawing object {

```

```
public string name;
public void shape() {
}
}
import java.util.List;
public class text {
    public string name;
    public List<group> group;
    public void size() {
    }
}
public class geometric object extends drawing object {
    public string name;
    public void shape() {
    }
}
import java.util.List;
public class group {
    public string name;
    public List<text> text;
    public void size() {
    }
}
```

C++:

```
#ifndef drawing object_h
#define drawing object_h
#include "string.h"
class drawing object {
public:
    virtual void shape();
public:
    string name;
};
#endif // drawing object_h
#ifndef text_h
#define text_h
#include "string.h"
class group;
class text {
public:
    virtual void size();
public:
    string name;
public:
    /**
     * @element-type group
     */
    group *mygroup;
```

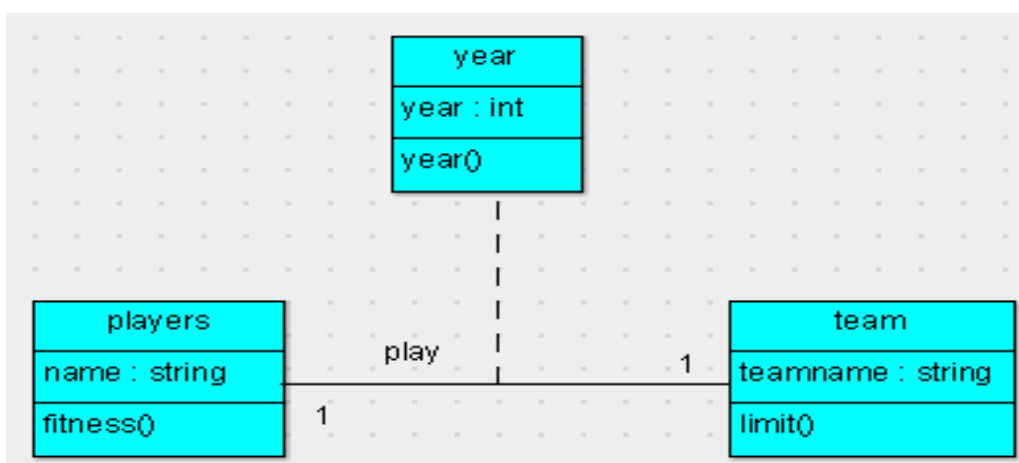


```

};
#endif // text_h
#ifndef geometric object_h
#define geometric object_h
#include "drawing object.h"
#include "string.h"
class geometric object : public drawing object {
public:
    virtual void shape();
public:
    string name;
};
#endif // geometric object_h
#ifndef group_h
#define group_h
#include "string.h"
class text;
class group {
public:
    virtual void size();
public:
    string name;
public:
    /**
     * @element-type text
     */
    text *mytext;
};
#endif // group_h

```

- A person plays for a team in a certain year:



Skeleton code:

Java:

```

public class year {
    public int year;
    public players play;
}

```

```
    public team team;
    public void year() {
    }
}
import java.util.List;
public class players {
    public string name;

    public team team;
    public List<year> year;
    public team team;
    public void fitness() {
    }
}
import java.util.List;
public class team {
    public string teamname;
    public players play;
    public List<year> year;
    public players play;
    public void limit() {
    }
}
```

C++:

```
#ifndef year_h
#define year_h
#include "int.h"
class players;
class team;
class year {
public:
    virtual void year();
public:
    int year;
public:
    players *play;
    team *myteam;
};
#endif // year_h
#ifndef players_h
#define players_h
#include "string.h"
class team;
class year;
class players {
public:
    virtual void fitness();
public:
```

```

string name;
public:
    team *myteam;
/**
 * @element-type year
 */
year *myyear;
team *myteam;
};
#endif // players_h
#ifndef team_h
#define team_h
#include "string.h"
class players;
class year;
class team {
public:
    virtual void limit();
public:
    string teamname;
public:
    players *play;
/**
 * @element-type year
 */
year *myyear;
players *play;
};
#endif // team_h

```

A route connects two cities:



Skeleton code:

Java:

```

import java.util.List;

public class cities {

    public string city;

```

```
public List<routes> l..*;  
  
public void route() {  
  
}  
  
}  
  
public class routes {  
  
public string names;  
  
public cities connected;  
  
  
  
public void path() {  
  
}  
  
}
```

C++:

```
#ifndef cities_h  
  
#define cities_h  
  
#include "string.h"  
  
class routes;  
  
class cities {  
  
public:  
  
    virtual void route();  
  
public:  
  
    string city;  
  
public:  
  
    /**  
  
    * @element-type routes  
  
    */  
  
    routes *l..*;  
  
};  
  
#endif // cities_h  
  
#ifndef routes_h  
  
#define routes_h
```

```
#include "string.h"

class cities;

class routes {

public:

    virtual void path();

public:

    string names;

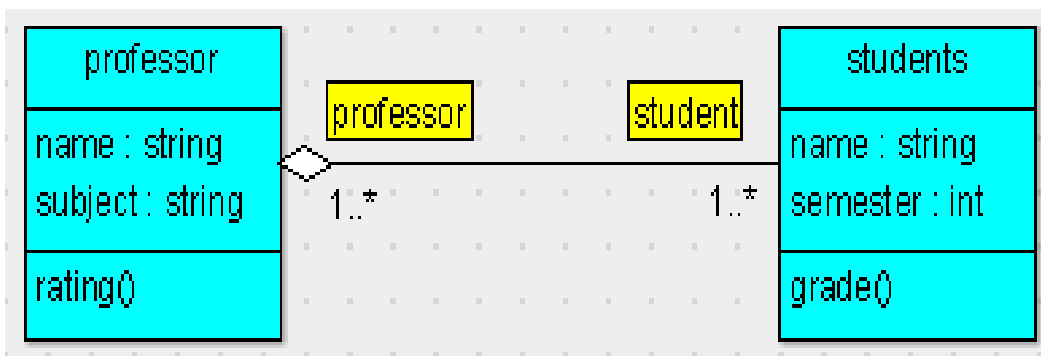
public:

    cities *connected;

};

#endif // routes_h
```

- **A student takes a course from a professor:**



Skeleton code:

Java:

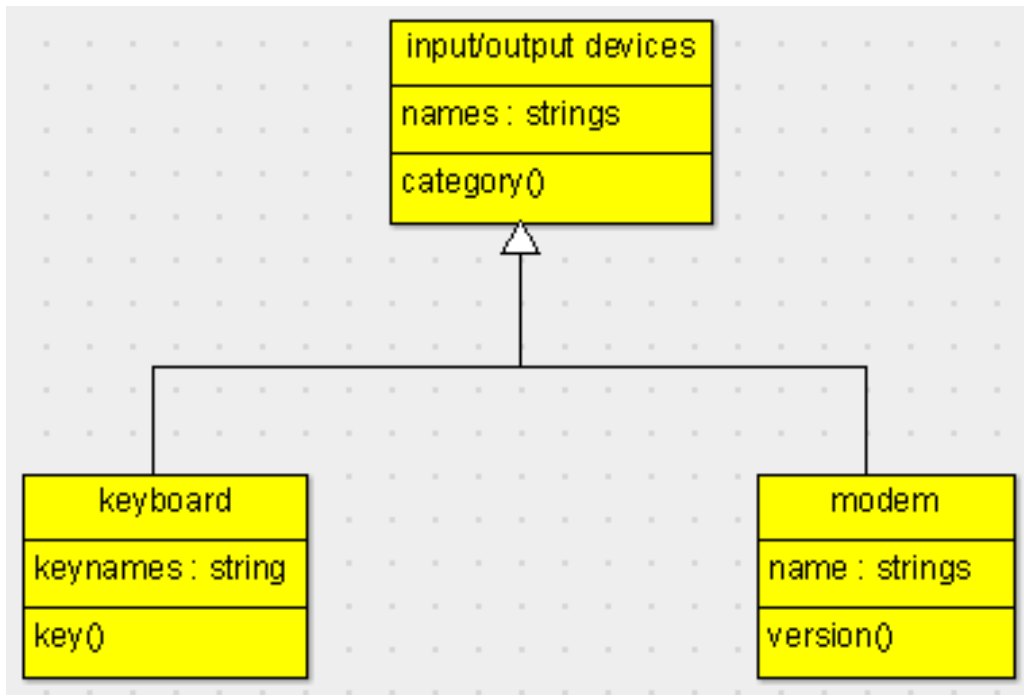
```
import java.util.List;
public class professor {
    public String name;
    public String subject;
    /**
     *
     *
     */
    public List<students> students;
    public void rating() {
    }
}
import java.util.List;
public class students {
    public String name;
    public int semester;
    /**
```

```
*  
*  
*/  
public List<professor> professor;  
public void grade() {  
}  
}
```

C++:

```
#ifndef professor_h  
#define professor_h  
#include <vector>  
#include "string.h"  
class students;  
class professor {  
public:  
    virtual void rating();  
public:  
    string name;  
    string subject;  
public:  
    /**  
     * @element-type students  
     */  
    std::vector< students* > mystudents;  
};  
#endif // professor_h  
#define students_h  
#include <vector>  
#include "int.h"  
#include "string.h"  
class professor;  
class students {  
public:  
    virtual void grade();  
public:  
    string name;  
    int semester;  
public:  
    /**  
     * @element-type professor  
     */  
    std::vector< professor* > myprofessor;  
};  
#endif // students_h
```

• Modems and keyboards are input/output devices:



Skeleton code:

Java:

```

import java.util.List;
public class keyboard {
    public string keynames;
    public List<modem> modem;
    public void key() {
    }
}
import java.util.List;
public class modem extends input/output devices {
    public strings name;
    public List<keyboard> keyboard;
    public void version() {
    }
}
  
```

C++:

```

#ifndef input/output_devices_h
#define input/output_devices_h
#include "string.h"
class input/output devices {
public:
    virtual void category();
public:
    string names;
};
  
```



