Society of Computer Training and Research's

Pune Institute of Computer Technology, Pune

DEPARTMENT OF INFORMATION TECHNOLOGY

(Accredited by NBA)

LABORATORY MANUAL

Computer Laboratory VIII

BE (I.T) SEMESTER – I (2020-21)

Course 414459: Computer Laboratory VIII

Teaching Scheme:	**Examination Scheme:**	
Practical Session : 4hrs/week	TW : 50 marks	Oral: 50 marks

Prerequisites:

1. Problem Solving & Object-Oriented Programming
2. Software Engineering and Project Management

Course Objectives:

1. To teach the student Unified Modeling Language (UML 2.0), in terms of "how to use" it for the purpose of specifying and developing software.
2. To teach the student how to identify different software artifacts at analysis and design phase.
3. To explore and analyze use case modeling.
4. To explore and analyze domain/ class modeling.
5. To teach the student Interaction and Behavior Modeling.
6. To Orient students with the software design principles and patterns.

Course Outcomes(COs)

Course Code: <414459>

Course: C15407

Course Name: Computer Laboratory - VIII - 2015 Course

Course Outcomes(CO)	**Statement**
C15407.1	Students will be able to identify Stakeholders and Actors for the chosen Problem of enough complexity and describe the problem by writing use case specification and drawing Use case Diagram and Activity Diagram by using UML 2.0 Notations.
C15407.2	Student will be able model the behavior of system and describe interactions among various entities using Sequence Model-by appropriately identifying various scenarios and State Model-by identifying states and showcase this behavior by implementation using Object oriented Programming Language.
C15407.3	Student should be able to demonstrate the knowledge and ability to Construct and implement the Analysis model and Design model of the system, by Identifying analysis level classes and refining the system using design level classes and class diagram using advanced UML notations.
C15407.4	Student should be able to describe Design Pattern and refine Design Model by applying appropriate GOF and GRASP Design pattern.

CO-PO,PSO Mapping

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
C15407.1	**2**	**3**	**2**	**3**	**1**	**3**	**2**	**1**	**3**	**3**	**3**	**3**	**3**	**3**	**1**
C15407.2	**2**	**3**	**3**	**3**	**2**	**-**	**-**	**-**	**3**	**3**	**-**	**3**	**2**	**3**	**1**
C15407.3	**2**	**3**	**3**	**3**	**2**	**-**	**-**	**-**	**3**	**3**	**-**	**3**	**2**	**3**	**1**
C15407.4	**2**	**3**	**3**	**3**	**2**	**-**	**3**	**-**	**3**	**3**	**2**	**3**	**2**	**3**	**1**
Avg	**2**	**3**	**3**	**3**	**2**	**3**	**2.5**	**1**	**3**	**3**	**-**	**3**	**2**	**3**	**1**

PUNE INSTITUTE OF COMPUTER TECHNOLOGY

DHANKAWADI, PUNE - 411043.

ACADEMIC YEAR: 2020-21

DEPARTMENT: INFORMATION TECHNOLOGY

LAB MANUAL

CLASS : B.E. **SEMESTER: I**

SUBJECT: Computer Laboratory VIII

INDEX OF LAB EXPERIMENTS

Sr. No.	Name of Assignments	Revised On
1	**Write Problem Statement for System / Project** Identify Project of enough complexity, which has at least 4-5 major functionalities. Identify stakeholders, actors and write detail problem statement for your System.	15/06/2018
2	**Prepare Use Case Model** Identify Major Use Cases, Identify actors. Write Use Case specification for all major Use Cases. Draw detail Use Case Diagram using UML2.0 notations.	15/06/2018
3	**Prepare Activity Model** Identify Activity states and Action states. Draw Activity diagram with Swim lanes using UML2.0 Notations for major Use Cases.	15/06/2018
4	**Prepare Analysis Model-Class Model** Identify Analysis Classes and assign responsibilities. Prepare Data Dictionary. Draw Analysis class Model using UML2.0 Notations. Implement Analysis class Model-class diagram with a suitable object oriented language.	15/06/2018

5	**Prepare a Design Model from Analysis Model** Study in detail working of system/Project. Identify Design classes/ Evolve Analysis Model. Use advanced relationships. Draw Design class Model using OCL and UML2.0 Notations. Implement the design model with a suitable object-oriented language.	15/06/2018
6	**Prepare Sequence Model.** Identify at least 5 major scenarios (sequence flow) for your system. Draw Sequence Diagram for every scenario by using advanced notations using UML2.0. Implement these scenarios by taking reference of design model implementation using suitable object-oriented language.	15/06/2018
7	**Prepare a State Model** Identify States and events for your system. Study state transitions and identify Guard conditions. Draw State chart diagram with advanced UML 2 notations. Implement the state model with a suitable object-oriented language	15/06/2018
8	**Identification and Implementation of GRASP pattern** Apply any two GRASP pattern to refine the Design Model for a given problem description Using effective UML 2 diagrams and implement them with a suitable object oriented language	15/06/2018
9	**Identification and Implementation of GOF pattern** Apply any two GOF pattern to refine Design Model for a given problem description Using effective UML 2 diagrams and implement them with a suitable object oriented language.	15/06/2018

Subject Co-ordinator
(Shyam B. Deshmukh)

Head of Department
(Information Technology)

ASSIGNMENT NO. – 01 **Revised on: 15/06/18**

TITLE	**WRITE PROBLEM STATEMENT FOR SYSTEM/PROJECT**
PROBLEM STATEMENT/ DEFINITION	Write Problem Statement for System / Project Identify Project of enough complexity, which has at least 4-5 major functionalities. Identify stakeholders, actors and write detail problem statement for your System.
OBJECTIVE	• Identify Project of enough complexity, which has at least 4-5 major functionalities. • Write Problem Statement for System / Project • Identify stakeholders, actors • Prepare Software Requirement Specification Document.
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o LibreOffice/ Any Document Editor. • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

Software Requirements Specification (Template)

1. Introduction

1.1 Purpose

<Identify the product whose software requirements are specified in this document.>

1.2DocumentConventions

<Describe any standards or typographical conventions that were followed when writing this SRS>

1.3 Intended Audience and Reading Suggestions

<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized.>

1.4 Product Scope

<Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies.>

1.5 References

<List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

2.1 Product Perspective

<Describe the context and origin of the product being specified in this SRS. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

2.2 Project Description /Product Functions

Example for Employee management

Many employees are working in a Company. As per rule these employees cannot work for any other company while in the job. An employee can be a worker, a supervisor or a manager (Other roles are not covered for simplicity). Company maintains record for each employee. Each employee record has a unique employee Id, name(title + first name + middle name + last name) and an address. Address consists of a house number, road, ward no and pin code. Employee record is maintained even for those employees who have left the service (for retirement benefits).

Each employee gets a monthly salary. The worker will get the salary as basic pay and additional daily allowance (based on attendance), the supervisor gets a basic pay and supervision allowance and manager gets a basic pay, a grade pay as 50 percent of basic pay and 30 percent of basic pay as travel allowance. Each employee marks her daily attendance in an attendance register. Employee is paid her salary proportional to her attendance.

On 1st of every month the Accounts department of the company computes the salary of all the employees for previous month. For calculating the salary, Accounts department gets the attendance details (total no of days for which each employee is present in a month) from the attendance register. The Accounts department transfers salary through Bank Money Transfer (BMT) facility of Bank and generates a consolidated statement of salary as employe Id, employee name, employee bank account, and the amount deposited in that account.

<Summarize the major functions the product must perform or must let the user perform.>

2.3 User Classes and Characteristics

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

2.4 Design and Implementation Constraints

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

2.5 User Documentation

<List the user documentation components (such as user manuals, on-line help, and tutorials) that will be delivered along with the software. Identify any known user documentation delivery formats or standards.>

2.6 Assumptions and Dependencies

<List any assumed factors (as opposed to known facts) that could affect the requirements stated in the SRS. The project could be affected if these assumptions are incorrect, are not shared, or change. .>

3. External Interface Requirements

3.1 User Interfaces

<Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.>

3.2 Hardware Interfaces

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

3.3 Software Interfaces

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

3.4 Communications Interfaces

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

4. Other Nonfunctional Requirements

4.1 Performance Requirements

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

4.2 Safety Requirements

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

4.3 Security Requirements

<Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any

user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.>

4.4 Software Quality Attributes

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

4.5 Business Rules

<List any operating principles about the product, such as which individuals or roles can perform which functions under specific circumstances. These are not functional requirements in themselves, but they may imply certain functional requirements to enforce the rules.>

REFERENCES	• Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw Hill, Seventh Edition
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Detail Problem Statement • Software Requirement Specification Document • Conclusion

ASSIGNMENT NO. – 02 **Revised on: 15/06/18**

TITLE	PREPARE USE CASE MODEL
PROBLEM STATEMENT/ DEFINITION	• Prepare Use Case Model • Identify Major Use Cases, Identify actors. • Write Use Case specification for all major Use Cases. • Draw detail Use Case Diagram using UML2.0 notations.
OBJECTIVE	• To Identify Major Use Cases, Identify actors. • To Write Use Case specification. • To Draw detail Use Case Diagram.
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

1 Use-case Diagrams

In UML, use-case diagrams model the behavior of a system and help to capture the requirements of the system.
Use-case diagrams describe the high-level functions and scope of a system. These diagrams also identify the interactions between the system and its actors. The use cases and actors in use-case diagrams describe what the system does and how the actors use it, but not how the system operates internally.
Use-case diagrams illustrate and define the context and requirements of either an entire system or the important parts of the system. You can model a complex system with a single use-case diagram, or create many use-case diagrams to model the components of the system. You would typically develop use-case diagrams in the early phases of a project and refer to them throughout the development process.
Use-case diagrams are helpful in the following situations:

- Before starting a project, you can create use-case diagrams to model a business so that all participants in the project share an understanding of the workers, customers, and activities of the business.
- While gathering requirements, you can create use-case diagrams to capture the system requirements and to present to others what the system should do.
- During the analysis and design phases, you can use the use cases and actors from your use-case diagrams to identify the classes that the system requires.

- During the testing phase, you can use use-case diagrams to identify tests for the system.

2 Creating use-case diagrams

Use-case diagrams describe the main functions of a system and identify the interactions between the system and its external environment, represented by actors. These actors can be people, organizations, machines, or other external systems.

- **Defining the boundaries of a system**

  A system boundary is a rectangle that you can draw in a use-case diagram to separate the use cases that are internal to a system from the actors that are external to the system. A system boundary is an optional visual aid in the diagram; it does not add semantic value to the model.
- **Specifying relationships in diagrams**
  In UML, relationships identify the semantic ties that exist between model elements. To add relationships to a model, you must work either in the diagram editor or in the Project Explorer view.
- **Extending the behavior of use cases**
  In UML modeling, when you draw use-case diagrams; you communicate the essence of the system or application without exposing unnecessary details. It's generally good practice to hide the complexity of the system, but there are times when you might need to expose details. You can depict these details in their own use cases and connect them to base use cases, which enables you to reveal information that would otherwise be hidden.

The use case diagram is composed of following model elements:

- **Use cases**

  A use case describes a function that a system performs to achieve the user's goal. A use case must yield an observable result that is of value to the user of the system.
- **Actors**
  An actor represents a role of a user that interacts with the system that you are modeling. The user can be a human user, an organization, a machine, or another external system.
- **Sub systems**
  In UML models, subsystems are a type of stereotyped component that represent independent, behavioral units in a system. Subsystems are used in class, component, and use-case diagrams to represent large-scale components in the system that you are modeling.
- **Relationships in use-case diagrams**
  In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between the model elements.

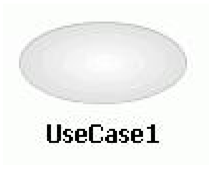
3 Use cases

A use case describes a function that a system performs to achieve the user's goal. A use case must yield an observable result that is of value to the user of the system.

Use cases contain detailed information about the system, the system's users, relationships between the system and the users, and the required behavior of the system. Use cases do not describe the details of how the system is implemented.
Each use case describes a particular goal for the user and how the user interacts with the system to achieve that goal. The use case describes all possible ways that the system can achieve, or fail to achieve, the goal of the user. You can use use cases for the following purposes:

- Determine the requirements of the system
- Describe what the system should do
- Provide a basis for testing to ensure that the system works as intended

In models that depict businesses, use cases represent the processes and activities of the business. In models that depict software systems, use cases represent the capabilities of the software.
Each use case must have a unique name that describes the action that the system performs. Use case names are often short phrases that start with a verb, such as Place Order Online.
As the following figure illustrates, a use case is displayed as an oval that contains the name of the use case.



You can add the following features to use cases:

- Attributes that identify the properties of the objects in a use case
- Operations that describe the behavior of objects in a use case and how they affect the system
- Documentation that details the purpose and flow of events in a use case

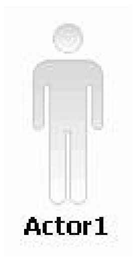
4 Actors

An actor represents a role of a user that interacts with the system that you are modeling. The user can be a human user, an organization, a machine, or another external system.

You can represent multiple users with a single actor and a single user can have the role of multiple actors. Actors are external to the system. They can initiate the behavior described in the use case or be acted upon by the use case. Actors can also exchange data with the system.
In models that depict businesses, actors represent the types of individuals and machines that interact with a business. In models that depict software

applications, actors represent the types of individuals, external systems, or machines that interact with the system.
You would typically use actors in use-case diagrams, but you can also use them in class and sequence diagrams.

As the following figure illustrates, an actor is displayed as a line drawing of a person.



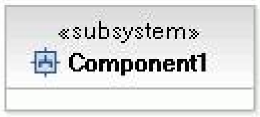
Each actor has a unique name that describes the role of the user who interacts with the system.
You can add documentation that defines what the actor does and how the actor interacts with the system.

5 Subsystems

In UML models, subsystems are a type of stereotyped component that represent independent, behavioral units in a system. Subsystems are used in class, component, and use-case diagrams to represent large-scale components in the system that you are modeling.
You can model an entire system as a hierarchy of subsystems. You can also define the behavior that each subsystem represents by specifying interfaces to the subsystems and the operations that support the interfaces.
In diagrams, compartments display information about the attributes, operations, provided interfaces, required interfaces, realizations, and internal structure of the subsystem.

Typically, a subsystem has a name that describes its contents and role in the system.

As the following figure illustrates, a subsystem is displayed as a rectangle that contains the name of the subsystem. It also contains the keyword «Subsystem» and the subsystem icon.

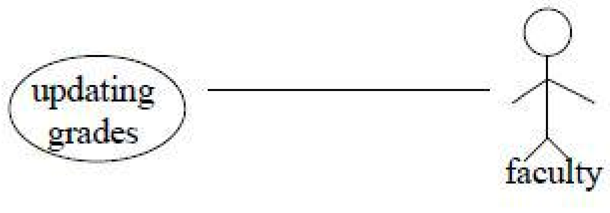


6 Relationships in use-case diagrams

In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between the model elements.

7 Relationships between Use Cases and Actors

Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.



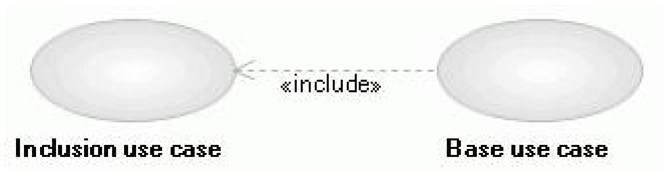
8 Include relationships

In UML modeling, an include relationship is a relationship in which one use case (the base use case) includes the functionality of another use case (the inclusion use case). The include relationship supports the reuse of functionality in a use-case model.

You can add include relationships to your model to show the following situations:

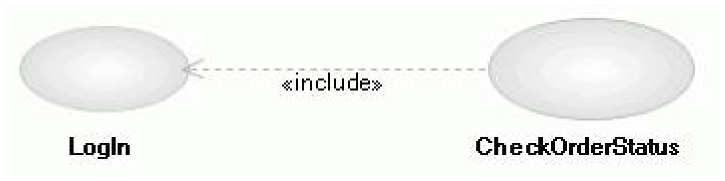
- The behavior of the inclusion use case is common to two or more use cases.
- The result of the behavior that the inclusion use case specifies, not the behavior itself, is important to the base use case.

Include relationships usually do not have names. If you name an include relationship, the name is displayed beside the include connector in the diagram. As the following figure illustrates, an include relationship is displayed in the diagram editor as a dashed line with an open arrow pointing from the base use case to the inclusion use case. The keyword «include» is attached to the connector.



Example of include relationship

The following figure illustrates an e-commerce application that provides customers with the option of checking the status of their orders. This behavior is modeled with a base use case called CheckOrderStatus that has an inclusion use case called LogIn. The LogIn use case is a separate inclusion use case because it contains behaviors that several other use cases in the system use. An include relationship points from the CheckOrderStatus use case to the LogIn use case to indicate that the CheckOrderStatus use case always includes the behaviors in the LogIn use case.



9 Extend relationships

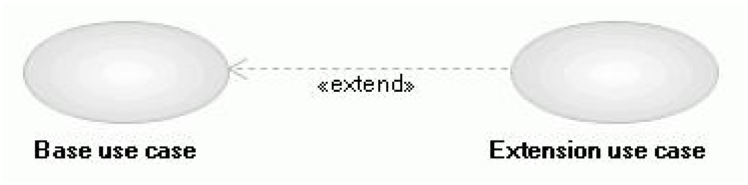
In UML modeling, you can use an extend relationship to specify that one use case (extension) extends the behavior of another use case (base). This type of relationship reveals details about a system or application that are typically hidden in a use case.

The extend relationship specifies that the incorporation of the extension use case is dependent on what happens when the base use case executes. The extension use case owns the extend relationship. You can specify several extend relationships for a single base use case.
While the base use case is defined independently and is meaningful by itself, the extension use case is not meaningful on its own.
The extension use case consists of one or several behavior sequences (segments) that describe additional behavior that can incrementally augment the behavior of the base use case.
Each segment can be inserted into the base use case at a different point, called an extension point.
The extension use case can access and modify the attributes of the base use case; however, the base use case is not aware of the extension use case and, therefore, cannot access or modify the attributes and operations of the extension use case. You can add extend relationships to a model to show the following situations:

- A part of a use case that is optional system behavior
- A subflow is executed only under certain conditions
- A set of behavior segments that may be inserted in a base use case

Extend relationships do not have names.

As the following figure illustrates, an extend relationship is displayed in the diagram editor as a dashed line with an open arrowhead pointing from the extension use case to the base use case. The arrow is labeled with the keyword «extend».



Example

You are developing an e-commerce system in which you have a base use case called Place Online Order that has an extending use case called Specify Shipping Instructions. An extend relationship points from the Specify Shipping Instructions use case to the Place Online Order use case to indicate that the behaviors in the Specify Shipping Instructions use case are optional and only occur in certain circumstances.

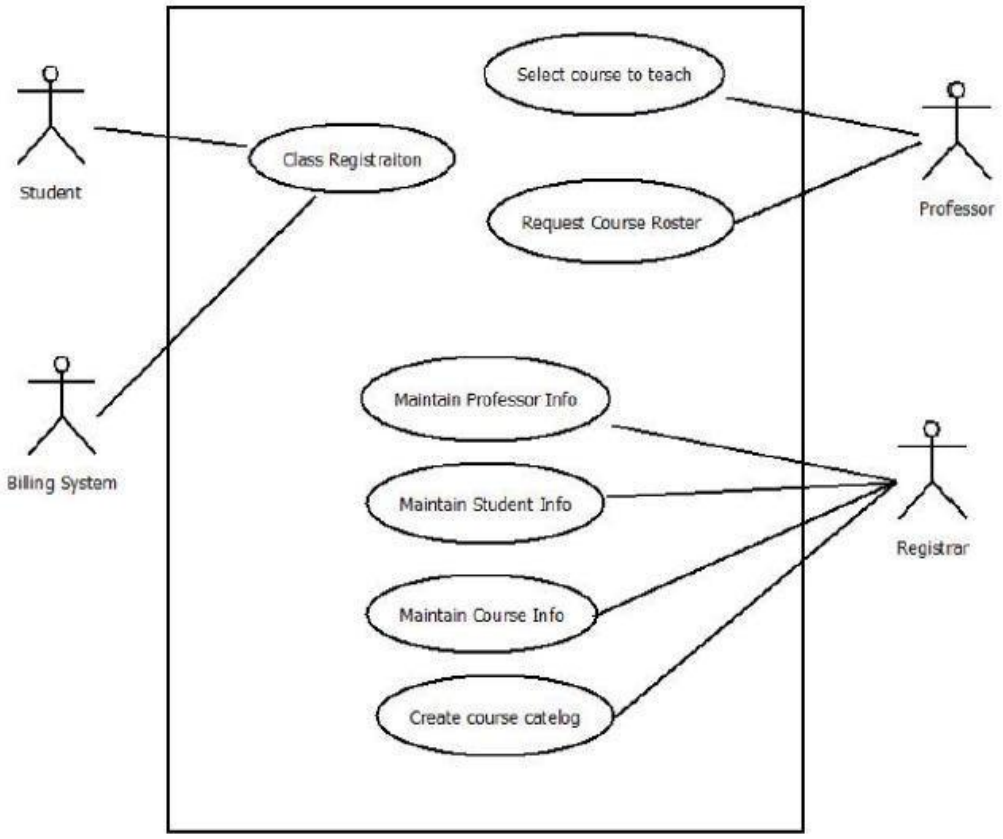
Example OF Use Case Diagram : Altered State University (ASU) Registration System

1. Professors indicate which courses they will teach on-line.

2. A course catalog can be printed
3. Allow students to select on-line four courses for upcoming semester.
4. No course may have more than 10 students or less than 3 students.
5 .When the registration is completed, the system sends information to the billing system.

