SREEVAHINI INSTITUTE OF SCIENCE & TECHNOLOGY::TIRUVURU

2020-21 1st Semester

LABORATORY MASTER MANUAL

of

SOFTWARE ARCHITECTURE & DESIGN PATTERN LAB

Prepared by

D.MANI MOHAN

Assoc Professor

for

IV B.Tech

CSE(R16)

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

INDEX

PROGRAM OUTCOMES (POs)

PO1 Engineering Knowledge: Apply knowledge of mathematics and science, with fundamentals of Computer Science and Engineering to be able to solve complex engineering problems related to CSE.

PO2 Problem Analysis: Identify, Formulate, review research literature and analyze complex engineering problems related to Computer Science and Engineering and reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences

PO3 Design/Development of solutions: Design solutions for complex engineering problems related to Computer Science and Engineering and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety and the cultural societal and environmental considerations

PO4 Conduct Investigations of Complex problems: Use research–based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5 Modern Tool Usage: Create, Select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to computer science related complex engineering activities with an understanding of the limitations

PO6 The Engineer and Society: Apply Reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the Computer Science and Engineering professional engineering practice

PO7 Environment and Sustainability: Understand the impact of the Computer Science and Engineering professional engineering solutions in societal and environmental contexts and demonstrate the knowledge of, and need for sustainable development

PO8 Ethics: Apply Ethical Principles and commit to professional ethics and responsibilities and norms of the engineering practice

PO9 Individual and Team Work: Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary Settings

PO10 Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large such as able to comprehend and with write effective reports and design documentation, make effective presentations and give and receive clear instructions.

PO11 Project Management and Finance: Demonstrate knowledge and understanding of the engineering management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multi-disciplinary environments

PO12 Life-Long Learning: Recognize the need for and have the preparation and ability to engage in independent and life-long learning the broadest context of technological change.

PROGRAM EDUCATIONAL OBJECTIVES (PEO's)

The educational objectives of UG program in Computer Science and Engineering are:

PEO - 1: To graduate students with core competencies by strengthening their mathematical, scientific and basic engineering fundamentals.

PEO - 2: To Produce graduates who are prepared for lifelong learning and successful career as Computer Science and Engineering.

PEO - 3: To inculcate high professionalism among the students by providing technical and soft skills.

PEO - 4: To promote collaborative learning and spirit of team work through multidisciplinary projects and diverse professional activities.

PEO - 5: To encourage students for higher studies, research activities and entrepreneurial skills by imparting interactive quality teaching and conferences, seminars, workshops and technical discussions.

PROGRAM SPECIFIC OUTCOMES (PSO's)

The Computer Science and Engineering Program will demonstrate:

PSO1: Professional Skills: The ability to understand, analyze and develop solutions to various problems.

PSO2: Problem-Solving Skills: The ability to apply standard practices and strategies in cloud computing big data analytics and network security.

PSO3: Successful Career: The ability to use techniques, skills and modern engineering tools.

DETAILS OF ASSOCIATED COURSES

COURSE OUT COMES:

Course Name: SADP LAB

CO1	Understand interrelationships, principles and guidelines governing architecture and evolution over time
CO2	Analyze the architecture and build the system from the components
CO3	Prepare creational patterns that deal with object creation mechanisms
CO4	Prepare structural patterns that ease the design by identifying a simple way to realize relationships among entities.
CO5	**L**earn behavioral patterns that identify common communication patterns between objects and realize these patterns.
CO6	Classify various case studies

MAPPING LEVELS:

Correlation levels 1, 2 or 3 as defined below:

1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)

CO/PO/PSO MAPPING

CO/PO /PSO	**P01**	**P02**	**P03**	**P04**	**P05**	**P06**	**P07**	**P08**	**P09**	**P10**	**P11**	**P12**	**PS O1**	**PS O2**	**PS O3**
CO1	3	2	1	-	-	-	-	-	-	-	-	-	3		2

