Army Institute of Technology,Pune

DEPARTMENT OF INFORMATION TECHNOLOGY

LABORATORY MANUAL

Computer Laboratory –VIII

BE (I.T) SEMESTER – I

(2019-20)

Lab Incharge

Prof G.M.Walunjkar

Teaching Scheme	**Examination Scheme**
Practical Session : 4hrs/week	Term Work : 50 marks Oral : 50 Marks Credits:02

LABORATORY OVERVIEW

CL-VIII Lab provides an exposure to the students with hands on experience on various software / tools such as database tool -Oracle, design tool – Enterprise Architecture / Rational Rose - UML 2.0.

The main focus of this lab is to understand the "software development life cycle" so that students will understand Object Oriented Software Development. The lab teaches the basics of UML to create different system prototypes – specification, behavior, components and deployment.

There are total 9 assignments based on design and testing in the lab sessions.

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 fo 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:

By the end of the course, students should be able to

1. Draw, discuss different UML 2.0 diagrams, their concepts, notation, advanced notation, forward and reverse engineering aspects.

2. Identify different software artifacts used to develop analysis and design model from requirements.

3. Develop use case model.

4. Develop, implement analysis model and design model.

5. Develop, implement Interaction and behavior Model.

6. Implement an appropriate design pattern to solve a design problem

List of Assignments Computer Lab VIII

I N D E X

Sr. No.	Name of Assignments
1	**Write Problem Statement for System / Project**
2	**Prepare Use Case Model**
3	**Prepare Activity Model**
4	**Prepare Analysis Model-Class Model**
5	**Prepare a Design Model from Analysis Model**
6	**Prepare Sequence Model.**
7	**Prepare a State Model**
8	**Identification and Implementation of GRASP pattern**
9	**Identification and Implementation of GOF pattern**

Assignment 1: Write Problem Statement for System / Project

Description: Identify Project of enough complexity, which has at least 4-5 major functionalities.
Identify stakeholders, actors and write detail problem statement for your system.

Aim **:** To Prepare Problem Statement for the System

Objective **:** To learn to identify stakeholders, actors and prepare SRS.

Software Requirements Specification (Template)

1. Introduction

1.1 Purpose

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

1.2 Document Conventions

<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. Overall Description

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.>

Assignment 2: Prepare Use Case Model

Description :Identify Major Use Cases, Identify actors ,Write Use Case Specification for all major Use cases .Draw detail Use Case Diagram using UML 2.0

Theory : Use case diagrams describe required usages of a system, or what a system is supposed to do. The key concepts that take part in a use case diagram are actors, use cases,

and subjects. A subject represents a system under consideration with which the actors and other subjects interact. The required behavior of the subject is described by the use cases.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

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

How to Draw a Use Case Diagram?

Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases.

We can say that use cases are nothing but the system functionalities written in an organized manner. The second thing which is relevant to use cases are the actors. Actors can be defined as something that interacts with the system.

Actors can be a human user, some internal applications, or may be some external applications. When we are planning to draw a use case diagram, we should have the following items identified.

- Functionalities to be represented as use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram

- The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.
- Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
- Use notes whenever required to clarify some important points.

Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases **(Order, SpecialOrder, and NormalOrder)** and one actor which is the customer.

The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship. Another important point is to identify the system boundary, which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.

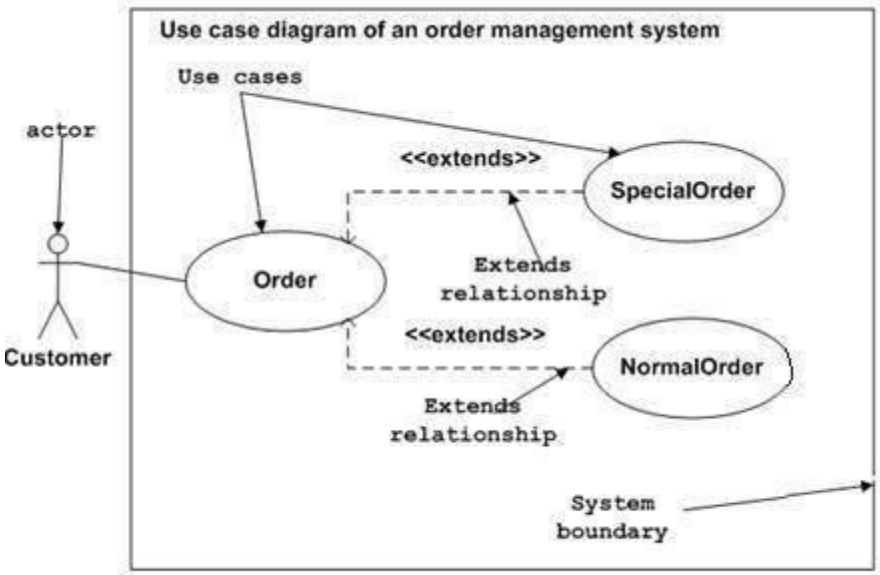


Figure: Sample Use Case diagram

Use Case Template

Use Case ID	

Use Case Name:	
Actor:	
Description:	
Preconditions:	
Postconditions:	
Priority:	
Frequency of Use:	
Normal Course of Events:	
Alternative Courses:	
Exceptions:	
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

Guidance for Use Case Template

Document each use case using the template shown in the above table. This section provides a description of each section in the use case template.

1. Actor

   An actor is a person or other entity external to the software system being specified who interacts with the system and performs use cases to accomplish tasks. Different actors often correspond to different user classes, or roles, identified from the customer community that will use the product. Name the actor(s) that will be performing this use case.

2. Description: Provide a brief description of the reason for and outcome of this use case, or a high-level description of the sequence of actions and the outcome of executing the use case.

3. Preconditions: List any activities that must take place, or any conditions that must be true, before the use case can be started. Number each precondition. Examples:

   a. User's identity has been authenticated. User's computer has sufficient free memory available to launch task.

4. Post conditions: Describe the state of the system at the conclusion of the use case execution. Number each postcondition. Examples: Document contains only valid SGML tags.Price of item in database has been updated with new value.
5. Priority : Indicate the relative priority of implementing the functionality required to allow this use case to be executed. The priority scheme used must be the same as that used in the software requirements specification.
6. Frequency of Use : Estimate the number of times this use case will be performed by the actors per some appropriate unit of time.
7. Normal Course of Events : Provide a detailed description of the user actions and system responses that will take place during execution of the use case under normal, expected conditions. This dialog sequence will ultimately lead to accomplishing the goal stated in the use case name and description. This description may be written as an answer to the hypothetical question, "How do I <accomplish the task stated in the use case name>?" This is best done as a numbered list of actions performed by the actor, alternating with responses provided by the system.
8. Alternative Courses: Document other, legitimate usage scenarios that can take place within this use case separately in this section. State the alternative course, and describe any differences in the sequence of steps that take place. Number each alternative course using the Use Case ID as a prefix, followed by "AC" to indicate "Alternative Course". Example: X.Y.AC.1.
9. Exceptions: Describe any anticipated error conditions that could occur during execution of the use case, and define how the system is to respond to those conditions. Also, describe how the system is to respond if the use case execution fails for some unanticipated reason. Number each exception using the Use Case ID as a prefix, followed by "EX" to indicate "Exception". Example: X.Y.EX.1.
10. Includes: List any other use cases that are included ("called") by this use case. Common functionality that appears in multiple use cases can be split out into a separate use case that is included by the ones that need that common functionality.
11. Special Requirements: Identify any additional requirements, such as nonfunctional requirements, for the use case that may need to be addressed during design or implementation. These may include performance requirements or other quality attributes.
12. Assumptions: List any assumptions that were made in the analysis that led to accepting this use case into the product description and writing the use case description.
13. Notes and Issues: List any additional comments about this use case or any remaining open issues or TBDs (To Be Determines) that must be resolved. Identify who will resolve each issue, the due date, and what the resolution ultimately is.

Example: Employee Management System

What kind of queries/ transactions will be required?

1. Add Employee to EMS (new Employee joins). Add Employee to Attendance register also.

2. Mark employee record when employee is released. (not removed)
3. Remove employee record from attendance register after employee leaves the company.
4. Employee wants to record attendance once a day.
5. Accounts department computes salary for each employee.
6. Accounts department makes a salary payment to each employee and transfers salary to account using bank (BMT) gateway.
7. Accounts department generates a total salary payment report.
8. Mark attendance in the register.

Use cases

1. Enroll new employee
2. Relieve Employee
3. Mark attendance
4. Distribute Salary

Assignment 3: Prepare Activity Model

Description : Identify Activity states and Action states. Draw Activity diagram with Swim lanes using UML2.0 Notations for major Use Cases

Aim **:** To Prepare the Activity Model

Objective **:** To learn behavioral modeling and creation of activity.

Theory **:**

01] About Activity Diagram

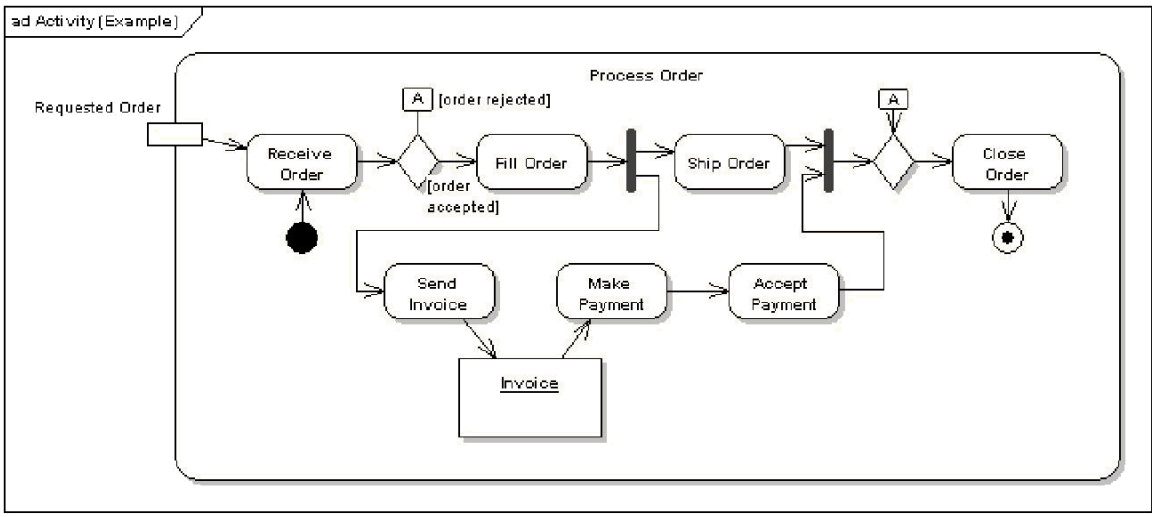
a) Contents

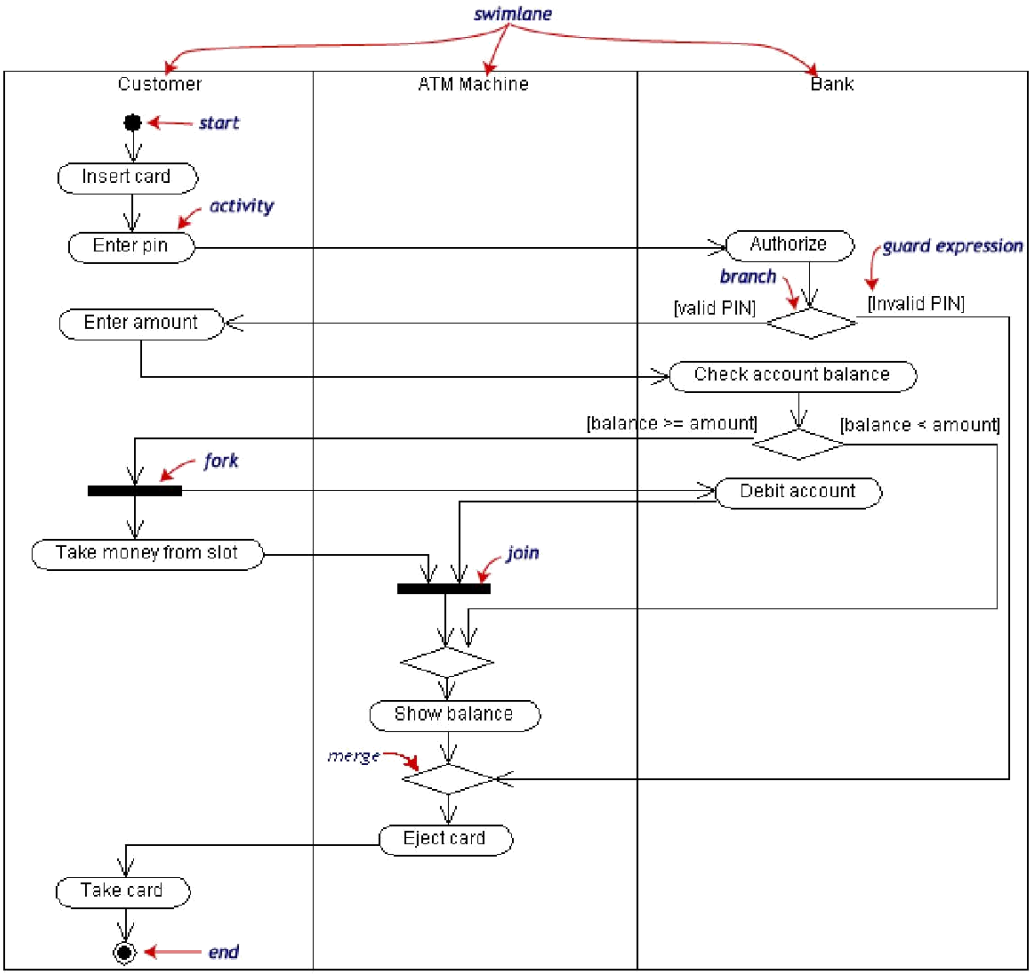
b) Common Uses.