6. Professors can obtain course rosters on-line.
7. Students can add or drop classes on-line.

Use case Diagram



REFERENCES	• Grady Booch, James Rumbaugh, Ivor Jacobson, "The Unified Modeling Language User Guide", Second Edition, Addison Wesley Object Technology Series • Tom Pender, UML 2 Bible, Wiley India • Ali Bahrami, Object Oriented System Development: Using Unified Modeling Language, McGraw-Hill
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concepts of Elements of Use case Diagram and Model. • Use case Diagram for the System. • Use Case specification for all major Use Cases • Conclusion

EXPERIMENT NO. 3 **Revised on: 15/06/18**

TITLE	**PREPARE ACTIVITY MODEL**
PROBLEM STATEMENT/ DEFINITION	• Prepare Activity Model, Identify Activity states and Action states. Draw Activity diagram with Swim lanes using UML2.0 Notations for major Use Cases.
OBJECTIVE	• To Identify activites involved within the proposed software system • Design the Activity Diagram.
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

Activity Diagrams

In UML, an activity diagram provides a view of the behavior of a system by describing the sequence of actions in a process. Activity diagrams are similar to flowcharts because they show the flow between the actions in an activity; however, activity diagrams can also show parallel or concurrent flows and alternate flows.

In activity diagrams, you use activity nodes and activity edges to model the flow of control and data between actions.
Activity diagrams are helpful in the following phases of a project:

- Before starting a project, you can create activity diagrams to model the most important workflows.
- During the requirements phase, you can create activity diagrams to illustrate the flow of events that the use cases describe.
- During the analysis and design phases, you can use activity diagrams to help define the behavior of operations.

The Activity diagram is composed of following model elements:

- **Activities**
  In UML, activities are container elements that describe the highest level of behavior in an activity diagram. Activities contain several activity nodes and activity edges that represent the sequence of tasks in a workflow that result in a behavior.

- **Actions**
  In UML, an action represents a discrete unit of functionality in an activity.
- **Control nodes**

In activity diagrams, a control node is an abstract activity node that coordinates the flow of control in an activity.

- **Object nodes**
  In activity diagrams, an object node is an abstract activity node that helps to define the object flow in an activity. An object node indicates that an instance of a classifier might be available at a particular point in the activity.

- **Activity edges**
  In activity diagrams, an activity edge is a directed connection between two activity nodes. When a specific action in an activity is complete, the activity edge continues the flow to the next action in the sequence.

10 Activities

In UML, activities are container elements that describe the highest level of behavior in an activity diagram. Activities contain several activity nodes and activity edges that represent the sequence of tasks in a workflow that result in a behavior.

An activity is composed of a series of discrete activity nodes, such as actions, that are connected by control flows or object flows. The actions in an activity are invoked when one of the following events occur:

- Other actions finish executing
- Objects and data become available to the action
- Events that are external to the flow occur

Each activity has a unique name that describes the purpose for the activity.
You can use activities to do the following things:

- Create organizational models of business processes where events can originate from inside or outside the system
- Create information system models that specify the system-level processes that occur

11 Actions

In UML, an action represents a discrete unit of functionality in an activity. Actions have incoming and outgoing activity edges that specify the flow of control and data to and from other activity nodes. The actions in an activity start when all of the input conditions are met. You can add input pins and output pins to specify values that are passed to and from the action when it starts. " Each action has a unique name that describes the behavior.

The Rational Software Architect provides several different types of actions that you can use when you create an activity diagram to describe a particular workflow. The actions are organized into the following groups in the Palette:

- Structured Activity
- Accept Actions
- Invocation Actions

- Link Actions
- Object Actions
- Structural Feature Actions
- Variable Actions

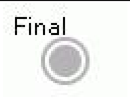
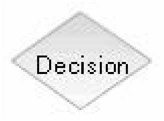
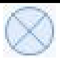
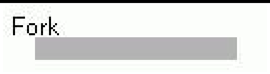
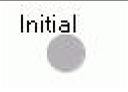
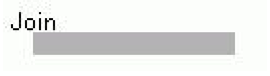
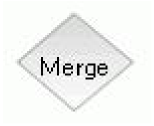
The following table lists a few of the more commonly used actions.

Type of action	**Description**
Opaque action	Opaque actions are a type of action that you can use to represent implementation information. You can also use them as placeholder actions until you determine the specific type of action to use.
Call behavior	Call behaviors are a type of action that you can use to reference behaviors in other activity, state machine, or interaction diagrams in a model. You can also add unspecified call behaviors to activity diagrams, and then specify a type later. Call behaviors reference the behavior, instead of referencing an operation which then invokes a behavior. Input pins and output pins are created for the input and output parameters of the behavior.
Call operation	Call operations are actions that you can use to invoke operations in a model. The referenced operation defines a type of behavior, such as a transformation or a query, that the target object can perform. Each call operation has a unique name that is synchronized with the operation that is referenced. A call operation contains the following pins: • «target» input pin - Represents the target object to which the request is sent; for example, the classifier that owns the operation. • Input pin - There is one for each in parameter. The input value must be compatible with the operation parameters and the class that owns the operation. • Output pin - There is one for each out parameter.
Structured activity	Structured activities are a type of node that you can use to create logical groups of activity nodes and edges. You can add activity nodes and edges to a structured activity; however, these nodes and edges belong only to the structured activity. They are not shared with other structured activities. When a structured activity is invoked, the activity nodes in the structured activity do not start until all of the input data is received. The output data from a structured activity is not available to other nodes in the activity, and flow does not continue through the activity, until all the actions in the structured activity have finished running.
Accept event	Accept events are a type of action that you can use to represent the processing of an event. This type of action waits for the occurrence of

Type of action	Description
	an event that meets specific conditions.
Send signal	Send signals are a type of action that creates an instance of a signal from its inputs, and sends it to the target object, which might invoke a state machine transition or another activity.

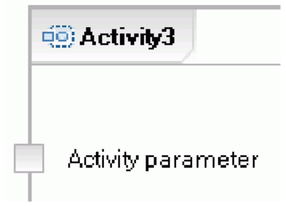
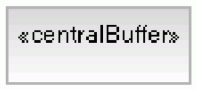
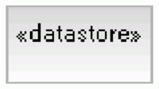
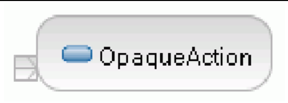
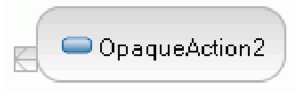
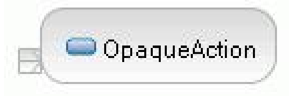
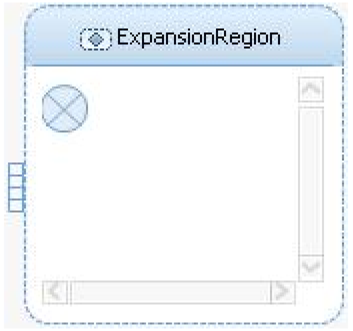
12 Control nodes

In activity diagrams, a control node is an abstract activity node that coordinates the flow of control in an activity.
The following table describes the types of control nodes that you can use in activity diagrams.

Control node type	**Description**	**Icon**
Activity final	This node represents a point where all flows in an activity stop. An activity can have several activity final nodes. When the first activity final node is encountered, all flows stop and the activity terminate.	
Decision	This node represents a point in an activity where a single incoming edge divides into several outgoing edges. You typically use constraints, also called guard conditions, on the outgoing edges to determine which edge should be followed.	
Flow final	This node represents a point where one flow in an activity terminates, but does not affect the other flows in the activity.	
Fork	This node represents a point in an activity where a single incoming flow divides into several outgoing flows.	
Initial	This node marks the point at which flow begins when the activity or structured activity node is invoked. An activity can have several initial nodes, which means that several flows with start when the activity is invoked - one for each initial node.	
Join	This node represents a point in an activity where several incoming flows are synchronized into a single outgoing flow.	
Merge	This node represents a point in an activity where several incoming edges come together into a single outgoing edge.	

Object nodes

In activity diagrams, an object node is an abstract activity node that helps to define the object flow in an activity. An object node indicates that an instance of a classifier might be available at a particular point in the activity.
The following table describes the types of object nodes that you can use to model object flow in activity diagrams.

Object node type	Description	Icon
Activity parameter	This node specifies the input and output parameters for the activity.	
Central buffer	This node consolidates data from several sources and destinations. Central buffer nodes do not connect directly to actions.	
Data store	This node models data flow in which the data is persistent.	
Input pin	This node represents the input parameters on an action. The input pin holds the input values that are received from other actions.	
Output pin	This node represents the output parameters on an action. The output pin holds the output values that an action produces. Object flow edges deliver the output values to other actions.	
Value pin	This node represents the input parameter on an action. The action is enabled when the value pin, which holds a specific value, is evaluated. The result of a "successful" evaluation is the input to the action.	
Expansion node	This node represents a flow across the boundary of an expansion region. An expansion node is represented by a series of squares attached to the side of an expansion region.	

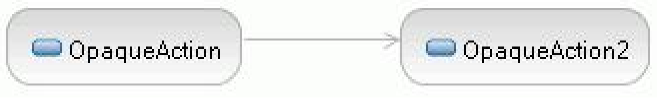
13 Activity edges

In activity diagrams, an activity edge is a directed connection between two activity nodes. When a specific action in an activity is complete, the activity edge continues the flow to the next action in the sequence.
You can use two types of activity edges to model the flow in activities:

- Control flow edges model the movement of control from one node to another.

- Object flow edges model the flow of objects or data from one node to another.

Typically, activity edges do not have names; however, you can add a name to describe the purpose of each edge.
As the following figure illustrates, an activity edge is displayed as a solid line with an open arrowhead that points in the direction of the flow.



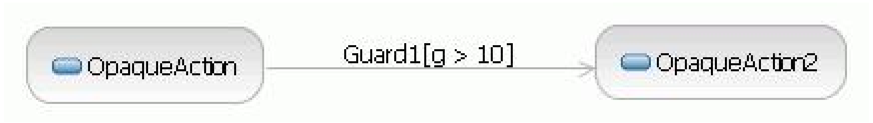
Guard conditions on activity edges

You can add a guard condition to an activity edge between two nodes, where the guard condition defines a condition that must be satisfied before the target activity node can be invoked. You can define the guard condition in the following ways:

- name [guard condition] - The guard condition is created and is assigned a name.

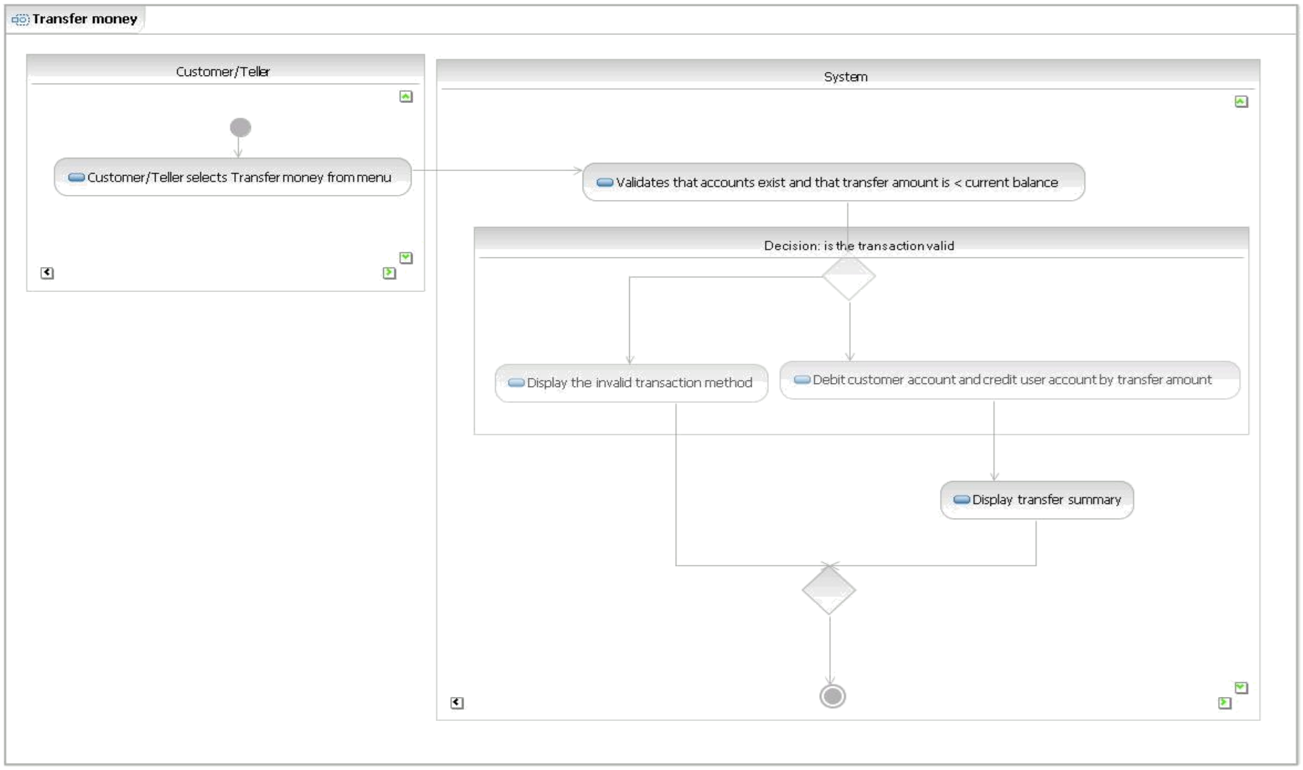
- [guard conditon] - The guard condition is created, but is not assigned a unique name.

In the following figure, one activity node called OpaqueAction is connected to a second activity node, called OpaqueAction2. A guard condition, called Guard1, specifies that the value of g coming from OpaqueAction must be greater than 10 for OpaqueAction2 to be invoked.



14 Example of an activity diagram

This example of an activity diagram is taken from the PiggyBank activity diagram sample that is available in the Samples section of the help and describes the TransferMoney use case from the PiggyBank online banking scenario. The TransferMoney use case describes the steps that occur when a user transfers money from one account to another account.


Transfer money
Customer/Teller
System
Customer/Teller selects Transfer money from menu
Validates that accounts exist and that transfer amount is < current balance
Decision: is the transaction valid
Display the invalid transaction method
Debit customer account and credit user account by transfer amount
Display transfer summary

REFERENCES	• Grady Booch, James Rumbaugh, Ivor Jacobson, "The Unified Modeling Language User Guide", Second Edition, Addison Wesley Object Technology Series • Tom Pender, UML 2 Bible, Wiley India • Ali Bahrami, Object Oriented System Development: Using Unified Modeling Language, McGraw-Hill
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concept of Activity Diagram • Activity Diagrams with Advanced Notations • Conclusion

EXPERIMENT NO. 4 **Revised on: 15/06/18**

TITLE	**DESIGN AND IMPLEMENTATION OF ANALYSIS MODEL- CLASS MODEL**
PROBLEM STATEMENT/ DEFINITION	• Prepare Analysis Model-Class Model • Identify Analysis Classes and assign responsibilities. • Prepare Data Dictionary. • Draw Analysis class Model using UML2.0 Notations. • Implement Analysis class Model-class diagram with a suitable object oriented language.
OBJECTIVE	• To Identify Analysis Classes and assign responsibilities. • To Draw Analysis class Model • To Implement Analysis class Model-class diagram
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool o Eclipse with JAVA ADT • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

Class Diagram

The Class diagram shows the building blocks of any object-orientated system. Class diagrams depict the static view of the model or part of the model, describing what attributes and behaviors it has rather that detailing the methods for achieving operations.

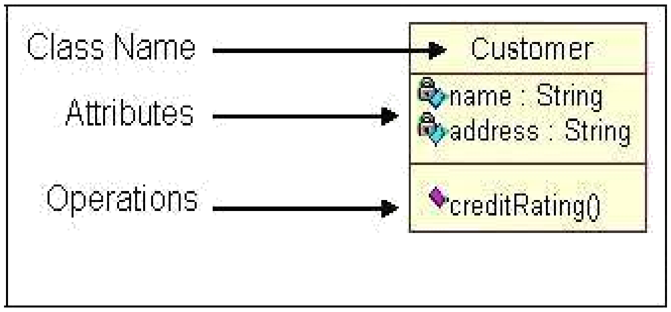
Class diagrams are most useful to illustrate relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections, respectively.

Classes

A class is an element that defines the attributes and behaviors that an object is able to generate. The behavior is the described by the possible messages the class is able to understand along with operations that are appropriate for each message. Classes may also contain definitions of constraints tagged values and stereotypes.

Class Notation

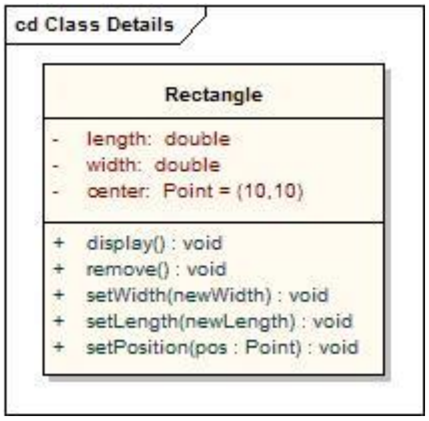
Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations. Additionally constraints, initial values and parameters may be assigned to classes.
Classes are composed of three things: a name, attributes, and operations. Below is an example of a class.

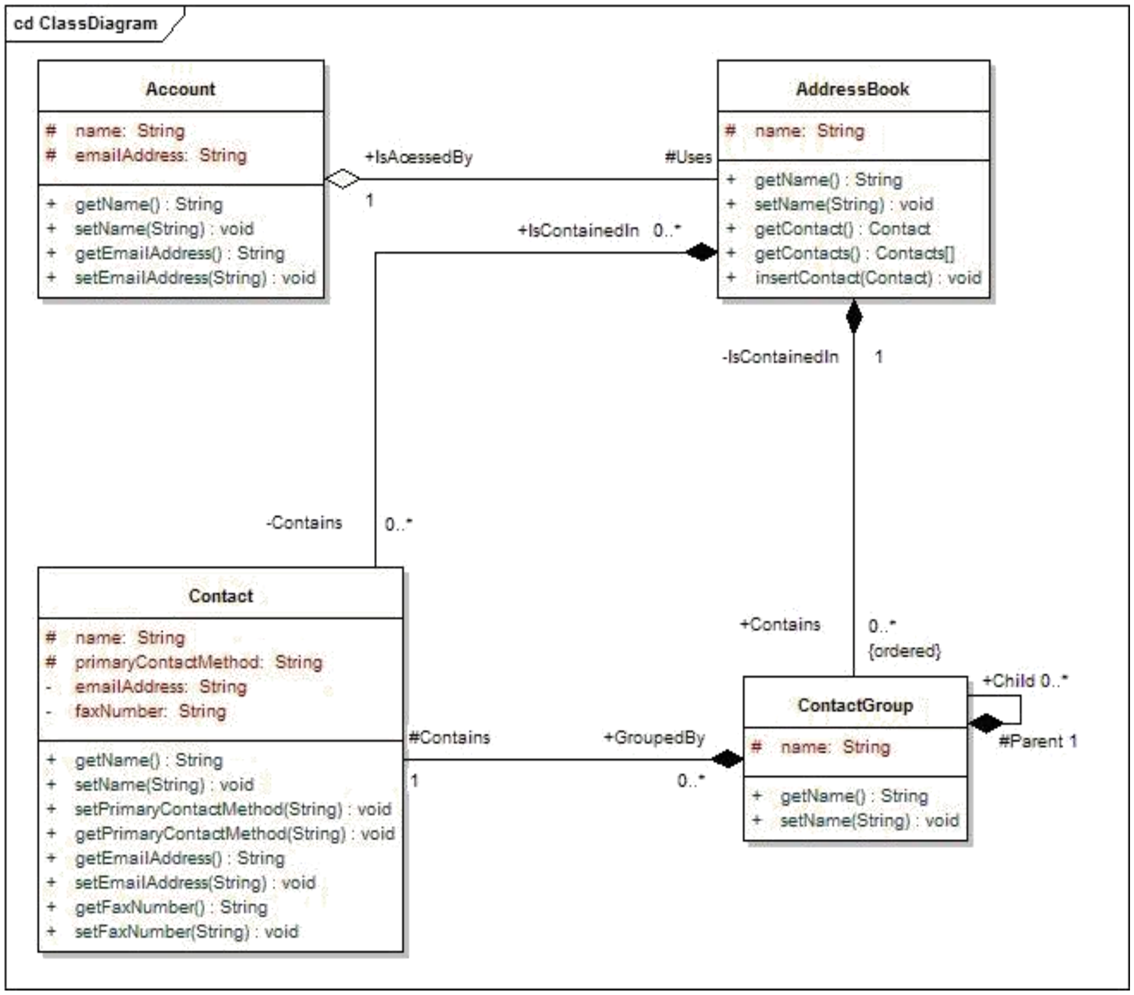


For Example

The diagram below illustrates aggregation relationships between classes. The lighter aggregation indicates that the class Account uses AddressBook, but does not necessarily contain an instance of it.
The strong, composite aggregations by the other connectors indicate ownership or containment of the source classes by the target classes, for example Contact and ContactGroup values will be contained in AddressBook.
In the diagram the class contains the class name in the topmost compartment, the next compartment details the attributes, with the "center" attribute showing initial values.