```
#endif // input/output devices_h
#ifndef keyboard_h
#define keyboard_h
#include "string.h"
class modem;
class keyboard {
public:
    virtual void key();
public:
    string keynames;
public:
    /**
     * @element-type modem
     */
    modem *mymodem;
};
#endif // keyboard_h
#ifndef modem_h
#define modem_h
#include "input/output devices.h"
#include "strings.h"
class keyboard;
class modem : public input/output devices {
public:
    virtual void version();
public:
    strings name;
public:
    /**
     * @element-type keyboard
     */
    keyboard *mykeyboard;
};
#endif // modem_h
```

RESULT:

All the given class diagrams have been drawn and code is generated successfully by using argo UML.

EXP NO: 3**DATE:** _____**AIM:**

To create a Point of Sale System.

DESCRIPTION:**Modeling steps for Use case Diagram:**

- Draw the lines around the system and actors lie outside the system.
- Identify the actors which are interacting with the system.
- Separate the generalized and specialized actors.
- Identify the functionality the way of interacting actors with system and specify the behavior of actor.
- Functionality or behavior of actors is considered as use cases.
- Specify the generalized and specialized use cases.
- Se the relationship among the use cases and in between actor and usecases.
- Adorn with constraints and notes.
- If necessary, use collaborations to realize use cases.

ACTORS:

- Customer
- cashier

USECASES:

- Bar code scanning
- Process sale
- close sale
- Pay Bill
- Tax calculation
- Buy product
- Update Inventory.

ALGORITHMIC PROCEDURE:

STEP 1: Start the application.

STEP 2: Create the require actors and usecases in the browser window

STEP 3: Go to new usecase view and then click the usecase view and open a new package

STEP 4: Rename the new package with the package with required names

STEP 5: Create two packages actor and usecase

USECASE DIAGRAM:RESULT:

Usecase diagram for a given point of sale problem is drawn successfully by using argo UML.

EXP NO:4**DATE:** _____**AIM:**

To develop a domain model for a given problem.

DESCRIPTION:**Modeling steps for Domain model diagram:**

- Identify conceptual classes
- Draw the class diagram
- Add any associations between classes
- Add attributes(properties)to the classes.

ALGORITHMIC PROCEDURE:

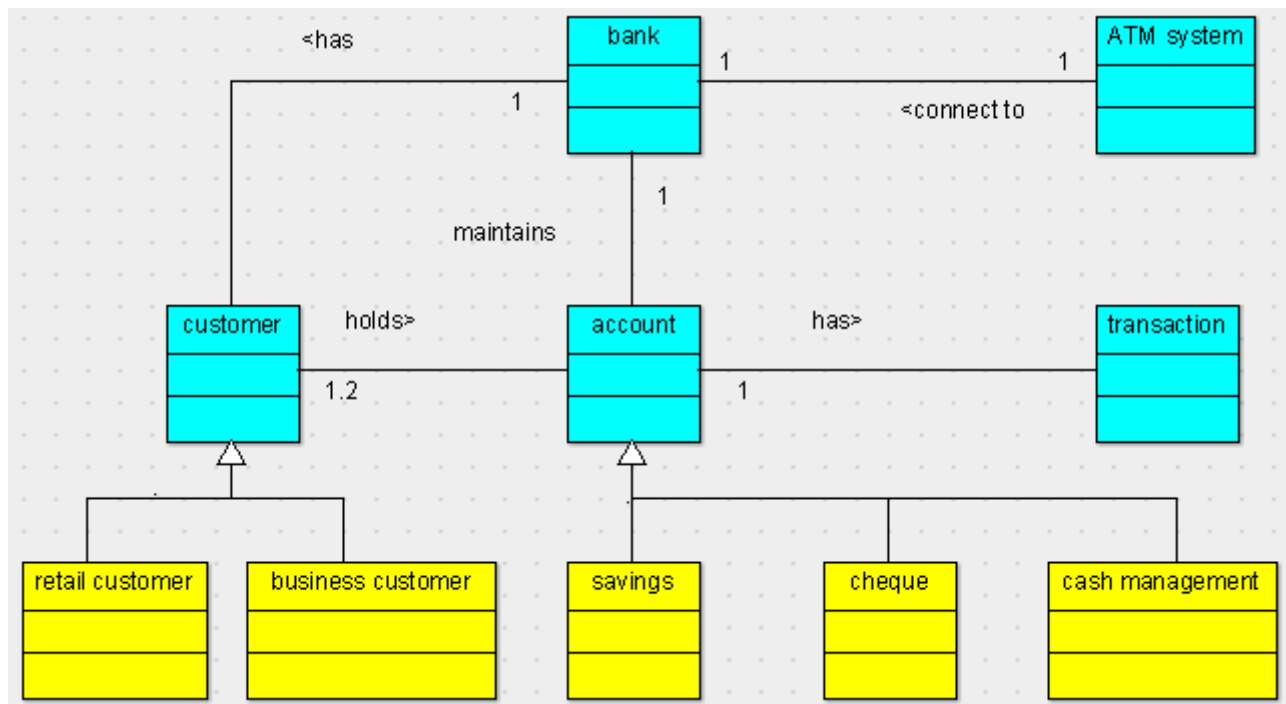
STEP 1: Start the application.

STEP 2: Make a list of candidate domain classes.

STEP 3: Draw these classes in a UML class diagrams.

STEP 4: Identify the necessary associations.

STEP 5: Group domain classes by category into packages.

CASE STUDY-1:**DOMAIN MODEL FOR BANKING SYSTEM**

CASE STUDY-2:**CUSTOMER SUPPORTING SYSTEM (Online Book store)**

a) Aim: Identify and analyze events.

Procedure:

Generally event is specification of a significances occurrence that has a location in time & space .The Flow of events of a usecase contains the most important information derived from usecase modeling work.it should describe the usecases flow of events clearly enough for an outsider to easily understand it.Remember flow of events should present what the system does,not how the system is design to perform the required behavior.

Guidelines for the contents of the flow of events are,

- 1)Describe how the usecase starts and ends.
- 2)Describe what data is exchanged between the actor and the usecase
- 3)Do not describe the details of the user interface,unless it is necessary to understand the behavior of the system.

b) Aim: Identify use cases:

Use case description for online book store: in these we have

Login: here we will our credentials and login to the online book store.

Browser catalog: here user can browse his required book according to his need such as amount ,author, version etc.....

Order book: after selecting the book can order the book on checking the option “buy now”.

Manage account: here we will have the option of online payment through our account were it will manage our accountdetails like sending OTPS checking balance ect.

Ship book: after the payment of money to the book, the book is shipped to our address in various steps.this is managed bythe system.

s in this the admin will check and analyze the details of shipping books.

Login**Search book****Browse catalog**

Order book

browse catalog

order book

Manage account

manage account

Ship book

ship book

Manage catalog

manage catalog

c) Aim: Draw Event table.

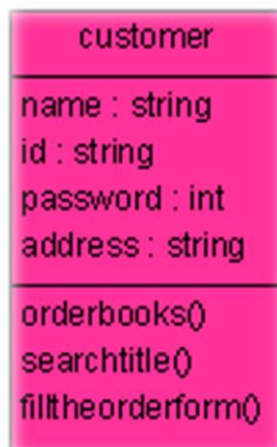
Event table:

Event	Actor	Description	Object
login	customer	Here we will open the webpage and by using our credentials we can login to the website of the online book store.	login
Search book	customer	In search books the users will search books according to their requirements	Book
Browse catalog	customer	The user can browse his required book according to their needs such as author,amount,version	Book
Order book	customer	After selecting the book the customer can order the book on clicking the commands such as buy new or add to cart	Book
Manage account	customer	Here we will have the option of cash on delivery or online payment through our account, it will manage our account details like sending OTP,checking	Account

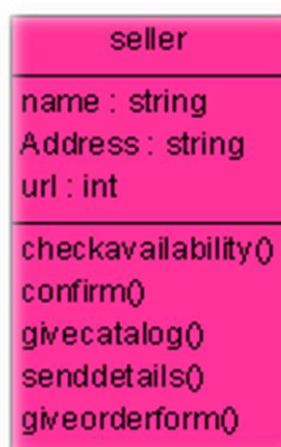
		balance etc.	
Ship book	System	After the payment of money to the book which the customer had selected the book is shipped to the customer address in various steps.	Books
Manage catalog	administrator	In managing catalog the administrator will check and analyze the details of shipping books like wheather the book is being delivered or not	Books

d)Aim: Identify and analyze domain classes:

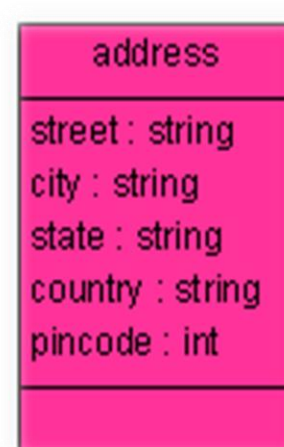
Customer



seller



address



Login



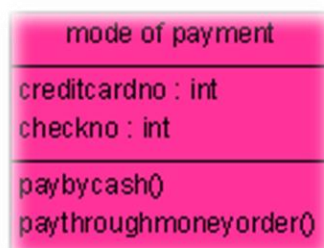
Logout



Book



Mode of payment

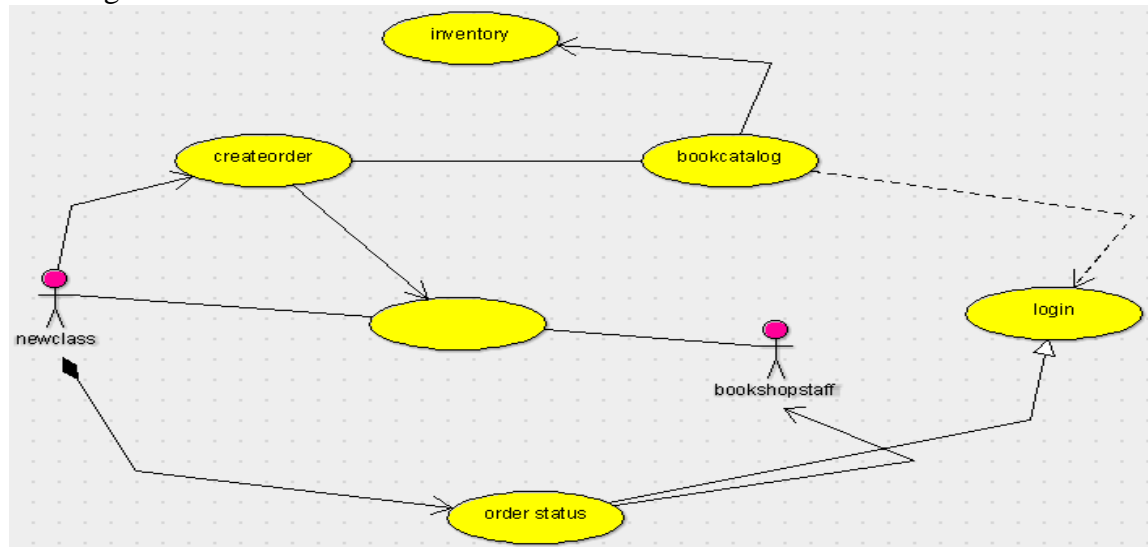


order form

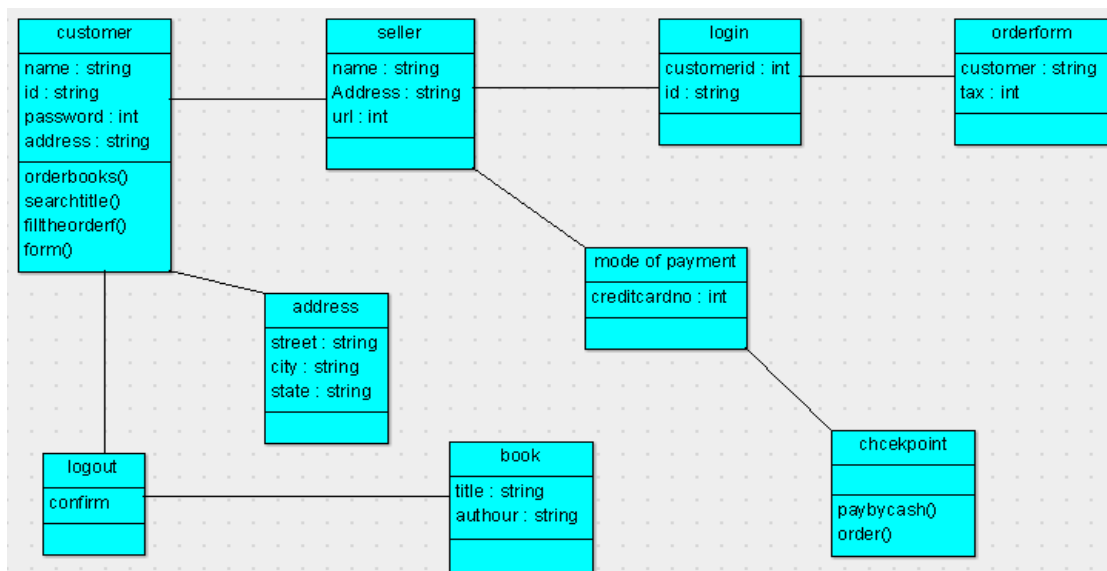


e) **Aim** :Represent usecases and domain class diagram

-usecase diagram:



-class diagram:



f) **Aim**: Develop CRUD matrix to represent relationship between use cases and problem domain classes.

Entity /function	custo mer	lo gi n	addr ess	log out	sel ler	or der	bo ok	Mode of paym ent
Inventory					C. R			
Create order	C.R. D				C	C		C.U
Book catalog	R				C		R	
Login		C. R			C. R			
Manange ment payment	R				C		C	

Order status			R		C	C		
--------------	--	--	---	--	---	---	--	--

CASE STUDY-3:**POINT OF SALE TERMINAL**

a) Aim: Identify and analyze events