Input: Refer UML User Guide – Rambaugh for Object Oriented Analysis and Design (OOAD)

Output: Activity Diagram.

In UML an activity diagram is used to display the sequence of activities. Activity Diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity. They may be used to detail situations where parallel processing may occur in the execution of some activities. Activity Diagrams are useful for Business Modeling where they are used for detailing the processes involved in business activities.


ad Activity (Example)
Process Order
Requested Order
A
[order rejected]
Receive Order
[order accepted]
Fill Order
Ship Order
A
Close Order
Send Invoice
Make Payment
Accept Payment
Invoice


swimlane
Customer
ATM Machine
Bank
start
Insert card
activity
Enter pin
Authorize
guard expression
branch
[valid PIN]
[Invalid PIN]
Enter amount
Check account balance
[balance >= amount]
[balance < amount]
fork
Debit account
Take money from slot
join
Show balance
merge
Eject card
Take card
end

Assignment 4: Prepare Analysis Model-Class Model

Aim **:** To Prepare the Class and Implement it.

Objective **:** To find classes, relationship, to prepare the class Diagram.

Theory **:**

01] About Classes:

a) Symbol

b) Syntax for Attribute and Operations.(Scope, Visibility, Property Strings)

02] Relationships with example.

03] About Class Diagram

a) Contents

b) Common Uses.

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.

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.

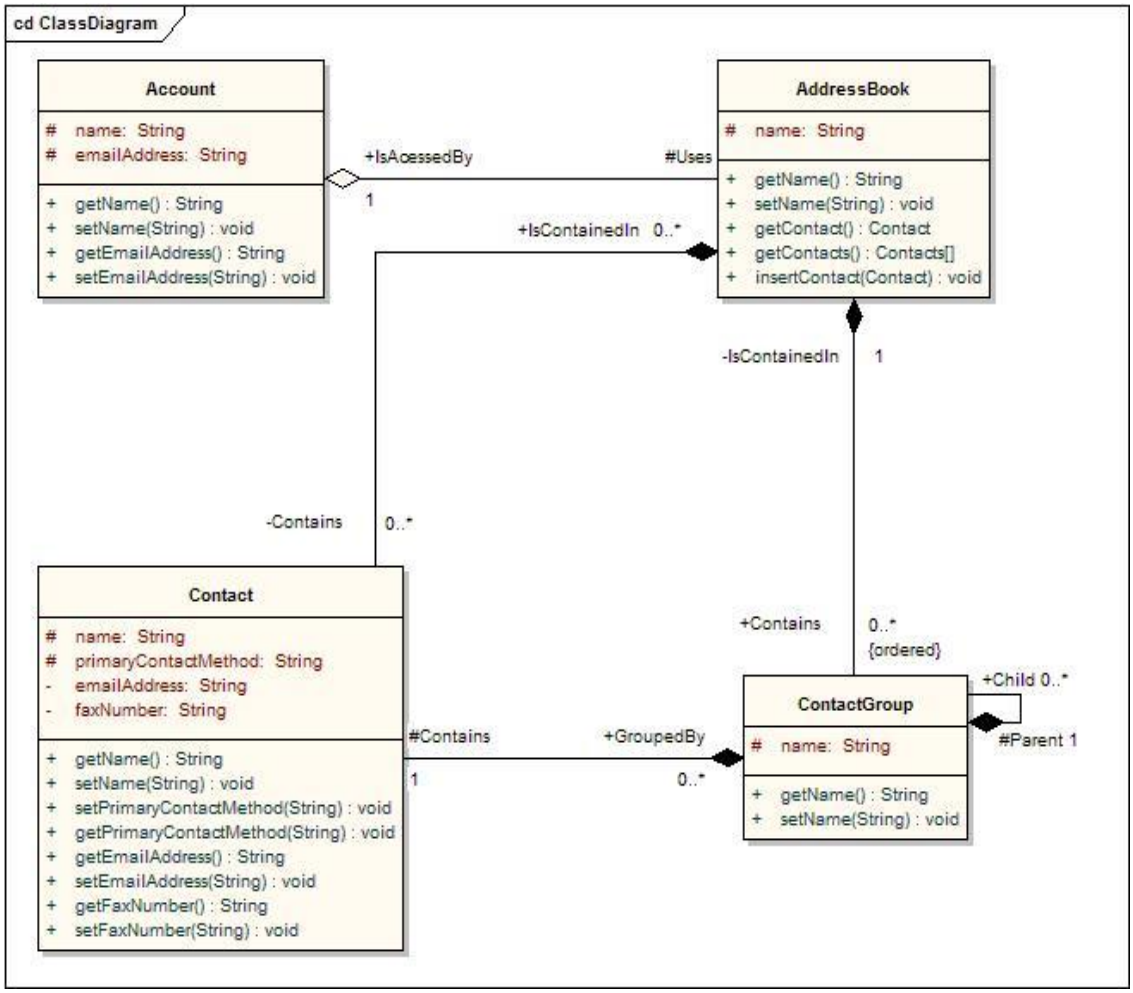
Classes

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.

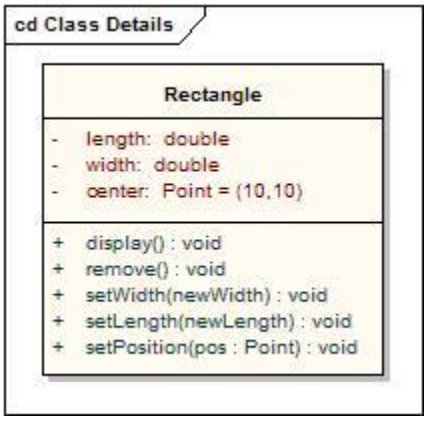
Class Notation

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.



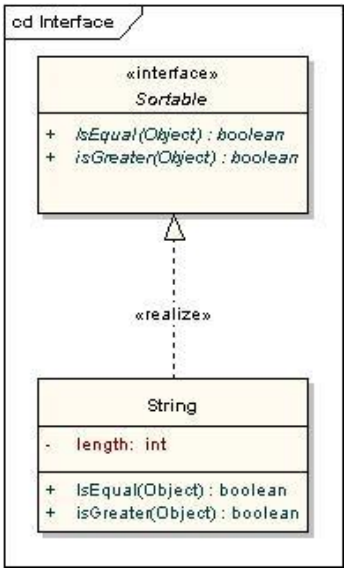
In the diagram the class contains the class name in the topmost compartment, the next compartment details the attributes, with the "center" attribute showing initial values. The final compartment shows the operations, the setWidth, setLength and setPosition operations

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 visibility.

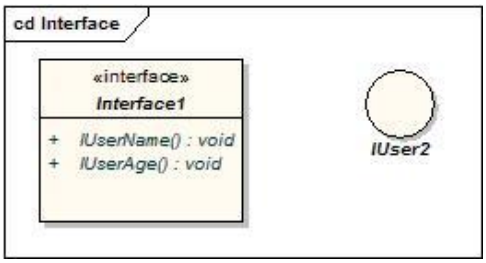


Interfaces

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.

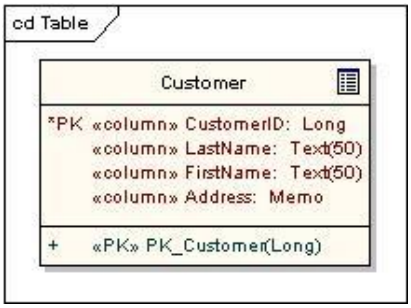


Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.



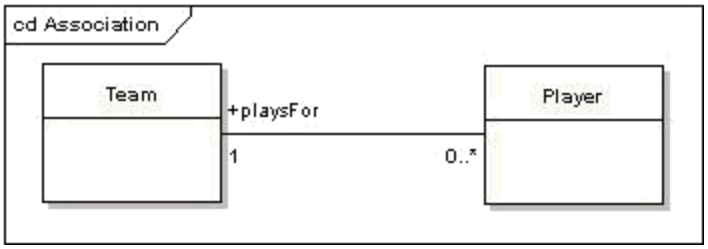
Tables

A table is a stereotyped class. It is drawn with a small table icon in the upper right corner. Table attributes are stereotyped «column». Most tables will have a primary key, being one or more fields that form a unique combination used to access the table, plus a primary key operation which is stereotyped «PK». Some tables will have one or more foreign keys, being one or more fields that together map onto a primary key in a related table, plus a foreign key operation which is stereotyped «FK».



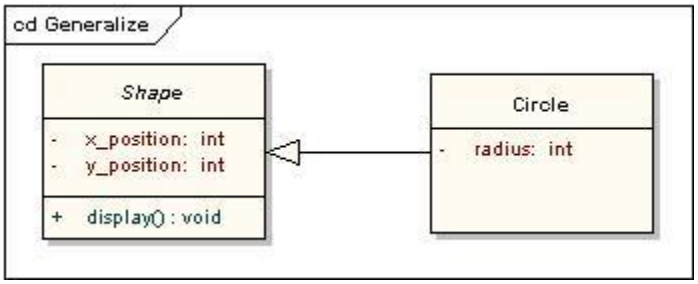
Associations

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, cardinality, direction and constraints. Association is the general relationship type between elements. For more than two elements, a diagonal representation toolbox element can be used as well. When code is generated for class diagrams, associations become instance variables in the target class.

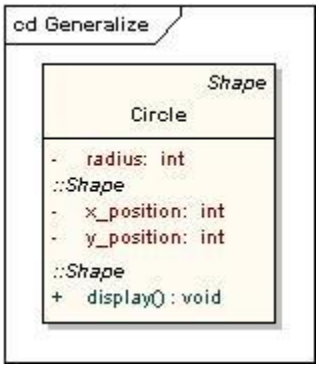


Generalizations

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(). Note that the class Shape is abstract, shown by the name being italicized.



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

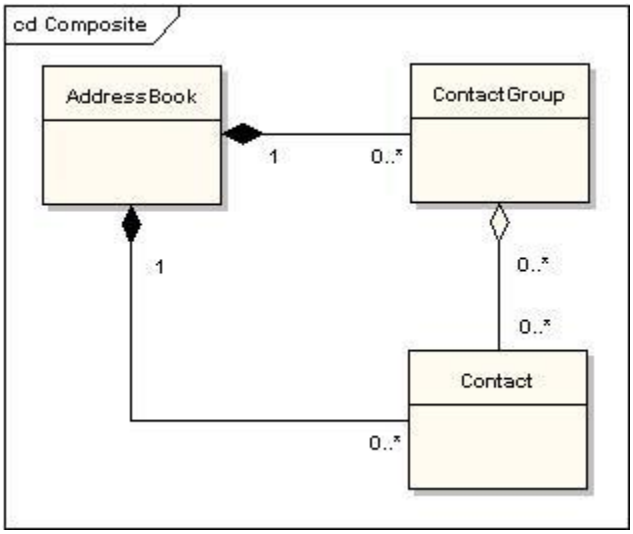


Aggregations

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.