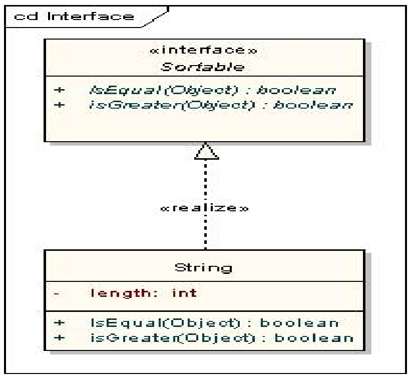
The final compartment shows the operations, the setWidth, setLength and setPosition operations showing their parameters.
The notation that precedes the attribute or operation name indicates the visibility of the element,
if the + symbol is used the attribute or operation has a public level of visibility, if a - symbol is used the attribute or operation is private. In addition the # symbol allows an operation or attribute to be defined as protected and the ~ symbol indicates package or default visibility.


cd Class Details
Rectangle
- length: double
- width: double
- center: Point = (10,10)
+ display() : void
+ remove() : void
+ setWidth(newWidth) : void
+ setLength(newLength) : void
+ setPosition(pos : Point) : void

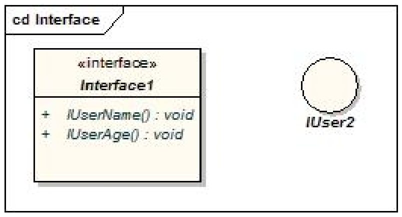


Interfaces

An interface is a specification of behavior that implementers agree to meet. It is a contract. By realizing an interface, classes are guaranteed to support a required behavior, which allows the system to treat non-related elements in the same way – i.e. through the common interface.

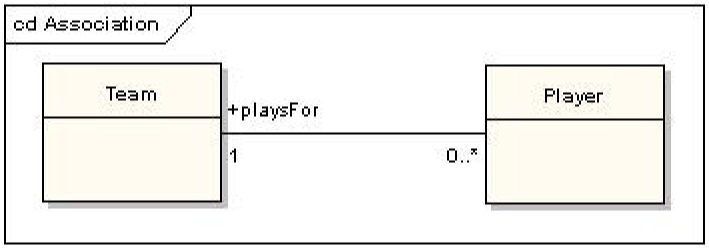

cd Interface
«interface»
Sortable
+ IsEqual(Object) : boolean
+ isGreater(Object) : boolean
«realize»
String
- length: int
+ IsEqual(Object) : boolean
+ isGreater(Object) : boolean

Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.



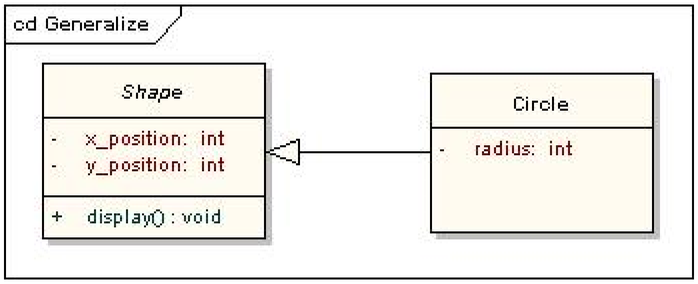
Associations

An association implies two model elements have a relationship - usually implemented as an instance variable in one class.
This connector may include named roles at each end, multiplicity or cardinality, direction and constraints.
Association is the general relationship type between elements. For more than two elements, When code is generated for class diagrams, associations become instance variables in the target class.

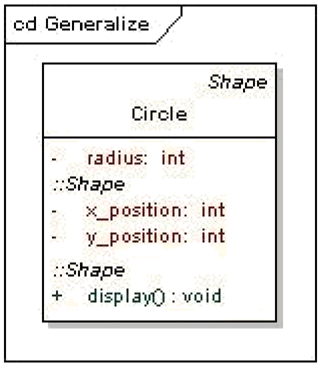


Generalizations

A generalization is used to indicate **inheritance.** Drawn from the specific classifier to a general classifier, the generalize implication is that the source inherits the target's characteristics.
The following diagram shows a parent class generalizing a child class. Implicitly, an instantiated object of the Circle class will have attributes x_position, y_position and radius and a method display().



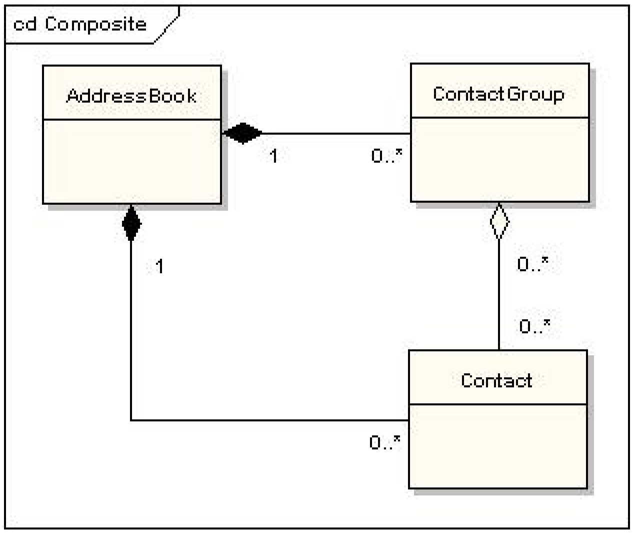
The following diagram shows an equivalent view of the same information.



Aggregations

Aggregations are used to depict elements which are made up of smaller components. Aggregation relationships are shown by a white diamond-shaped arrowhead pointing towards the target or parent class.
A stronger form of aggregation - **a composite aggregation** - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time.
If the parent of a composite aggregation is deleted, usually all of its parts are deleted with it; however a part can be individually removed from a composition without having to delete the entire composition. Compositions are transitive, asymmetric relationships and can be recursive.
The following diagram illustrates the difference between weak and strong aggregations.

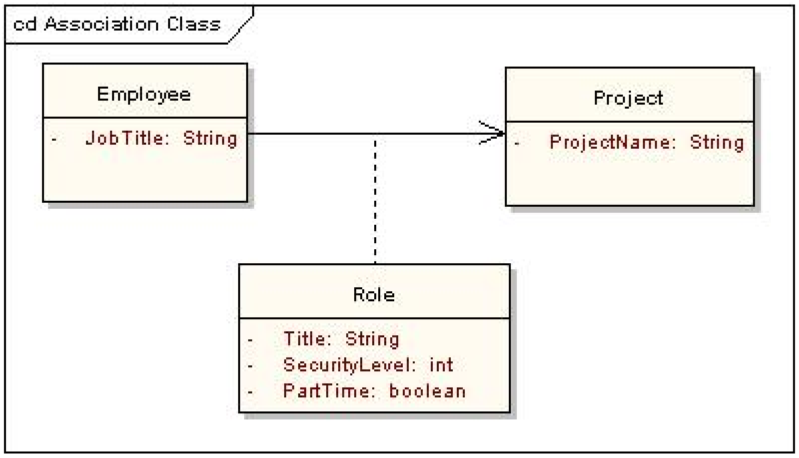
An address book is made up of a multiplicity of contacts and contact groups. A contact group is a virtual grouping of contacts; a contact may be included in more than one contact group.
If you delete an address book, all the contacts and contact groups will be deleted too; if you delete a contact group, no contacts will be deleted.



Association Classes

An association class is a construct that allows an association connection to have operations and attributes. The following example shows that there is more to allocating an employee to a project than making a simple association link between the two classes: the role that the employee takes up on the project is a complex entity in its own right and contains detail that does not belong in the employee or project class.

For example, an employee may be working on several projects at the same time and have different job titles and security levels on each.



Dependencies

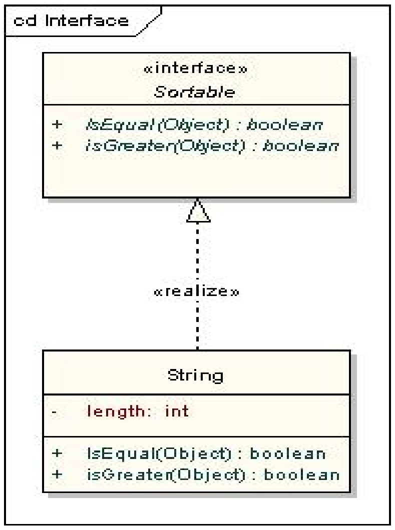
A dependency is used to model a wide range of dependent relationships between model elements. It would normally be used early in the design process where it is known that there is some kind of link between two elements but it is too early to know exactly what the relationship is.
Later in the design process, dependencies will be stereotyped (stereotypes available include «instantiate», «trace», «import» and others) or replaced with a more specific type of connector.

Traces

The trace relationship is a specialization of a dependency, linking model elements or sets of elements that represent the same idea across models. Traces are often used to track requirements and model changes. As changes can occur in both directions, the order of this dependency is usually ignored. The relationship's properties can specify the trace mapping, but the trace is usually bi-directional, informal and rarely computable.

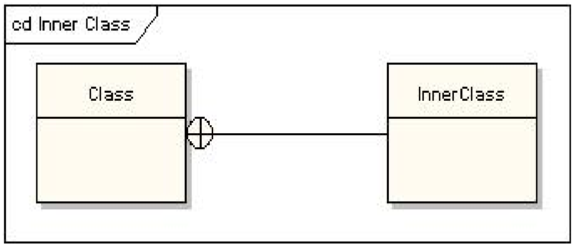
Realizations

The source objects implements or realizes the destination. Realize is used to express traceability and completeness in the model - a business process or requirement is realized by one or more use cases which are in turn realized by some classes, which in turn are realized by a component, etc. Mapping requirements, classes, etc. across the design of your system, up through the levels of modeling abstraction, ensures the big picture of your system remembers and reflects all the little pictures and details that constrain and define it. A realization is shown as a dashed line with a solid arrowhead and the «realize» stereotype.


cd Interface
«interface»
Sortable
+ IsEqual(Object) : boolean
+ isGreater(Object) : boolean
«realize»
String
- length: int
+ IsEqual(Object) : boolean
+ isGreater(Object) : boolean

Nestings

A nesting is connector that shows that the source element is nested within the target element. The following diagram shows the definition of an inner class although in EA it is more usual to show them by their position in the Project View hierarchy.



REFERENCES	• Grady Booch, James Rumbaugh, Ivor Jacobson, “The Unified Modeling Language User Guide”, Second Edition • Tom Pender, UML 2 Bible, Wiley India • Ali Bahrami, Object Oriented System Development: Using Unified Modeling Language, McGraw-Hill • UML2 Toolkit
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concept of Analysis classes • Data Dictionary of System. • Analysis level class Diagram. • Implementation of Analysis Model • Conclusion

EXPERIMENT NO. 5 **Revised on: 15/06/18**

TITLE	**DESIGN AND IMPLEMENTATION DESIGN MODEL FROM ANALYSIS MODEL.**
PROBLEM STATEMENT/ DEFINITION	• Prepare a Design Model from Analysis Model • Study in detail working of system/Project. • Identify Design classes/ Evolve Analysis Model. Use advanced relationships. Draw Design class Model using OCL and UML2.0 Notations. Implement the design model with a suitable object-oriented language.
OBJECTIVE	• To Identify Design level Classes. • To Draw Design level class Model using analysis model. • To Implement Design Model-class diagram
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool o Eclipse with JAVA ADT • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

The analysis model is refined and formalized to get a design model.
Design modeling, means refinement to analysis level models to adapt to the actual implementation environment.
In design space, yet another new dimension has been added to the analysis space to include the implementation environment.
This means that we want to adopt our analysis model to fit in the implementation model at the same time as we refine it.

1 Creating Design level Class Diagrams

Class diagrams model the static structure of a package or of a complete system. As the blueprints of system, class diagrams model the objects that make up the system, allowing to display the relationships among those objects and to describe what the objects can do and the services they can provide.

2 Class diagrams

In UML, class diagrams are one of six types of structural diagram. Class diagrams are fundamental to the object modeling process and model the static structure of a system. Depending on the complexity of a system, you can use a single class diagram to model an entire system, or you can use several class diagrams to model the components of a system.

Class diagrams are the blueprints of your system or subsystem. You can use class diagrams to model the objects that make up the system, to display the relationships between the objects, and to describe what those objects do and the services that they provide.

Class diagrams are useful in many stages of system design. In the analysis stage, a class diagram can help you to understand the requirements of your problem domain and to identify its components. In an object-oriented software project, the class diagrams that you create during the early stages of the project contain classes that often translate into actual software classes and objects when you write code. Later, you can refine your earlier analysis and conceptual models into class diagrams that show the specific parts of your system, user interfaces, logical implementations, and so on. Your class diagrams then become a snapshot that describes exactly how your system works, the relationships between system components at many levels, and how you plan to implement those components.

You can use class diagrams to visualize, specify, and document structural features in your models. For example, during the analysis and design phases of the development cycle, you can create class diagrams to perform the following functions:

- Capture and define the structure of classes and other classifiers
- Define relationships between classes and classifiers
- Illustrate the structure of a model by using attributes, operations, and signals

- Show the common classifier roles and responsibilities that define the behavior of the system
- Show the implementation classes in a package
- Show the structure and behavior of one or more classes
- Show an inheritance hierarchy among classes and classifiers
- Show the workers and entities as business object models

During the implementation phase of a software development cycle, you can use class diagrams to convert your models into code

3 Relationships in class diagrams

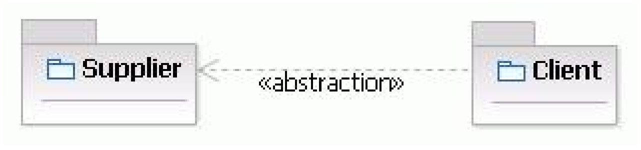
4 Abstraction relationships

An abstraction relationship is a dependency between model elements that represents the same concept at different levels of abstraction or from different viewpoints.

In an abstraction relationship, one model element, the client, is more refined or detailed than the other, the supplier. T

he different types of abstraction relationships include derivation, realization, refinement, and trace relationships.
All abstraction relationships can connect model elements that are in the same model or in different models.
Abstraction relationships do not usually have names and appear as a dashed line with an open arrow pointing from the detailed model element to the general model element.

As the following figure illustrates, when you create an abstraction relationship, the «abstraction» keyword appears beside the connector.



You can assign the following stereotypes to an abstraction relationship to identify the type of abstraction in a model:

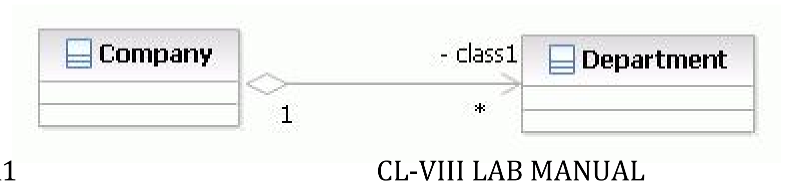
- «derive»
- «realize»
- «refine»
- «trace»

5 Aggregation relationships

In UML models, an aggregation relationship shows a class as a part of or subordinate to another class.
An aggregation is a special type of association in which objects are assembled or configured together to create a more complex object.
Data flows from the whole classifier, or aggregate, to the part. A part classifier can belong to more than one aggregate classifier and it can exist independently of the aggregate.

For example, a Department class can have an aggregation relationship with a Company class, which indicates that the department is part of the company. Aggregations are closely related to compositions.
You can name an association to describe the nature of the relationship between two classifiers; however, names are unnecessary if you use association end names.

As the following figure illustrates, an aggregation association appears as a solid line with an unfilled diamond at the association end, which is connected to the classifier that represents the aggregate. Aggregation relationships do not have to be unidirectional.



6 Association relationships

In UML models, an association is a relationship between two classifiers, such as classes, that describes the reasons for the relationship and the rules that govern the relationship.

An association represents a structural relationship that connects two classifiers.
Like attributes, associations record the properties of classifiers.
For example, in relationships between classes, you can use associations to show the design decisions that you made about classes in your application that contain data, and to show which of those classes need to share data. You can use an association's navigability feature to show how an object of one class gains access to an object of another class or, in a reflexive association, to an object of the same class.

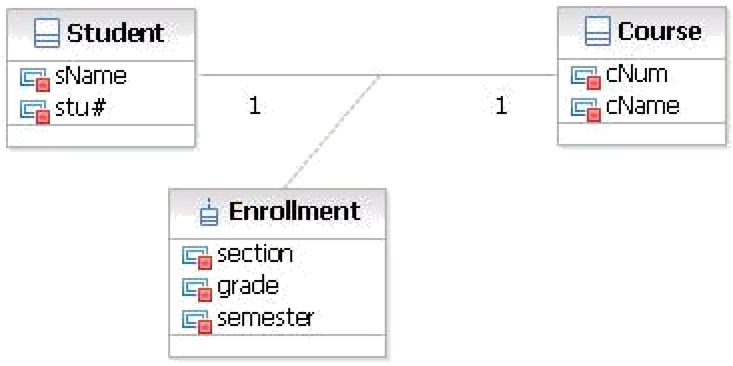
The name of an association describes the nature of the relationship between two classifiers and should be a verb or phrase.
In the diagram editor, an association appears as a solid line between two classifiers.

Association ends
An association end specifies the role that the object at one end of a relationship performs. Each end of a relationship has properties that specify the role of the association end, its multiplicity, visibility, navigability, and constraints.
Example
In an e-commerce application, a customer class has a single association with an account class. The association shows that a customer instance owns one or more instances of the account class. If you have an account, you can locate the customer that owns the account.

7 Association classes

In UML diagrams, an association class is a class that is part of an association relationship between two other classes.
You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations.

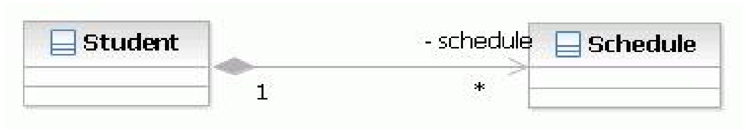
For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course. An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.
As the following figure illustrates, an association class is connected to an association by a dotted line.



8 Composition Association Relationships

A composition association relationship represents a whole–part relationship and is a form of aggregation. A composition association relationship specifies that the lifetime of the part classifier is dependent on the lifetime of the whole classifier.
In a composition association relationship, data usually flows in only one direction (that is, from the whole classifier to the part classifier). For example, a composition association relationship connects a Student class with a Schedule class, which means that if you remove the student, the schedule is also removed.

As the following figure illustrates, a composition association relationship appears as a solid line with a filled diamond at the association end, which is connected to the whole, or composite, classifier.

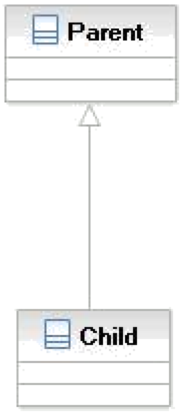


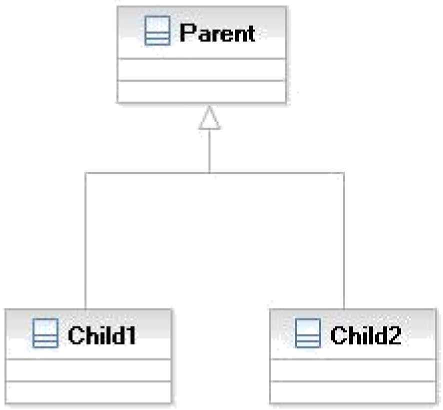
9 Generalization Relationships

In UML modeling, a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent).
You can add generalization relationships to capture attributes, operations, and relationships in a parent model element and then reuse them in one or more child model elements. Because the child model elements in generalizations inherit the attributes, operations, and relationships of the parent, you must only define for the child the attributes, operations, or relationships that are distinct from the parent.

The parent model element can have one or more children, and any child model element can have one or more parents. It is more common to have a single parent model element and multiple child model elements. Generalization relationships do not have names.

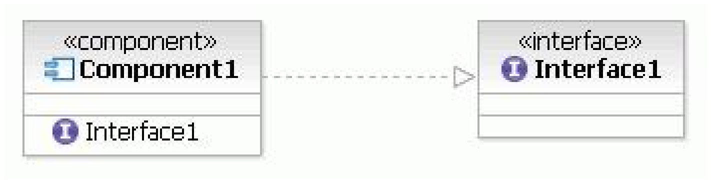
As the following figures illustrate, a generalization relationship as a solid line with a hollow arrowhead that points from the child model element to the parent model element.