Procedure:

The Flow of events of a usecase contains the most important information derived from usecase modeling work.it should describe the usecases flow of events clearly enough for an outsider to easily understand it.Remember flow of events should present what the system does,not how the system is design to perform the required behavior.

Guidelines for the contents of the flow of events are,

- 1)Describe how the usecase starts and ends.
- 2)Describe what data is exchanged between the actor and the usecase
- 3)Do not describe the details of the user interface,unless it is necessary to understand the behavior of the system.

For example,it is often good to use a limited set of web specific terminology when it is known before hand that the application is going to be web based otherwise your run the risk that the usecase text is being perceived as too abstract.words to include in your terminology could be “navigate”,”browser”,”hyperlink”,”page”,”submit”,and,”browser”.however it is not asvisible to include references to frames or “webpages” in such a way that you are making assumptions about the boundaries between them this is a critical design decision.

- Describe the flow of events ,not only the functionality to enforce this,start every action with “when the actor”.
- Describe only the events that belong to the usecase and not what happens in other usecases or outside of the system.
- Avoid vague terminology such as “for example”,”information”.
- Detail the flow of events all whats should be answered .remember that test designers are to be use this text to identify test cases.

If you have used certain terms in other use cases ,be sure to use the exact same terms in this usecase,and that theirintented meaning is the same.to manage comman terms ,put them in a glossary.

b)Aim: Identify Usecases

Procedure:

- 1)Buy Product
- 2)Bar Code scanning
- 3)Pay Bill
- 4)Process Sale
- 5)Close Sale
- 6)Update Inventory
- 7)Tax Calculator

1)Buy product:

Among the multiple products we will select which product we should buy

2)barcode scanning:

Here the barcode is scanned for the product which we are selected.the barcode will be ready predefined on each and every product and according to that the price of the product will be displayed.

3)pay bill:

Pay bill is a bill in which the price of the product is printed ,according to that price the customers will pay

4)process sale:

After receiving the bill the products which we have selected are processed to the cashier counter and they will crosscheck our products according to our bill.

5)close sale:

In this close sale the customer will pay the bill and cashier place a stamp on that bill,that is represents the bill is closed.


6)update inventory:

In this the cashier will update the database according to the products in that store.

7)tax calculation:

The cashier will calculate the tax in various forms such as transportation cost,maintenance cost.

Graphical Representation:

Buy product:

buy product

Barcode scanning:


barcode scanning

Pay bill:

pay bill

Process sale:

process sale

Close sale:

close sale

Update inventory:

update inventory

Tax calculation:

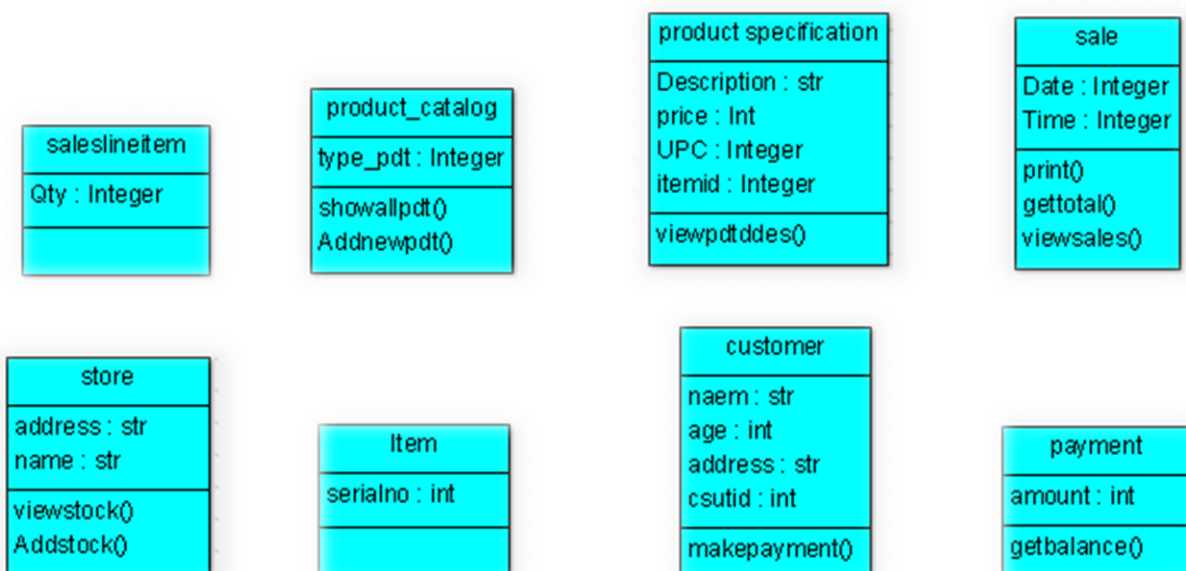
tax calculation

c) Aim: Develop event table Event table**Event table:**

Usecase	Description
Buy product	Among the multiple products we will select product we should buy
Barcode scanning	Here the barcode is scanned for the product which we are selected
Pay bill	In this price of the product is printed
Process sale	The bill, product are processed to the cashier counter
Close sale	The customer will pay the bill and cashier place a stamp in that bill
Update inventory	The customer will update the database according to the product in that store
Tax calculation	The cashier will calculate the tax in various forms such as transport cost, maintenance cost etc

d) Aim: Identify and analyze domain classes**Domain classes:**

A Domain class is a class its description of set of objects that share the same attribute, operations, relationships. A class implements one or more interfaces.

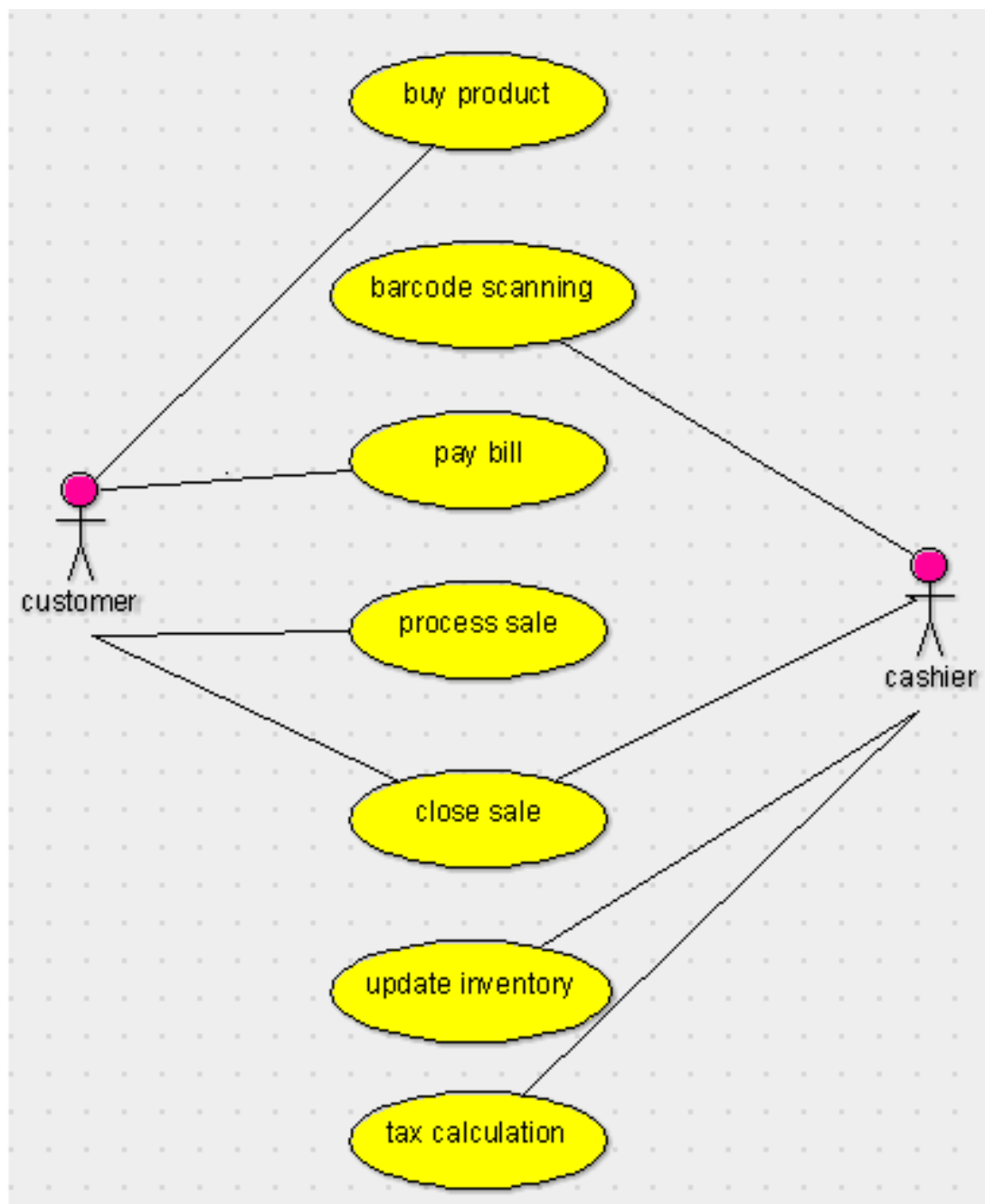


manager
name : str
age : int
address : str
managestock()
updatestock()
addnewitem()
deleteitem()
managesales()
viewsales()

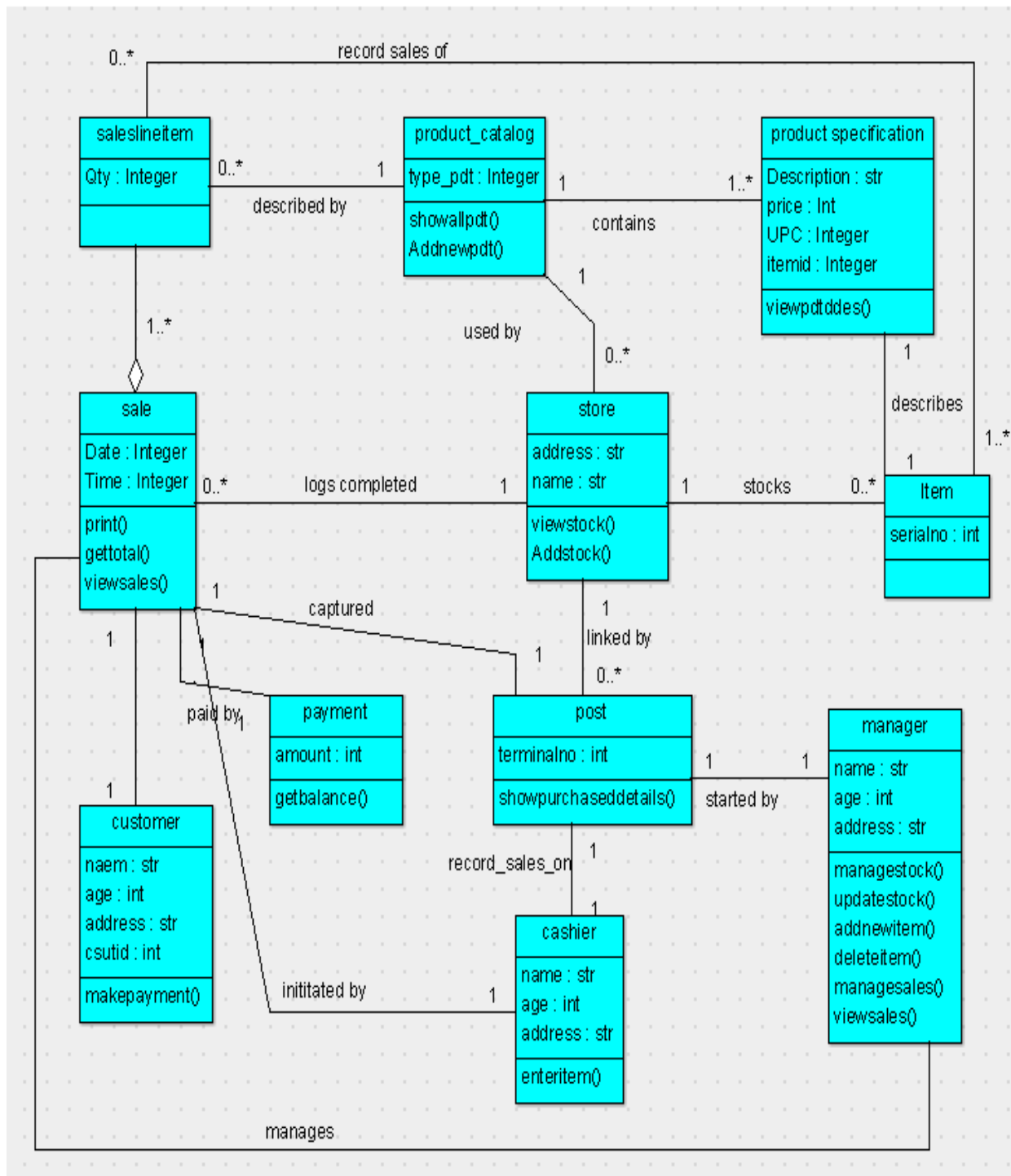
cashier
name : str
age : int
address : str
enteritem()

e)**Aim:** Represent usecases and domain class diagram

-USECASE DIAGRAM:



-CLASS DIAGRAM:



f) Aim: Develop CRUD matrix to represent relationships between usecases and problem domain classes.

Entity	customer	customer	Product catalog	Product specification	Sale	store	payment	post	manager
Buy product	C,D	R	C,R,U	R	U	U		R	R
Barcode		R			R		C		

scanning									
Pay bill	R	C					R		
Process sale			R		C				
Close sale			C,R	R	R	U		R	R
Update inventory			C,R	U	U	U		R	U
Tax calculation		C					R	R	C,R

CASE STUDY-3:**LIBRARY MANAGEMENT SYSTEM**

a) Aim: Identify and analyze events

The flow of events of a usecase contains the most important information derived from usecase modeling work. It should describe the usecases flow of events clearly enough for an outsider to easily understand it. Remember flow of events should present what the system does, not how the system is designed to perform the required behavior.

Guidelines for the contents of the flow of events are,

- 1) Describe how the usecase starts and ends.
- 2) Describe what data is exchanged between the actor and the usecase
- 3) Do not describe the details of the user interface, unless it is necessary to understand the behavior of the system.

b) Aim: Identify usecases

Procedure:

a) Register member: here the members who can access the library are registered based on the id nos.

issue book:

verify member: before issuing book the specific member is verified whether he/she is registered or not

check availability or book: after verifying the person then the availability of book which he/she selected is checked whether the book is available or not. after verifying & checking the book is issued

c) Return book: after the specific time is over the person should return book during returning

Calculate fine: the fine is calculated

d) Enquiry: in this the member is checked if fine is existed or not

e) Maintaining books:

This is done by librarian that how many books are present, person or members login id and their registrations etc.

Graphical Representation:



return book