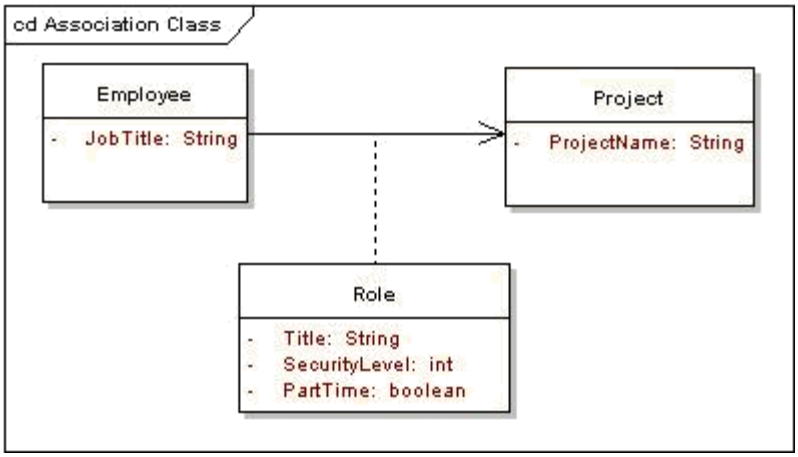
A stronger form of aggregation - a composite aggregation - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time. If the parent of a composite aggregation is deleted, usually all of its parts are deleted with it; however a 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

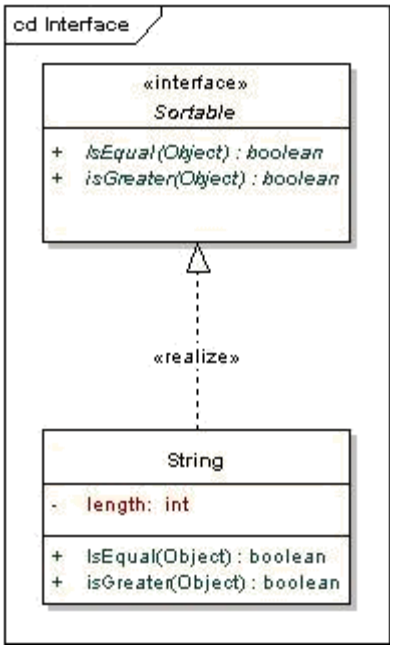
A dependency is used to model a wide range of dependent relationships between model elements. It would normally be used early in the design process where it is known that there is some kind of link between two elements but it is too early to know exactly what the relationship is. Later in the design 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.

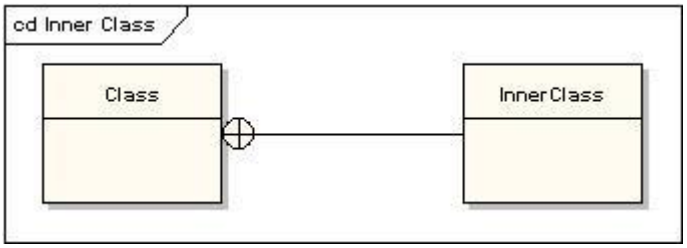
Realizations

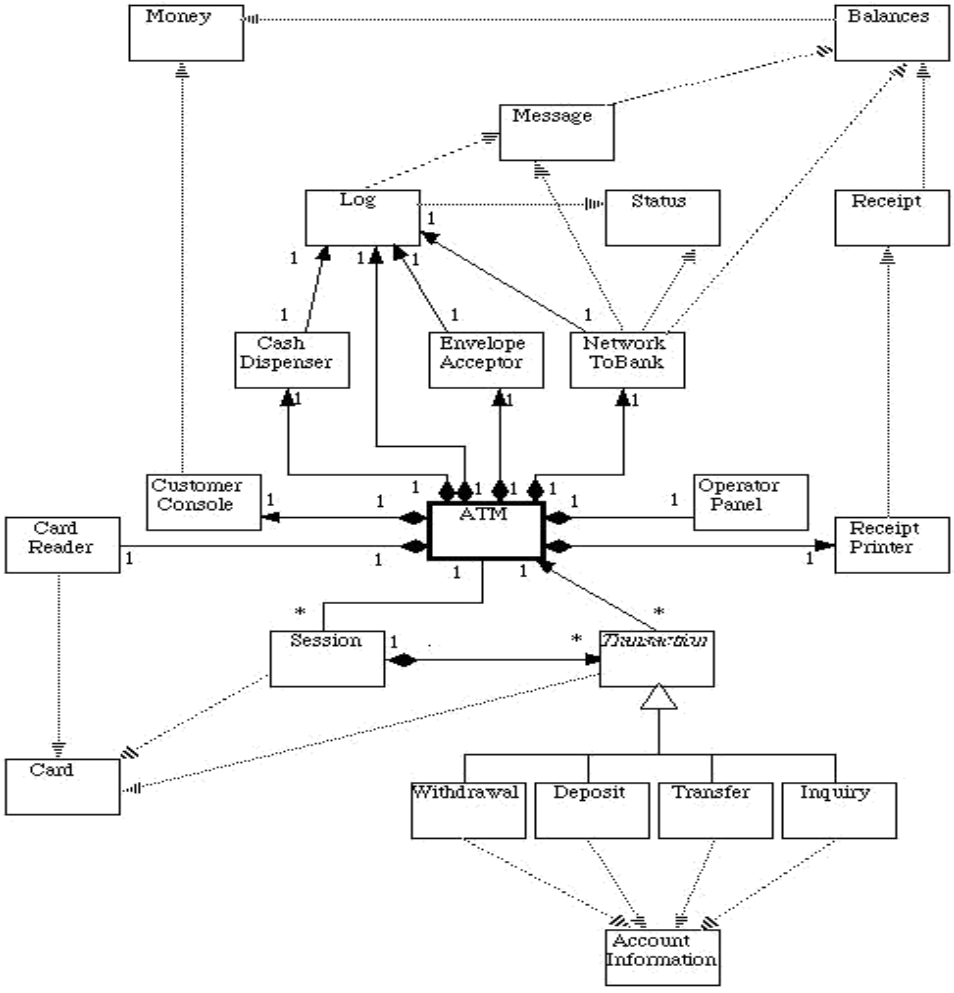
The source object implements or realizes the destination. Realize is used to express traceability and completeness in the model - a business process or requirement is realized by one or more use cases which are in turn realized by some classes, which in turn are realized by a component, etc. Mapping requirements, classes, etc. across the design of your system, up through the levels of modelling 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.



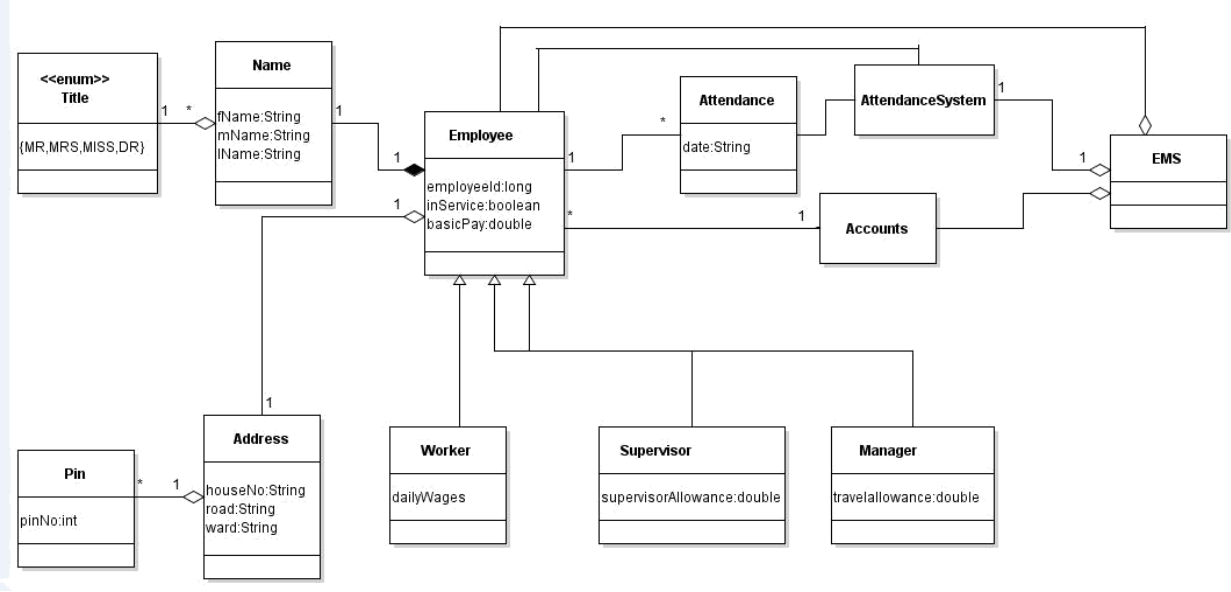
Nestings

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.





Class Diagram for ATM System



Class Diagram for Employee Management System

Input: Refer UML User Guide – Rambaugh for Object Oriented Analysis and Design (OOAD) Output: Class Diagram.

Assignment 5: Prepare a Design Model from Analysis Model

Description: Identify Analysis Classes and assign responsibilities, Prepare Data Dictionary.

Problem Statement: 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 employeeId, employee name, employee bank account, and the amount deposited in that account.

Part – II

What kind of queries/ transactions will be required?

Add Employee to EMS (new Employee joins). Add Employee to Attendance register also.

Mark employee record when employee is released. (not removed)

Remove employee record from attendance register after employee leaves the company. Employee wants to record attendance once a day.

Accounts department computes salary for each employee.

Accounts department makes a salary payment to each employee and transfers salary to account using bank (BMT) gateway.

Accounts department generates a total salary payment report.
Mark attendance in the register.

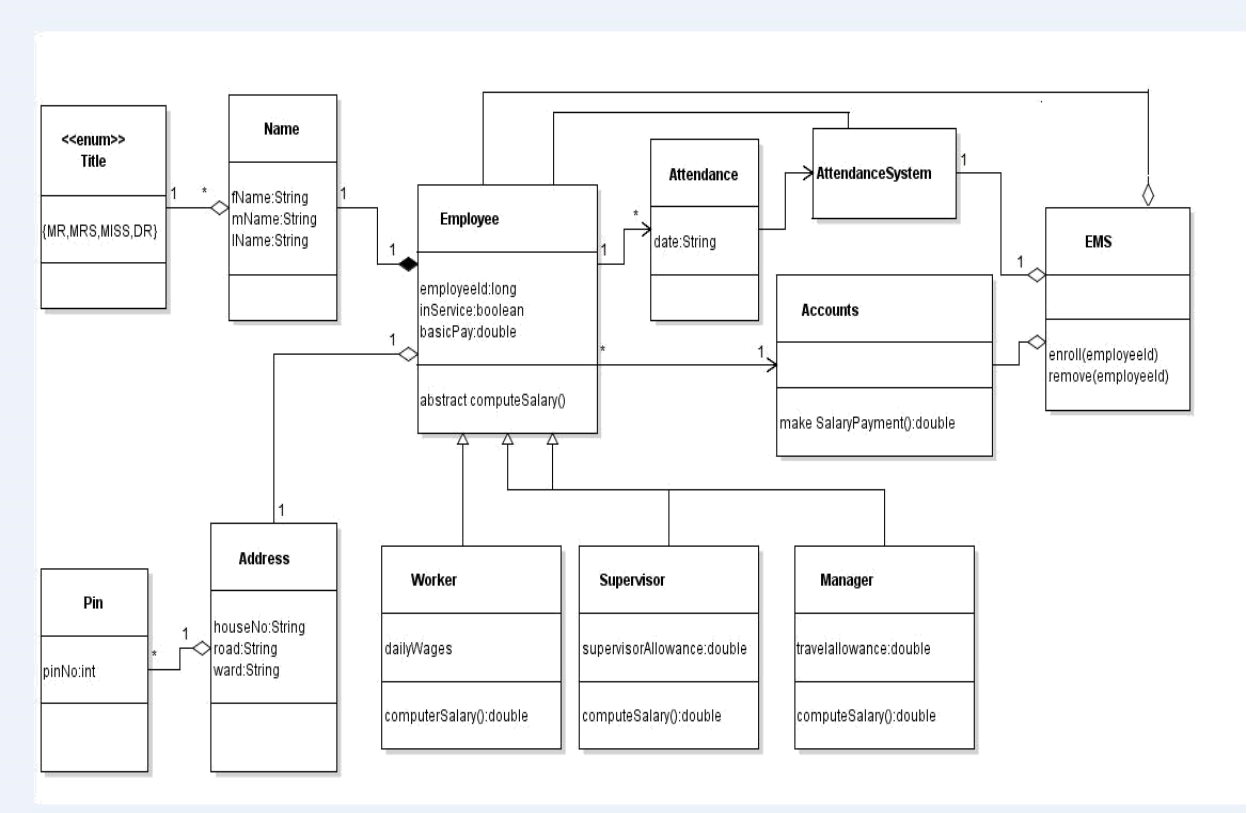
Part - III

For yearly performance appraisal Worker reports to supervisor, supervisor reports to manager. Manager reports another manager or none.

A performance assessment cycle starts when HR department starts the appraisal cycle in the Appraisal system. Once started Appraisal system is available to all employees for making self-assessment. As a self-apraisal, each employee has to mark a self-rating (1-5) in the system based on her self-assessment. After self-assessment the employee-appraisal is available to reporting officer (Appraiser) to mark appraiser's rating (1-5). After appraiser's rating is complete, employee-appraisal is available to the HR system for computing the Pay revision and Promotions. For simplicity the rule for pay revision is 10% increment in Basic if the appraiser's rating is 3 to 5. Promotion rule is if the Appriser's rating is 5 then worker becomes supervisor and supervisor becomes Manager. Managers simply get 10% additional basic pay in place of promotion..