10 Interface Realization Relationships

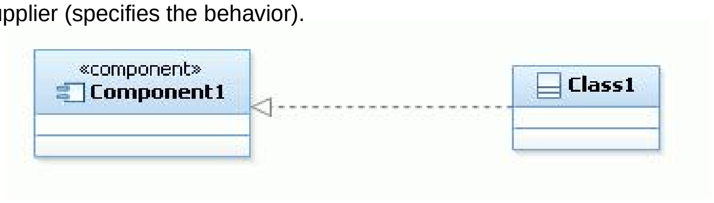
In UML diagrams, an interface realization relationship is a specialized type of implementation relationship between a classifier and a provided interface. The interface realization relationship specifies that the realizing classifier must conform to the contract that the provided interface specifies.
Typically, interface realization relationships do not have names. If you name an interface realization, the name is displayed beside the connector in the diagram. As the following figure illustrates, an interface realization relationship is displayed in the diagram editor as a dashed line with a hollow arrowhead. The interface realization points from the classifier to the provided interface.



11 Realization Relationships

In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes the behavior that the other model element (the supplier) specifies. Several clients can realize the behavior of a single supplier.

As the following figure illustrates, a realization as a dashed line with an unfilled arrowhead that points from the client (realizes the behavior) to the supplier (specifies the behavior).



12 Qualifiers on Association Ends

In UML, qualifiers are properties of binary associations and are an optional part of association ends.

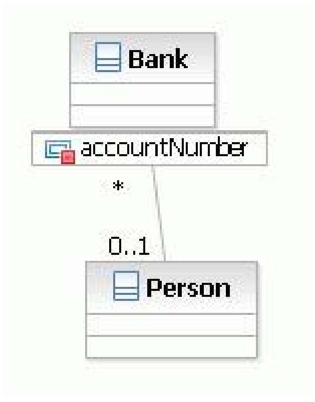
A qualifier holds a list of association attributes, each with a name and a type. Association attributes model the keys that are used to index a subset of relationship instances.

A qualifier is visually represented as a rectangle attached to the qualified end of the association relationship. The list of association attributes is displayed in the qualifier box.

Qualifiers identify subsets of related instances in association navigations; they provide a model of indices or keys for association ends. It is rare to see qualifiers on both ends of an association, because only the unqualified element typically varies in multiplicity.

In a navigation context, qualifiers are used to select a specific object pair from the set of all related objects in that association. In an implementation context, each qualifier value points to a unique target object. Generally, if an application requires the retrieval of data based on search keys, the model should use qualified associations.

Example
In a banking application, a class called Bank represents a banking institution and has an association with a class called Person, which represents an individual. Each individual is associated with the bank through several bank accounts. The account number qualifies the association, and it enables the indexing of many associations between the Person and Bank classes.
As the following figure illustrates, the qualifier is attached to the association end that corresponds to the Bank class.



Visibility in Class Diagram
Use visibility indicates who can access the information contained within a class.
There are four types of visibilities :
1. **Private visibility** hides information from anything outside the class partition. UML notation for private is +
2. **Public visibility** allows all other classes to view the marked information. UML notation for public is -.
3. **Protected visibility** allows child classes to access information they inherited from a parent class. UML notation for protected is #.

4. **Package or default visibility** allows classes to access information within same package. UML notation for package visibility is ~ symbol

13 Attributes in Class

In UML models, attributes represent the information, data, or properties that belong to instances of a classifier.
A classifier can have any number of attributes or none at all. Attributes describe a value or a range of values that instances of the classifier can hold. You can specify an attribute's type, such as an integer or Boolean, and its initial value. You can also attach a constraint to an attribute to define the range of values it holds.

Attribute names are short nouns or noun phrases that describe the attribute. The UML syntax for an attribute name incorporates information in addition to its name, such as the attribute's visibility, type, and initial value as shown in the following example.

```
visibility «stereotype» name : type-expression = initial-value
```

Example
In an e-commerce application, a Customer class has an attribute that holds the amount of money in the customer's balance as shown in the following example.

```
- balance : MoneyType = 0.00
```

- - is the attribute's visibility, in this case, private
- balance is the attribute's name, which describes a particular property of the Customer class
- MoneyType is the attribute's type, in this case an instance of the MoneyType class
- 0.00 is the attribute's initial value, zero

14 Operations in Class

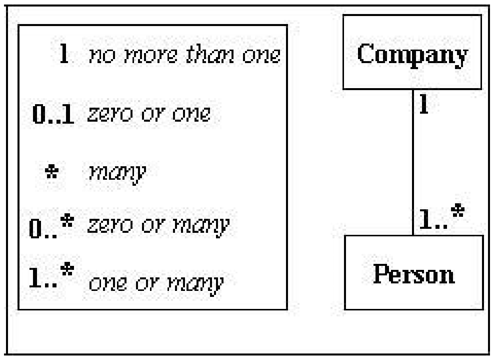
In UML models, operations represent the services or actions that instances of a classifier might be requested to perform.
A classifier can have any number of operations or none at all. Operations define the behavior of an instance of a classifier
You can add operations to identify the behavior of many types of classifier in your model. In classes, operations are implementations of functions that an object might be required to perform. Well-defined operations perform a single task.
Operations can have exceptions, elements that are created when the operation encounters an error.
Every operation in a classifier must have a unique signature. A signature comprises the operation's name and its ordered list of parameter types. The UML syntax for an operation name is as follows:

```
Visibility «stereotype» name(parameter list) : return-type
```

For example, in an e-commerce application a Customer class has the following operation:
- getBalance([in] day: Date) : MoneyType.

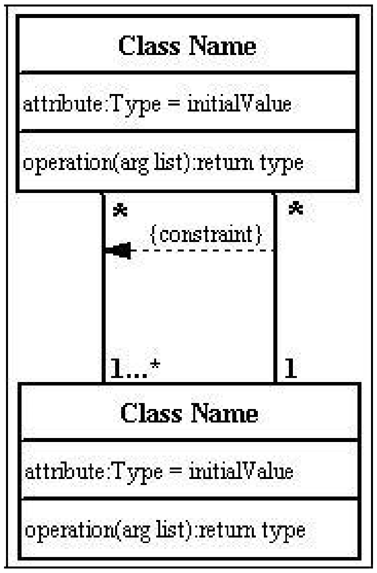
❖ **Multiplicity (Cardinality)**

The multiplicity of the association denotes how many the number of objects from one class can associated with how many no of objects from another class.
. For example, one company will have one or more employees, but each employee works for one company only.



❖ **Constraint:**

Constraints are Boolean expressions that must get evaluated true. Constraints can be applied on objects, Links and Generalization. Place constraints inside curly braces {}.Example for the same is given below.


Class Name
attribute:Type = initialValue
operation(arg list):return type
*
{constraint}
*
1...*
1
Class Name
attribute:Type = initialValue
operation(arg list):return type

REFERENCES	• Grady Booch, James Rumbaugh, Ivor Jacobson, “The Unified Modeling Language User Guide”, Second Edition, • Tom Pender, UML 2 Bible, Wiley India • Ali Bahrami, Object Oriented System Development: Using Unified Modeling Language, McGraw-Hill • UML2 Toolkit
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concept of Design model. • Design level class diagram. • Implementation of design model. • Conclusion

EXPERIMENT NO. 6 **Revised on: 15/06/18**

TITLE	**PREPARE AND IMPLEMENT SEQUENCE MODEL.**
PROBLEM STATEMENT/ DEFINITION	• Prepare Sequence Model. • Identify at least 5 major scenarios (sequence flow) for your system. Draw Sequence Diagram for every scenario by using advanced notations using UML2.0 • Implement these scenarios by taking reference of design model implementation using suitable object-oriented language.
OBJECTIVE	• To study and use of communication. • Draw sequence diagram • To implement sequence diagram.
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool o Eclipse with JAVA ADT • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM..

RELEVANT THEORY

1 Sequence Diagrams

A sequence diagram is a Unified Modeling Language (UML) diagram that illustrates the sequence of messages between objects in an interaction.
A sequence diagram consists of a group of objects that are represented by lifelines, and the messages that they exchange over time during the interaction. A sequence diagram shows the sequence of messages passed between objects. Sequence diagrams can also show the control structures between objects.
For example, lifelines in a sequence diagram for a banking scenario can represent a customer, bank teller, or bank manager. The communication between the customer, teller, and manager are represented by messages passed between them. The sequence diagram shows the objects and the messages between the objects.

The sequence diagram consists of following elements:

- **Interaction frames**
  In sequence diagrams and communication diagrams, an interaction frame provides a context or boundary to the diagram in which you create diagram elements, such as lifelines or messages, and in which you observe behavior.

- **Lifelines in UML diagrams**
  In UML diagrams, such as sequence or communication diagrams, lifelines represent the objects that participate in an interaction. For example, in a banking scenario, lifelines can represent objects such as a bank system or customer. Each instance in an interaction is represented by a lifeline.
- **Messages in UML diagrams**
  A message is an element in a Unified Modeling Language (UML) diagram that defines a specific kind of communication between instances in an interaction. A message conveys information from one instance, which is represented by a lifeline, to another instance in an interaction.
- **Combined fragments in sequence diagrams**
  In sequence diagrams, combined fragments are logical groupings, represented by a rectangle, which contain the conditional structures that affect the flow of messages. A combined fragment contains interaction operands and is defined by the interaction operator.
- **Interaction uses in sequence diagrams**
  In sequence diagrams, interaction uses enable you to reference other existing interactions. You can construct a complete and complex sequence from smaller simpler interactions.

2 Interaction frames

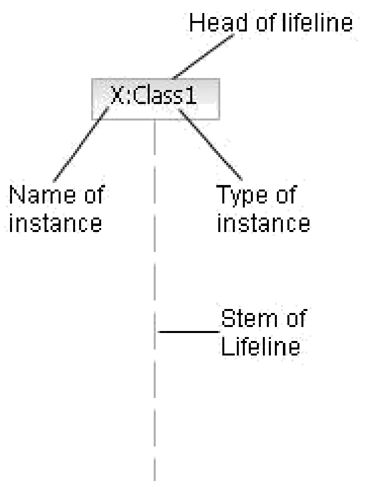
In sequence diagrams and communication diagrams, an interaction frame provides a context or boundary to the diagram in which you create diagram elements, such as lifelines or messages, and in which you observe behavior.

The frame and its contents represent an interaction in a sequence diagram or communication diagram. The heading label of the interaction frame is the name of the interaction that the diagram represents. In sequence diagrams, a frame can represent combined fragments, which represent scenario constructs, and interaction uses, which represent an interaction within an interaction.

3 Lifelines in UML diagrams

In UML diagrams, such as sequence or communication diagrams, lifelines represent the objects that participate in an interaction. For example, in a banking scenario, lifelines can represent objects such as a bank system or customer. Each instance in an interaction is represented by a lifeline.

As the following figure illustrates, a lifeline in a sequence diagram is displayed with its name and type in a rectangle, which is called the head. The head is located on top of a vertical dashed line, called the stem, which represents the timeline for the instance of the object.



Messages, which are sent and received by the instance, appear on the lifeline in sequential order. You can create new lifelines, create lifelines from existing elements, or assign element types to existing lifelines.
As the following figure illustrates, a lifeline in a communication diagram is represented by a rectangle that contains the instance name and the type.

4 Messages in UML diagrams

A message is an element in a Unified Modeling Language (UML) diagram that defines a specific kind of communication between instances in an interaction. A message conveys information from one instance, which is represented by a lifeline, to another instance in an interaction.

5 Types of messages

A message specifies a sender and receiver, and defines the kind of communication that occurs between lifelines. For example, a communication can invoke, or call, an operation by using a synchronous call message or asynchronous call message, can raise a signal using an asynchronous signal, and can create or destroy a participant.

You can use the five types of messages that are listed in the following table to show the communication between lifelines in an interaction.

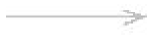
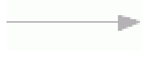
Message type	Description	Example
Create	A create message represents the creation of an instance in an interaction. The create message is represented by the keyword «create». The target lifeline begins at the point of the create message.	In a banking scenario, a bank manager might start a credit check on a client by sending a create message to the server.

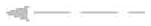
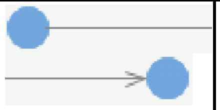
Messag e type	Description	Example
Destroy	A destroy message represents the destruction of an instance in an interaction. The destroy message is represented by the keyword «destroy». The target lifeline ends at the point of the destroy message, and is denoted by an X.	A bank manager, after starting a credit check, might close or destroy the credit program application for a customer.
Synchro nous call	Synchronous calls, which are associated with an operation, have a send and a receive message. A message is sent from the source lifeline to the target lifeline. The source lifeline is blocked from other operations until it receives a response from the target lifeline.	A bank teller might send a credit request to the bank manager for approval and must wait for a response before further serving the customer.
Asynchr onous call	Asynchronous calls, which are associated with operations, typically have only a send message, but can also have a reply message. In contrast to a synchronous message, the source lifeline is not blocked from receiving or sending other messages. You can also move the send and receive points individually to delay the time between the send and receive events. You might choose to do this if a response is not time sensitive or order sensitive.	A bank customer could apply for credit but can receive banking information over the phone or request money from an ATM, while waiting to hear about the credit application.
Asynchr onous signal	Asynchronous signal messages are associated with signals. A signal differs from a message because no operation is associated with the signal. A signal can represent an interrupt condition or error condition. To specify a signal, you create an asynchronous call message and change the type in the message	A credit agency could send an error signal message to the bank manager that states a failure to connect to the credit bureau.

Messag e type	Description	Example
	properties view.	
Lost and found	A lost message is a message that has a known sender but the receiver is not known. A found message is a message that does not have a known sender but has a receiver.	An outside actor sends a message to a bank manager. The actor is outside the scope of the sequence diagram and is therefore a found message. A lost message can occur when a message is sent to an element outside the scope of the UML diagram.

An asynchronous message is the only message type for which you can individually move the sending and receiving points. You can move the points of an asynchronous message to manipulate the time delay between the sending event and the receiving event; the result is called a skewed message. You can create an asynchronous message with or without a behavior execution specification.

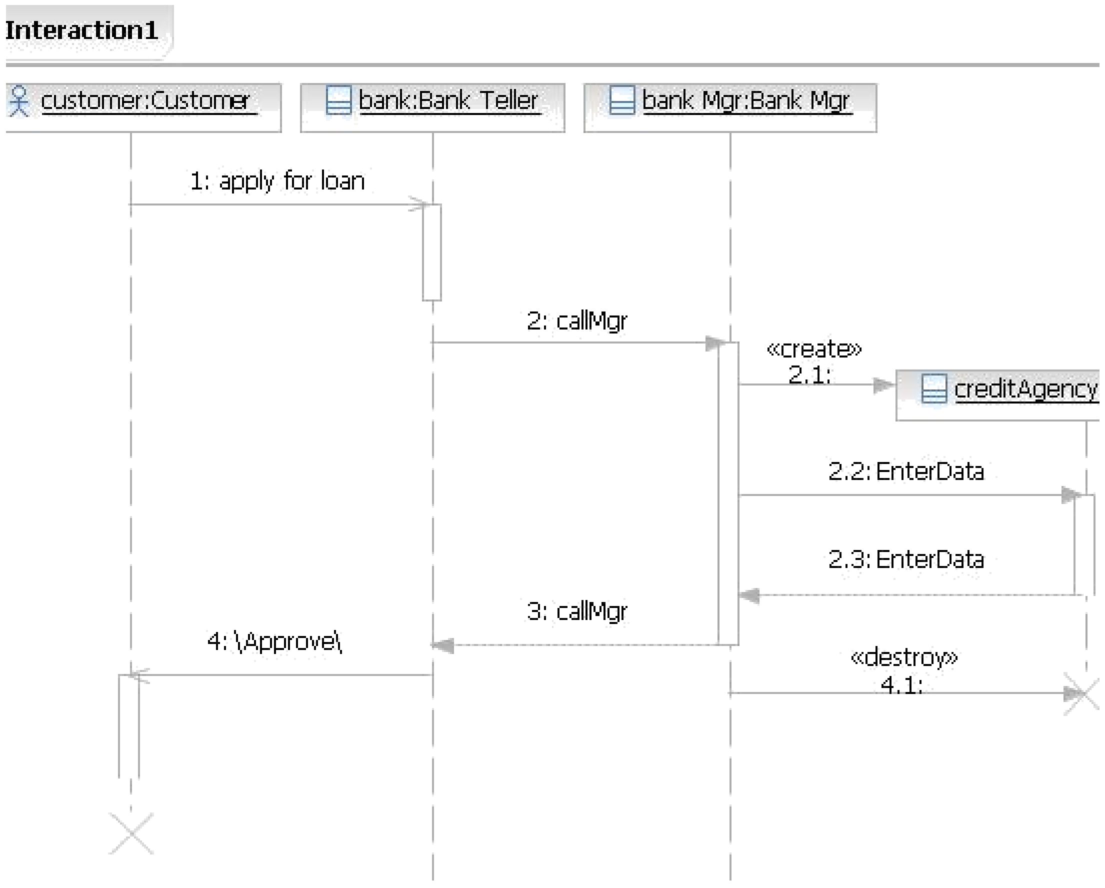
A self-directed message is a message that is sent from the source lifeline to itself. A self-directed message could be a recursive call or a call to another operation or signal that belongs to the same object.
The message that the source lifeline sends to the target lifeline represents an operation or a signal that the target lifeline implements. You can name and order messages. The appearance of the line or arrowhead reflects the properties of the message. The following table shows the graphics that represent messages in UML diagrams.

Message type	Graphic	Description	Representation
Asynchronous		A line with an open arrowhead	This graphic represents an asynchronous signal or an asynchronous call in which the source object sends the message and immediately continues with the next step.
Synchronous		A line with a solid arrowhead that points toward the receiving lifeline	This graphic represents a synchronous call operation in which the source object sends a message and waits for a return message from the target before the source can continue.

Synchronous return		A dashed line with a solid arrowhead that points toward the originating lifeline	This graphic represents a return message from a call to a procedure. When you create a synchronous message, a return message is created by default. You can change this default in the Preferences window.
Lost and found		A line with an open arrowhead and that contains a dot at either end.	This graphic represents a lost or found message. A lost message contains a dot at the end of the arrowhead to indicate the destination is unknown. A dot at the source of the message indicates a found message with an unknown sender.

As the following figure illustrates, messages are displayed as a line with an arrow that points in the direction in which the message is sent; that is, from the sending message end to the receiving message end. The following example shows how messages are displayed in a sequence diagram that represents a banking scenario in which a bank customer applies for a loan by following this process.

- A customer gives an application for the loan to a bank teller.
- The bank teller sends the application to the bank manager to processed and waits for the manager to finish.
- The bank manager starts the credit check program, enters the data, and waits for the credit agency to send the results.
- The bank manager receives a response from the credit agency and sends a message to the bank teller that states the decision.
- The bank teller sends a message to the customer that states whether the loan was approved.
- The bank manager closes the credit agency program and the customer completes the transaction.



REFERENCES	• Grady Booch, James Rumbaugh, Ivor Jacobson, "The UML User Guide", Second Edition, Addison Wesley • Tom Pender, UML 2 Bible, Wiley India • Ali Bahrami, Object Oriented System Development: Using Unified Modeling Language, McGraw-Hill
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concept of communication Diagrams • Sequence Diagram. • Implementation of Sequence Diagram. • Conclusion

EXPERIMENT NO. 7 **Revised on: 15/06/18**

TITLE	**PREPARE AND IMPLEMENT STATE MODEL**
PROBLEM STATEMENT/ DEFINITION	• Prepare a State Model. • Identify States and events for your system. • Study state transitions and identify Guard conditions. • Draw State chart diagram with advanced UML 2 notations. • Implement the state model with a suitable OO language
OBJECTIVE	• To Identify States Transitions, events in the system flow. • Draw State Diagram and Implement Model.
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool o Eclipse with JAVA ADT • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

State Machine Diagram:

A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.
A State chart diagram describes a state machine. Now to clarify it state machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.
As State chart diagram defines states it is used to model lifetime of an object.

Purpose:

State chart diagram is one of the five UML diagrams used to model dynamic nature of a system. They define different states of an object during its lifetime. And these states are changed by events.

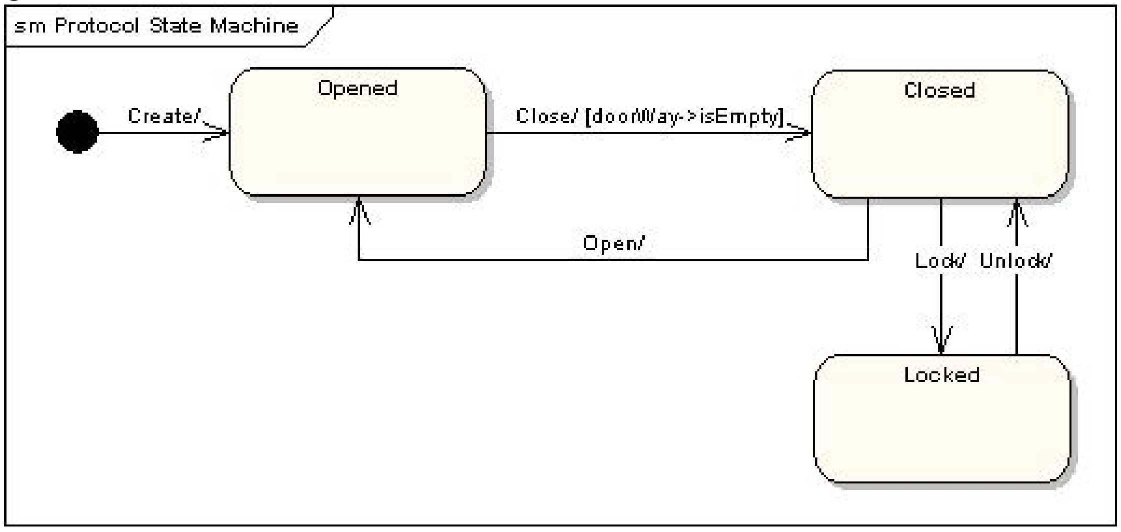
So State chart diagrams are useful to model reactive systems. Reactive systems can be defined as a system that responds to external or internal events.
State chart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. So the most important purpose of State chart diagram is to model life time of an object from creation to termination.
State chart diagrams are also used for forward and reverse engineering of a system.