CO2	2	3	1	-		-	-	-	-	-	-	-	3		2
CO3	1	2	3	-	2	-	-	-	-	-	-	-	3		3
CO4	1	2	3	-	2	-	-	-	-	-	-	-	3		3
CO5	3	2	1	-	-	-	-	-	-	-	-	-	3		2
CO6	1	2	3	-	-	-	-	-	-	-	-	-	3		2

CO/PO/PSO JUSTIFICATION

MAPPING	CORRELATION LEVELS	JUSTIFICATION
C412.1-PO1	3	knowledge of interrelationships, principles and guidelines of Architecture
C412.1-PO2	2	Usecase modeling and Risk Analysis
C412.1-PO3	1	Usage of Rational Rose Tool with Rational Rose Software Architect.
C412.2-PO1	2	Understanding various static structure of the classes.
C412.2-PO2	3	Analyze and construct various classes and classes needed for the system.
C412.2-PO3	1	Design use case, class diagram using Rational Rose tool
C412.3- PO1	1	Knowing instances of classes and interface for creating a factory of related objects.
C412.3 -PO2	2	Provide an interface for creating families of related or dependent objects without specifying their concrete classes
C412.3-PO3	3	Design abstract factory and builder patterns
C412.3-PO5	2	Use Rational Rose to design this patterns.
C412.4-PO1	1	Knowing various interfaces and classes and their relationships to build these pateerns
C412.4-PO2	2	Identifying various complexities of classes and providing interfaces.
C412.4-PO3	3	Design various structural patterns
C412.4-PO5	2	Use Rational Rose tool to design this patterns.

C412.5-PO1	3	Understanding various objects and their properties to represent the patterns
C412.5-PO2	2	Formulate various relationships between the objects
C412.5-PO3	1	Design various behavioral patterns
C412.6-PO1	1	Knowing various actors and usecases to represent case studies
C412.6-PO2	2	Identifying the relationship between the actors and usecase and their representation.
C412.6-PO3	3	Design various diagrams for various case studies.

IV Year – I SEMESTER	T	P	C
	0	3	2

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS LAB

Week 1:

Presentation of the Project: Weather Mapping System.

Week 2:

Design of the Use Case View. Risk Analysis

Week 3:

Design of the Logical View of the Weather Mapping System (WMS).

Week 4:

Integration of selected architectural and design patterns in the logical view obtained previously

Week 5:

Design of the implementation, process, and deployment views for the Weather Mapping System

Week 6:

Generation from the previous architecture design of CORBA Interfaces and Components Definitions

Week 7:

Implementation of the Weather Mapping System (Java & C++), with a particular emphasis on the Inter process communication mechanism and the software components identified.

Week 8:

- Use case Diagram for Librarian Scenario
- Using UML design Abstract factory design pattern

Week 9:

- Using UML design Adapter-class Design pattern
- Using UML design Adapter-object Design pattern

Week 10:

- Using UML design Strategy Design pattern
- Using UML design Builder Design pattern

Week 11:

- Using UML design Bridge Design pattern
- Using UML design Decorator Design pattern

Week 12:

- User gives a print command from a word document. Design to represent this chain of responsibility Design pattern
- Design a Flyweight Design pattern

Week 13:

- Using UML design Facade Design pattern
- Using UML design Iterator Design pattern

Week 14:

- Using UML design Mediator Design pattern
- Using UML design Proxy Design pattern

Week-15:

Using UML design Visitor Design pattern

Exp no:1

Presentation of the Project: Weather Mapping System:

Summary of the Requirements for the Weather Mapping System (WMS)

System Overview

A weather mapping system (WMS) is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellite. Weather stations transmit their data to the area computer in response to a request from that machine.

The area computer system validates the collected data and integrates the data from different sources. The integrated data is archived and, using data from this archive and a digitized map database, a set of local weather maps is created. Maps may be printed for distribution on a special-purpose map printer or may be displayed in a number of different formats.