Use cases

Enroll new employee
Relieve Employee
Mark attendance
Distribute Salary



Class diagram for Employee management System (Design Model)

Assignment 6: Prepare Sequence Model.

Description: 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

Aim **:** To Prepare the Sequence and Collaboration Diagram.

Objective **:** To learn behavioral modeling and creation of sequence diagram.

Theory **:**

01] About Sequence and Collaboration Diagram

a) Contents

b) Common Uses.

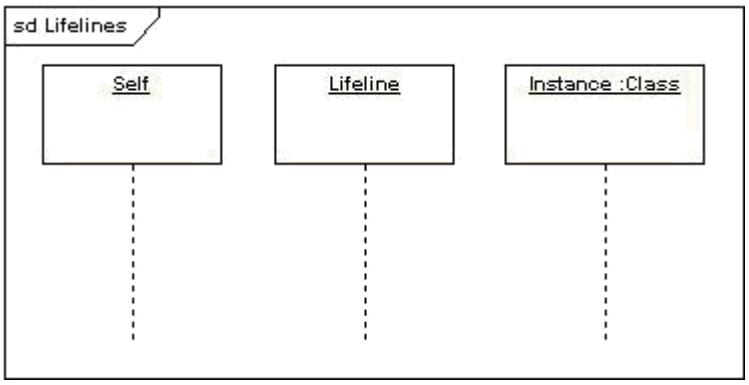
Input: Refer UML User Guide – Rambaugh for Object Oriented Analysis and Design (OOAD)

Output: Sequence and Collaboration Diagram.

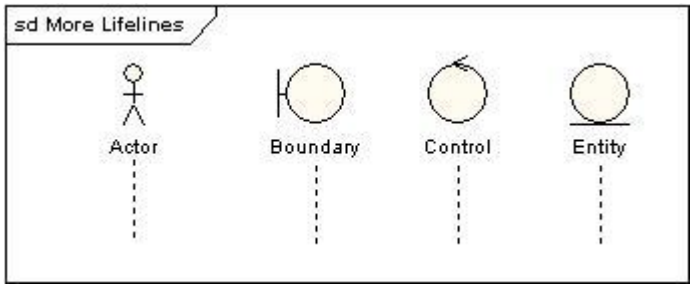
A sequence diagram is a form of interaction diagram which shows objects as lifelines running down the page and with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good at showing which objects communicate with which other objects and what messages trigger those communications. Sequence diagrams are not intended for showing complex procedural logic.

Lifelines

A lifeline represents an individual participant in a sequence diagram. A lifeline will usually have a rectangle containing its object name. If its name is self then that indicates that the lifeline represents the classifier which owns the sequence diagram..

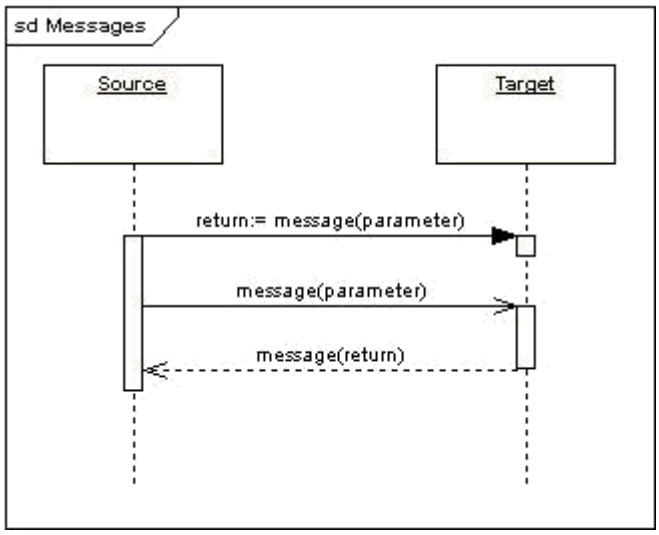


Sometimes a sequence diagram will have a lifeline with an actor element symbol at its head. This will usually be the case if the sequence diagram is owned by a use case. Boundary, control and entity elements from robustness diagrams can also own lifelines.



Messages

Messages are displayed as arrows. Messages can be complete, lost or found; synchronous or asynchronous; call or signal. In the following diagram, the first message is a synchronous message (denoted by the solid arrowhead) complete with an implicit return message; the second message is asynchronous (denoted by line arrowhead) and the third is the asynchronous return message (denoted by the dashed line).

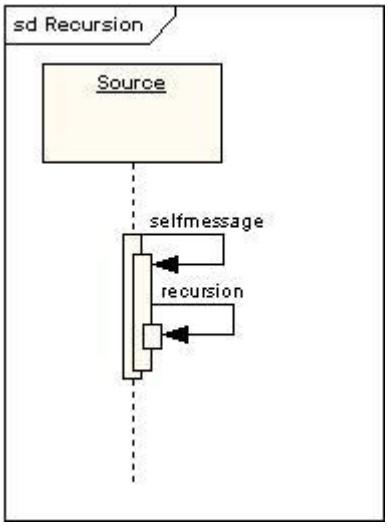


ExecutionOccurrence

A thin rectangle running down the lifeline denotes the execution occurrence or activation of a focus of control. In the previous diagram, there are three execution occurrences. The first is the source object sending two messages and receiving two replies; the second is the target object receiving a synchronous message and returning a reply; and the third is the target object receiving an asynchronous message and returning a reply.

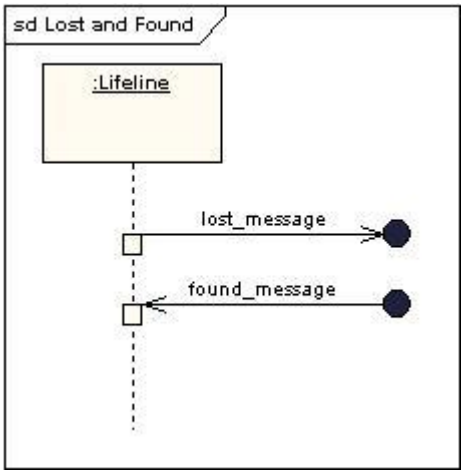
SelfMessage

A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object. It is shown as creating a nested focus of control in the lifeline's execution occurrence.



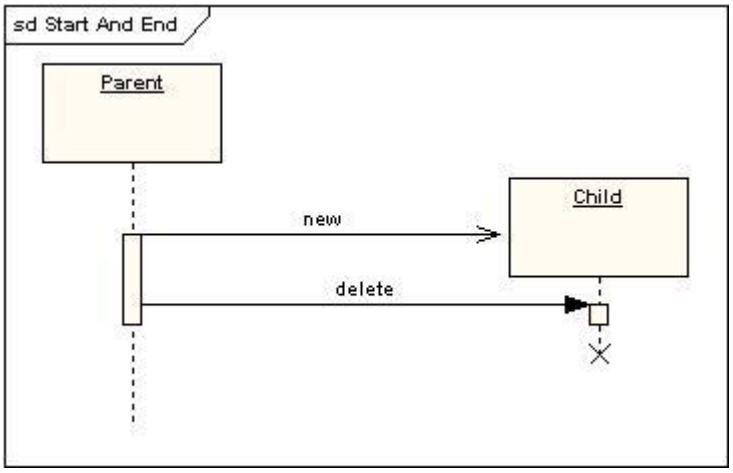
Lost and Found Messages

Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram. Found messages are those that arrive from an unknown sender, or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.



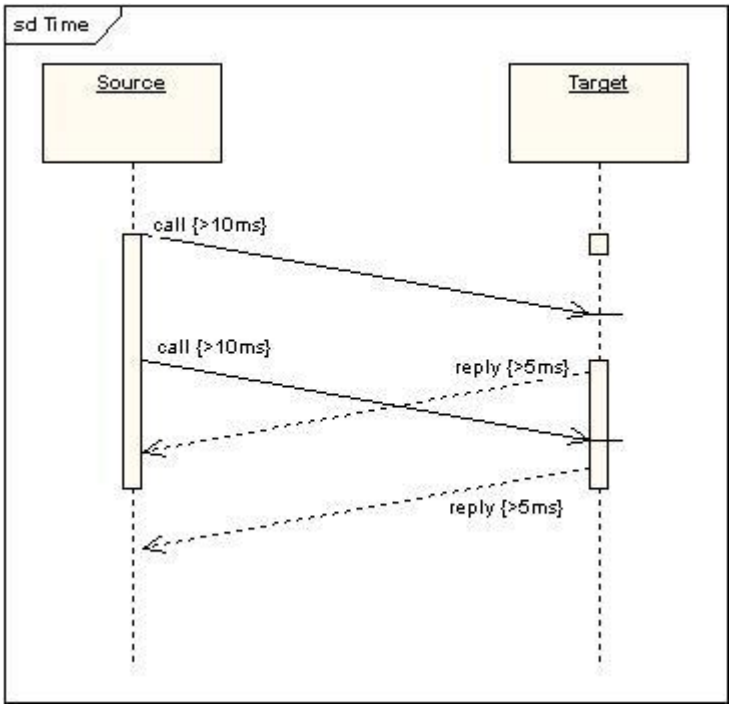
Lifeline Start and End

A lifeline may be created or destroyed during the timescale represented by a sequence diagram. In the latter case, the lifeline is terminated by a stop symbol, represented as a cross. In the former case, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation. The following diagram shows an object being created and destroyed.



Duration and Time Constraints

By default, a message is shown as a horizontal line. Since the lifeline represents the passage of time down the screen, when modeling a real-time system, or even a time-bound business process, it can be important to consider the length of time it takes to perform actions. By setting a duration constraint for a message, the message will be shown as a sloping line.



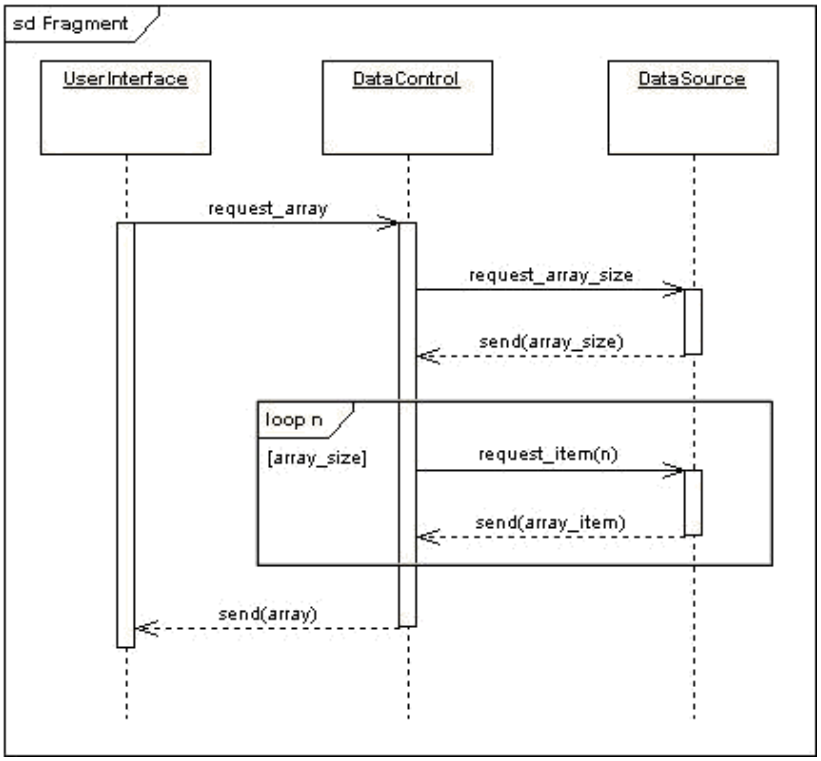
Combined Fragments

It was stated earlier that Sequence diagrams are not intended for showing complex procedural logic. While this is the case, there are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments. A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances. The fragments available are:

- Alternative fragment (denoted “alt”) models if…then…else constructs.
- Option fragment (denoted “opt”) models switch constructs.
- Break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.

- Parallel fragment (denoted “par”) models concurrent processing.
- Weak sequencing fragment (denoted “seq”) encloses a number of sequences for which all the messages must be processed in a preceding segment before the following segment can start, but which does not impose any sequencing within a segment on messages that don’t share a lifeline.
- Strict sequencing fragment (denoted “strict”) encloses a series of messages which must be processed in the given order.
- Negative fragment (denoted “neg”) encloses an invalid series of messages.
- Critical fragment encloses a critical section.
- Ignore fragment declares a message or message to be of no interest if it appears in the current context.
- Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.
- Assertion fragment (denoted “assert”) designates that any sequence not shown as an operand of the assertion is invalid.
- Loop fragment encloses a series of messages which are repeated.