But the main purpose is to model reactive system.
Following are the main purposes of using State chart diagrams:

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.

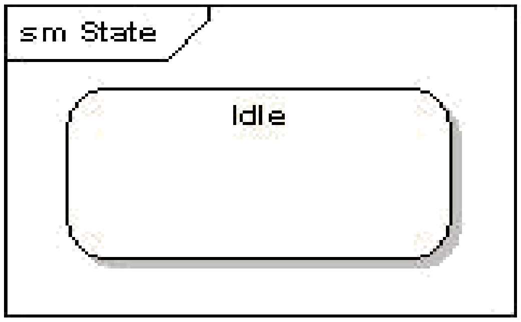
As an example, the following state machine diagram shows the states that a door goes through during its lifetime. The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock. Notice that not all events are valid in all states;
For example, if a door is opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled.

The syntax and conventions used in state machine diagrams will be discussed in full in the following sections.



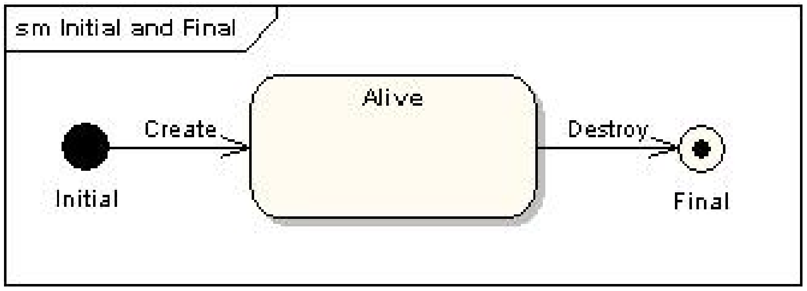
States

A state is denoted by a round-cornered rectangle with the name of the state written inside it.



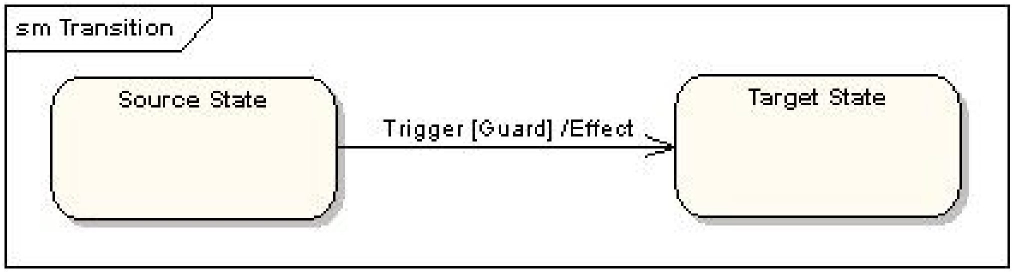
Initial and Final States

The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name.



Transitions

Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



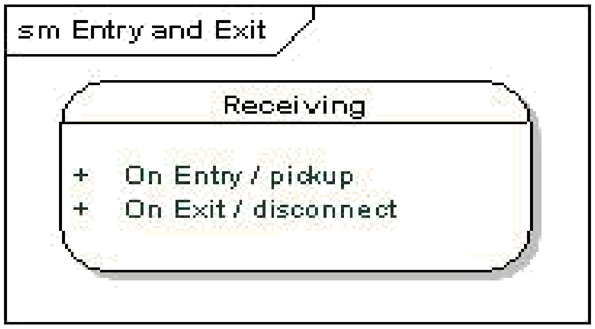
"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time.

"Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Actions

In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions.

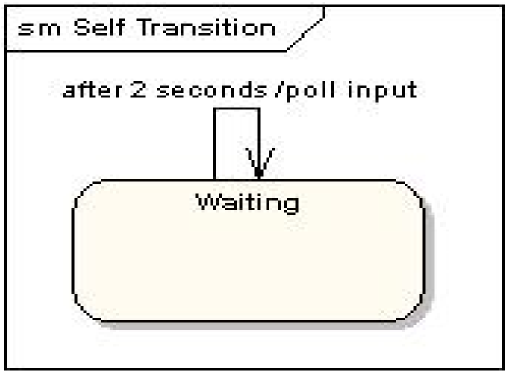
This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.



It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.

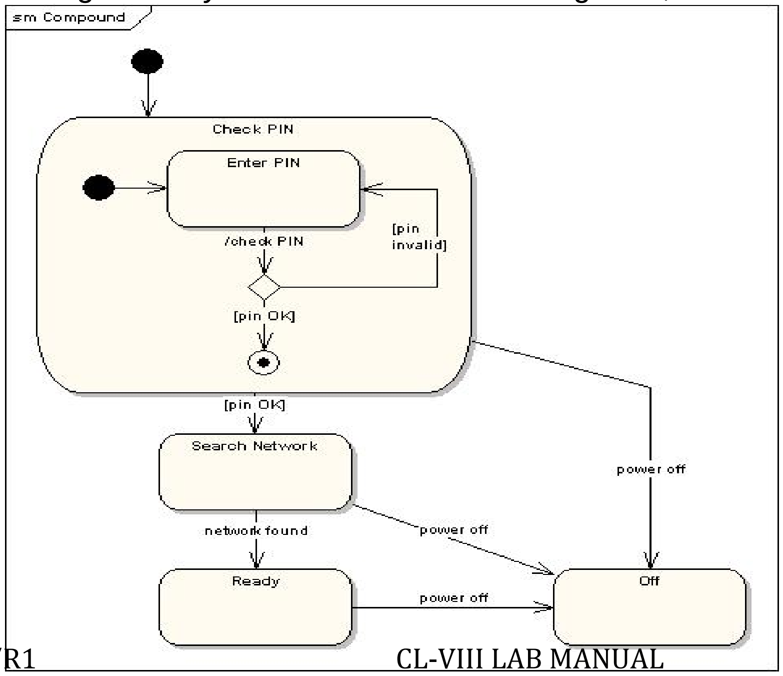
Self-Transitions

A state can have a transition that returns to itself, as in the following diagram.
This is most useful when an effect is associated with the transition.

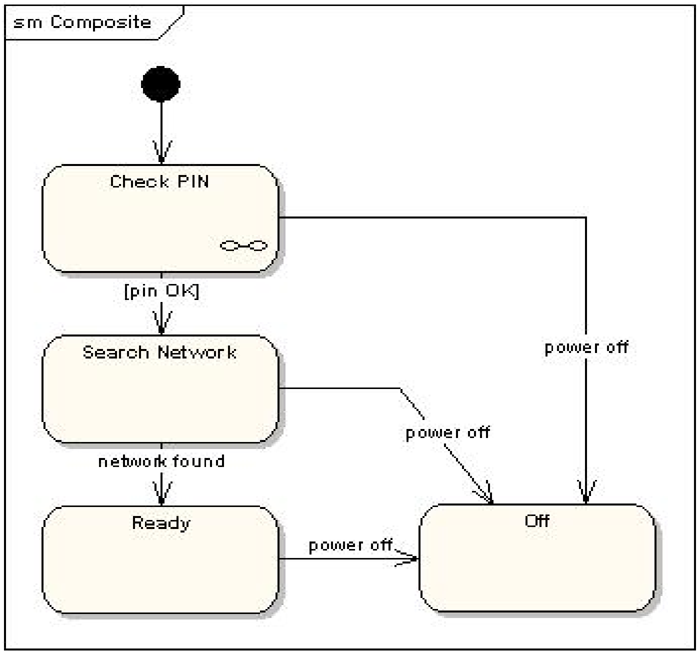


Compound States

A state machine diagram may include sub-machine diagrams, as in the example below.



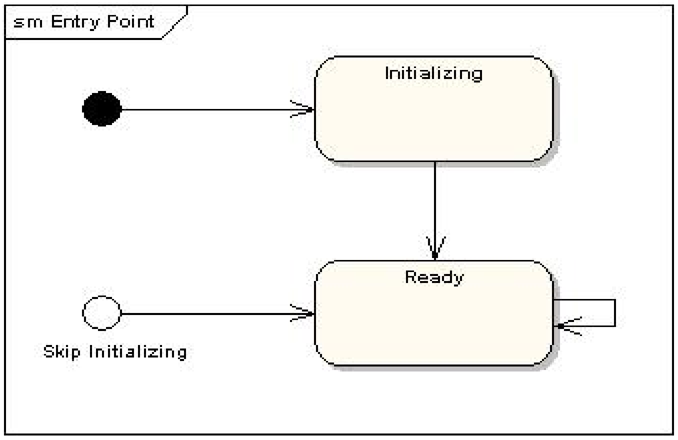
The alternative way to show the same information is as follows.



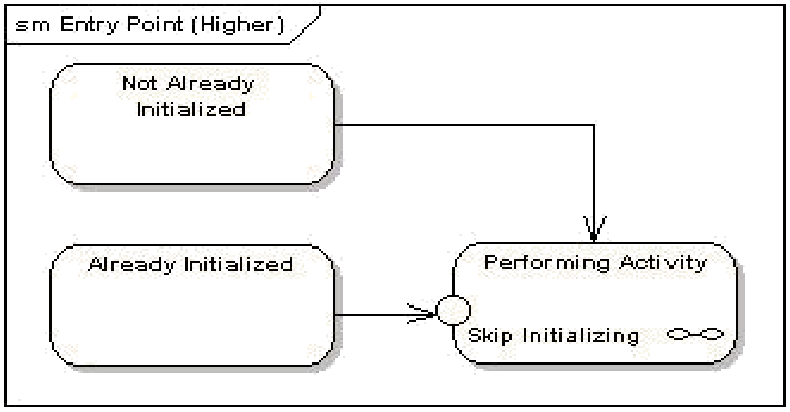
The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

Entry Point

Sometimes you won't want to enter a sub-machine at the normal initial state. For example, in the following sub-machine it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.

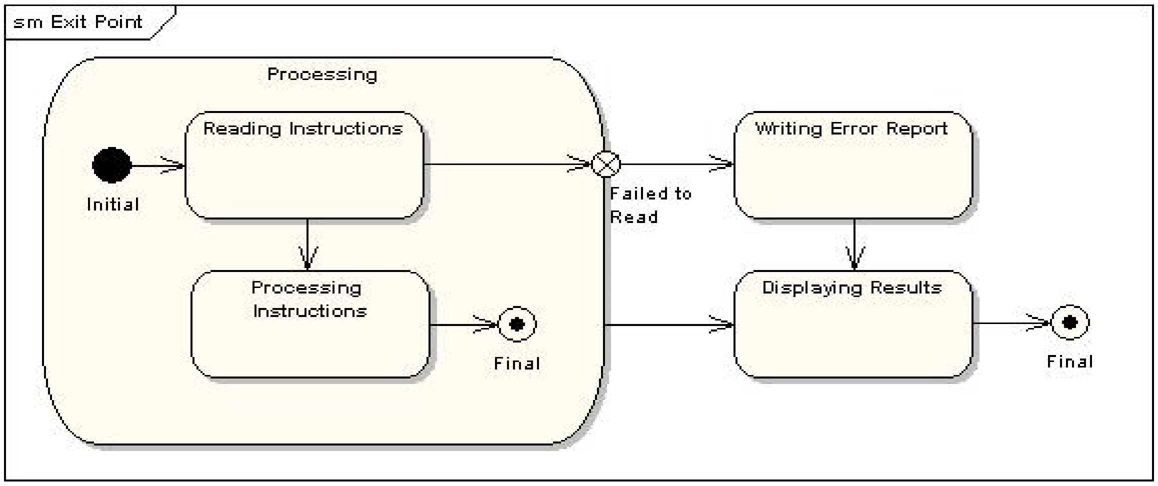


The following diagram shows the state machine one level up.



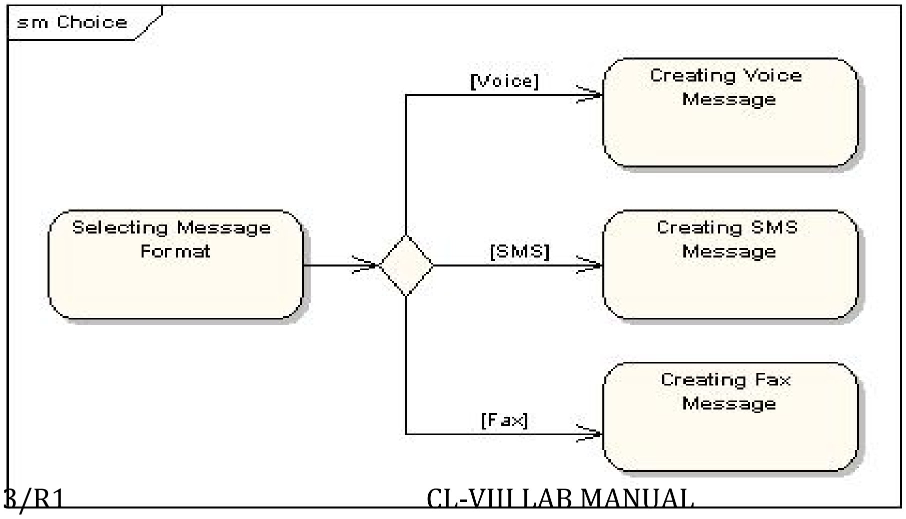
Exit Point

In a similar manner to entry points, it is possible to have named alternative exit points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.



Choice Pseudo-State

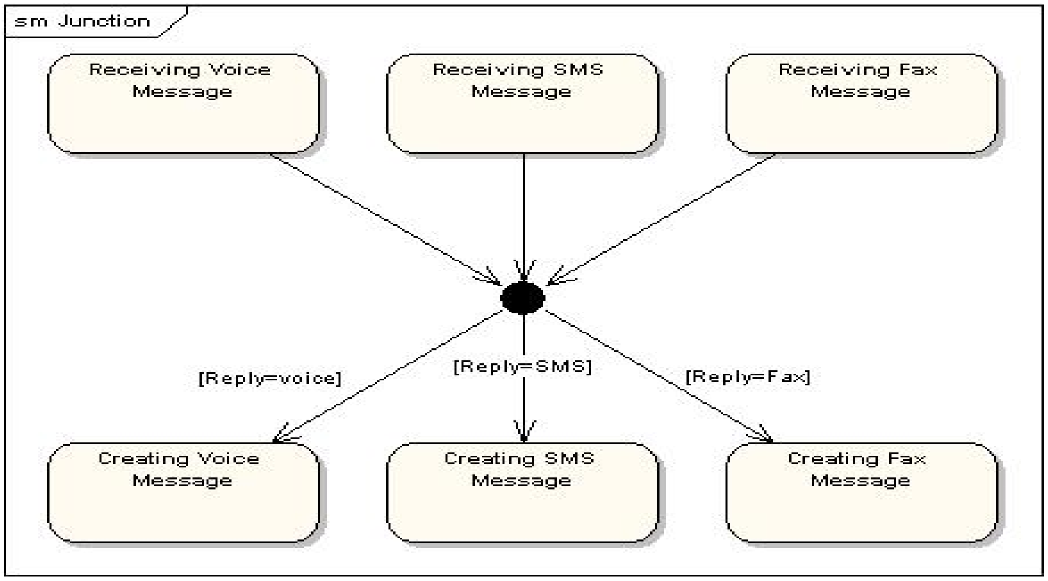
A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.



Junction Pseudo-State

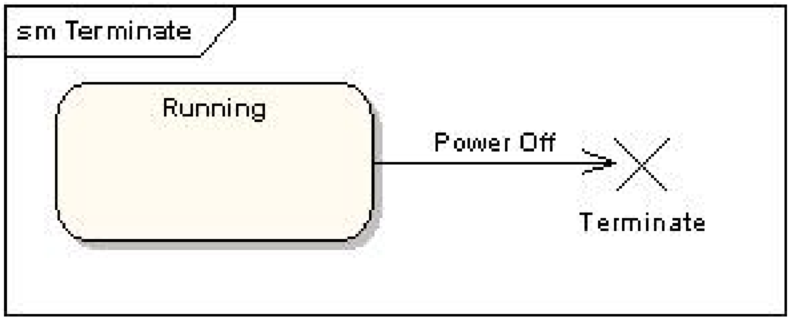
Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free.

A junction which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.



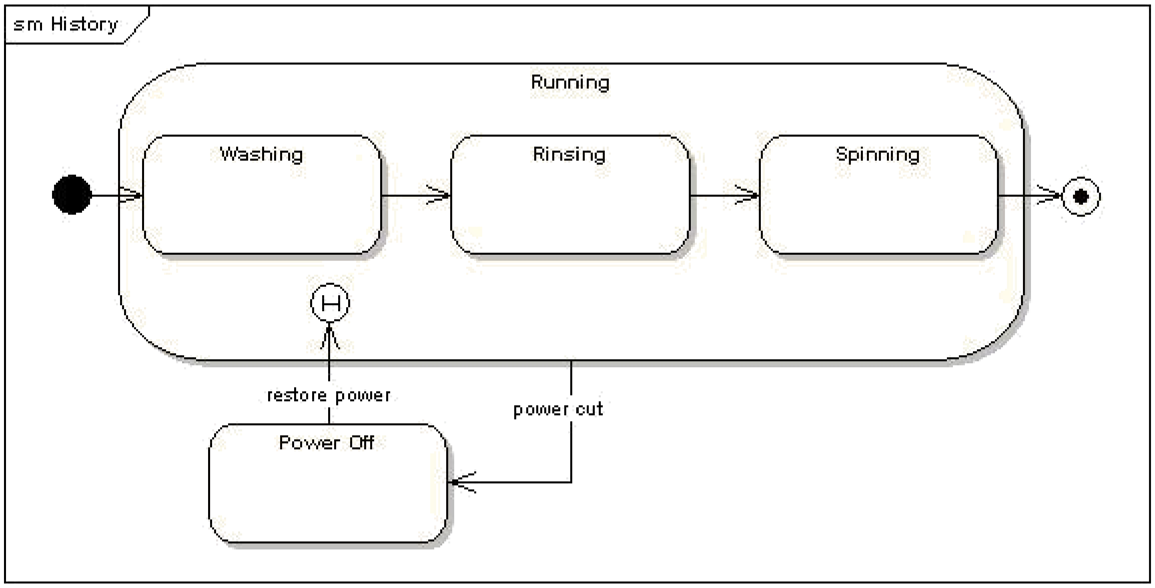
Terminate Pseudo-State

Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated as a cross.



History States

A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.

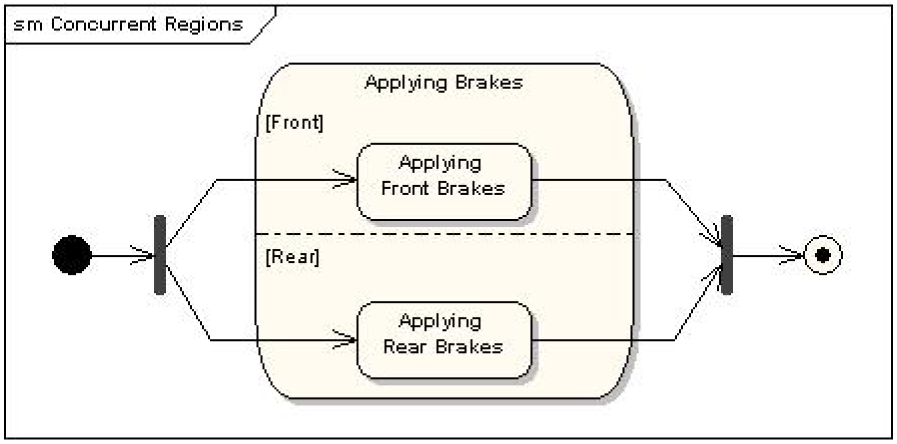


In this state machine, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning".

If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

Concurrent Regions

A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.



How to draw State chart Diagram?