enquiry

calculate fine

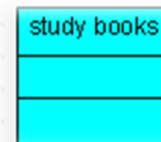
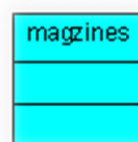
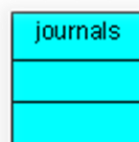
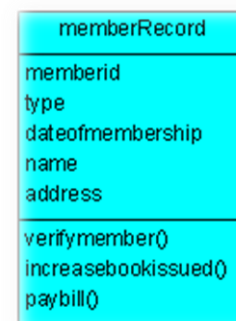
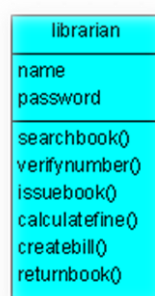
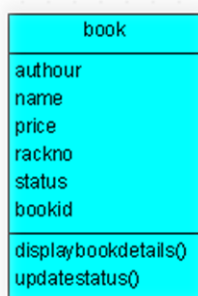
maintaining books

c) **Aim:** Develop event table

Event table:

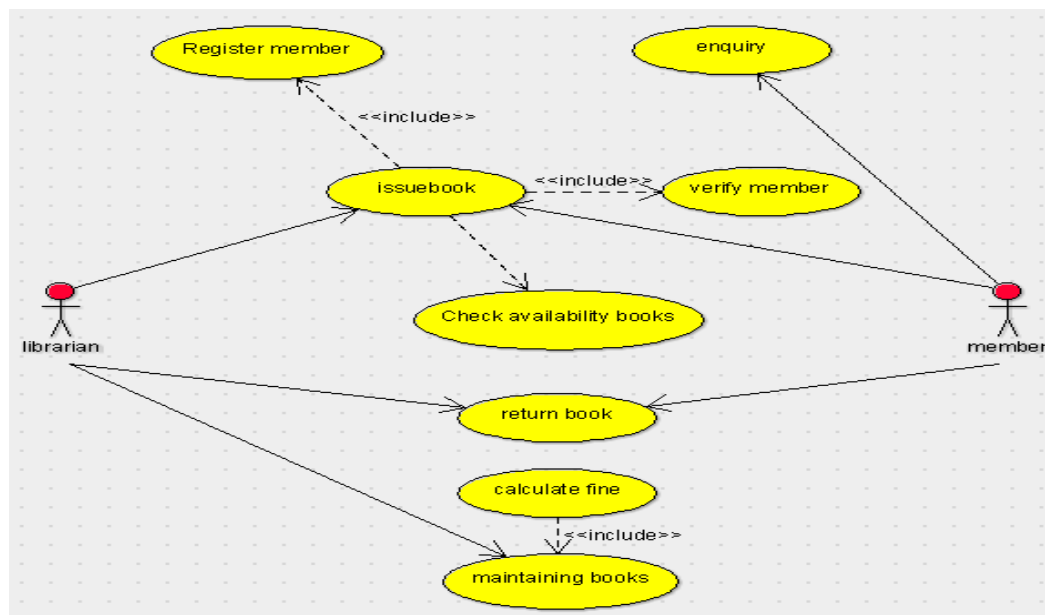
Usecase	Description
Register member	The members who can access the library registered based on the ids
Issue book verify member	Before issuing the book the specific member is verified whether he/she is registered or not
Check availability of book	Check the book is available or not
Return book	After the specific time is over the person should return
Calculate time	The fine is calculated in their fine
Enquiry	In this the member is checked if fine is existed or not
Maintaining books	This is done by the librarian to cross checking the books

d)**Aim:** Identify and analyze domain classes.

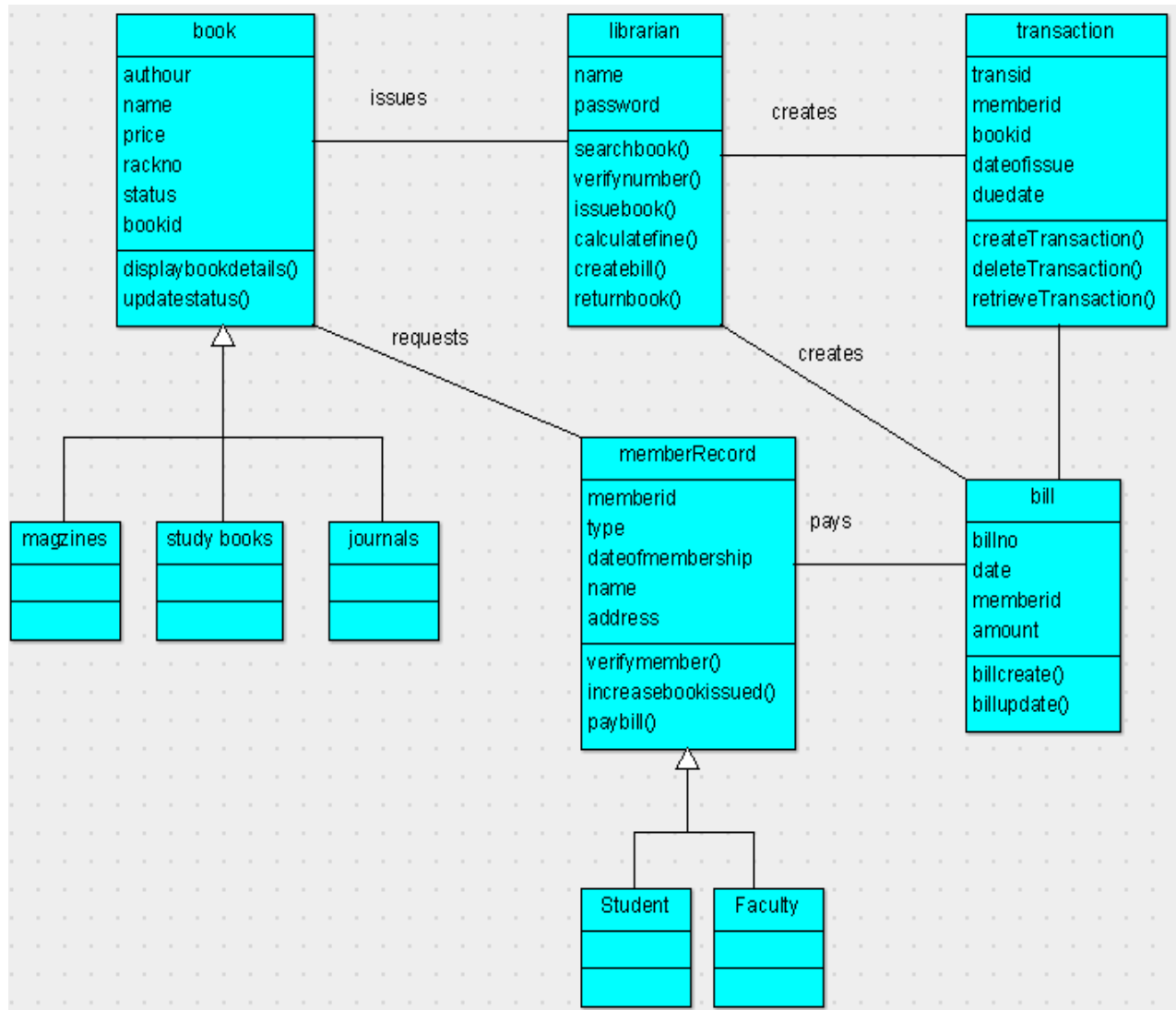


d) **Aim:** Represent class diagram for library management system.

-USECASE DIAGRAM:



-USECASE DIAGRAM:



f) **Aim:** Develop CRUD matrix to represent relationships between usecase and problem domain classes.

Entity /function	Libra-rian	Trans - action	book	Jou- rnals	Study books	Mag zines	bil l	Membe r record	Studen t	facult y
Register member	U,C, D									
Enquiry	C							R	R	R
Issue	C		R	R	R	R				
Book										
Verify	R									
Member										
Check			R	R	R	R		R	R	R
Availabilit y										
Of book										
Return book	R	R							C	C
Calculate	C	C							R	R
Fine										
Maintain	C									
books										

RESULT:

Domain model for the given problems is drawn and the design is successfully completed by using argo UML.

EXP NO:5**DATE:** _____**AIM:**

To develop sequence and collaboration diagram for a given problem.

DESCRIPTION:**Modeling steps for Sequence Diagrams:**

- Set the context for the interactions, system, subsystem, classes, object or use cases.
- Set the stages for the interactions by identifying objects which are placed as actions in interaction diagrams.
- Lay them out along the X-axis by placing the important object at the left side and others in the next subsequent.
- Set the lifelines for each and every object by sending create and destroy messages.
- Start the message which is initiating interactions and place all other messages in the increasing order of items.
- Specify the time and space constraints.
- Set the pre and post conditioned.

Modeling steps for Collaboration Diagrams:

- Set the context for interaction, whether it is system, subsystem, operation or class or one scenario of use case or collaboration.
- Identify the objects that play a role in the interaction. Lay them as vertices in graph, placing important objects in centre and neighboring objects to outside.
- Set the initial properties of each of these objects. If the attributes or tagged values of an object changes in significant ways over the interaction, place a duplicate object, update with these new values and connect them by a message stereotyped as become or copy.
- Specify the links among these objects. Lay the association links first represent a structural connection. Lay out other links and adorn with stereotypes.
- Starting with the message that initiates this interaction, attach each subsequent message to appropriate link, setting sequence number as appropriate.
- Adorn each message with time and space constraints if needed
- Attach pre & post conditions to specify flow of control formally.

ALGORITHMIC PROCEDURE:

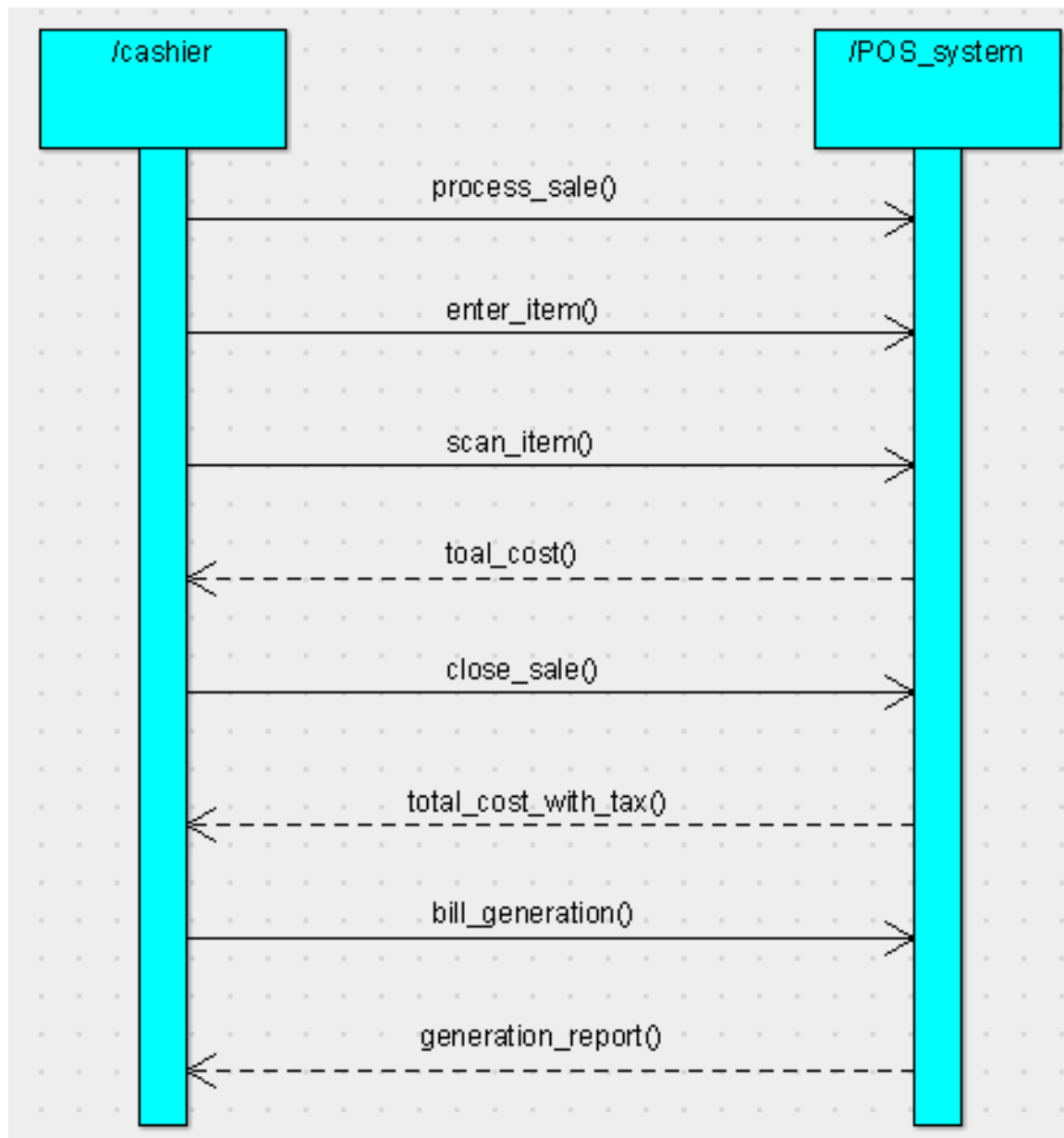
STEP 1: Start the application

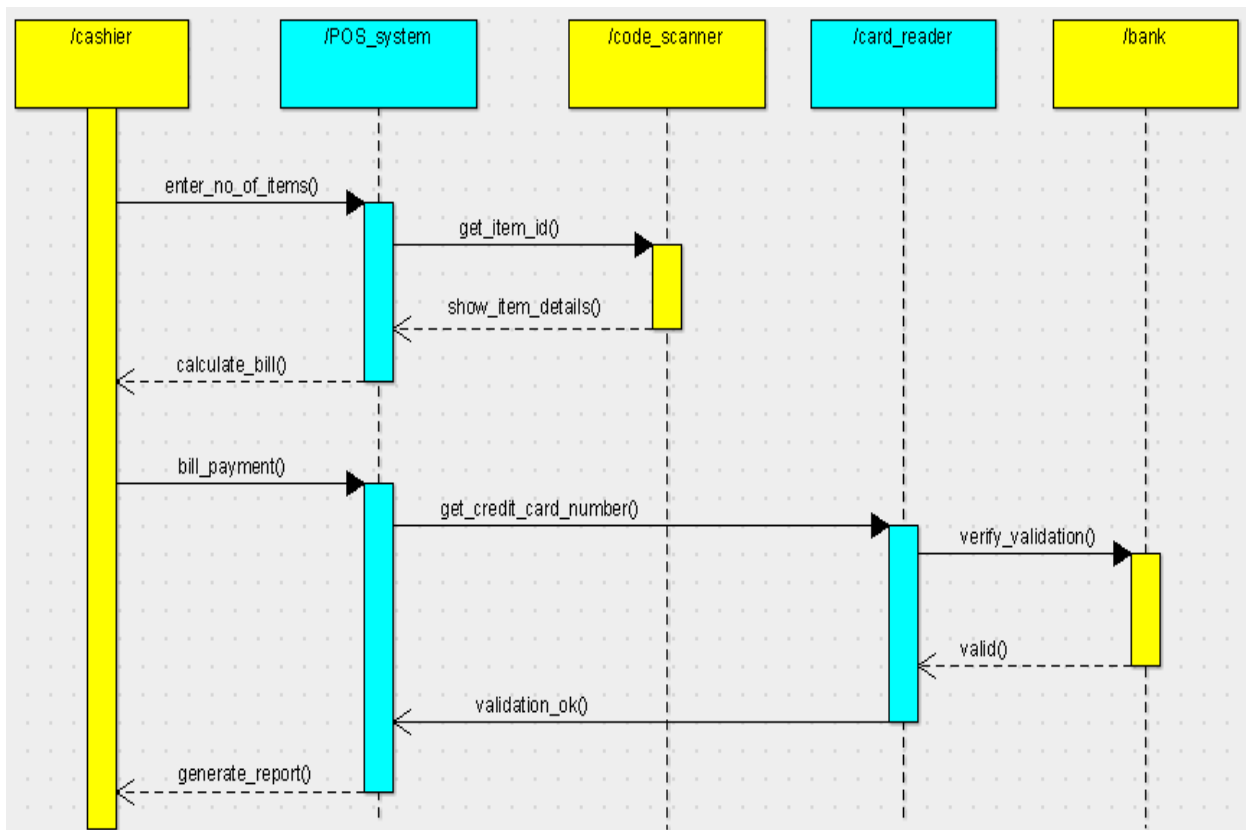
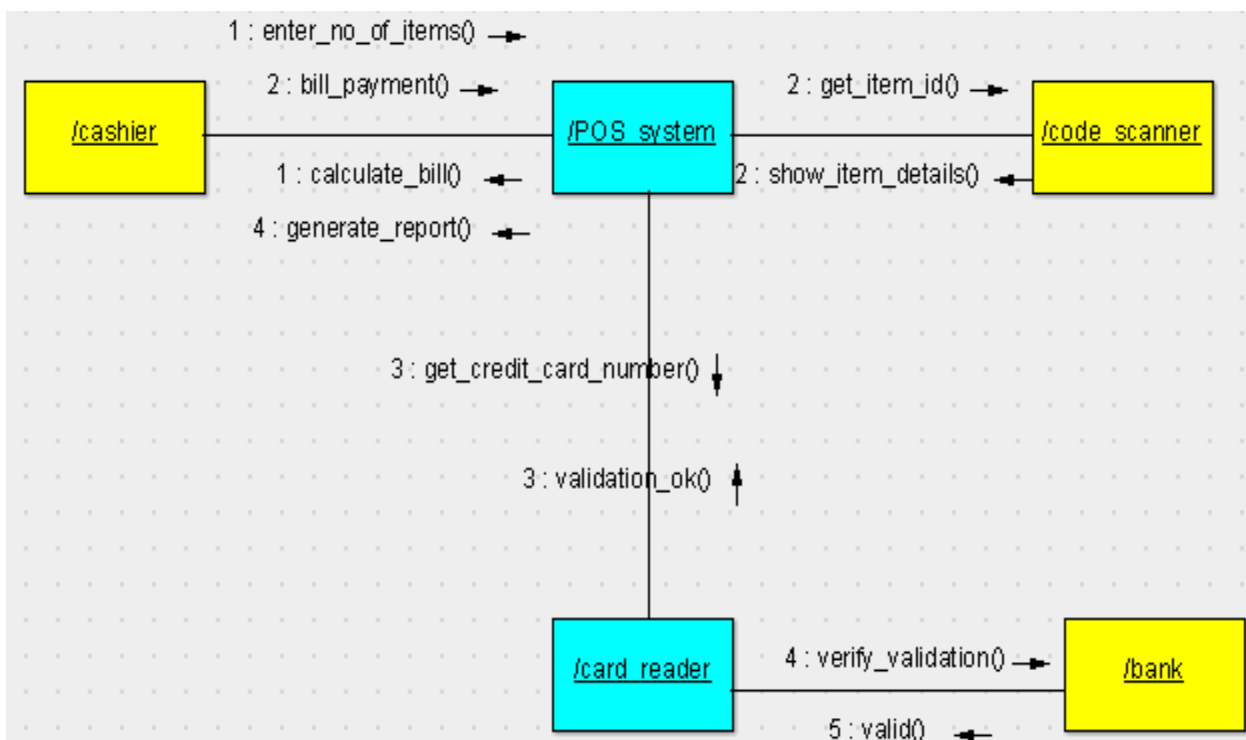
STEP 2: Create the require actors and usecases in the browser window

STEP 3: Go to new usecase view and then click the usecase view and open a new package

STEP 4: Rename the new package with the package with required names&4

STEP 5: Create two packages actor and usecase.

POS system sequence diagram:

Process Sale Sequence Diagram:**Process Sale Collaboration Diagram:****RESULT:**

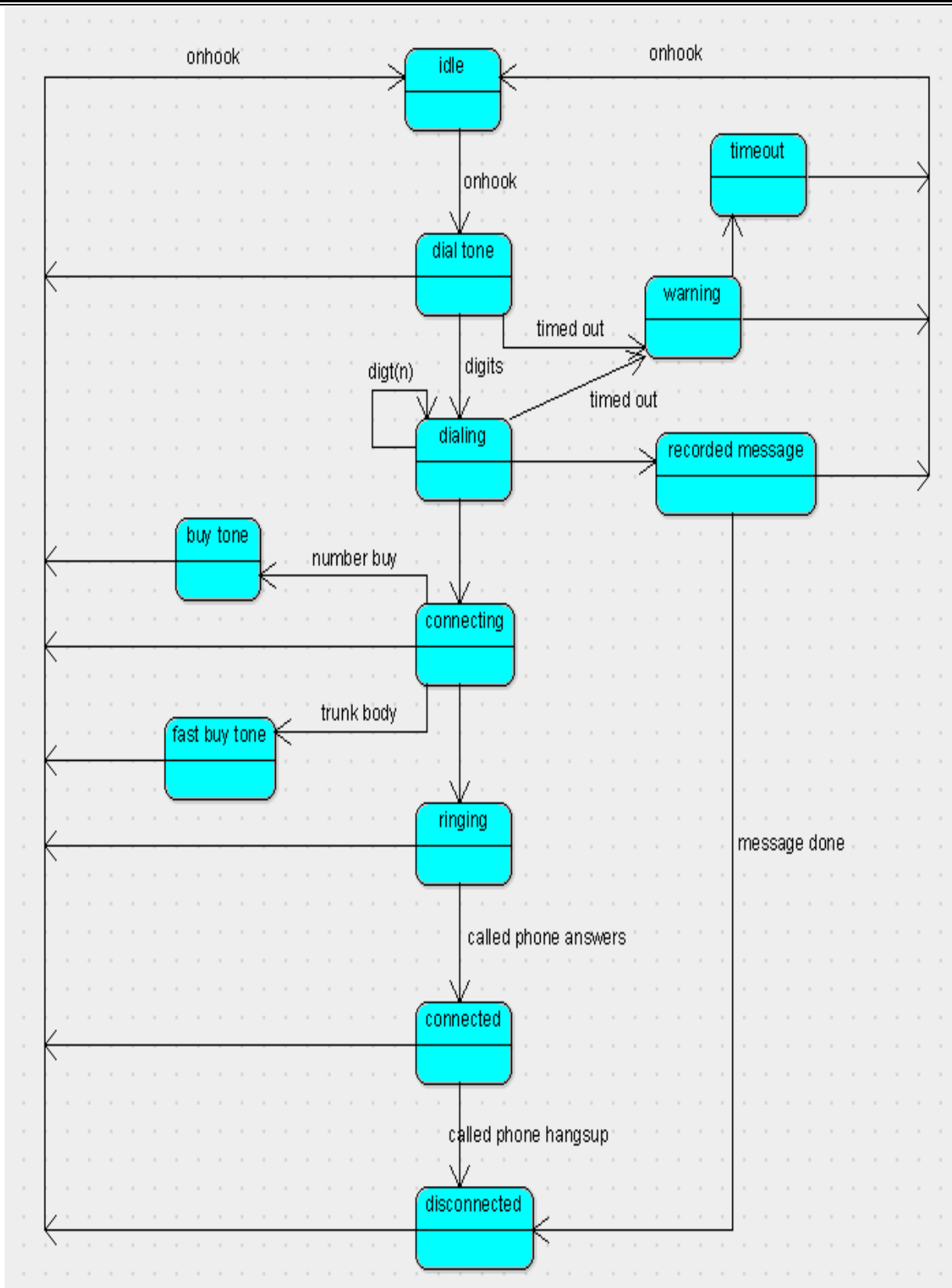
Sequence and Collaboration diagrams for a given problem are drawn successfully by using argo UML.

EXP NO: 6**DATE:_____****AIM:**

To develop a state model diagram for a given problem.

DESCRIPTION:**Modeling steps for Statechart Diagram**

- Choose the context for state machine, whether it is a class, a usecase, or the system as a whole.
- Choose the initial & final states of the objects.
- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high level states of the objects & only then consider its possible substates.
- decide on the meaningful partial ordering of stable states over the lifetime of the object.
- Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
- Attach actions to these transitions and/or to these states.
- Consider ways to simplify your machine by using substates, branches, forks, joins and history states.
- Check that all states are reachable under some combination of events.
- Check that no state is a dead from which no combination of events will transition the object out of that state.
- Trace through the state machine, either manually or by using tools, to check it against expected sequence of events & their responses.

**RESULT:**

State chart model for the given problem is drawn successfully by using argo UML.

EXP NO:7**DATE:**_____**AIM:**

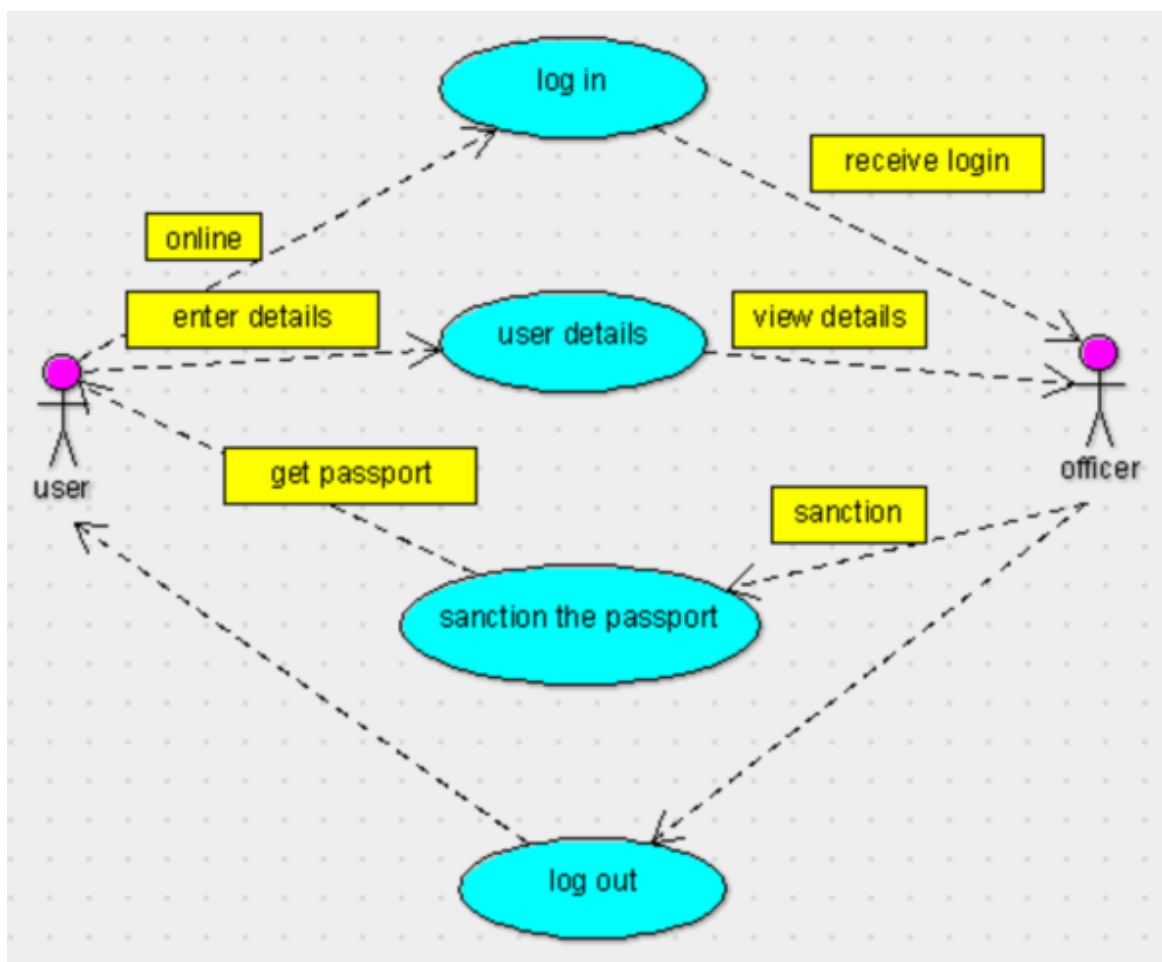
To generate JAVA/C++ code for the design solution developed for the given problems.

DESCRIPTION:

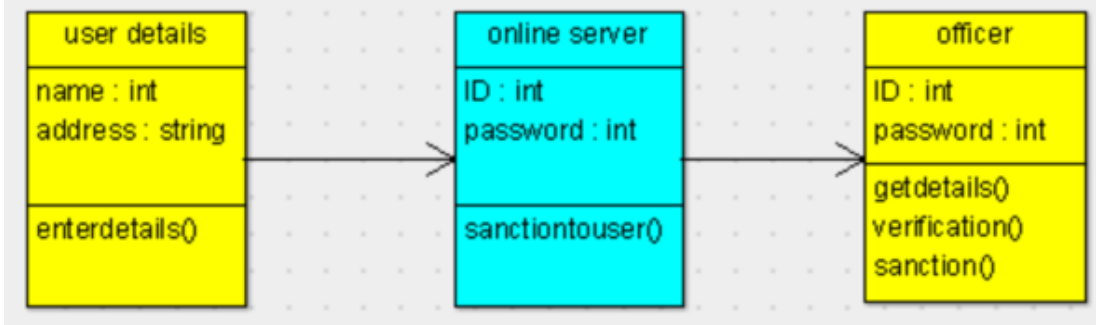
Passport automation system for various diagrams is shown below.

1.PASSPORT AUTOMATION SYSTEM:**USECASE DIAGRAM:**

- Use case diagrams are a way to capture the system's functionality and requirements in UML diagrams.
- It captures the dynamic behavior of a live system.
- A use case diagram consists of a use case and an actor.

**CLASS DIAGRAM:**

- A class diagram is a type of diagram and part of a unified modeling language (UML) that defines and provides the overview and structure of a system in terms of classes, attributes and methods, and the relationships between different classes.
- It is used to illustrate and create a functional diagram of the system classes and serves as a system development resource within the software development life cycle.

**SKELETON CODE:**

JAVA:

```

import java.util.List;

public class user details {

    public int name;

    public string address;

    public List<online server> online server;

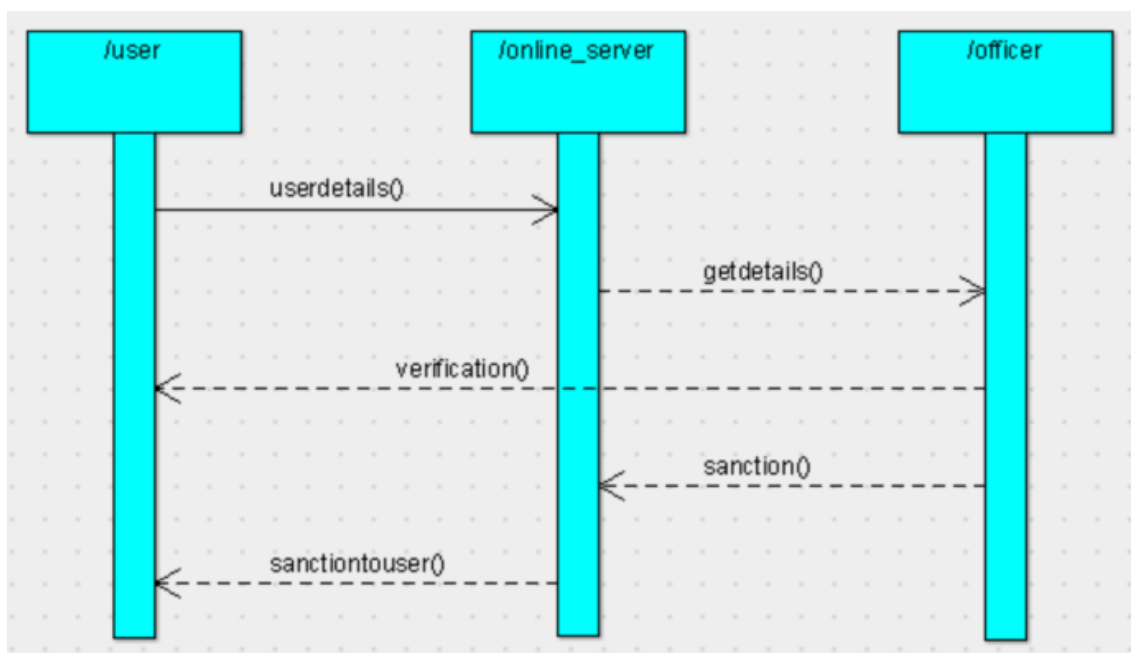
    public void enterdetails() {

    }

}
  
```

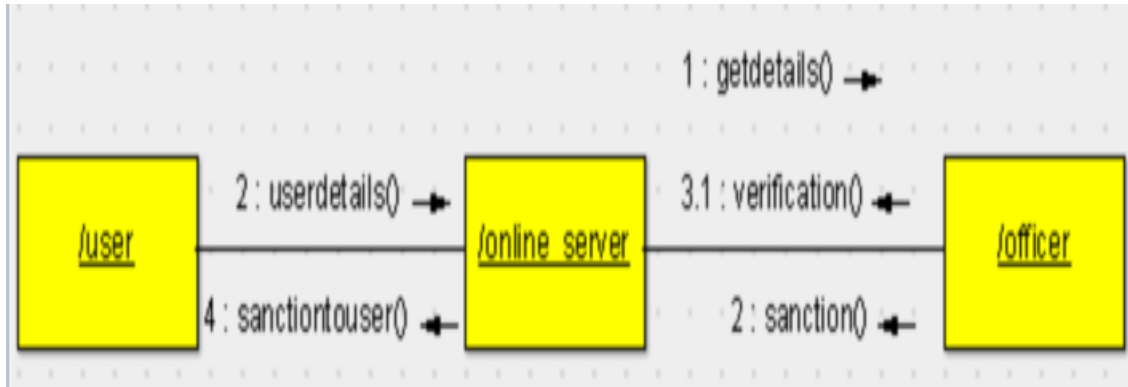
SEQUENCE DIAGRAM:

- A sequence diagram, in the context of UML, represents object collaboration and is used to define event sequences between objects for a certain outcome.
- A sequence diagram is an essential component used in processes related to analysis, design and documentation.

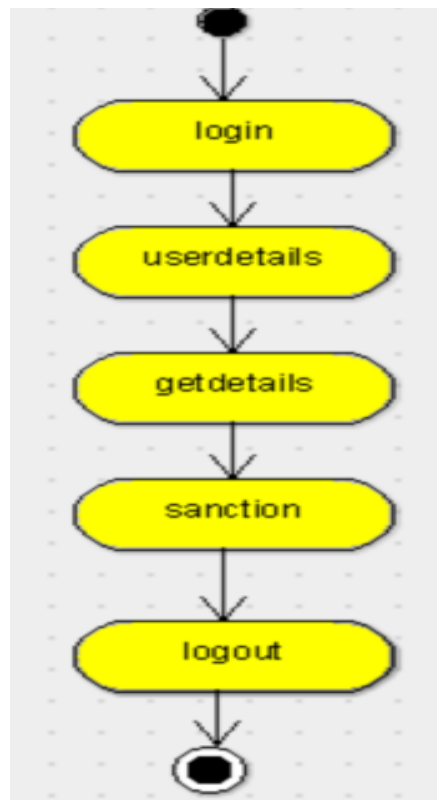


COLLABORATION DIAGRAM:

- The collaboration diagram is used to show the relationship between the objects in a system.
- Both the sequence and the collaboration diagrams represent the same information but differently.
- Instead of showing the flow of messages, it depicts the architecture of the object residing in the system as it is based on object-oriented programming. An object consists of several features.
- Multiple objects present in the system are connected to each other.
- The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.

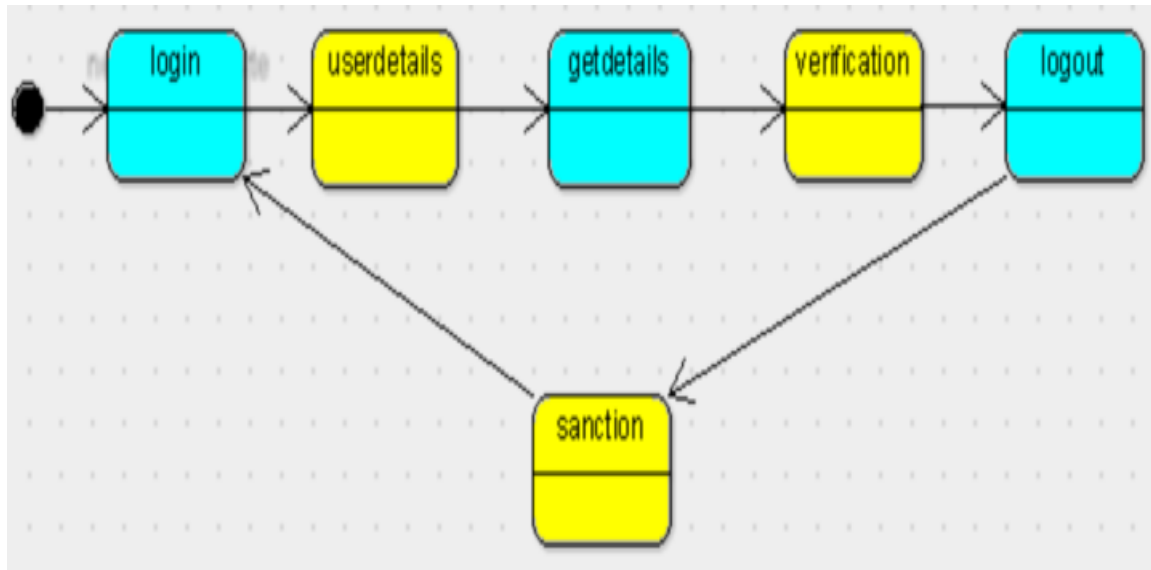
**ACTIVITY DIAGRAM:**

- The activity diagram is used to demonstrate the flow of control within the system rather than the implementation. It models the concurrent and sequential activities.
- It is also termed as an object-oriented flowchart.

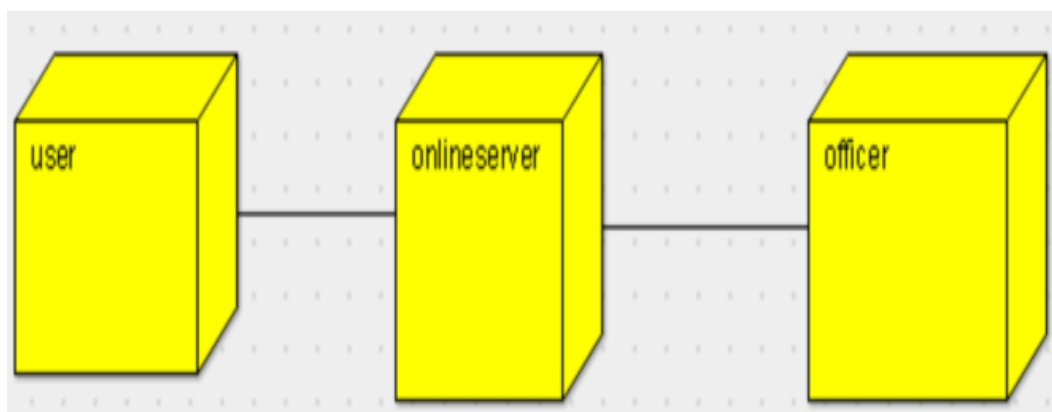


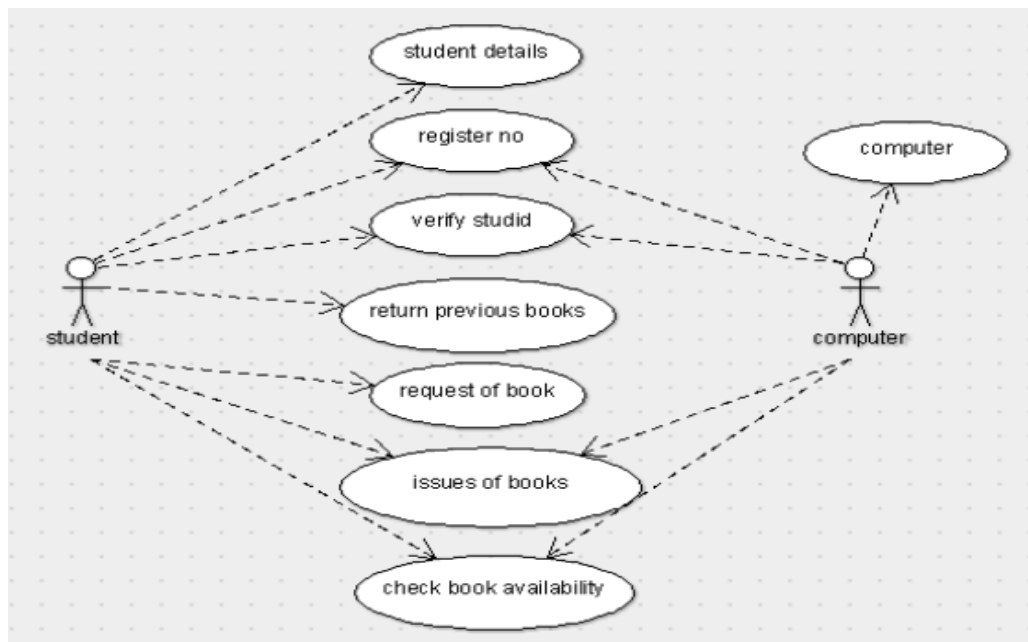
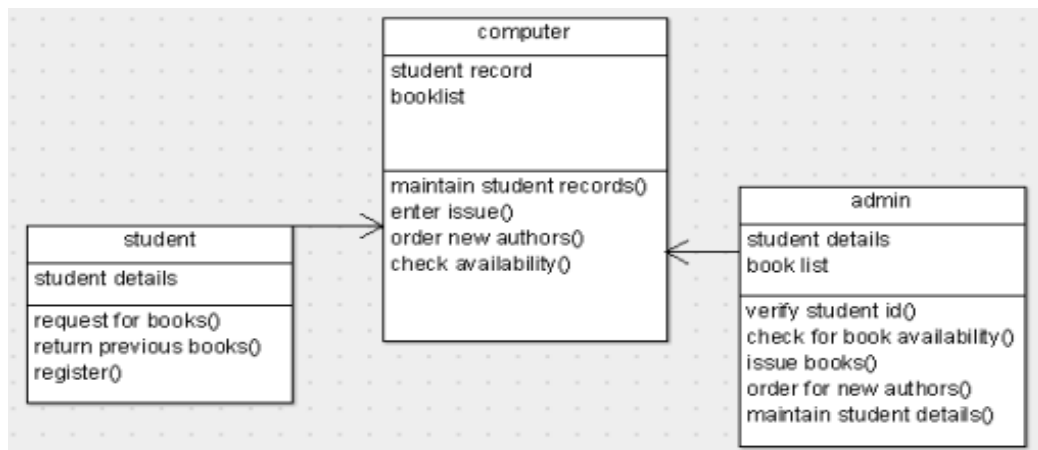
STATE-CHART DIAGRAM:

A state diagram, also known as a state machine diagram or statechart diagram, is an **illustration of the states an object can attain** as well as the transitions between those states in the Unified Modeling Language (UML)

**DEPLOYMENT DIAGRAM:**

- The deployment diagram visualizes the physical hardware on which the software will be deployed.
- It portrays the static deployment view of a system.
- It involves the nodes and their relationships.
- It ascertains how software is deployed on the hardware.
- It maps the software architecture created in design to the physical system architecture, where the software will be executed as a node.
- Since it involves many nodes, the relationship is shown by utilizing communication paths.



2.BOOK BANK REGISTRATION SYSTEM:**USECASE DIAGRAM:****CLASS DIAGRAM:****SKELETON CODE:**student.java

```

import java.util.vector;
public class student {
    private student details;
    public Vector mycomputer;
    public void request for books() {
    }
    public void return previous books() {
    }
    public void register() {
    }
}
  
```

computer.java

```

public class computer {
  
```

```

private student record;

public booklist;

public void maintain student records() {
}

public void enter issue() {
}

public void order new authors() {
}

public void check availability() {
}

}

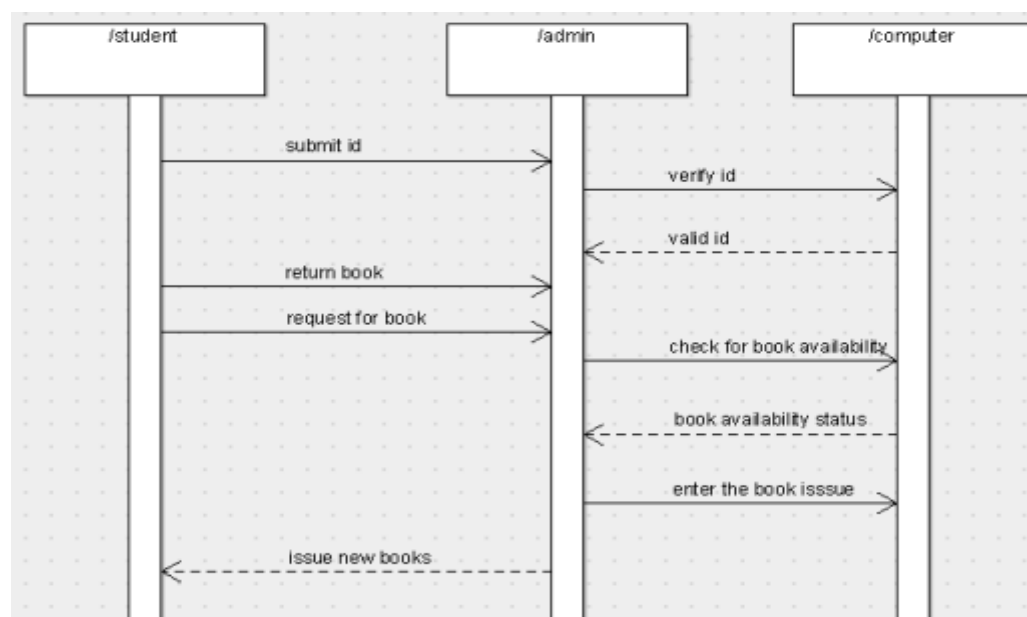
Import java.util.Vector;
public class admin {
private student details;
public book list;
public Vector mycomputer;
public void verify student id() {
}
public void check for book availability() {
}
public void issue books() {
}