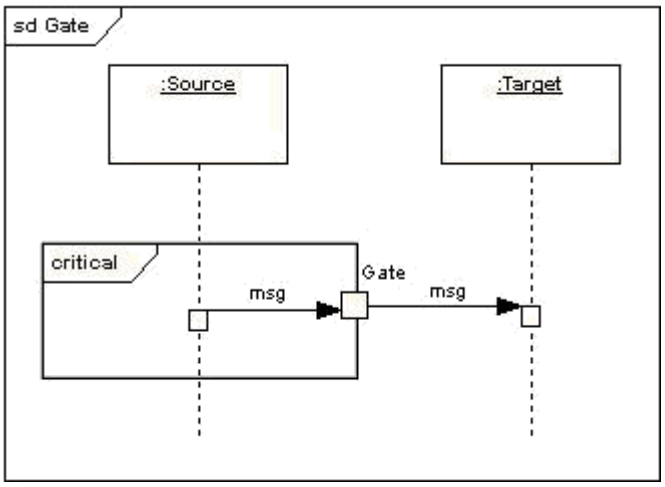
The following diagram shows a loop fragment.



There is also an interaction occurrence, which is similar to a combined fragment. An interaction occurrence is a reference to another diagram which has the word "ref" in the top left corner of the frame, and has the name of the referenced diagram shown in the middle of the frame.

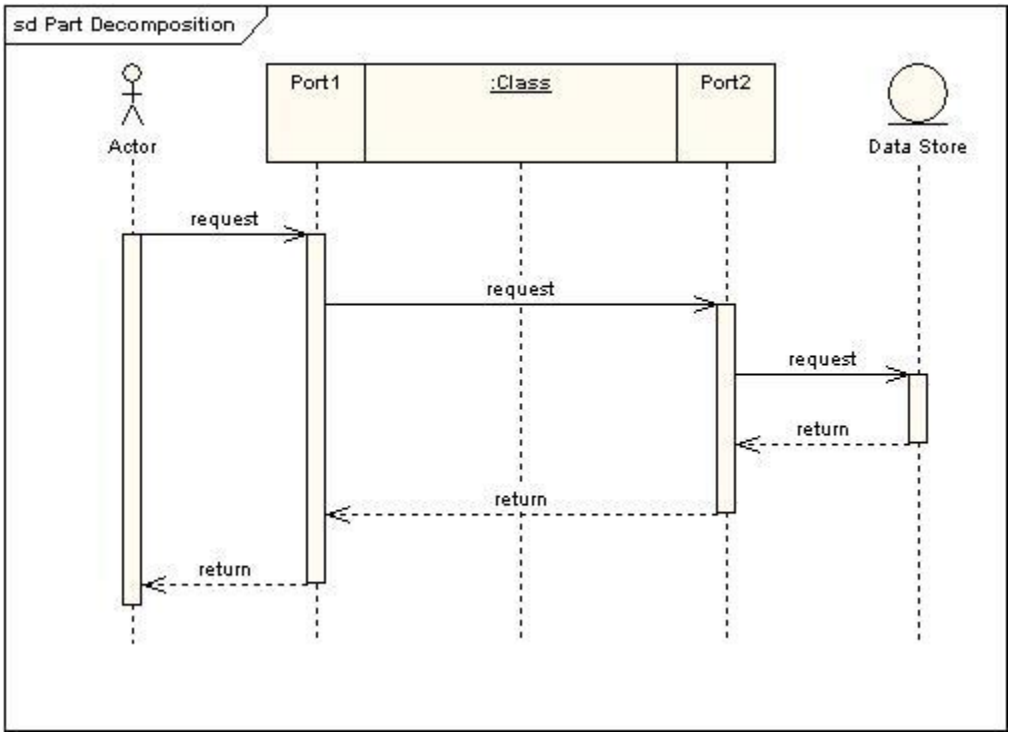
Gate

A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame.



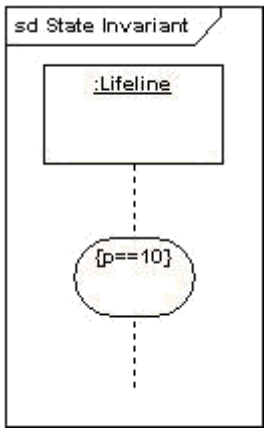
Part Decomposition

An object can have more than one lifeline coming from it. This allows for inter- and intra-object messages to be displayed on the same diagram.



State Invariant / Continuations

A state invariant is a constraint placed on a lifeline that must be true at run-time. It is shown as a rectangle with semi-circular ends.



A Continuation has the same notation as a state invariant but is used in combined fragments and can stretch across more than one lifeline.

Assignment 7: Prepare a State Model

Description : 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

Aim **:** To Prepare the State Diagram.

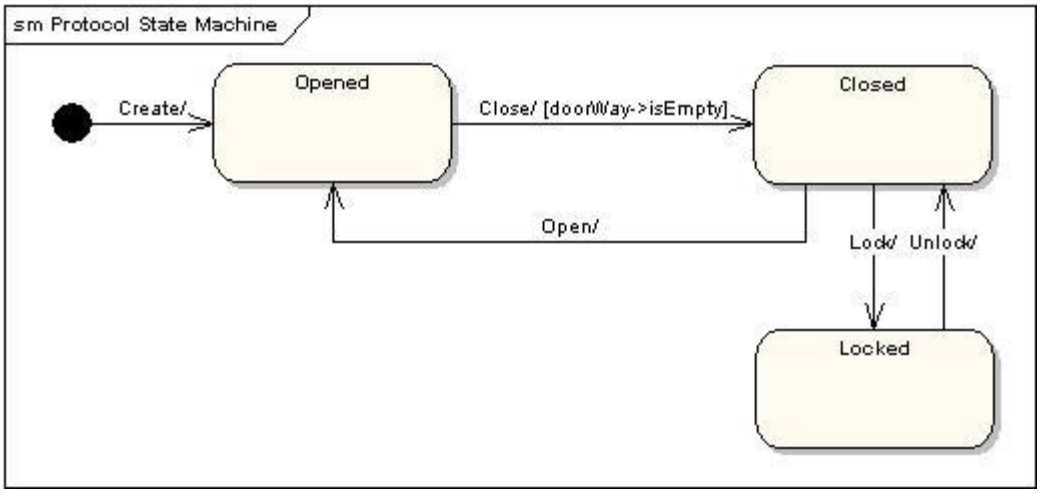
Objective **:** To learn behavioral modeling and creation of state diagram.

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.

As an example, the following state machine diagram shows the states that a door goes through during its lifetime.

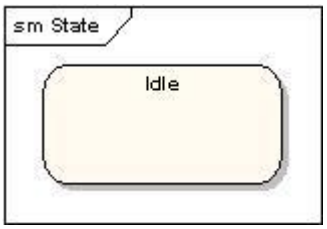


The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock. Notice that not all events are valid in all states; for example, if a door is opened, you cannot lock it until you close it. Also notice that a state transition can have 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.

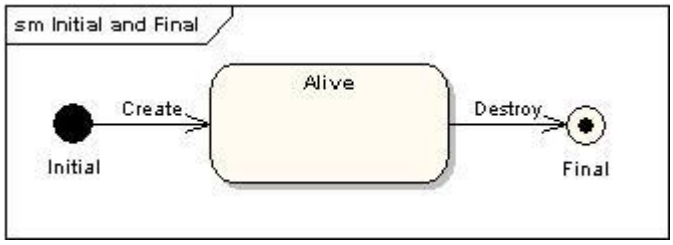
States

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



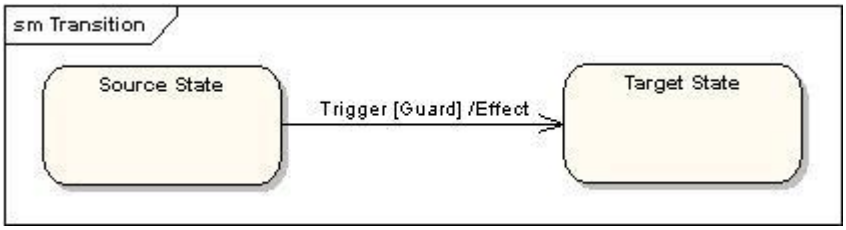
Initial and Final States

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.

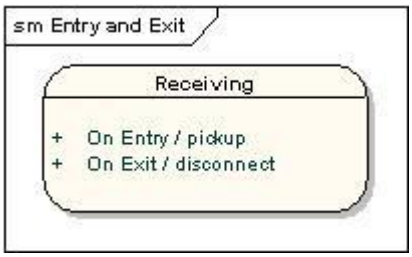


"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the 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,

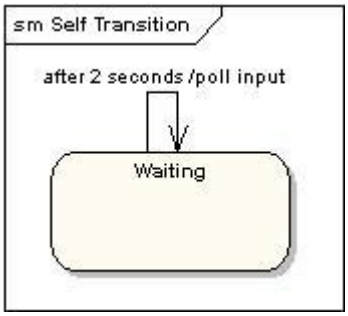
it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.



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.

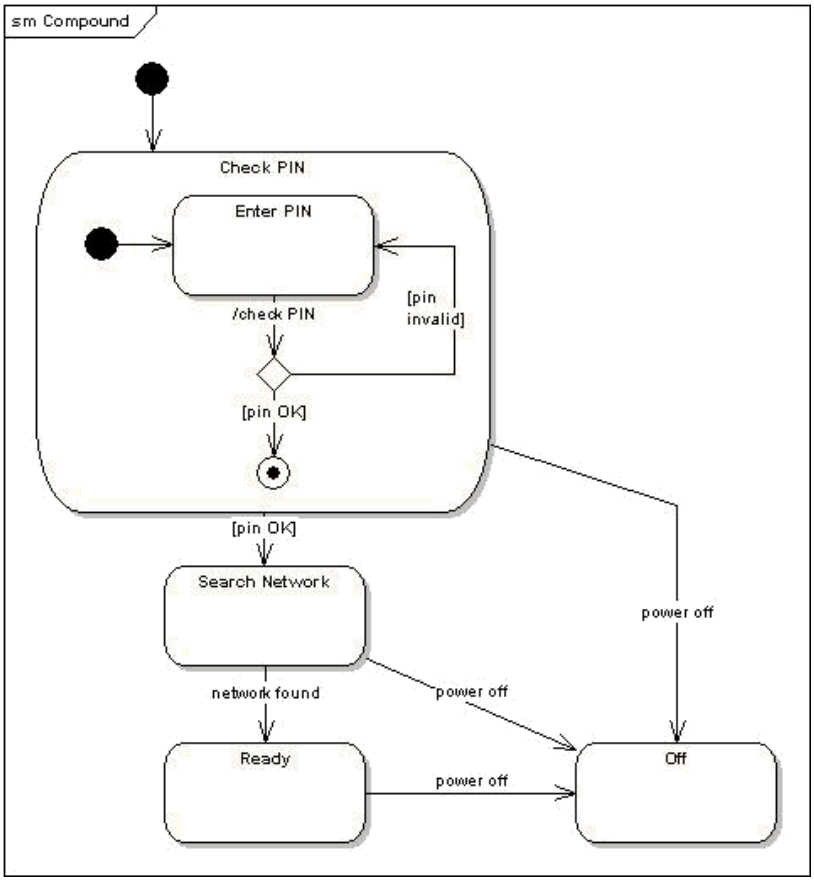
Self-Transitions

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.

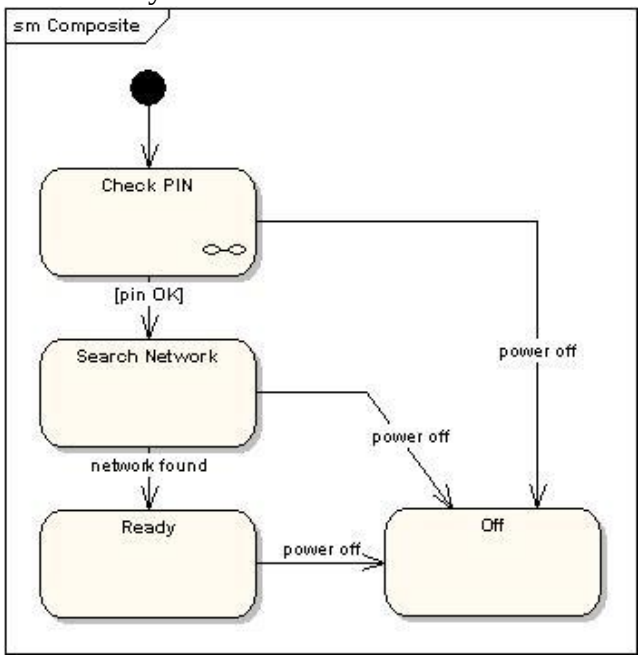


Compound States

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



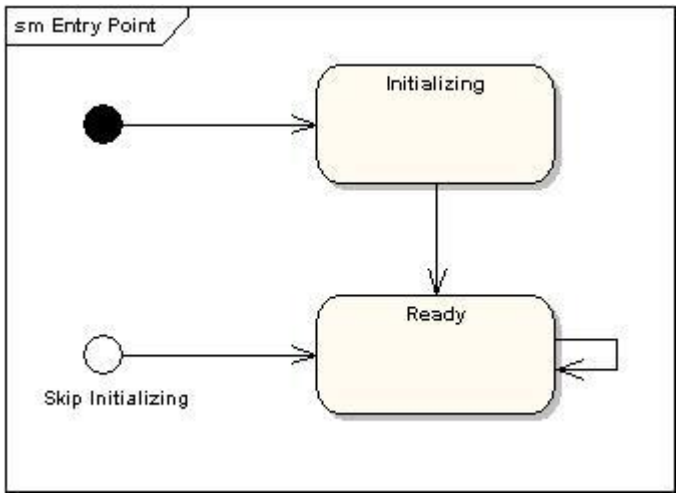
The alternative way to show the same information is as follows.