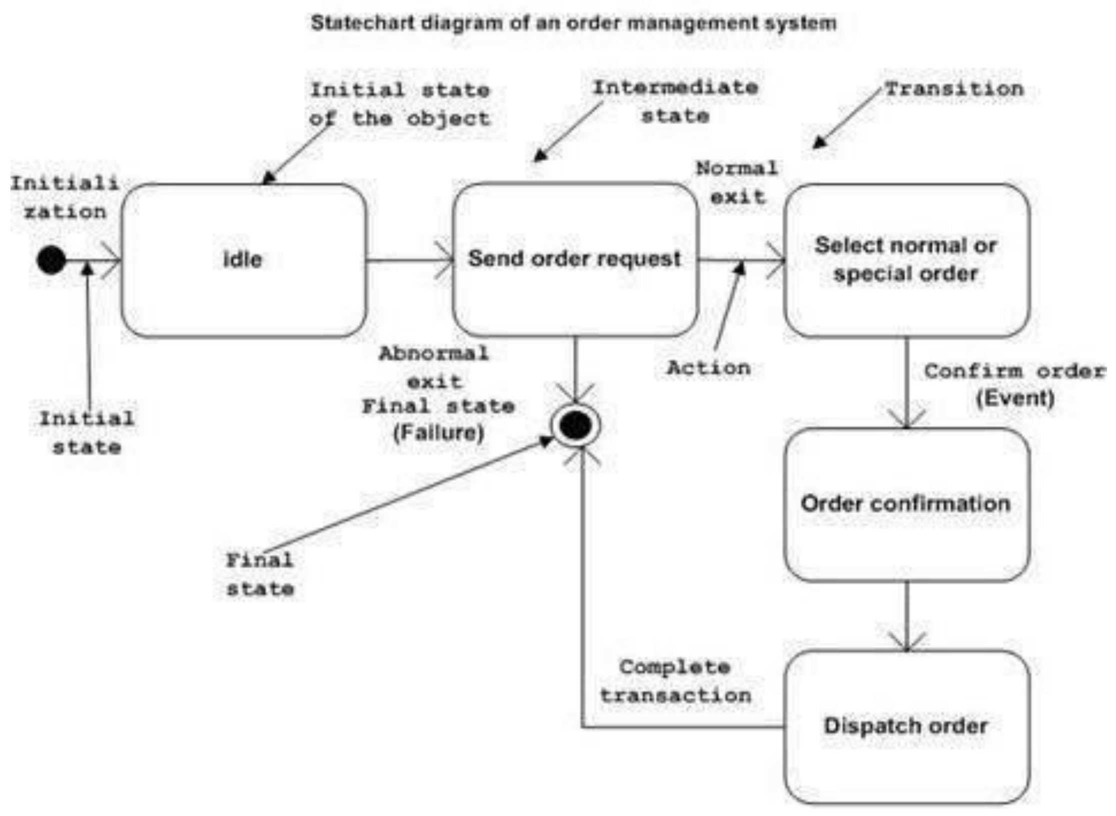
State chart diagram is used to describe the states of different objects in its life cycle. So the emphasis is given on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

State chart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

Before drawing a State chart diagram we must have clarified the following points:

- Identify important objects to be analyzed.
- Identify the states.
- Identify the events.

The following is an **example** of a State chart diagram where the state of Order object is analyzed.
The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for state changes of order object.
During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exists also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete it is considered as the complete transaction as mentioned below. The initial and final state of an object is also shown below.



Where to use State chart Diagrams?

From the above discussion we can define the practical applications of a State chart diagram. State chart diagrams are used to model dynamic aspect of a system like other four diagrams discussed in this tutorial. But it has some distinguishing characteristics for modeling dynamic nature.
State chart diagram defines the states of a component and these state changes are dynamic in nature.
So its specific purpose is to define state changes triggered by events. Events are internal or external factors influencing the system.
State chart diagrams are used to model states and also events operating on the system.

When implementing a system it is very important to clarify different states of an object during its life time and state chart diagrams are used for this purpose. When these states and events are identified they are used to model it and these models are used during implementation of the system.
If we look into the practical implementation of State chart diagram then it is mainly used to analyze the object states influenced by events.
This analysis is helpful to understand the system behavior during its execution.

So the main usages can be described as:

- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.
- Forward and reverse engineering.

REFERENCES	• Grady Booch, James Rumbaugh, Ivor Jacobson, “The UML User Guide”, Second Edition, Addison Wesley • Tom Pender, UML 2 Bible, Wiley India • Ali Bahrami, Object Oriented System Development: Using Unified Modeling Language, McGraw-Hill
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concept of States Transitions, events in the system flow. • State Diagram. • Implementation of StateModel • Conclusion •

EXPERIMENT NO. 8 **Revised on: 15/06/18**

TITLE	**IDENTIFY AND IMPLEMENT GRASP PATTERN**
PROBLEM STATEMENT/ DEFINITION	• Identification and Implementation of GRASP pattern • Apply any two GRASP pattern to refine the Design Model for a given problem description Using effective UML 2 diagrams and implement them with a suitable object oriented language
OBJECTIVE	• To Study GRASP patterns. • To implement system using any two GRASP Patterns.
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool o Eclipse with JAVA ADT • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

GRASP Patterns

What are GRASP Patterns?

They describe fundamental principles of object design and responsibility assignment, expressed as patterns.
After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.

The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way. This approach to understanding and using design principles are based on patterns of assigning responsibilities.

Responsibilities and Methods

The UML defines a responsibility as "a contract or obligation of a classifier". Responsibilities are related to the obligations of an object in terms of its behaviour. Basically, these responsibilities are of the following two types:

- knowing
- doing

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects

- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

Responsibilities are assigned to classes of objects during object design. For example, I may declare that "a Sale is responsible for creating SalesLineItems" (a doing), or "a Sale is responsible for knowing its total" (a knowing). Relevant responsibilities related to "knowing" are often inferable from the domain model because of the attributes and associations it illustrates.

POS (Point Of Sale) application is used to explain all the **GRASP Patterns**.

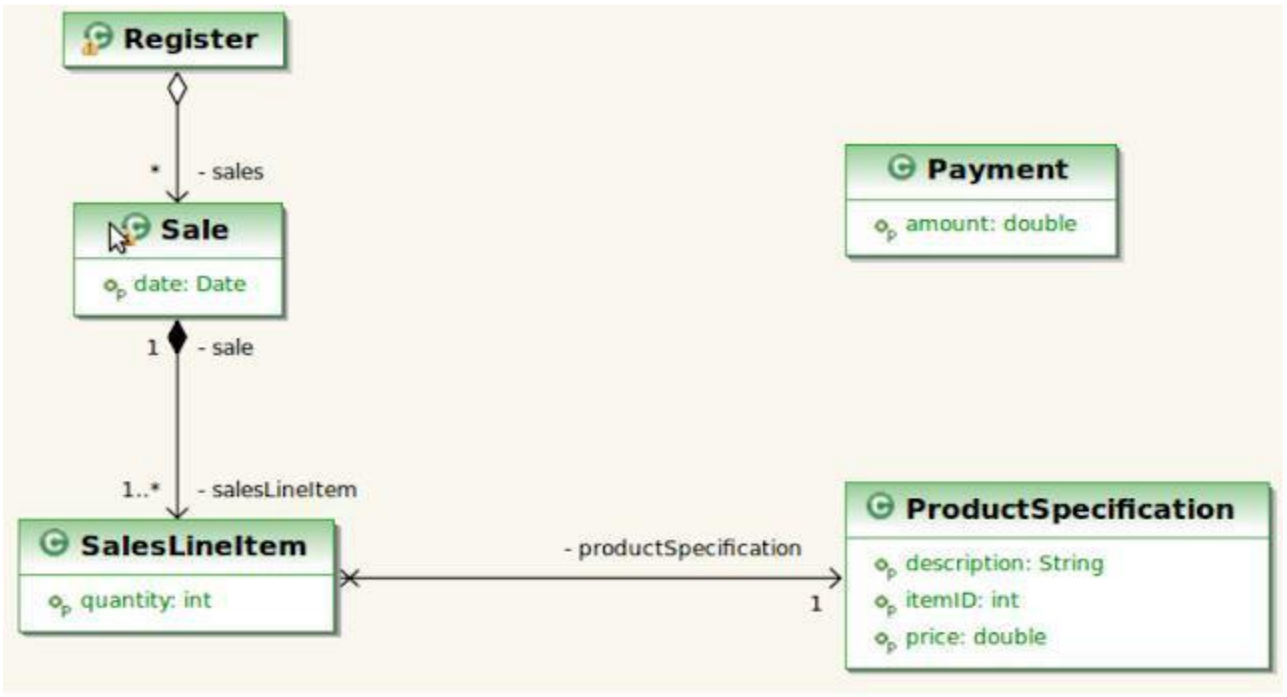
Let's understand the POS(Point Of Sale) application briefly and then apply **GRASP Patterns** to POS Application.

Point of Sale (POS) application

Let's consider Point of Sale (POS) application: a Brief overview of POS application.

1. Application for a shop, restaurant, etc. that registers sales.
2. Each sale is of one or more items of one or more product types and happens at a certain date.
3. A product has a specification including a description, unitary price, and identifier.

4. The application also registers payments (say, in cash) associated with sales.
5. A payment is for a certain amount, equal or greater than the total of the sale.



How to Apply the GRASP Patterns (General Responsibility Assignment Software Patterns)

Let's discuss five GRASP patterns. There is a separate Post for each **GRASP Pattern**

- **Information Expert**
- **Low Coupling**
- **High Cohesion**
- **Controller**
- **Creator**

Information Expert GRASP Pattern

Problem

What is a general principle of assigning responsibilities to objects? A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. Done well, systems tend to be easier to understand, maintain, and extend, and there is more opportunity to reuse components in future applications.

Solution

Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.

Example

Let's consider Point of Sale (POS) application: a Brief overview of POS application.

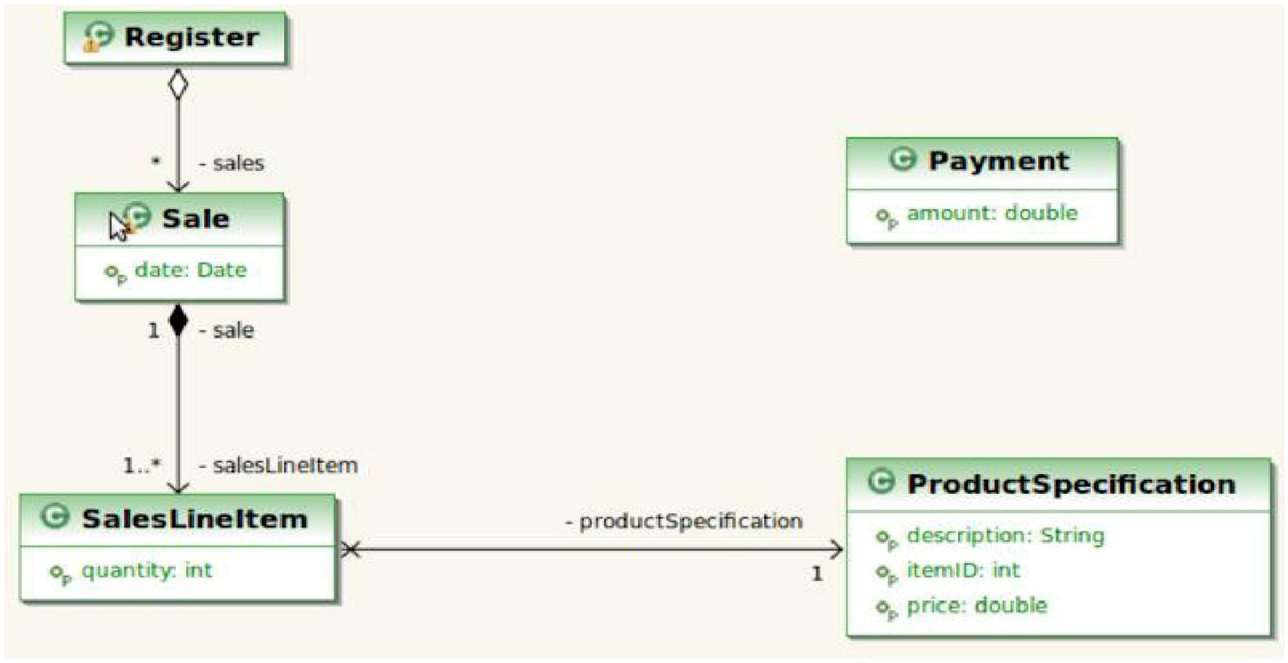
- Application for a shop, restaurant, etc. that registers sales.
- Each sale is of one or more items of one or more product types and happens at a certain date.
- A product has a specification including a description, unitary price, and identifier.
- The application also registers payments (say, in cash) associated with sales.
- A payment is for a certain amount, equal or greater than the total of the sale.

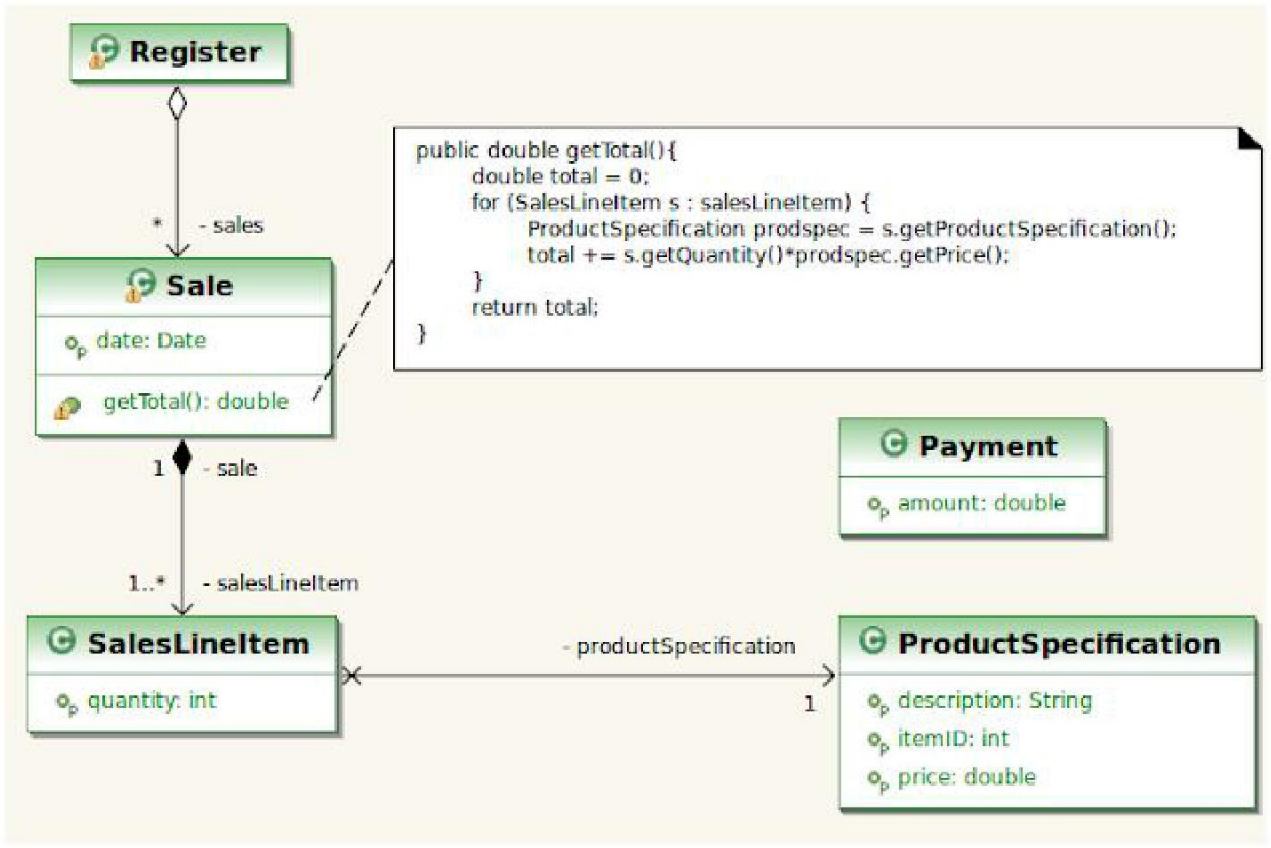
In the POS application, some class needs to know the grand total of a sale.

- Start assigning responsibilities by clearly stating the responsibility.
- By *Information Expert*, we should look for that class of objects that has the information needed to determine the total.
- It is necessary to know about all the **SalesLineItem** instances of a sale and the sum of their subtotals.
- A **Sale** instance contains these; therefore, by the guideline of Information Expert, **Sale** is a suitable class of object for this responsibility; it is an information expert for the work.

- We are not done yet. What information is needed to determine the line item subtotal? **SalesLineItem.quantity** and **ProductSpecification.price** are needed. The **SalesLineItem** knows its quantity and its associated ProductSpecification; therefore, by Expert, **SalesLineItem** should determine the subtotal; it is the information expert.

- In terms of an interaction diagram, this means that the **Sale** needs to send get-Subtotal messages to each of the **SalesLineItems** and sum the results; this design is shown in Figure :

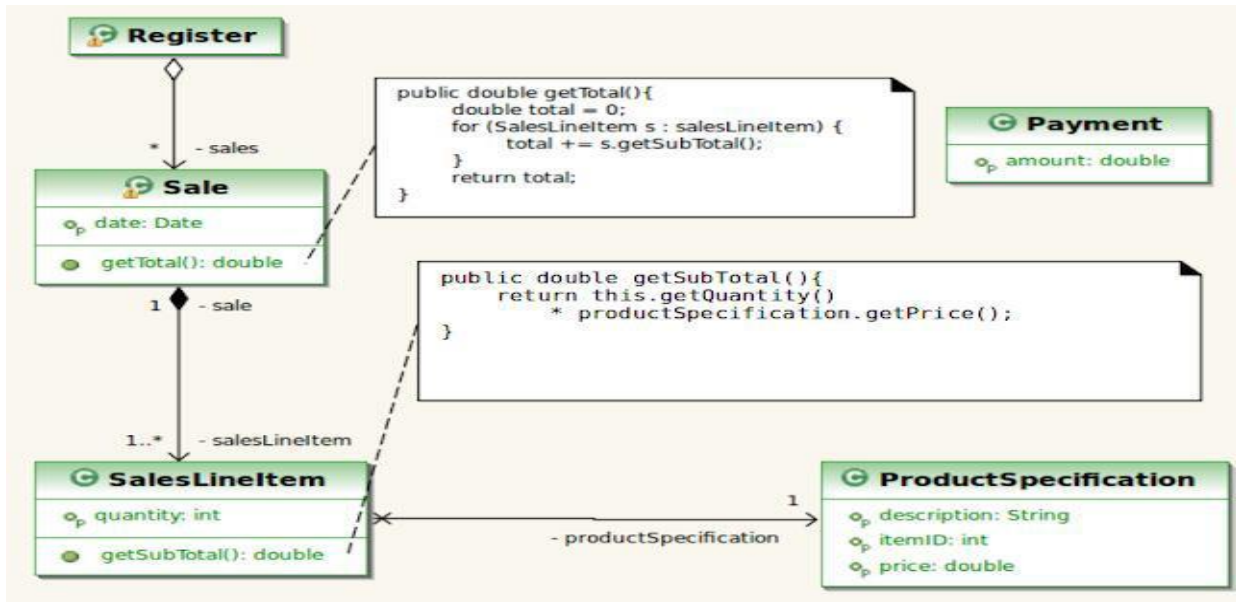




```
public double getTotal(){
        double total = 0;
        for (SalesLineItem s : salesLineItem) {
                ProductSpecification prodspec = s.getProductSpecification();
                total += s.getQuantity()*prodspec.getPrice();
        }
        return total;
}
```

To fulfill the responsibility of knowing and answering its subtotal, a Sales- LineItem needs to know the product price. The **ProductSpecification** is an information expert on answering its **price**; therefore, a message must be sent to it asking for its price.

Complete Class Diagram



```
public double getTotal(){
      double total = 0;
      for (SalesLineItem s : salesLineItem) {
            total += s.getSubTotal();
      }
      return total;
}
```

```
public double getSubTotal(){
     return this.getQuantity()
          * productSpecification.getPrice();
}
```

In conclusion, to fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects as follows

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

Sample Code

Let's write sample method in **Sale** Domain model class.

```
public class Sale {
 //...
 public double getTotal(){
  double total = 0;
  for (SalesLineItem s : salesLineItem) {
   ProductSpecification prodspec
   = s.getProductSpecification();
   total += s.getQuantity()*prodspec.getPrice();
  }
  return total;
 }
}
```

This is above code is enough? It is not enough right because here we need **getSubTotal** from **salesLineItem**.
Let's create the method in **SaleLineItem** domain model class.

```
public class SalesLineItem {
  //...
  public double getSubTotal(){
  return this.getQuantity()
   productSpecification.getPrice();
  }
 }
```

Observe **ProductSpecification** Domain model provides getPrice method gives a price. Fulfillment of a responsibility may require information spread across different classes, each expert on its own data.

Benefits

- Information encapsulation is maintained since objects use their own information to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems. (Low Coupling is also a GRASP pattern that is discussed in the following section).

- Behavior is distributed across the classes that have the required information, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain. High cohesion is usually supported (another pattern discussed later).

Low Coupling GRASP Pattern

Problem

How to support low dependency, low change impact, and increase reuse?