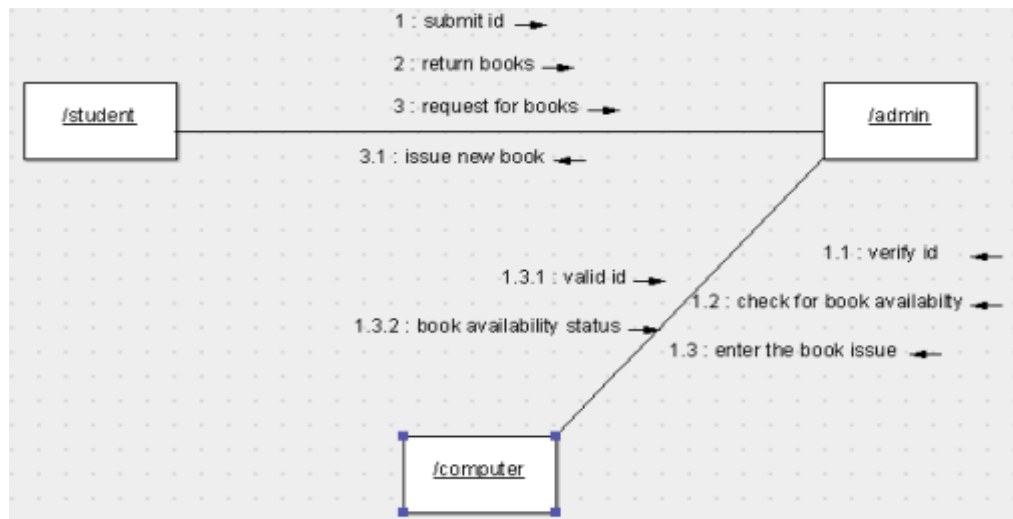
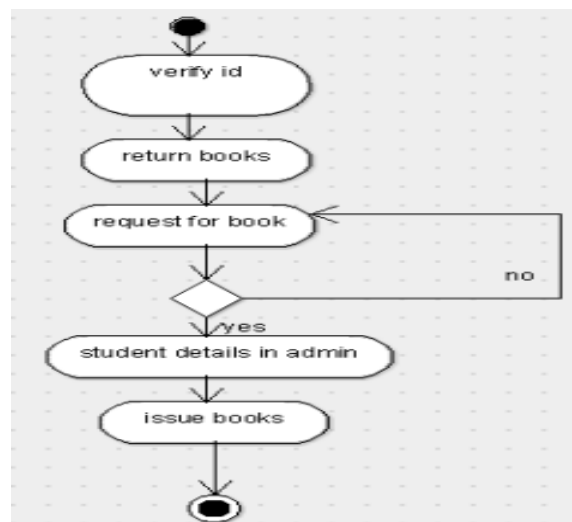
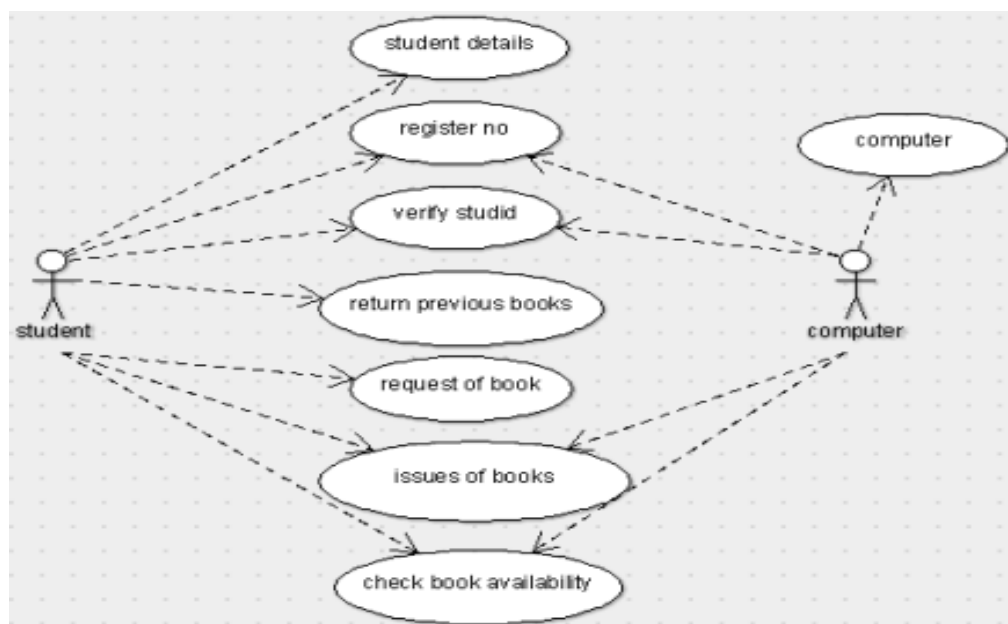
public void order for new authors() {
}

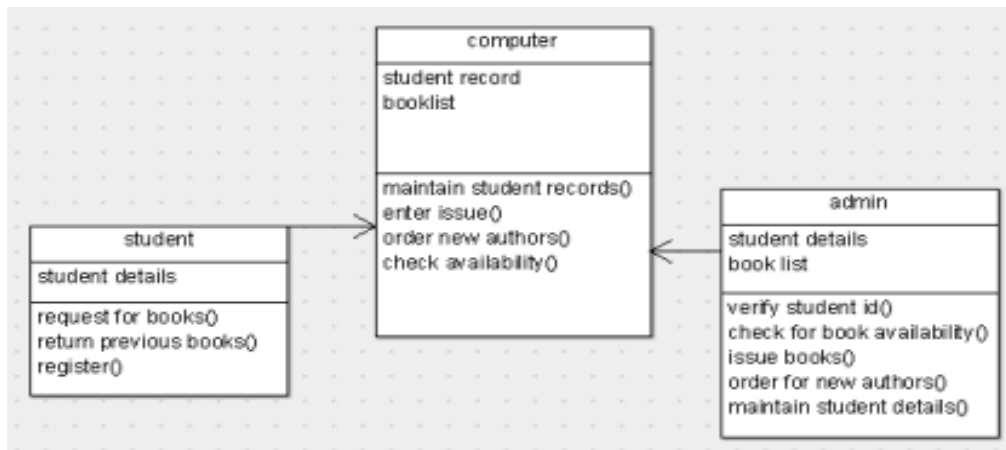
public void maintain student details() {
}
}

```

SEQUENCE DIAGRAM:



COLLABORATION DIAGRAM:**ACTIVITY DIAGRAM:****3. EXAM REGISTRATION SYSTEM:****USECASE DIAGRAM:**

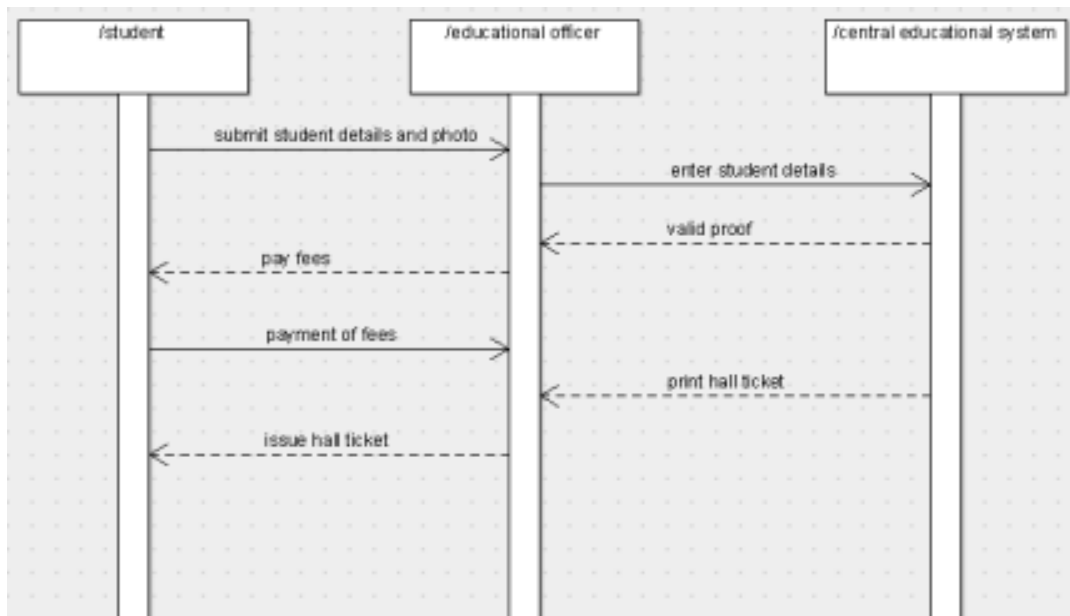
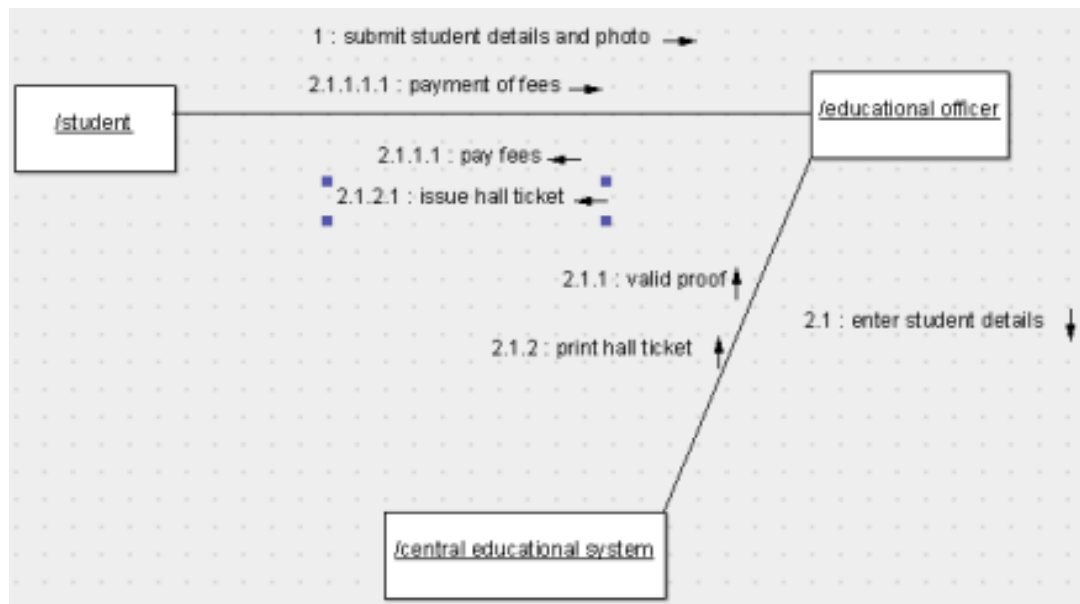
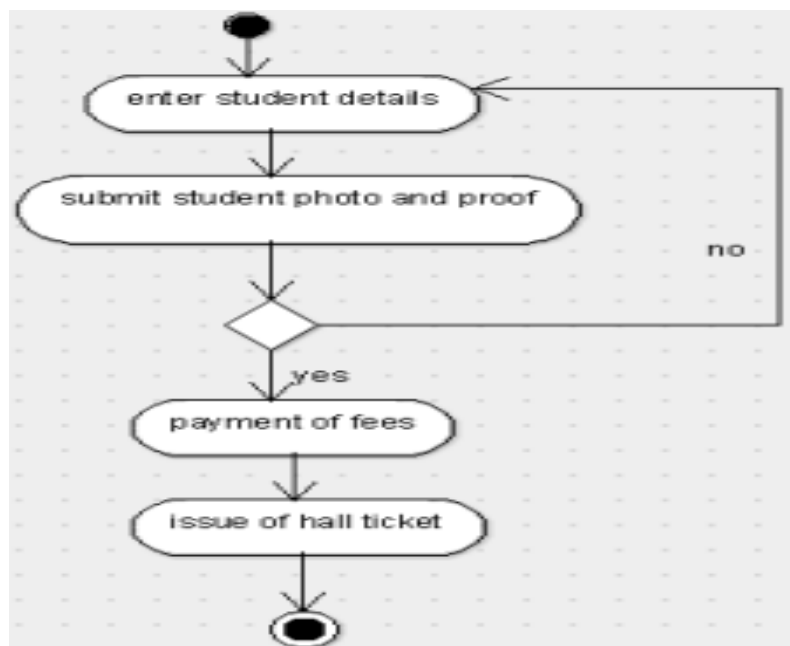
CLASS DIAGRAM:**SKELETON CODE:**

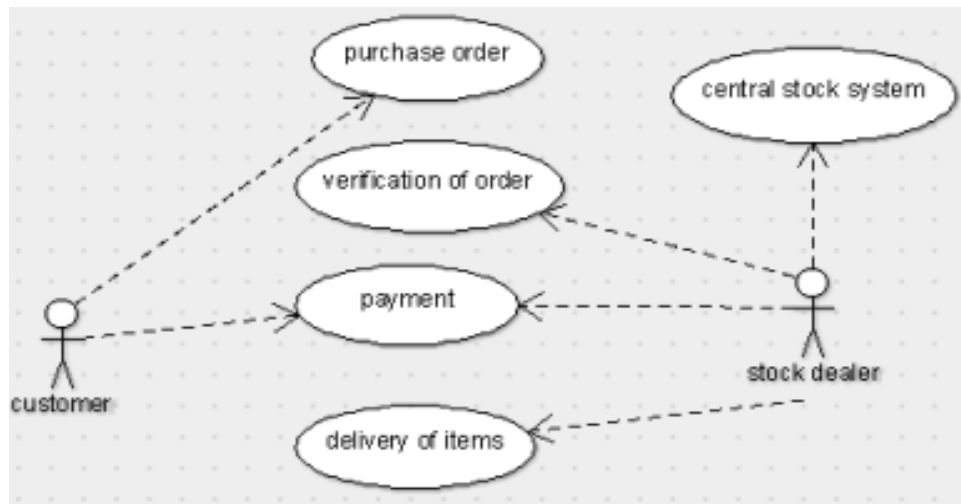
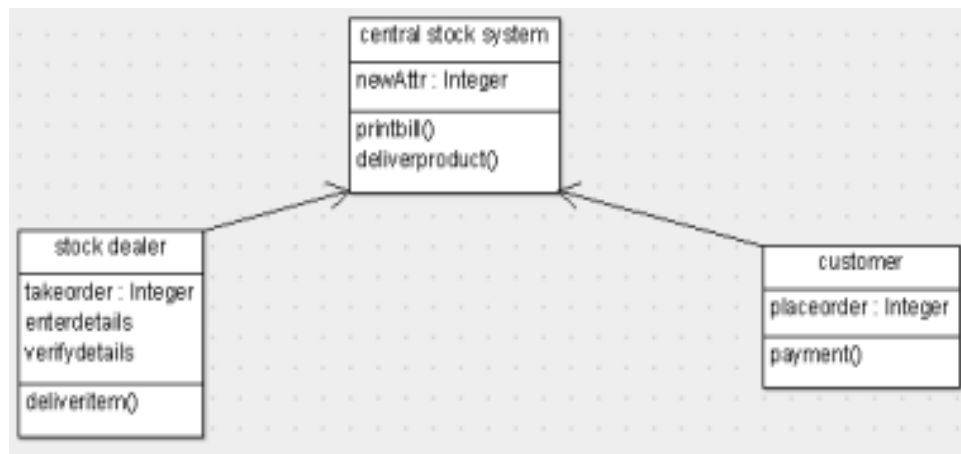
```

central educational system.java
public class central educational system {
private Integer details;
public void printhallticket() {
}
public void issuehallticket() {
}
public void verifydetails() {
}
}

eduofficer.java
import java.util.Vector;
public class edu officer {
private Integer details;
public Vector mycentral educational system;
private void issuehallticket() {
}
public void verifyproof() {
}
}

stud.java
import java.util.Vector;
public class stud {
public Integer submitdetails;
public submitphoto;
public Vector mycentral educational system;
public void paymentoffees() {
}
}
  
```

SEQUENCE DIAGRAM:**COLLABORATION DIAGRAM:****ACTIVITY DIAGRAM:**

4. STOCK MAINTAINCE SYSTEM:**USECASE DIAGRAM:****CLASS DIAGRAM:****SKELETON CODE:**