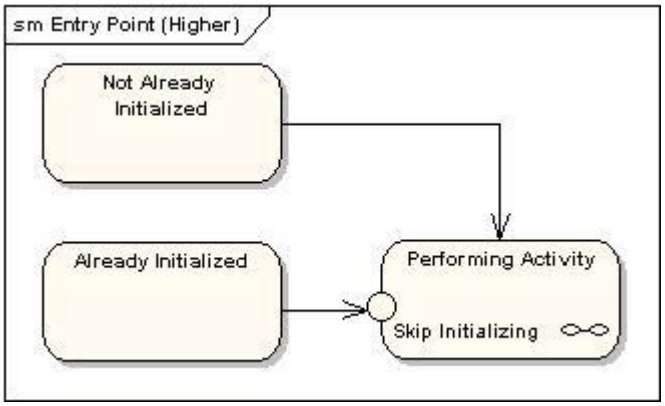
The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

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.

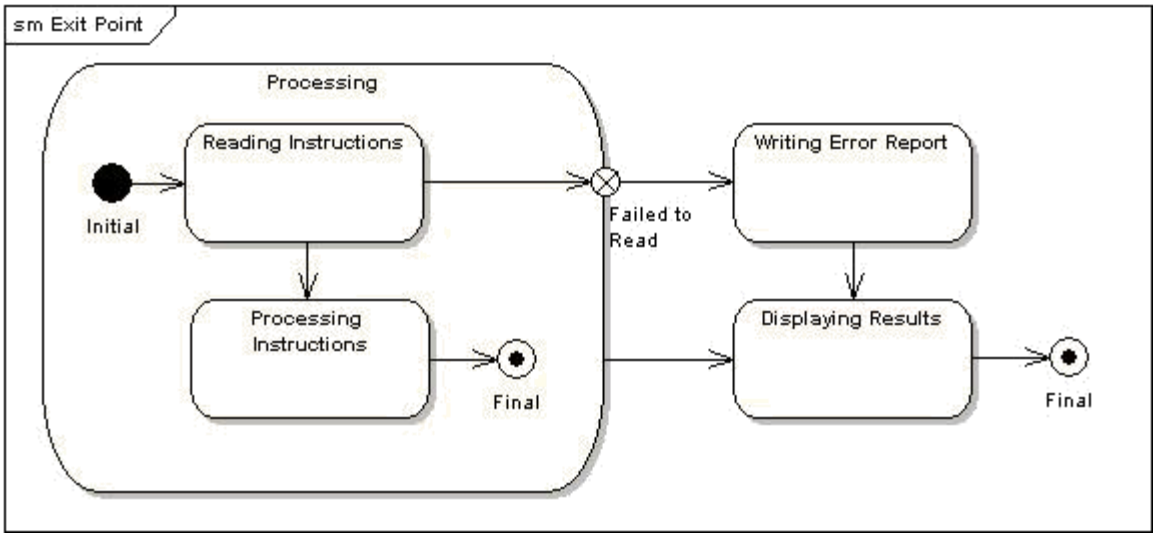


The following diagram shows the state machine one level up.



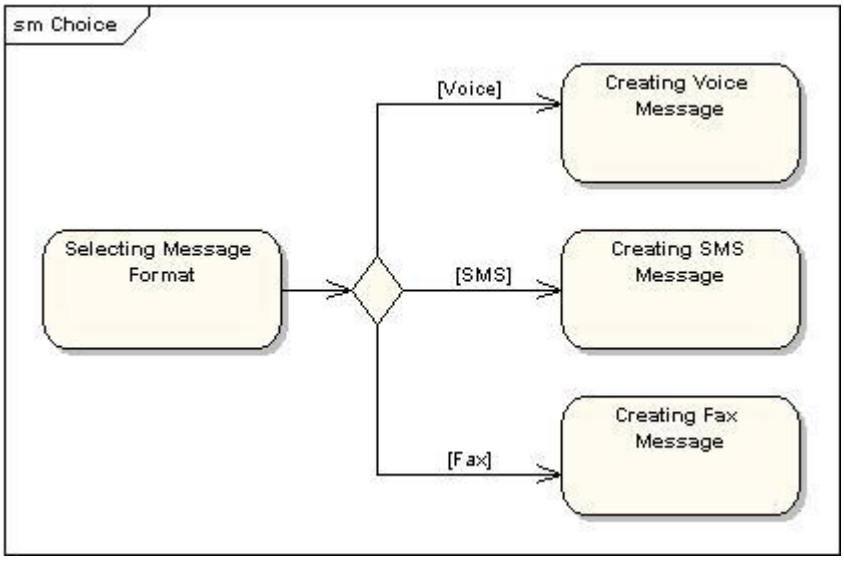
Exit Point

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.



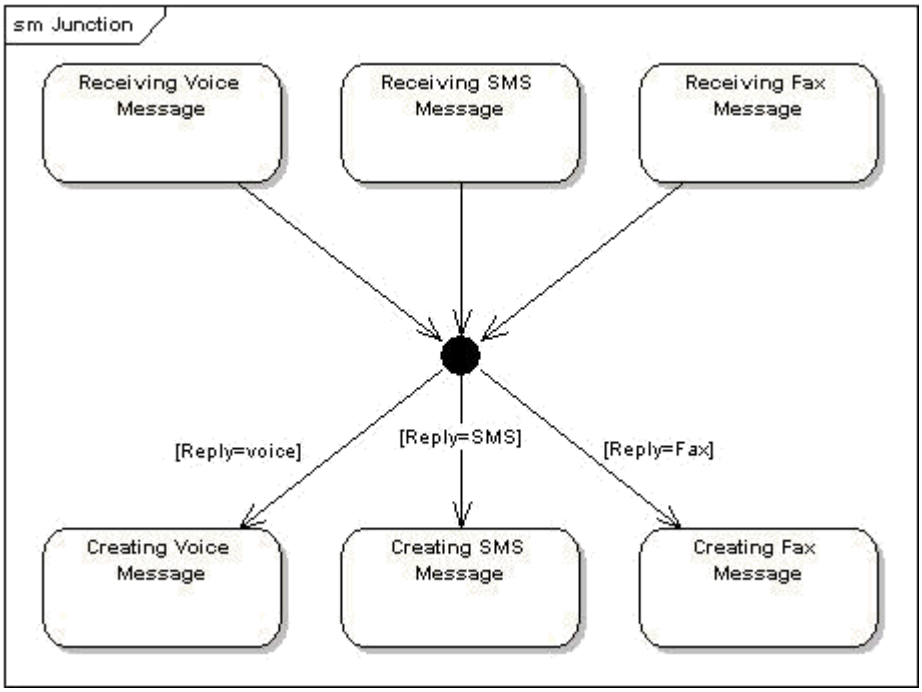
Choice Pseudo-State

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.



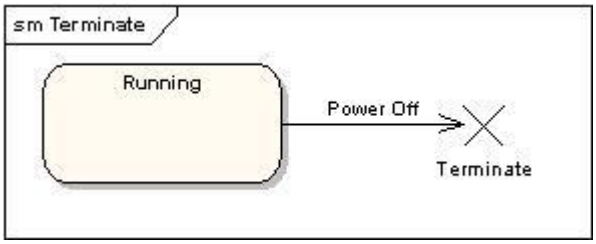
Junction Pseudo-State

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.



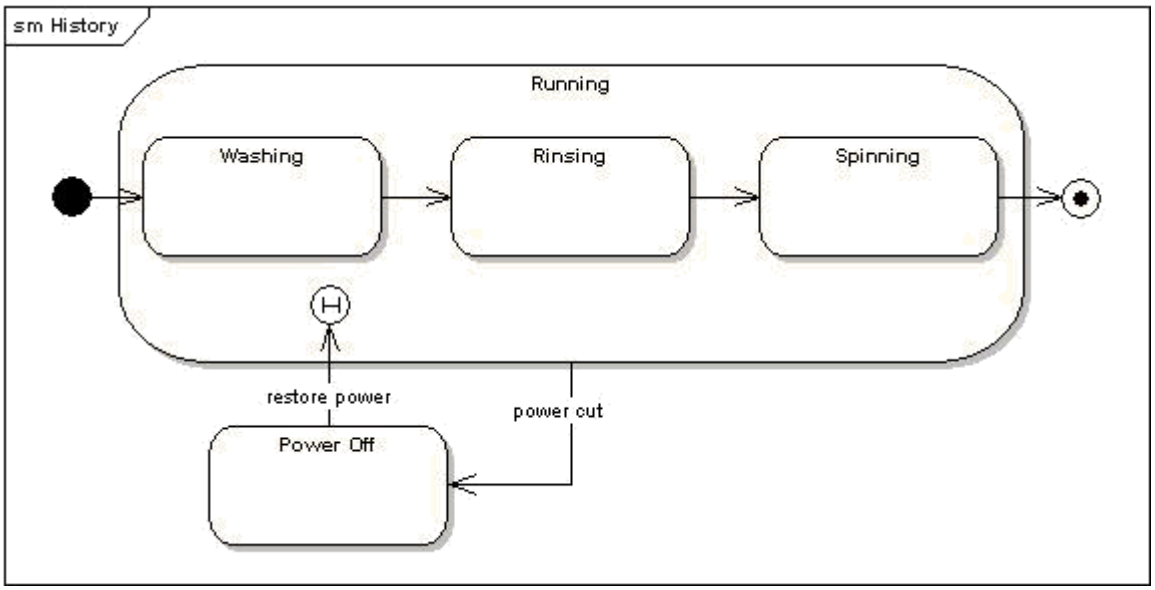
Terminate Pseudo-State

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

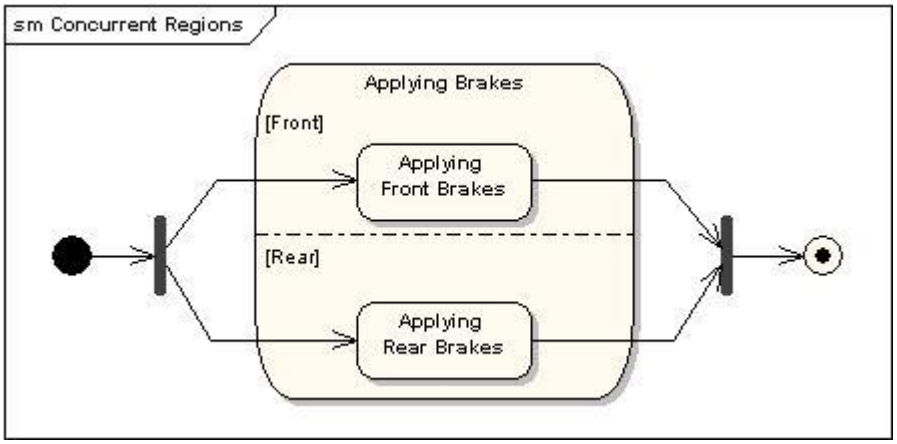
A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.