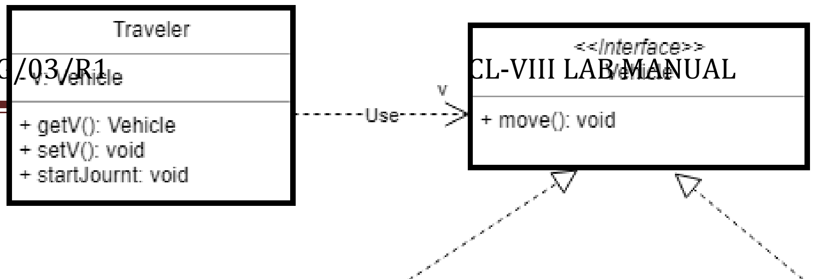
Solution

Assign responsibilities so that coupling remains low. Try to avoid one class to have to know about many others.

Key points about low coupling

- Low dependencies between “artifacts” (classes, modules, components).
- There shouldn’t be too much of dependency between the modules, even if there is a dependency it should be via the interfaces and should be minimal.
- Avoid tight-coupling for collaboration between two classes (if one class wants to call the logic of a second class, then first class needs an object of second class it means the first class creates an object of the second class).
- Strive for loosely coupled design between objects that interact.
- Inversion Of Control (IoC) / Dependency Injection (DI) - With DI objects are given their dependencies at creation time by some third party (i.e. Java EE CDI, Spring DI…) that coordinates each object in the system. Objects aren’t expected to create or obtain their dependencies—dependencies are injected into the objects that need them. The key benefit of DI—loose coupling.

Implementation



This is an example of loose coupling. In this class, **Traveler** class is not tightly coupled with **Car** or **Bike** implementation. Instead by applying dependency injection mechanism, the loose coupling implementation is achieved to allow start journey with any class which has implemented **Vehicle** interface.

Step 1: **Vehicle** interface to allow loose coupling implementation.

```
interface Vehicle {
 public void move();
}
```

Step 2: **Car** class implements Vehicle interface.

```
class Car implements Vehicle {
 @Override
 public void move() {
  System.out.println("Car is moving");
 }
}
```

Step 3: **Bike** class implements Vehicle interface.

```
class Bike implements Vehicle {
 @Override
 public void move() {
  System.out.println("Bike is moving");
 }
}
```

Step 4: Now create **Traveller** class which holds the reference to Vehicle interface.

```
class Traveler {
 private Vehicle v;
 public Vehicle getV() {
  return v;
 }
 public void setV(Vehicle v) {
     this.v = v;
 }
 public void startJourney() {
  v.move();
 }
}
```

Step 5: Test class for loose coupling example - Traveler is an example of loose coupling.

```
public static void main(String[] args) { Traveler traveler =
 new Traveler(); traveler.setV(new Car()); // Inject Car
 dependency traveler.startJourney(); // start journey by
 Car traveler.setV(new Bike()); // Inject Bike dependency
 traveler.startJourney(); // Start journey by Bike

}
```

High Cohesion GRASP Pattern

Problem

How to keep classes focused, understandable and manageable?

Solution

Assign responsibilities so that cohesion remains high. Try to avoid classes to do too much or too different things.

The term cohesion is used to indicate the degree to which a class has a single, well-focused responsibility. Cohesion is a measure of how the methods of a class or a module are meaningfully and strongly related and how focused they are in providing a well-defined purpose to the system.

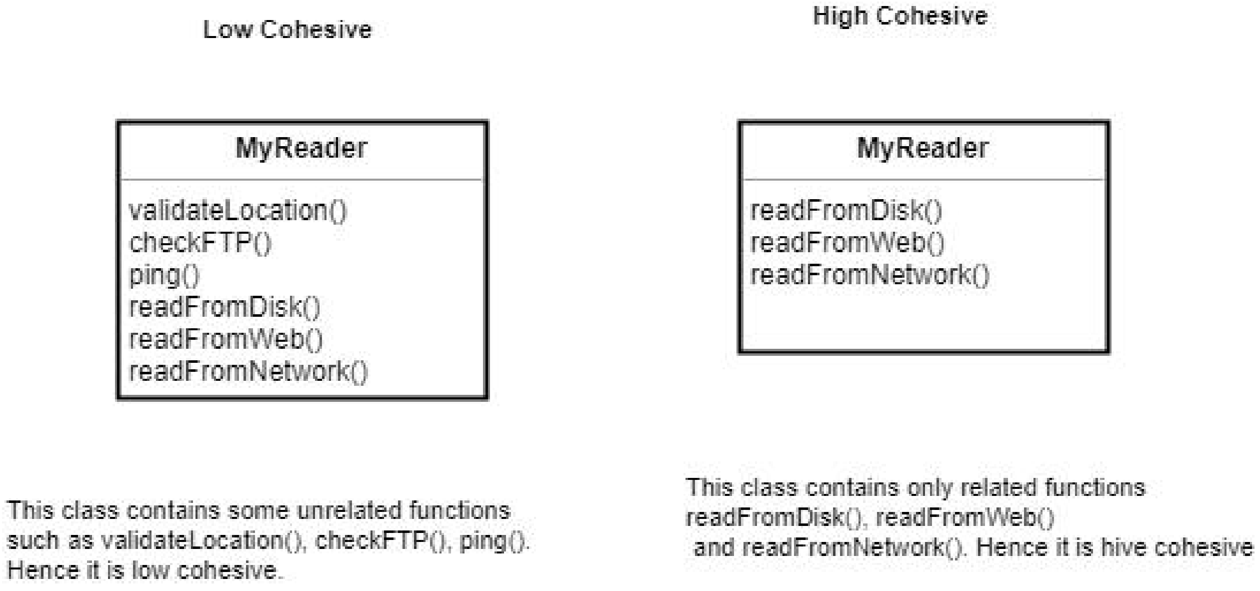
A class is identified as a low cohesive class when it contains many unrelated functions within it. And that what we need to avoid because big classes with unrelated functions hamper their maintaining. Always make your class small and with precise purpose and highly related functions.

Key points about high cohesion

- The code has to be very specific in its operations.
- The responsibilities/methods are highly related to class/module.
- The term cohesion is used to indicate the degree to which a class has a single, well-focused responsibility. Cohesion is a measure of how the methods of a class or a module are meaningfully and strongly related and how focused they are in providing a well-defined purpose to the system. The more focused a class is the higher its cohesiveness - a good thing.
- A class is identified as a low cohesive class when it contains many unrelated functions within it. And that what we need to avoid because big classes with unrelated functions hamper their maintaining. Always make your class small and with precise purpose and highly related functions.

Example

In this example, the purpose of **MyReader** class is to read the resource and it does that only. It does not implement other unrelated things. Hence it is highly cohesive.



This class contains some unrelated functions such as validateLocation(), checkFTP(), ping(). Hence it is low cohesive.

This class contains only related functions readFromDisk(), readFromWeb() and readFromNetwork(). Hence it is hive cohesive

```
class HighCohesive {
 //   -------------- functions related to read resource
 //   read the resource from disk
 public String readFromDisk(String fileName) {
  return "reading data of " + fileName;
 }

 // read the resource from web
 public String readFromWeb(String url) {
  return "reading data of " + url;
 }

 // read the resource from network
 public String readFromNetwork(String networkAddress) {
  return "reading data of " + networkAddress;
 }
}
```

Controller GRASP Pattern

Problem

Who should be responsible for handling an input system event?

An input system event is an event generated by an external actor. They are associated with system operations of the system in response to system events,

just as messages and methods are related. For example, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.

Who should be responsible for handling an input event, which object/s beyond the UI layer receives interaction?

Solution

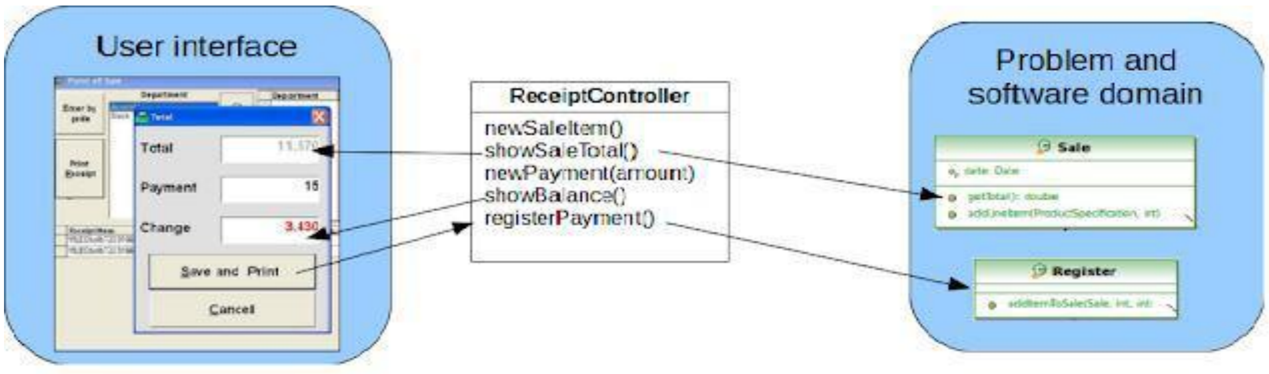
Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- Represents the overall system, device, or subsystem (facade controller).
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session (use-case or session controller).
- Use the same controller class for all system events in the same use case scenario.

- Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).

Corollary: Note that "window," "applet," "widget," "view," and "document" classes are not on this list. Such classes should not fulfill the tasks associated with system events, they typically receive these events and delegate them to a controller.

Example



Benefits

- Either the UI classes or the problem/software domain classes can change without affecting the other side.
- The controller is a simple class that mediates between the UI and problem domain classes, just forwards
- event handling requests
- output requests

Creator GRASP Pattern

Problem

Who should be responsible for creating a new instance of some class? The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability. Solution

Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B aggregates A objects.
- B contains A objects.
- B records instances of A objects.
- B closely uses A objects.
- B has the initializing data that will be passed to A when it is created (thus B is an Expert with respect to creating A).

B is a creator of A objects.

If more than one option applies, prefer a class B which aggregates or contains class A.

Example

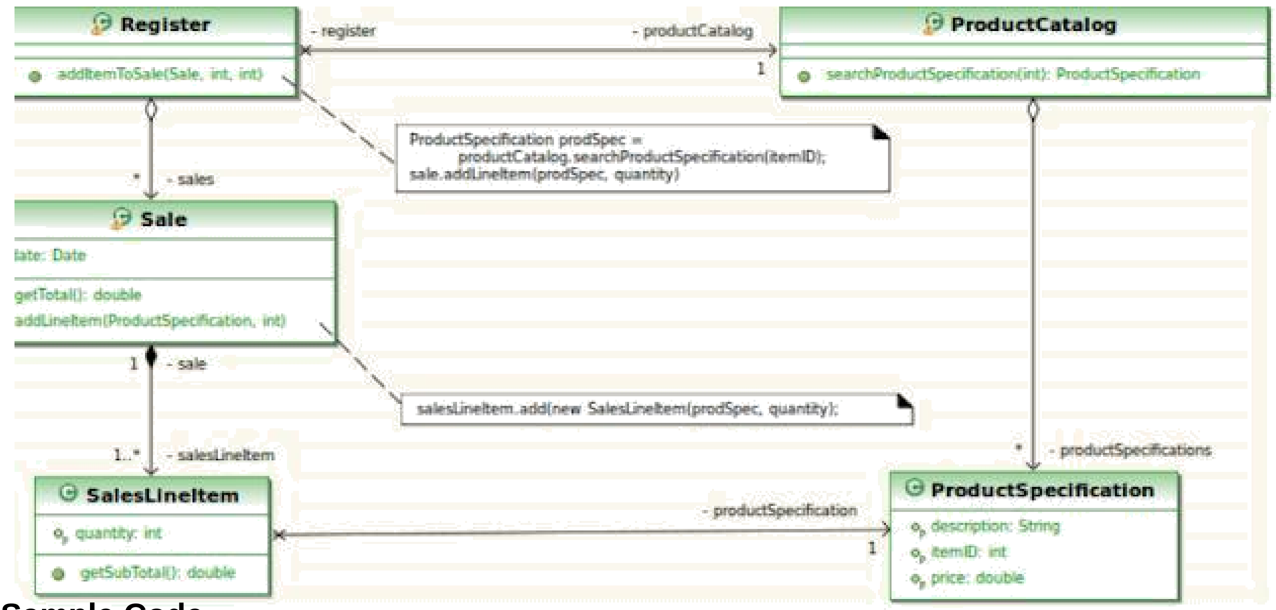
Let's consider Point of Sale (POS) application: a Brief overview of POS application.
Application for a shop, restaurant, etc. that registers sales.
Each sale is of one or more items of one or more product types and happens at a certain date.
A product has a specification including a description, unitary price, and identifier.

The application also registers payments (say, in cash) associated with sales.
A payment is for a certain amount, equal or greater than the total of the sale.

Problem statement: who should be responsible for creating a SalesLineItem instance?

In the POS application, who should be responsible for creating a SalesLineItem instance? By Creator, we should look for a class that aggregates, contains, and so on, SalesLineItem instances.

Since a Sale contains (in fact, aggregates) many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.
This leads to a design of object interactions: Creating a SalesLineItem


Register
addItemToSale(Sale, int, int)
- register
- productCatalog
1
ProductCatalog
searchProductSpecification(int): ProductSpecification
ProductSpecification prodSpec = productCatalog.searchProductSpecification(itemID); sale.addLineItem(prodSpec, quantity)
* - sales
Sale
date: Date
getTotal(): double
addLineItem(ProductSpecification, int)
1 - sale
salesLineItem.add(new SalesLineItem(prodSpec, quantity);
1..* - salesLineItem
* - productSpecifications
SalesLineItem
quantity: int
getSubTotal(): double
- productSpecification
1
ProductSpecification
description: String
itemID: int
price: double

Sample Code

```
class Sale {
    List<SalesLineItem> salesLineItem
    = new ArrayList<SalesLineItem>();
    //...
    public void addLineItem(ProductSpecification prodSpec,int quantity) {
        salesLineItem.add(new SalesLineItem(prodSpec, quantity);
 return salesLineItem;
    }
}
```

Benefits

- Low coupling (described next) is supported, which implies lower maintenance dependencies and higher opportunities for reuse. Coupling is probably not increased because the created class is likely already visible to the creator class, due to the existing associations that motivated its choice as creator.

REFERENCES	• Craig Larman, Applying UML and Patterns, Pearson Education, Second Edition. • Erich Gamma et al, Design Patterns: Elements of Reusable Object, Pearson, First Edition
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concept Design pattern -GRASP. • UML Diagram applicable to system using GRASP design pattern. • Implementation of system using patterns. • Conclusion

EXPERIMENT NO. 9 **Revised on: 15/06/18**

TITLE	**IDENTIFY AND IMPLEMENT GOF PATTERN**
PROBLEM STATEMENT/ DEFINITION	• Identification and Implementation of GOF pattern • Apply any two GOF pattern to refine Design Model for a given problem description Using effective UML 2 diagrams and implement them with a suitable object oriented language.
OBJECTIVE	• To Study GOF Patterns. • To identify applicability of GOF in the system • Implement System with pattern.
S/W PACKAGES AND H/W USED	• S/W: o Ubuntu/Linux OS o Any UML 2.0 tool o Eclipse with JAVA ADT • H/W: o Dual core Intel / AMD architecture machine with 2GB RAM.

RELEVANT THEORY

Strategy Design Pattern

In Software Engineering, the strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm's behavior to be selected at runtime.

The strategy pattern

- Defines a family of algorithms,
- Encapsulates each algorithm, and
- Makes the algorithms interchangeable within that family. Strategy

lets the algorithm vary independently from clients that use it.

Example 1:

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors.

These factors are not known for each case until run-time, and may require radically different validation to be performed.

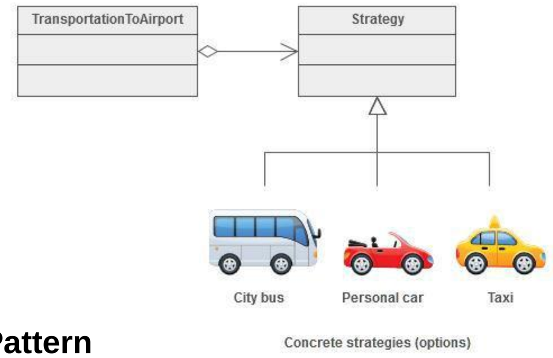
The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

Example 2 :

Modes of transportation to an airport are an example of a Strategy.

Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service.

For some airports, subways and helicopters are also available as a mode of transportation to the airport.
Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably.
The traveler must chose the Strategy based on trade-offs between cost, convenience, and time.

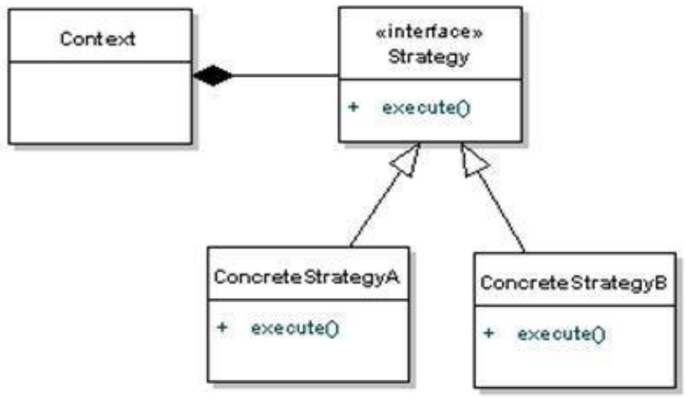


Definition: Strategy Pattern

- ❑ The Strategy pattern is known as a behavioral pattern - it's used to manage algorithms, relationships and responsibilities between objects.
- ❑ The definition of Strategy provided in the original **Gang of Four book** on Design Patterns states:

“Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior”

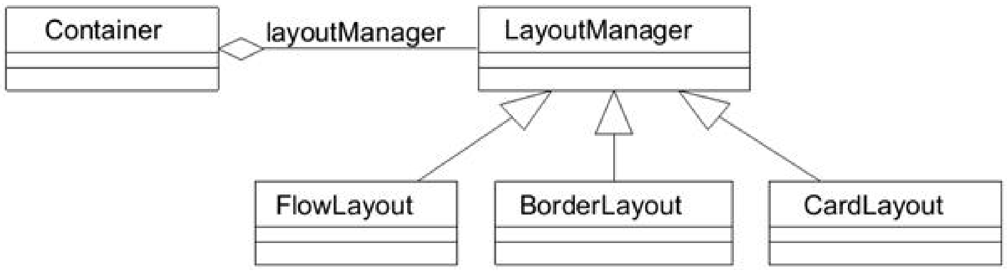
Now, let's take a look at the diagram definition of the Strategy pattern.



- ❑ Here we rely on composition instead of inheritance for reuse.
- ❑ In the above diagram **Context** is composed of a **Strategy**. Instead of implementing a behavior the **Context** delegates it to **Strategy**.
- ❑ The context could be anything that would require changing behaviours
- ❑ The **Strategy** is simply implemented as an interface, so that we can swap **ConcreteStrategy**s in and out without affecting our **Context.**

Strategy Pattern Example 3

Situation: A GUI container object wants to decide at run-time what strategy it should use to layout the GUI components it contains. Many different layout strategies are already available.
Solution: Encapsulate the different layout strategies using the Strategy pattern This is what the Java AWT does with its LayoutManagers



Advantages:

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.

2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a "switch" statement or a series of "if-else" statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

Disadvantages:

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

Steps to implementation

- ❑ Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
- ❑ Specify the signature for that algorithm in an interface.
- ❑ Bury the alternative implementation details in derived classes.
- ❑ Clients of the algorithm couple themselves to the interface.

We are going to create a ***Strategy*** interface defining an action and ***concrete strategy classes*** implementing the *Strategy* interface.

Context is a class which uses a Strategy.

StrategyPatternDemo**,** our demo class, will use *Context* and strategy objects to demonstrate change in Context behavior based on strategy it deploys or uses

Applications

- ❑ The Strategy pattern is to be used where you want to choose the algorithm to use at runtime.
- ❑ A good use of the Strategy pattern would be saving files in different formats, running various sorting algorithms, or file compression.
- ❑ The Strategy pattern provides a way to define a family of algorithms, encapsulate each one as an object, and make them interchangeable.

❖ **Exercises:**

Implement Strategy Pattern for Adding 2 integer numbers, multiplying two integer numbers, dynamically at runtime. Consider an interface called strategy with an operation called arithmetic operation which can be implemented by respective addition and multiplication class (Strategy concrete classes)

State Design Pattern

State pattern is one of the behavioral design patterns. State design pattern is used when an Object change its behavior based on its internal state. The state pattern is a behavioral software design pattern that implements a state machine in an object-oriented way.