```

central educational system.java
public class central educational system {
private Integer details;
public void printhallticket() {
}
public void issuehallticket() {
}
public void verifydetails() {
}
}
eduofficer.java
import java.util.Vector;
public class edu officer {
private Integer details;
public Vector mycentral educational system;
private void issuehallticket() {
}
public void verifyproof() {
}
}
stud.java
import java.util.Vector;
public class stud {

```

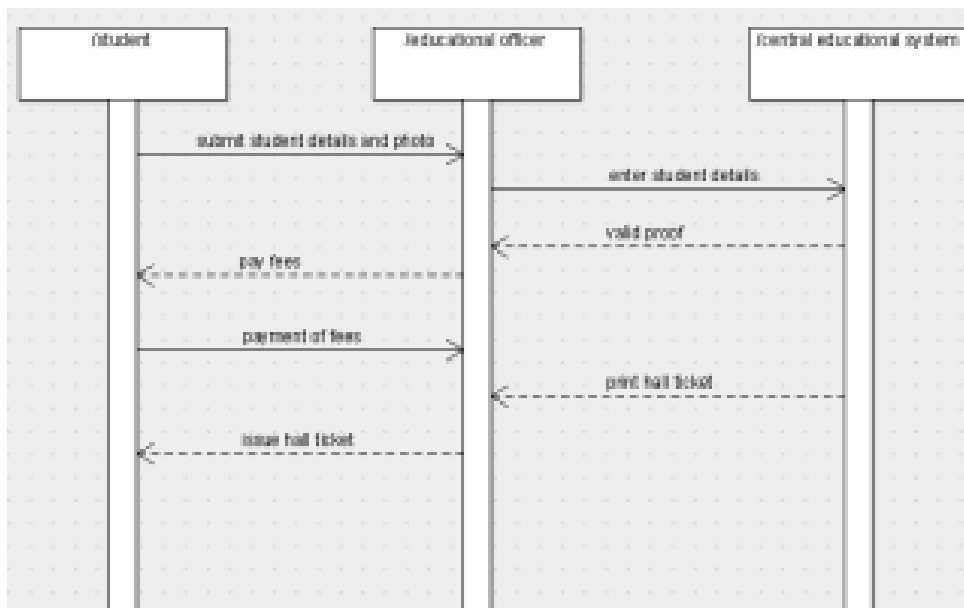


```

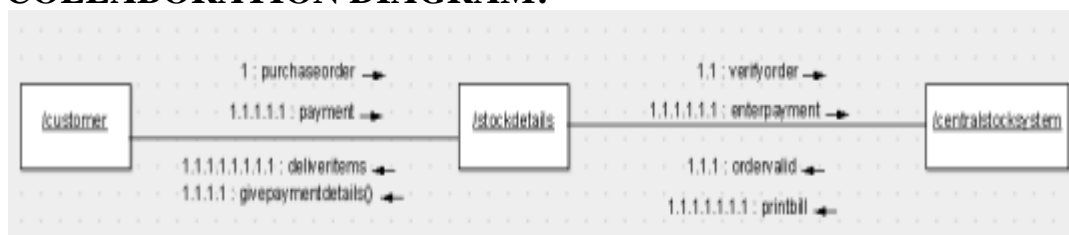
public Integer submitdetails;
public submitphoto;
public Vector mycentral educational system;
public void paymentoffees() {
}
}

```

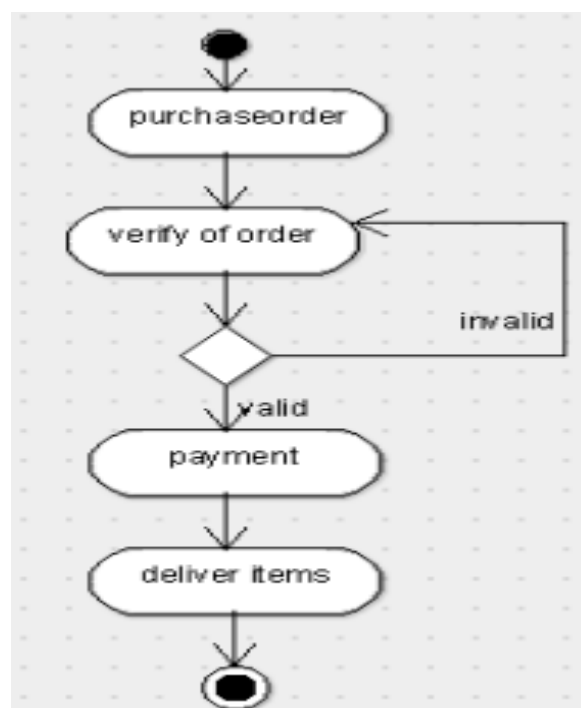
USECASE DIAGRAM:



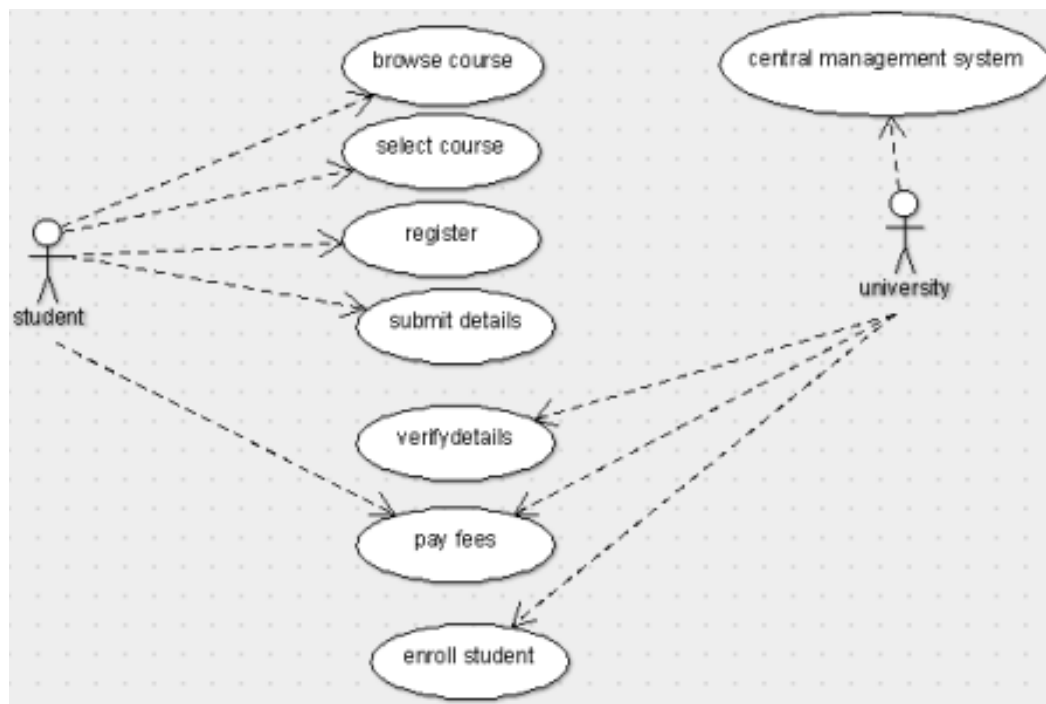
COLLABORATION DIAGRAM:



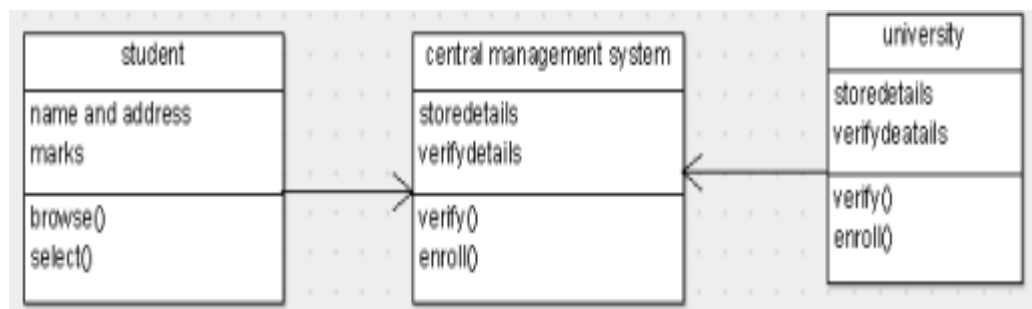
ACTIVITY DIAGRAM:



5. ONLINE COURSE RESERVATION SYSTEM: USECASE DIAGRAM:



CLASS DIAGRAM:



SKELETON CODE:

```

central management system.java
public class central management system {
private storedetails;
public verifydetails;
public void verify() {
}
public void enroll() {
}
}
student.java
import java.util.Vector;
public class student {
public name and
public marks;
public Vector mycentral management system;
public void browse() {
}
public void select() {
}
}
university.java
import java.util.Vector;

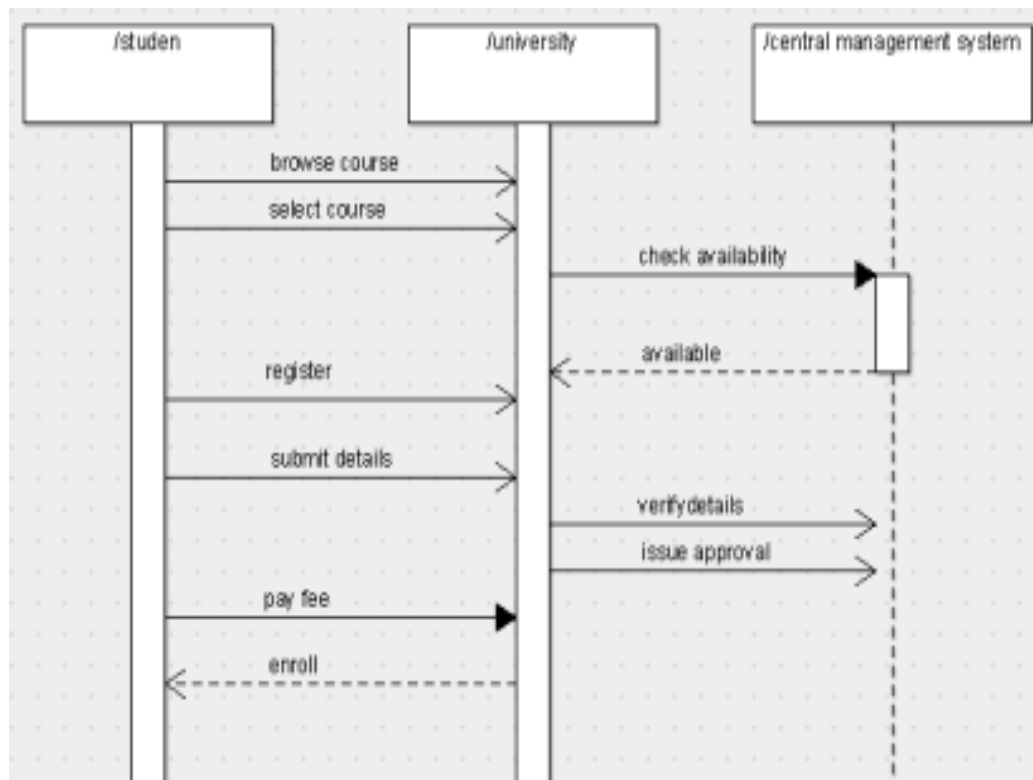
```

```

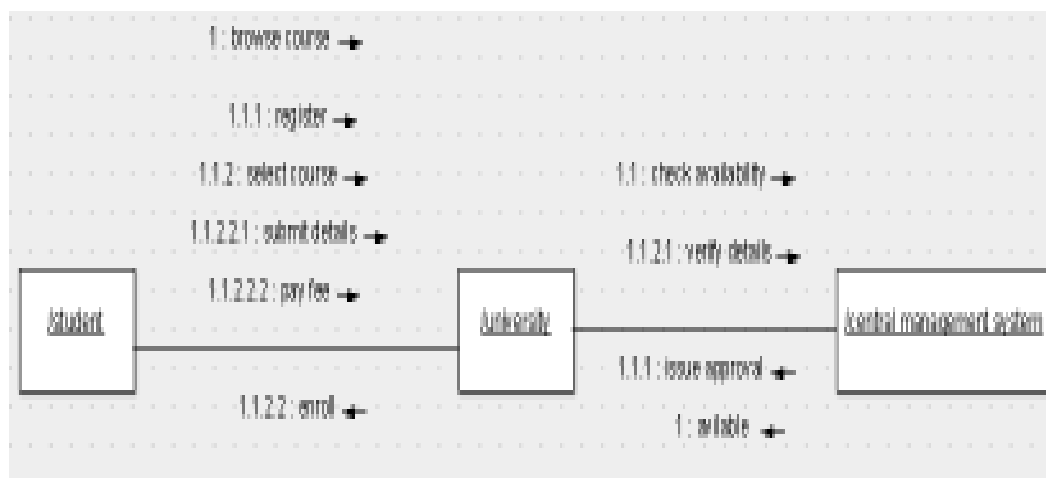
public class university {
public store details;
public verify deatails;
public Vector mycentral management system;
public void verify() {
}
public void enroll() {
}
}