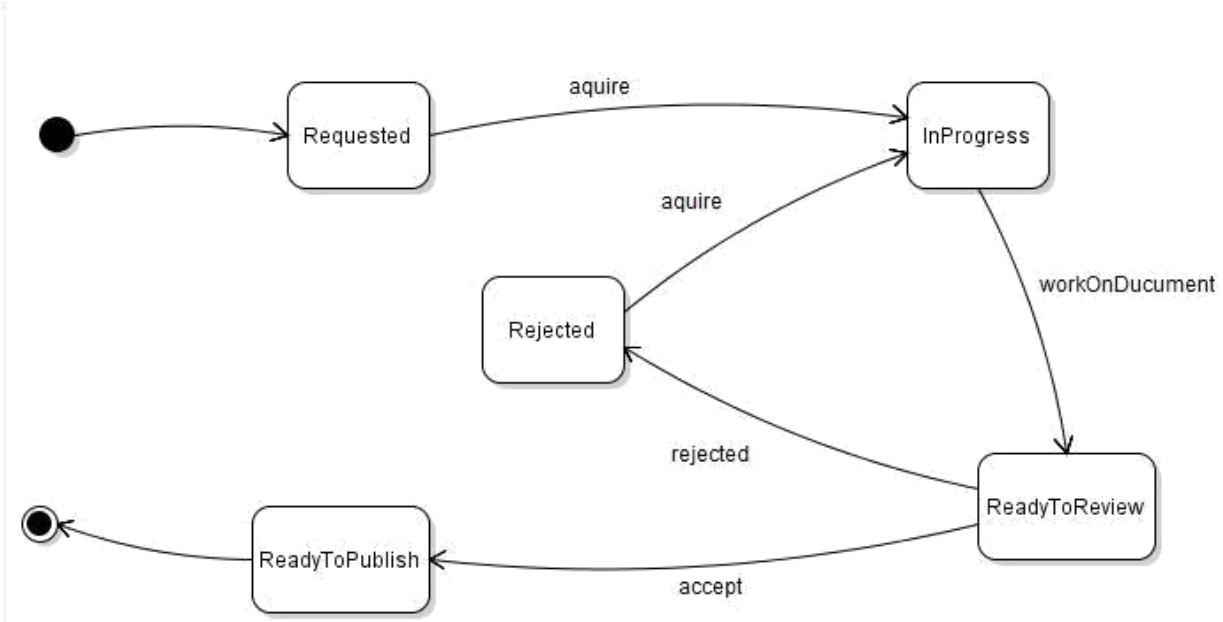


In this state machine, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning". If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History 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.




Requested
aquire
InProgress
aquire
Rejected
workOnDucument
rejected
ReadyToReview
ReadyToPublish
accept

Assignment 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

Theory :

General responsibility assignment software patterns (or **principles**), abbreviated **GRASP**, consist of guidelines for assigning responsibility to classes and objects in object-oriented design. The different patterns and principles used in GRASP are controller, creator, indirection, information expert, high cohesion, low coupling, polymorphism, protected variations, and pure fabrication. All these patterns answer some software problem, and these problems are common to almost every software development project. These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

Responsibilities are more general than methods

- Methods are implemented to fulfill responsibilities
- Example: Sale class may have a method to know its total but may require interaction with other objects

Computer scientist Craig Larman states that "the critical design tool for software development is a mind well educated in design principles. It is not UML or any other technology." Thus, GRASP are really a mental toolset, a learning aid to help in the design of object-oriented software.

In OO design, a pattern is a named description of a problem and solution that can be applied to new contexts; ideally, a pattern advises us on how to apply its solution in varying circumstances and considers the forces and trade-offs. Many patterns, given a specific category of problem, guide the assignment of responsibilities to objects.

GRASP principles – a learning aid for OO design with responsibilities

Pattern – a *named* and *well-known* problem/solution pair that can be applied in new contexts, with advice on how to apply it in new situations and discussion of its trade-offs, implementations, variations, etc

9 GRASP principles:

1. Information Expert
2. Creator
3. Low Coupling
4. Controller
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

1. Controller

The **controller** pattern assigns the responsibility of dealing with system events to a non- UI class that represents the overall system or a use case scenario. A controller object is a non-user interface object responsible for receiving or handling a system event.

A use case controller should be used to deal with *all* system events of a use case, and may be used for more than one use case (for instance, for use cases *Create User* and *Delete User*, one can have a single *UserController*, instead of two separate use case controllers).

It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate the work that needs to be done to other objects; it coordinates or controls the activity. It should not do much work itself. The GRASP Controller can be thought of as being a part of the application/service layer [2] (assuming that the application has made an explicit distinction between the application/service layer and

the domain layer) in an object-oriented system with common layers in an information system logical architecture.

Problem: Who should be responsible for handling a system event? (Or, what object receives and coordinates a system operation?)

Solution: Assign the responsibility for receiving and/or handling a system event to one of following choices:

- Object that represents overall system, device or subsystem (*façade controller*)
- Object that represents a use case scenario within which the system event occurs (a <UseCase>Handler)

– Input system event – event generated by an external actor associated with a system operation

– Controller – a non-UI object responsible for receiving or handling a system event

– During analysis can assign system operations to a class System

– That doesn't mean there will be a System class at time of design

– During design a controller class is given responsibility for system operations

– Controller is a *façade* into domain layer from interface layer

– Often use same controller class for all system events of one use case so that one can maintain state information, e.g. events must occur in a certain order

– Normally controller coordinates activity but delegates work to other objects rather than doing work itself

2 . Creator

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

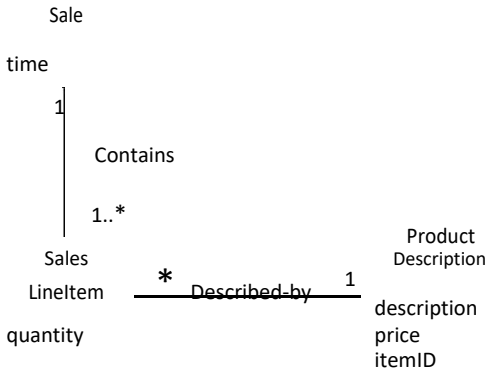
Problem: Who should be responsible for creating a new instance of a class?

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

- 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.

Example: (from POS system) Who should be responsible for creating a new SalesLineItem instance?

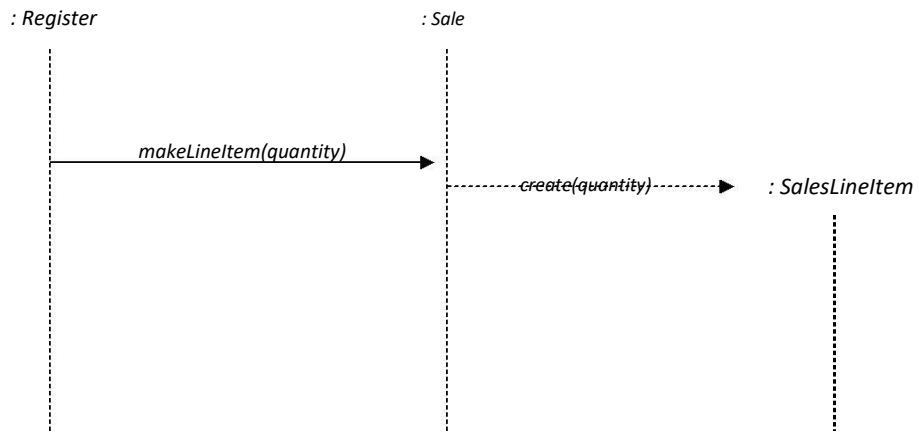
Start with domain model:



Example: (from POS system) Who should be responsible for creating a new SalesLineItem instance?

Since a Sale contains SalesLineItem objects it should be responsible according to the Creator pattern .

Note: there is a related design pattern called Factory for more complex creation situations



3) High cohesion

High cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and averse to change

Problem: How to keep complexity manageable?

Solution: Assign the responsibility so that cohesion remains high.

Cohesion – a measure of how strongly related and focused the responsibilities of an element (class, subsystem, etc.) are

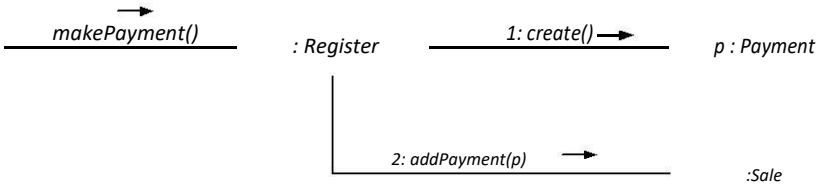
Problems from low cohesion (does many unrelated things or does too much work):

- Hard to understand/comprehend
- Hard to reuse

– Hard to maintain
– Brittle – easily affected by change

Example: (from POS system) Who should be responsible for creating a Payment instance and associate it with a Sale?

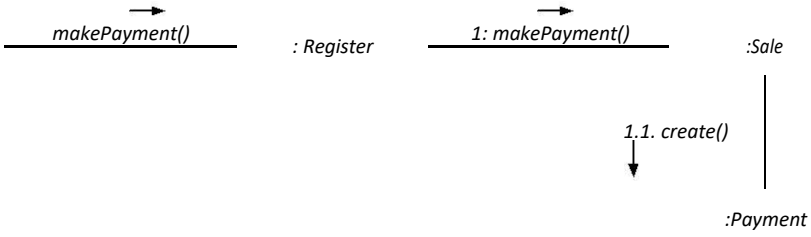
1. Register creates a Payment *p* then sends addPayment(*p*) message to the Sale



Register is taking on responsibility for system operation makePayment()

– In isolation no problem
– But if we start assigning additional system operations to Register then will violate **high cohesion**

– **Example:** (from POS system) Who should be responsible for creating a Payment instance and associate it with a Sale?
– Register delegates Payment creation to the Sale



This second approach will lead to higher cohesion for Register class.

Note: this design supports both low coupling and high cohesion

High cohesion, like low coupling, is an evaluative principle

Consider:

– Very low cohesion – a class is responsible for many things in different functional areas

– Low cohesion – a class has sole responsibility for a complex task in one functional area

– Moderate cohesion – a class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other

– High cohesion – a class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks

Typically high cohesion => few methods with highly related functionality

Benefits of high cohesion:

– Easy to maintain
– Easy to understand
– Easy to reuse

4.

The **indirection** pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view-controller pattern.

5.Information expert **(also** expert **or the** expert principle**) is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields, and so on.**

Using the principle of information expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.

Information expert will lead to placing the responsibility on the class with the most information required to fulfill it.

Problem: What is a general principle for assigning responsibilities to objects?

Solution: Assign a responsibility to the information expert, that is, the class that has the information necessary to fulfill the responsibility.

Example: (from POS system) Who should be responsible for knowing the grand total of a sale?

Example (cont.): Who should be responsible for knowing the grand total of a sale?

Summary of responsibilities:

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

- Information Expert => “objects do things related to the information they have”
- Information necessary may be spread across several classes => objects interact via messages

6) **Low coupling** is an evaluative pattern that dictates how to assign responsibilities to support

- lower dependency between the classes,
- change in one class having lower impact on other classes,
- higher reuse potential.

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

Problem: How to support low dependency, low change impact, increased reuse?

Solution: Assign a responsibility so coupling is low.

Coupling – a measure of how strongly one element is connected to, has knowledge of, or relies on other elements

A class with high coupling relies on many other classes – leads to problems:

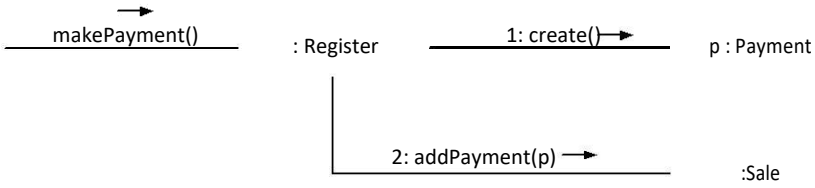
- Changes in related classes force local changes
- Harder to understand in isolation
- Harder to reuse

– Consider Payment, Register, Sale

– Need to create a Payment and associate it with a Sale, who is responsible?

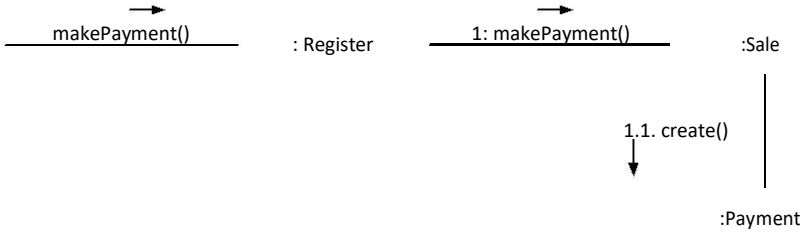
– Creator => since a Register "records" a Payment it should have this responsibility

– Register creates Payment *p* then sends *p* to a Sale => coupling of Register class to Payment class



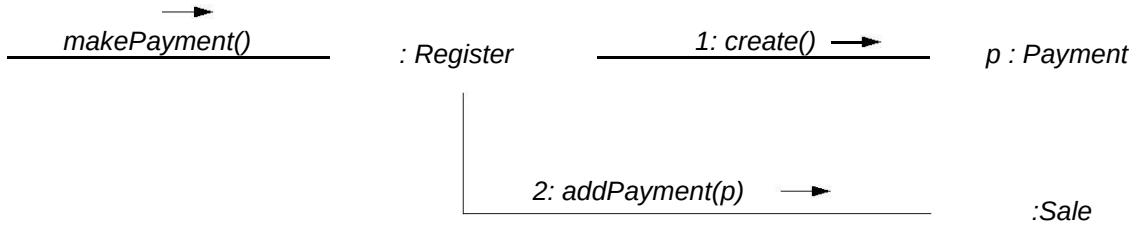
Alternate approach:

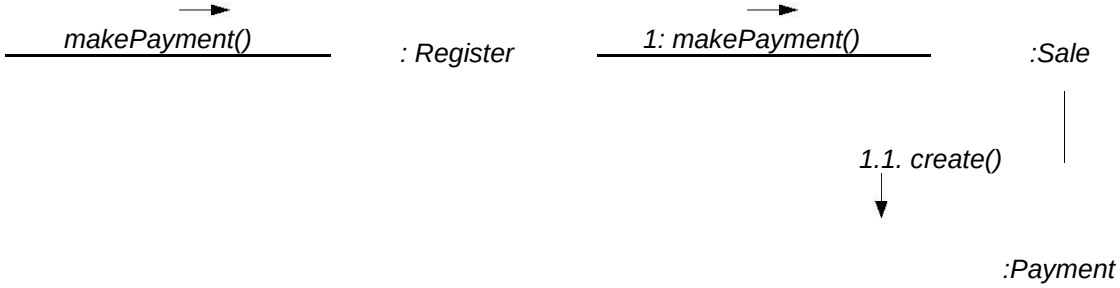
Register requests Sale to create the Payment



Consider coupling in two approaches:

- In both cases a Sale needs to know about a Payment
- However a Register needs to know about a Payment in first but not in second
- Second approach has lower coupling





7. **Polymorphism**

According to **polymorphism** principle, responsibility of defining the variation of behaviors based on type is assigned to the type for which this variation happens. This is achieved using polymorphic operations. The user of the type should use polymorphic operations instead of explicit branching based on type.