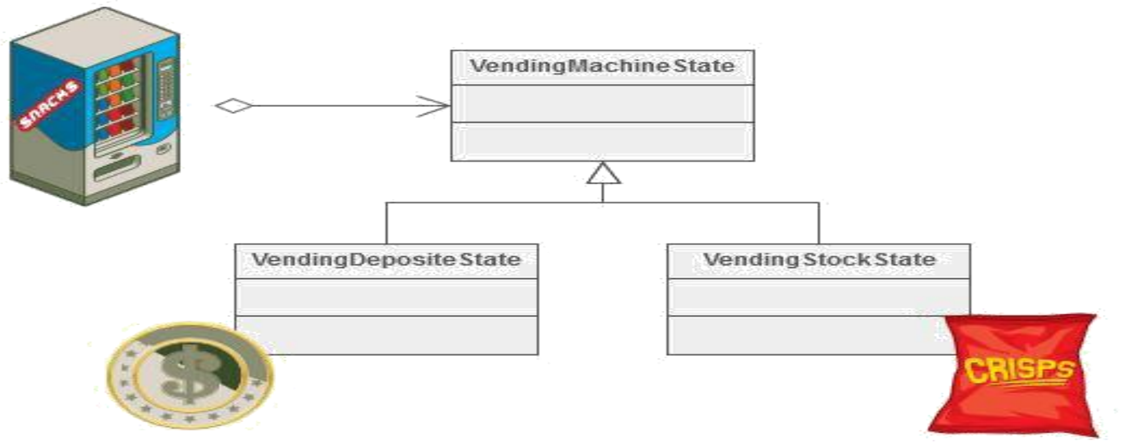
With the state pattern, a state machine is implemented by implementing each This pattern is used in computer programming to encapsulate varying behavior for the same object based on its internal state. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic /huge conditional statements and thus improve maintainability.

Example

The State pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine.

Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc.

When currency is deposited and a selection is made, a vending machine will either deliver a product and no change,



Problem in Software Design / Coding Style

- ❑ When writing code, our classes often go through a series of transformations.

- ❑ What starts out as a simple class will grow as behavior is added.
- ❑ if you didn't take the necessary precautions, your code will become difficult to understand and maintain.
- ❑ Too often, the state of an object is kept by creating multiple Boolean attributes and deciding how to behave based on the values.
- ❑ This can become cumbersome and difficult to maintain when the complexity of your class starts to increase.
- ❑ This is a common problem on most projects.

Suppose we want to implement a TV Remote object with a simple button to perform action, if the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV. We can implement it using if-else condition like below;

Notice that following are the problems in above design

- ❑ client code should know the specific values to use for setting the state of remote,

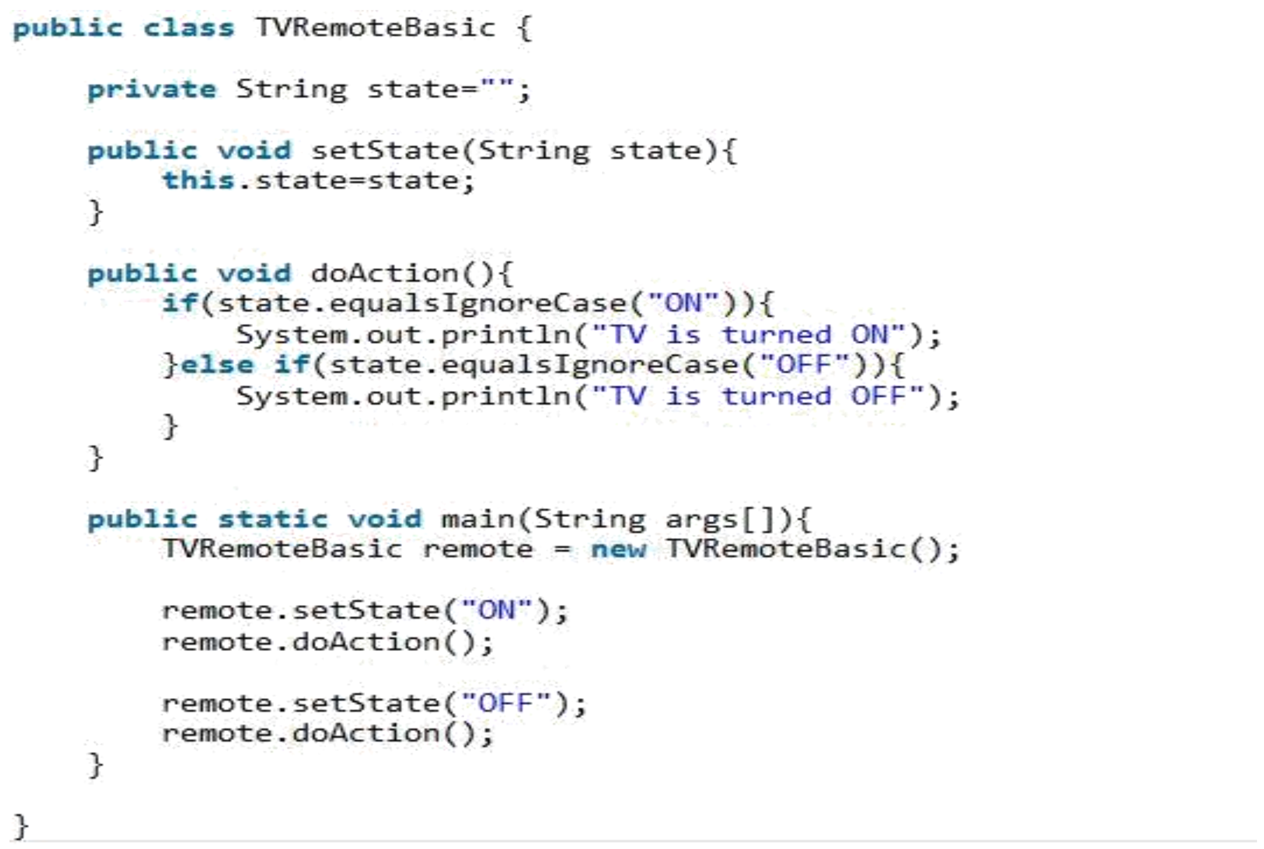
- ❑ Further more if number of states increases then the **tight coupling** between **implementation and the client code** will be very hard to maintain and extend.

❖ **How can we avoid this?**

- ❑ Design / Code will become more cleaner by using **Enums and Switches** or multiple if then else (Similar to our SavingAccount object in Assignment no.3)

- ❑ Here in this solution instead of a bunch of flags, we will just have one state_ field. We also flip the order of our branching.

But again code becomes **monolithic or as single block.**



```java
public class TVRemoteBasic {

    private String state="";

    public void setState(String state){
        this.state=state;
    }

    public void doAction(){
        if(state.equalsIgnoreCase("ON")){
            System.out.println("TV is turned ON");
        }else if(state.equalsIgnoreCase("OFF")){
            System.out.println("TV is turned OFF");
        }
    }

    public static void main(String args[]){
        TVRemoteBasic remote = new TVRemoteBasic();

        remote.setState("ON");
        remote.doAction();

        remote.setState("OFF");
        remote.doAction();
    }

}
```

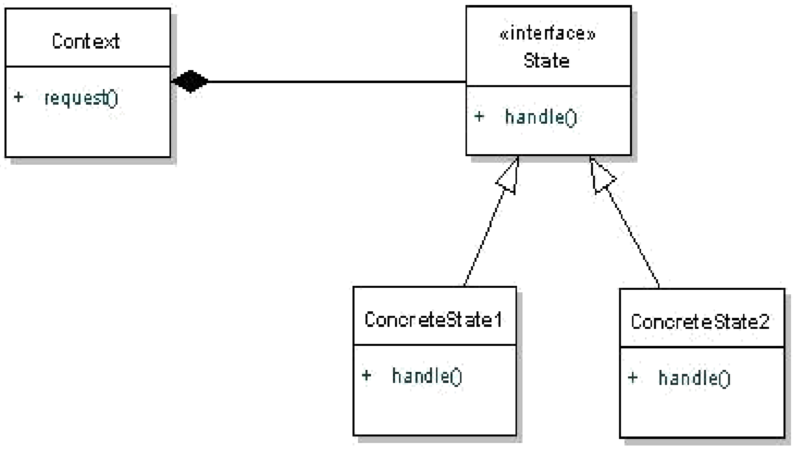
This is a common problem on most projects, and it is wise to model it with a Finite State Machine.
In fact, there is a **design pattern called State that address this very well**, so you can find hundreds of gems implementing this pattern.

Definition: State Pattern

- The State pattern is known as a behavioral pattern - it's used to manage algorithms, relationships and responsibilities between objects.
- In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern.
- In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.
- The definition of State provided in the original Gang of Four book on Design Patterns states:

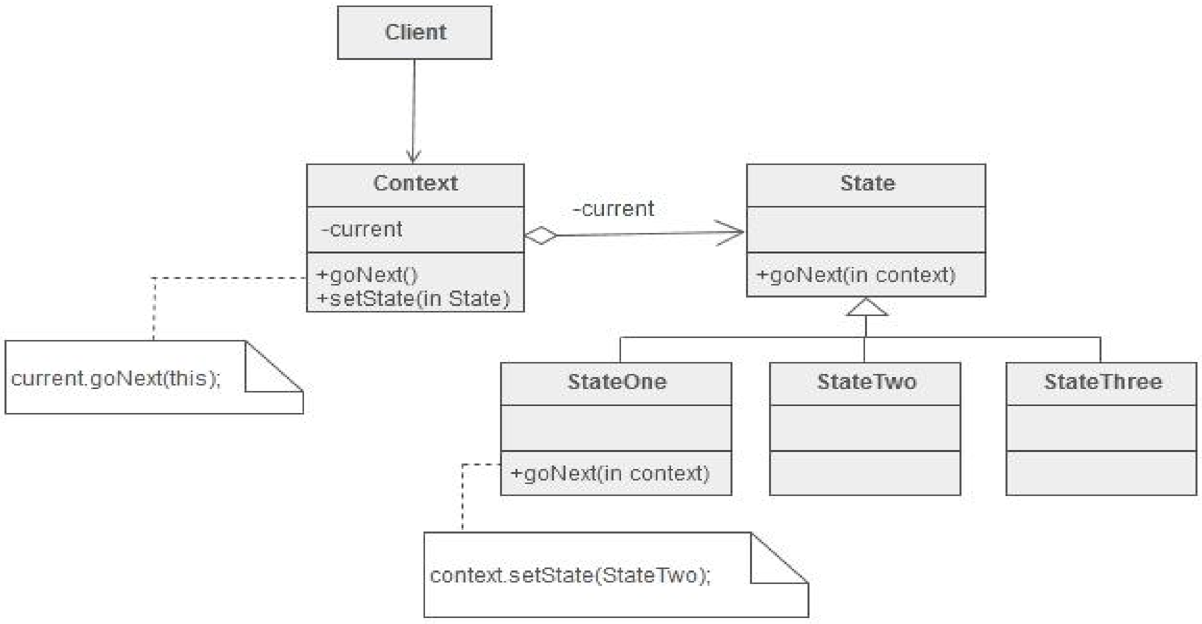
"Allows an object to alter its behavior when its internal state changes. The object will appear to change its class."

Let's take a look at the diagram definition:



- The **Context** can have a number of internal States, whenever the request() method is called on the Context, the message is delegated to the State to handle.

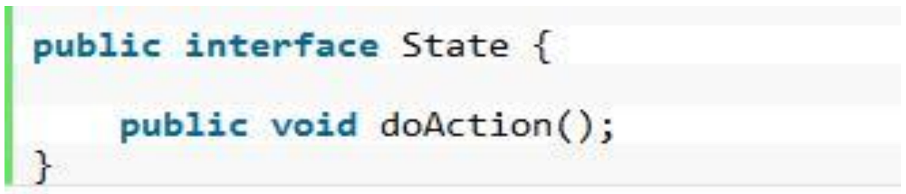
- The **State** interface defines a common interface for all concrete states, encapsulating all behavior associated with a particular state.
- The **ConcreteState** implements its own implementation for the request.
- When a Context changes state, what really happens is that we have a different ConcreteState associated with it.



Solution to above problem in terms of Sate Pattern

❖ **State Interface**

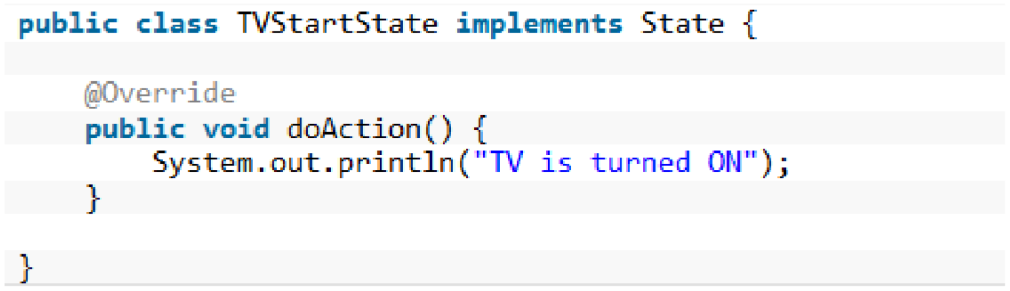
First of all we will create State interface that will define the method that should be implemented by different concrete



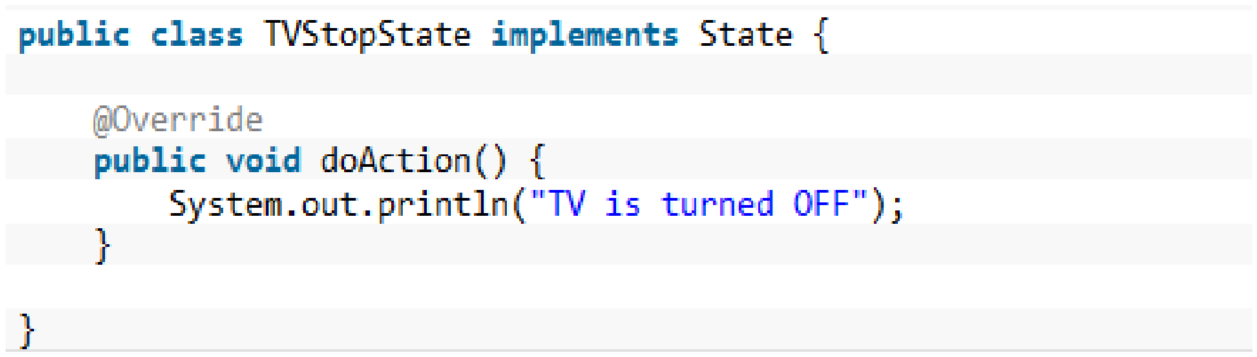
```
public interface State {

    public void doAction();
}
```

❖ **Concrete State Implementations**

In our example, we can have two states – one for turning TV on and another to turn it off. So we will create two concrete states



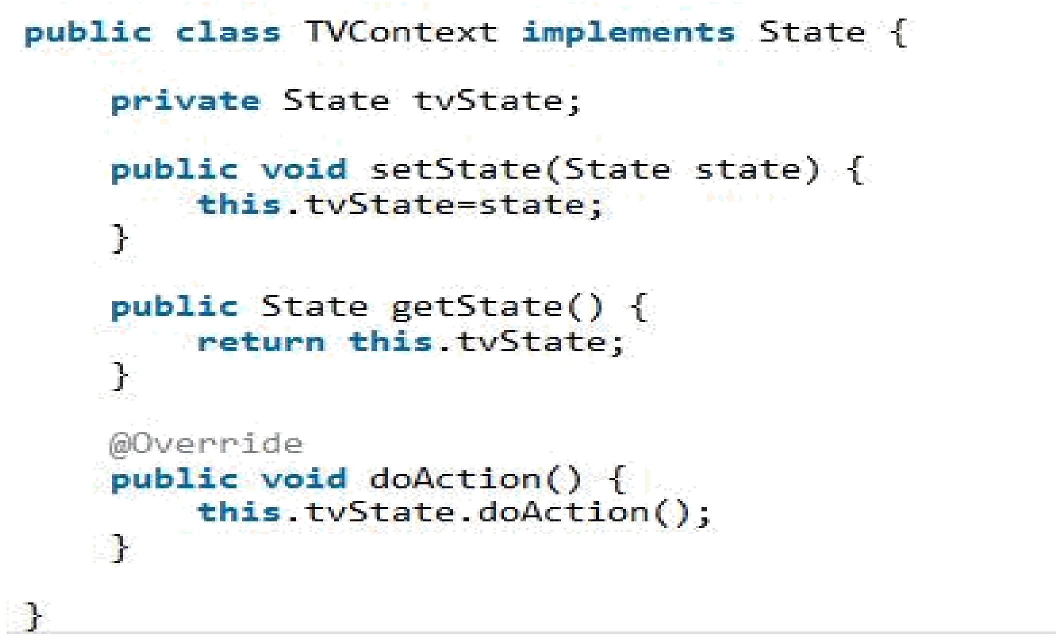
```
public class TVStartState implements State {

    @Override
    public void doAction() {
        System.out.println("TV is turned ON");
    }

}
```



```
public class TVStopState implements State {

    @Override
    public void doAction() {
        System.out.println("TV is turned OFF");
    }

}
```

❖ **Context Implementation**

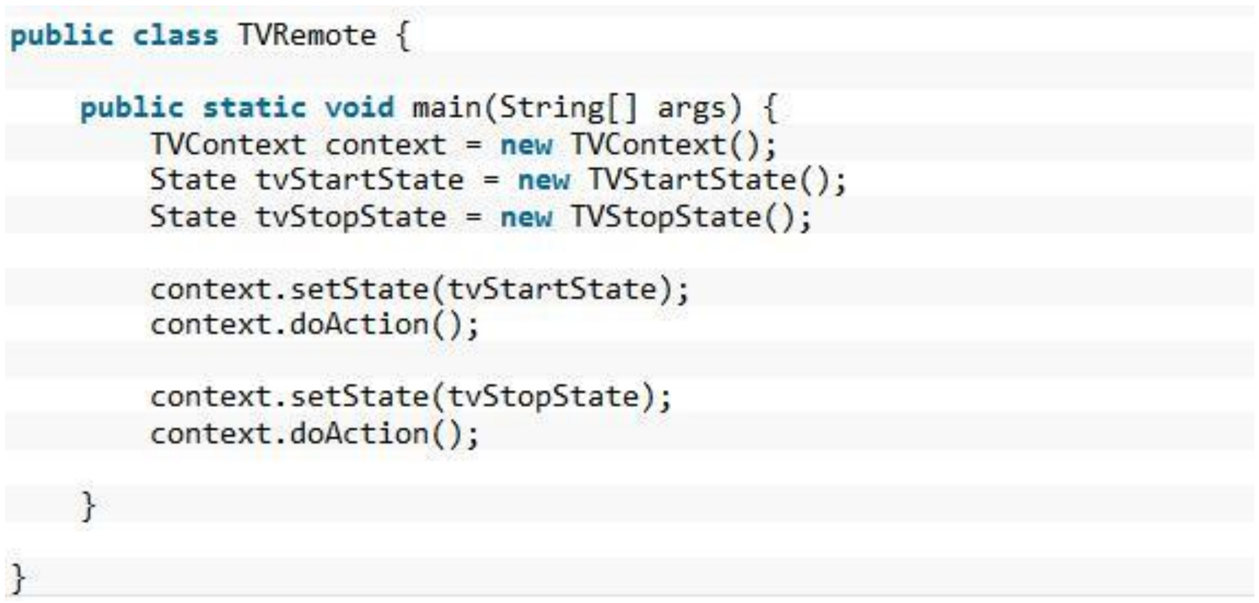
Now we are ready to implement our Context object that will change its behavior based on its internal state.



```java
public class TVContext implements State {

    private State tvState;

    public void setState(State state) {
        this.tvState=state;
    }

    public State getState() {
        return this.tvState;
    }

    @Override
    public void doAction() {
        this.tvState.doAction();
    }

}
```

❖ **Test Program /Client Program**

Now let’s write a simple program to test our implementation of TV Remote using State pattern.



```java
public class TVRemote {

    public static void main(String[] args) {
        TVContext context = new TVContext();
        State tvStartState = new TVStartState();
        State tvStopState = new TVStopState();

        context.setState(tvStartState);
        context.doAction();

        context.setState(tvStopState);
        context.doAction();

    }

}
```

Advantages:

- The benefits of using State pattern to implement polymorphic behavior is clearly visible,
- the chances of error are less and it’s very easy to add more states for additional behavior making it more robust,
- easily maintainable and flexible.
- Also State pattern helped in avoiding if-else or switch-case conditional logic in this scenario.
- Avoiding inconsistent states
- Putting all associated behavior together in one state object
- Removes monolithic if or case statements

Steps to implementation

- ❑ Create an interface /create our state interface
- ❑ Create concrete classes implementing the same interface.
- ❑ Create *Context* Class. /set up a context
- ❑ Use the *Context* to see change in behavior when *State* changes.

Applications

❖ **Where Would I Use This Pattern?**

- ❑ You should use the State pattern when the behavior of an object should be influenced by its state, and when complex conditions tie object behavior to its state.

❖ **Exercises:**

Problem Description

1. Design and Implement state design pattern for media player having start state and stop state. Consider an interface called state with an operation called do operation which can be implemented by respective state classes (State concrete classes). If state is START, it will turn on media player and if state is STOP, it will turn the media player off

❖ **FREQUENTLY ASKED QUESTIONS FOR ORAL EXAMINATION:**

- ➢ What do you mean by Design pattern?
- ➢ What are the different types of design pattern, explain each in detail?
- ➢ What is open and close principal for object oriented system
- ➢ Explain problem of multiplicity of design

REFERENCES	• Craig Larman, Applying UML and Patterns, Pearson Education, Second Edition. • Erich Gamma et al, Design Patterns: Elements of Reusable Object, Pearson, First Edition
INSTRUCTIONS FORWRITING JOURNAL	• Title • Problem Definition • Concept of GOF Patterns. • Class Diagram by applying GOF pattern. • Implementation of system using GOF patterns. • Conclusion