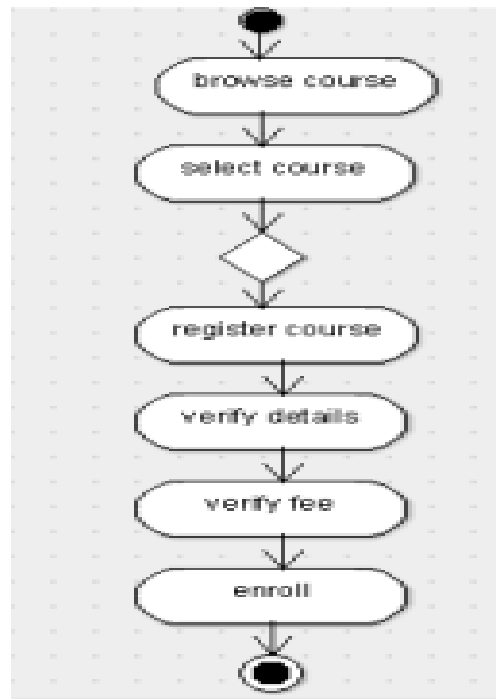
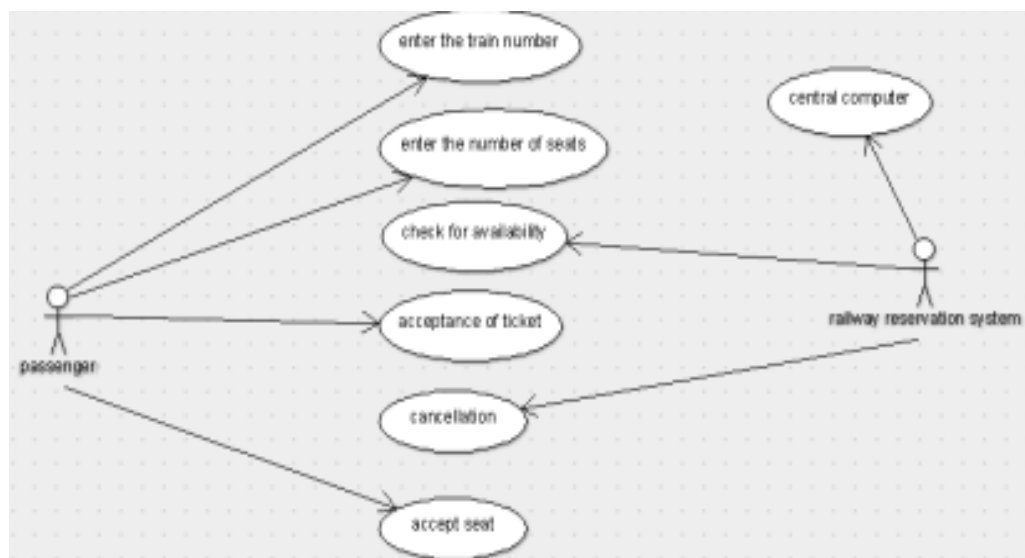
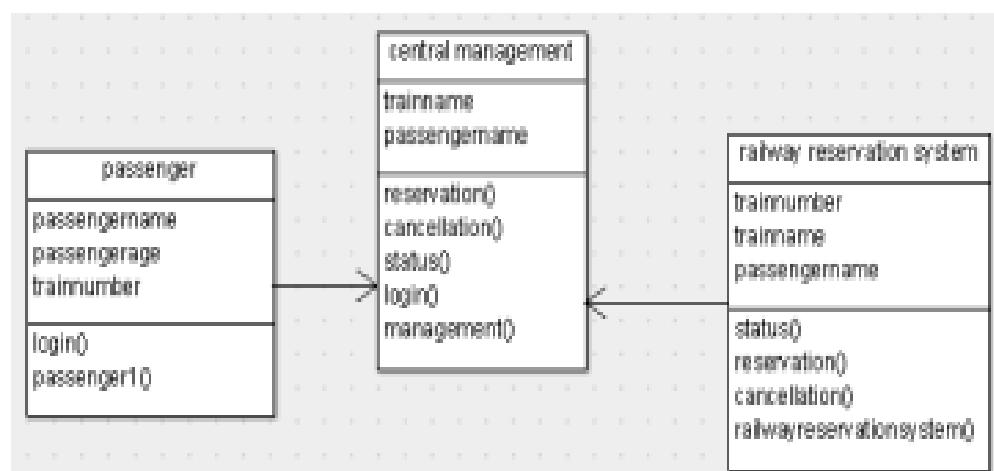
```

SEQUENCE DIAGRAM:



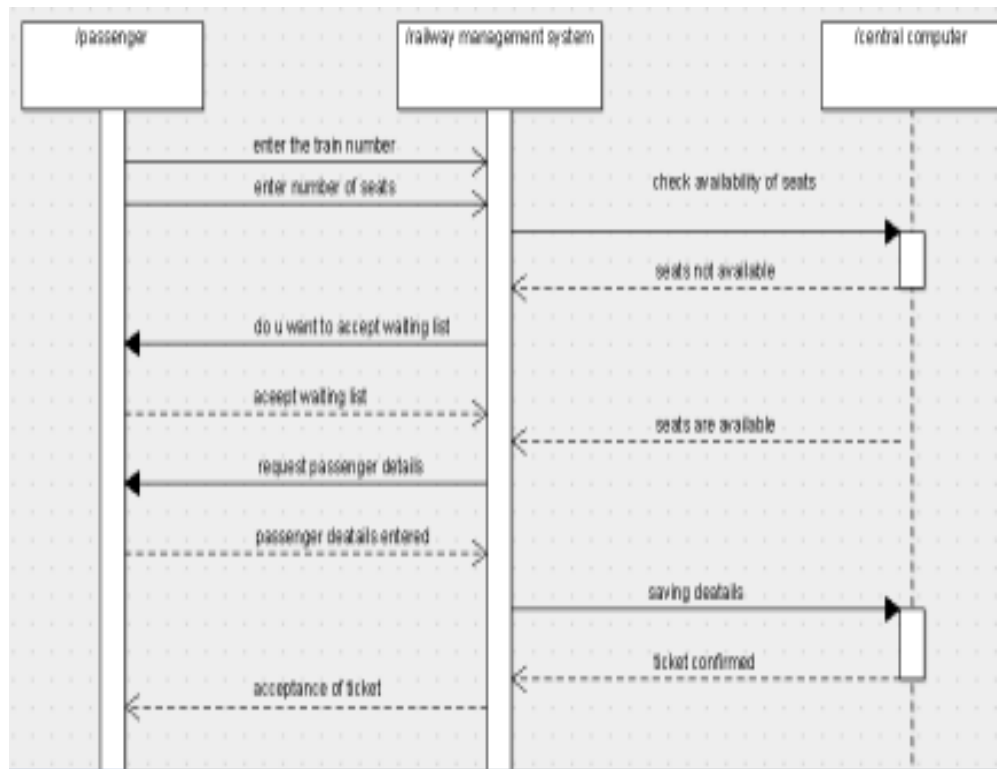
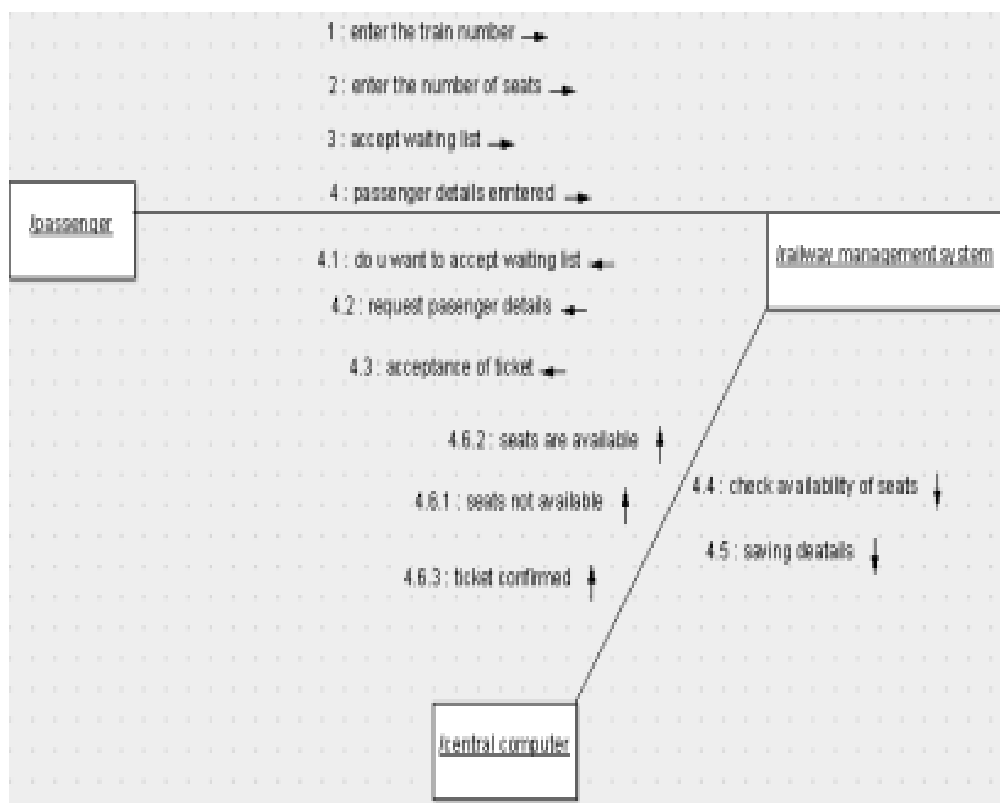
COLLABORATION DIAGRAM:

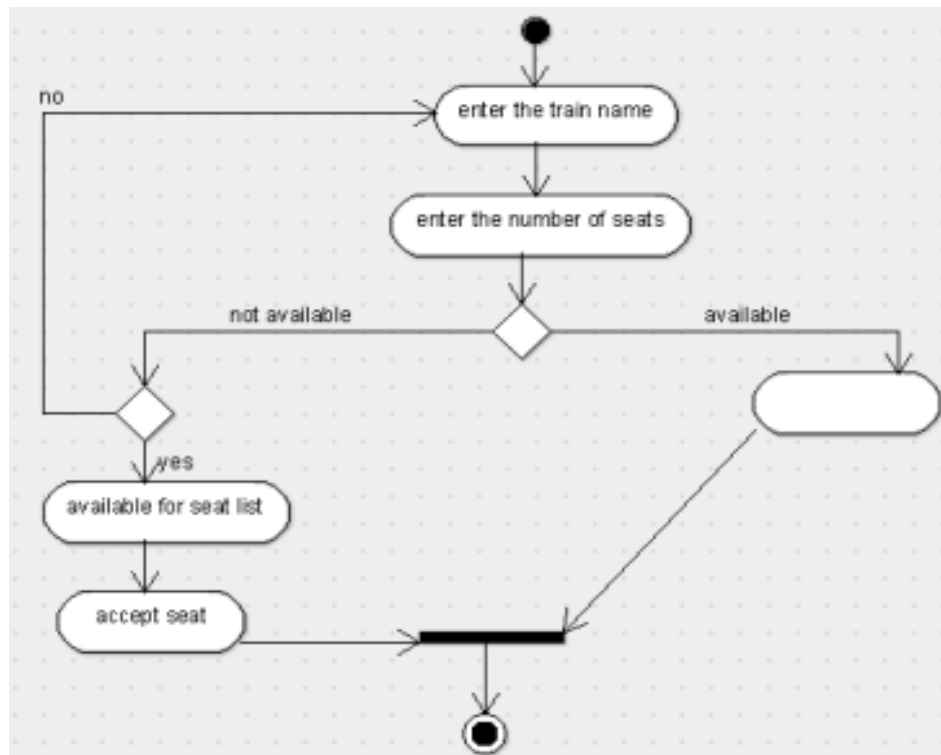


ACTIVITY DIAGRAM:**6.E-TICKETING:****USECASE DIAGRAM:****CLASS DIAGRAM:**

SKELETON CODE:

```
central management.java
public class central management {
public train name;
public passenger name;
public void reservation() {
}
public void cancellation() {
}
public void status() {
}
    public void login() {
}
private void management() {
}
}
passenger.java
import java.util.Vector;
public class passenger {
public passenger name;
public passenger age;
public train number;
public Vector my central management;
public void login() {
}
public void passenger1() {
}
}
railway reservation system.java
import java.util.Vector;
public class railway reservation system {
private train number;
public train name
public passenger name;
public Vector mycentral management;
public void status() {
}
public void reservation() {
}
public void cancellation() {
}
public void railway reservation system() {
}
}
```

SEQUENCE DIAGRAM:**COLLABORATION DIAGRAM:**

ACTIVITY DIAGRAM:**RESULT:**

All the given diagrams have successfully drawn for various uml diagrams and code is successfully generated for class diagrams by using argo uml.

EXP NO:8**DATE:**_____**AIM:**

To generate complete skeleton code by using argo uml.

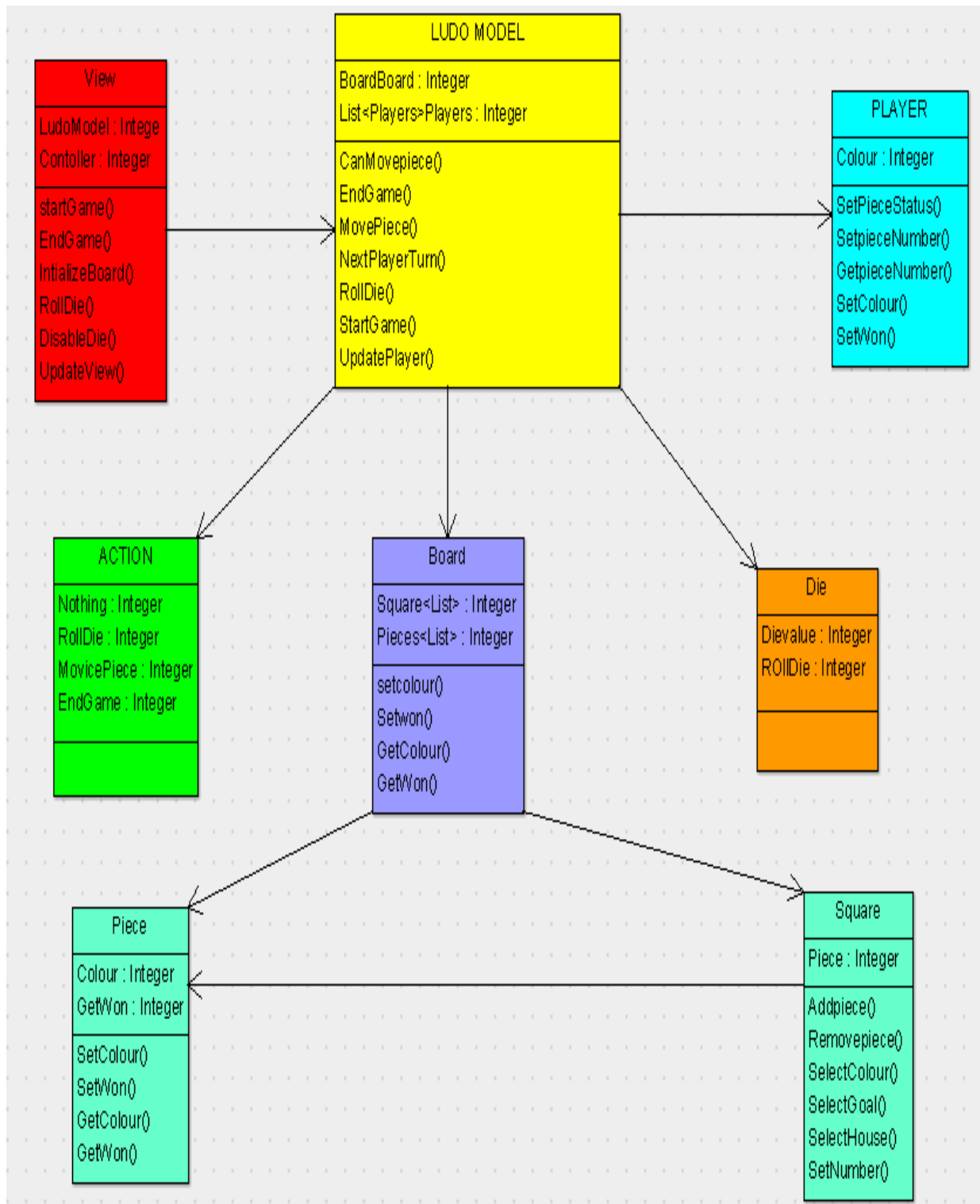
DESCRIPTION:**CLASS DIAGRAM:**

Class diagrams are used for only one of the UML static structure diagrams, the class diagram itself. Object diagrams are represented on the Argo UML deployment diagram.

- Helps you visualize the structural or static view of a system.
- Class diagrams show the relationships among class.
- Foundation for component and deployment diagrams.

Modeling steps for Class Diagrams:

- Identify the things that are interacting with class diagram.
- Set the attributes and operations.
- Set the responsibilities.
- Identify the generalization and specification classes.
- Set the relationship among all the things.
- Adron with tagged values, constraints and notes.

CLASS DIAGRAM FOR LUDO GAME:**SKELETON CODE:**Ludo model.h:

```

import java.util.List;
public class LUDO MODEL {
    public Integer BoardBoard;
    public Integer List<Players>Players;
    public List<PLAYER> pPLAYER;
    public List<ACTION> aACTION;
    public List<Board> board;
    public List<Die> die;
    public void CanMovepiece() {

```

```
}  
View.h:  
import java.util.List;  
    public class View {  
        public Integer LudoModel;  
    public Integer Contoller;  
        public List<LUDO MODEL> IUDO MODEL;  
        public void startGame() {  
    }  
    public void EndGame() {  
    }  
}
```

```
Action.h:  
public class ACTION {  
    public Integer Nothing;  
    public Integer RollDie;  
    public Integer MovicePiece;  
    public Integer EndGame;  
}  

```

```
Board.h:  
import java.util.List;  
public class Board {  
    public Integer Square<List>;  
    public Integer Pieces<List>;  
        public List<Piece> piece;  
        public List<Square> square;  
    public void setcolour() {  
    }  
    public void Setwon() {  
    }  
    public void GetColour() {  
    }  
    public void GetWon() {  
    }  
}  

```

```
Piece.h:  
public class Piece {  
    public Integer Colour;  
    public Integer GetWon;  
    public void SetColour() {  
    }  
    public void SetWon() {  
    }  
    public void GetColour() {  
    }  
    public void GetWon() {  
    }  
}  

```

```
Square.h:  
import java.util.List;  
public class Square {  
    public Integer Piece;  
        public List<Piece> piece;  
    public void Addpiece() {  
    }  
    public void Removepiece() {  

```

```
}  
public void SelectColour() {  
}  
public void SelectGoal() {  
}  
public void SelectHouse() {  
}  
public void SetNumber() {  
}  
}
```

Die.h:

```
public class Die {  
    public Integer Dievalue;  
    public Integer RollDie;  
}  
public class PLAYER {  
    public Integer Colour;  
    public void SetPieceStatus() {  
    }  
    public void SetpieceNumber() {  
    }  
    public void GetpieceNumber() {  
    }  
    public void SetColour() {  
    }  
    public void SetWon() {  
    }  
}
```

C++:

Ludo model.h:

```
#ifndef LUDO MODEL_h  
#define LUDO MODEL_h  
class PLAYER;  
class ACTION;  
class Board;  
class Die;  
class LUDO MODEL {  
public:  
    virtual void CanMovepiece();  
    virtual void EndGame();  
    virtual void MovePiece();  
    virtual void NextPlayerTurn();  
    virtual void RollDie();  
    virtual void StartGame();  
    virtual void UpdatePlayer();  
public:  
    Integer BoardBoard;  
    Integer List<Players>Players;  
public:  
  
    /**  
    * @element-type PLAYER  
    */  
    PLAYER *myPLAYER;
```

```
/**
 * @element-type ACTION
 */
ACTION *myACTION;
/**
 * @element-type Board
 */
Board *myBoard;
/**
 * @element-type Die
 */
Die *myDie;
};
#endif // LUDO MODEL_h
View.h:
#ifndef View_h
#define View_h
#include "Intege.h"
class LUDO MODEL;
class View {
public:
    virtual void startGame();
    virtual void EndGame()
action.h:
#ifndef ACTION_h
#define ACTION_h
class ACTION {
public:
    Integer Nothing;
    Integer RollDie;
    Integer MovicePiece;
    Integer EndGame;
};
#endif // ACTION_h
piece.h:
#ifndef Piece_h
#define Piece_h
class Piece {
public:
    virtual void SetColour();
    virtual void SetWon();
    virtual void GetColour();
    virtual void GetWon();
public:
    Integer Colour;
    Integer GetWon;
};
#endif // Piece_h
Board.h:
#ifndef Board_h
#define Board_h
class Piece;
class Square;
class Board {
public:
```

```
virtual void setcolour();
virtual void Setwon();
virtual void GetColour();
virtual void GetWon();
public:
    Integer Square<List>;
    Integer Pieces<List>;
public:
    /**
     * @element-type Piece
     */
    Piece *myPiece;

    /**
     * @element-type Square
     */
    Square *mySquare;
};
#endif // Board_h
Square.h:
#ifndef Square_h
#define Square_h
class Piece;
class Square {
public:
    virtual void Addpiece();
    virtual void Removepiece();
    virtual void SelectColour();
    virtual void SelectGoal();
    virtual void SelectHouse();
    virtual void SetNumber();
public:
    Integer Piece;
public:
    /**
     * @element-type Piece
     */
    Piece *myPiece;
};
#endif // Square_h
Die.h:
#ifndef Die_h
#define Die_h
class Die {
public:
    Integer Dievalue;
    Integer ROLLDie;
};
#endif // Die_h
player.h:
#ifndef PLAYER_h
#define PLAYER_h
class PLAYER {
public:
    virtual void SetPieceStatus();
```

```
virtual void SetpieceNumber();  
virtual void GetpieceNumber();  
virtual void SetColour();  
virtual void SetWon();  
public:  
    Integer Colour;  
};  
#endif // PLAYER_h
```

RESULT:

Complete skeleton codes is generated in java and c++ for the given task by using argo UML.