8. **Protected variations**

The **protected variations** pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface

9. Pure fabrication

A **pure fabrication** is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the *information expert* pattern does not). This kind of class is called a "service" in domain-driven design.

Assignment 9: Identification and Implementation of GOF pattern

Description : 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

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

What is Gang of Four *GOF*?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four *GOF***. According to these authors design patterns are primarily based on the following principles of object orientated design

Program to an interface not an implementation, Favor object composition over inheritance

Usage of Design Pattern : Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software** , there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns.

Pattern & Description

1 Creational Patterns :These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

2 Structural Patterns :These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

3 Behavioral Patterns :These design patterns are specifically concerned with communication between objects.

1. Abstract Factory

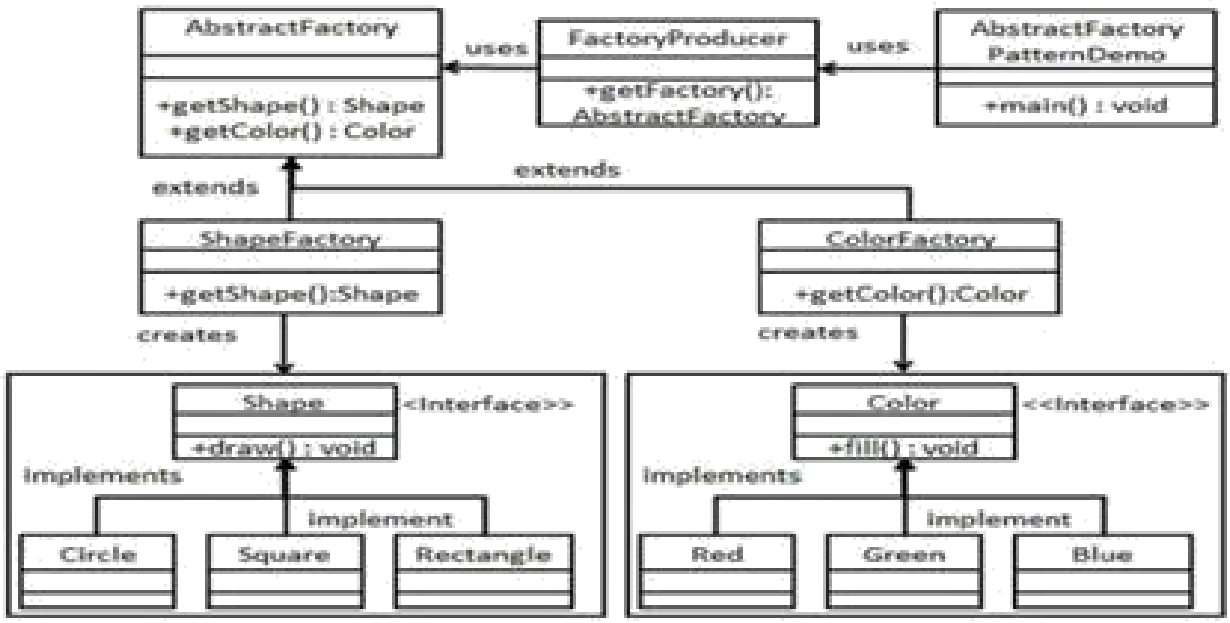
Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Implementation

We are going to create a *Shape* and *Color* interfaces and concrete classes implementing these interfaces. We create an abstract factory class *AbstractFactory* as next step. Factory classes *ShapeFactory* and *ColorFactory* are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* created.

AbstractFactoryPatternDemo, our demo class uses *FactoryProducer* to get a *AbstractFactory* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE* for *Shape*) to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN / BLUE* for *Color*) to *AbstractFactory* to get the type of object it needs.



Step 1

Create an interface for Shapes.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

Square.java

```
public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
```

Circle.java

```
public class Circle implements Shape {

    @Override
    public void draw() {
```

```
        System.out.println("Inside Circle::draw() method.");
    }
}
```

Step 3

Create an interface for Colors.

Color.java

```
public interface Color {
    void fill();
}
```

Step4

Create concrete classes implementing the same interface.

Red.java

```
public class Red implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Red::fill() method.");
    }
```

```
}
```

Green.java

```
public class Green implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Green::fill() method.");
    }
}
```

Blue.java

```
public class Blue implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Blue::fill() method.");
    }
}
```

Step 5

Create an Abstract class to get factories for Color and Shape Objects.

AbstractFactory.java

```
public abstract class AbstractFactory {
    abstract Color getColor(String color);
    abstract Shape getShape(String shape) ;
}
```

Step 6

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType){

        if(shapeType ==
            null){ return null;
```

```
        }

        if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();

        }else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();

        }else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }

        return null;
    }

    @Override
    Color getColor(String color) {
        return null;
    }
}
```

ColorFactory.java

```
public class ColorFactory extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType){
        return null;
    }
```

```
    @Override
    Color getColor(String color) {

        if(color == null){
            return null;
        }

        if(color.equalsIgnoreCase("RED")){
            return new Red();

        }else if(color.equalsIgnoreCase("GREEN")){
            return new Green();

        }else if(color.equalsIgnoreCase("BLUE")){
            return new Blue();
        }

        return null;
    }
}
```

Step 7

Create a Factory generator/producer class to get factories by passing an information such as Shape or Color

FactoryProducer.java

```
public class FactoryProducer {
    public static AbstractFactory getFactory(String choice){

        if(choice.equalsIgnoreCase("SHAPE")){
            return new ShapeFactory();

        }else if(choice.equalsIgnoreCase("COLOR")){
            return new ColorFactory();
```

```
        }

        return null;
    }
}
```

Step 8

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
    public static void main(String[] args) {

        //get shape factory
        AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

        //get an object of Shape Circle
        Shape shape1 = shapeFactory.getShape("CIRCLE");

        //call draw method of Shape
        Circle shape1.draw();

        //get an object of Shape Rectangle
        Shape shape2 = shapeFactory.getShape("RECTANGLE");

        //call draw method of Shape Rectangle
        shape2.draw();
```

```
        //get an object of Shape Square
        Shape shape3 = shapeFactory.getShape("SQUARE");

        //call draw method of Shape Square
        shape3.draw();

        //get color factory
        AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

        //get an object of Color Red
        Color color1 = colorFactory.getColor("RED");

        //call fill method of
        Red color1.fill();
```

```
        //get an object of Color Green
        Color color2 = colorFactory.getColor("Green");

        //call fill method of Green
        color2.fill();

        //get an object of Color Blue
        Color color3 = colorFactory.getColor("BLUE");

        //call fill method of Color
        Blue color3.fill();
    }
}
```

Step 9

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Inside Red::fill() method.

Inside Green::fill() method.

Inside Blue::fill() method.

2. Singleton pattern

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

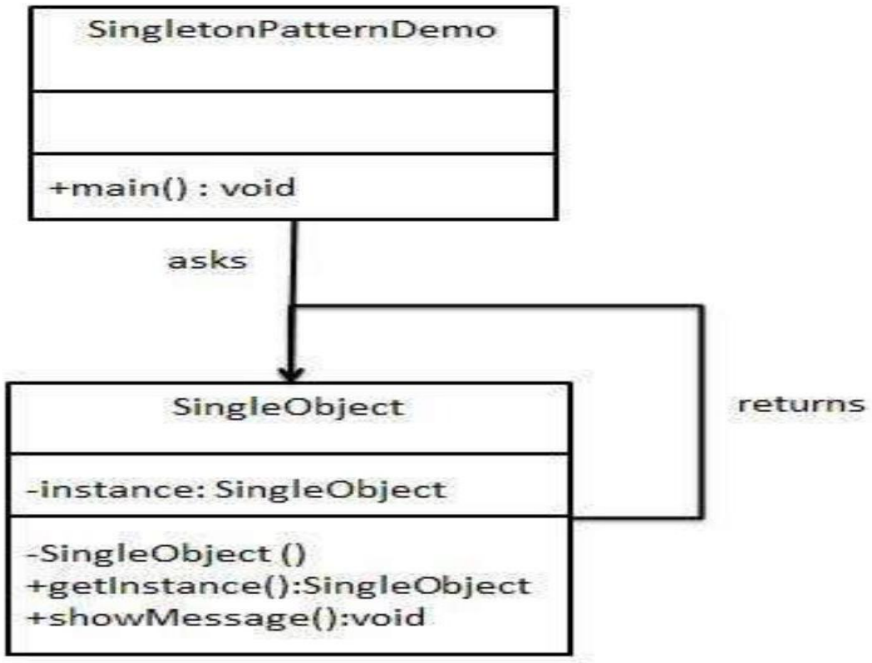
This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

SingleObject class provides a static method to get its static instance to outside world.

SingletonPatternDemo, our demo class will use *SingleObject* class to get a *SingleObject* object



Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }
```

```
    public void showMessage(){
        System.out.println("Hello World!");
    }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
    public static void main(String[] args) {

        //illegal construct

        //Compile Time Error: The constructor SingleObject() is not
        visible //SingleObject object = new SingleObject();


        //Get the only object available

        SingleObject object = SingleObject.getInstance();


        //show the message

        object.showMessage();

    }
}
```

Step 3

Verify the output.

Hello World!

3. Strategy Pattern

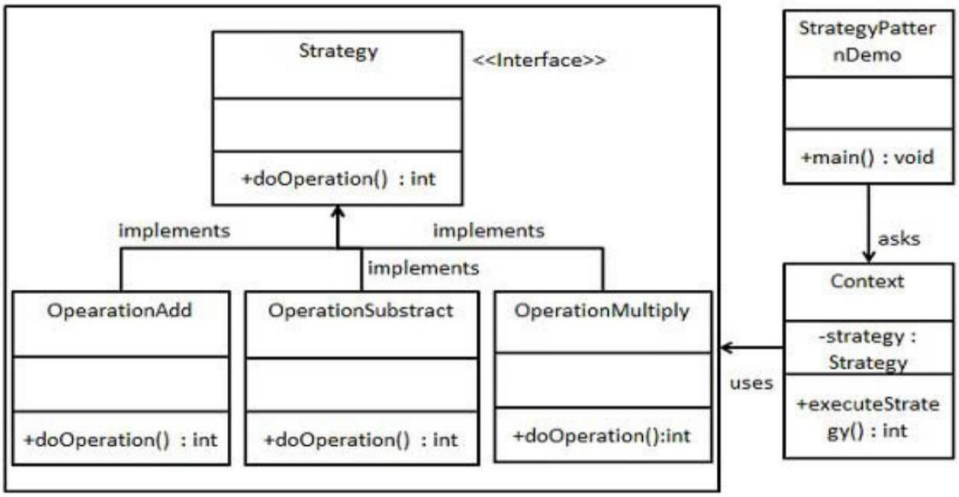
In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

Implementation

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 behaviour based on strategy it deploys or uses.



Step 1

Create an interface.

Strategy.java

```
public interface Strategy {
    public int doOperation(int num1, int num2);
}
```

Step 2

Create concrete classes implementing the same interface.

OperationAdd.java

```
public class OperationAdd
    implements Strategy{ @Override

    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
    @Override

    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
```

```
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
    @Override

    public int doOperation(int num1, int num2) {
        return num1 * num2;
    }
}
```

Step 3

Create *Context* Class.

Context.java

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}
```

Step 4

Use the *Context* to see change in behaviour when it changes its *Strategy*.

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

Step 5

Verify the output.

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```