Typically, the weather data collection system establishes a modem link with the weather station and requests transmission of the data.

Then, the weather station sends a summary of the weather data that has been collected from the instruments (e.g. anemometer, barometer, ground thermometer) in the collection. The data sent to the weather data collection system are the maximum, minimum and average ground temperatures, the maximum, minimum and average air pressures, the maximum, minimum, and average wind speeds. Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and be modified in the future. Weather stations are also automatically monitored by the system, for start-up, shutdown, and for instrument testing and calibration.

Another function carried out by weather stations is pollution monitoring. More specifically, there is an air quality meter that computes the amount of various pollutants in the atmosphere collected from pollution monitoring instruments (e.g. No meter, Smoke meter and Benzene meter). The pollution readings are transmitted at the same time as the weather data. The pollution data collected are NO, smoke and benzene rates.

Product Features

WMS provides a range of map processing operations (e.g. two-dimensional and three-dimensional mappings), weather data storage for the maps, and network access to the maps for remote viewers. Its key marketing features are the following:

- A user-friendly operator environment
- The throughput of weather data acquisition is 50% higher than previous product
- Maps display can be as fast as the maximum hardware speed
- Is designed for easy upgrade to new platforms
- Has open platform connectivity
- Can be connected to peripherals, including printers and digital imagers.

A single control panel is provided with each WMS unit. Additional independent viewing stations are also available.

The Future of WMS

WMS must be designed to be extensible, maintainable, and portable. Its design must be flexible enough to accommodate certain expected changes. The product requirements may change somewhat during development and certainly will change over the lifetime of the product. The physical characteristics of the weather data acquisition instruments may change as new models are introduced. As the processing power of the system increases, more and more work that was the responsibility of the users will shift to the system.

Other product features will also likely change. The product needs to remain compatible with new or evolving standards for file formats and communication of weather information.
In addition, the technology affecting the software components is likely to change over the lifetime of the system. WMS may have to be improved to handle upgrades to commercial components that are part of the product, and the target software environment is likely to change as upgrades are introduced.

The Company

The software company in charge of the development of the WMS has faced significant turnover of their skilled personnel in the last few years, mainly due to the competitive job market. Facing at the same time tight deadlines set by the customers to deliver useful business functionality; the company has decided to make use of significant third-party products and packages as an integral part of their application. However they will also maintain development and maintenance control of the mission-critical components of their applications.

Experiment-2:

Design of the Use Case View,Risk Analysis

Weather Mapping System (WMS)-Use Case View

In this lab session you'll start the analysis of the Weather Mapping System (WMS). The requirements of WMS are summarized in a separate document.

A. Use Case Modeling

1. Identify the actors involved in this problem, create them and provide appropriate documentation using Rational Rose. The documentation for an actor is a brief description that should identify the role the actor plays while interacting with the system. (20%)
2. Evaluate the needs that must be addressed by the system, and based on this information, identify the use cases for this problem. Create the use cases and provide appropriate documentation using Rational Rose. The documentation for a use case is a brief description that states the purpose of the use case in few sentences, and gives high-level definition of the functionality provided by the use case. (20%)
3. Create the main use case diagram in Rational Rose, by putting together the use cases and actors identified previously. Complete the diagram by defining and creating appropriate relationships among the actors and use cases involved. (10%)

B. Risk Analysis

1. One of the most important risks that the development of the weather mapping system is facing is Change management; failure to handle properly for instance changes in the product requirements may lead to the overall project failure. Identify using a cause-effect (sub) tree the risks hierarchy that may lead to the materialization of this risk. Describe these risks and corresponding reduction measures by providing a risk registry. (30%)
2. Not all use cases are architecturally significant. Only critical use cases that carry the most important risks, or the quality requirements or key functionalities are eligible for the architecture definition. One such use case encompasses functionality for collecting data from remote sources: we call it Collect data. Based on the system requirements, motivate briefly how the Collect data use case relates to the Change management risk specified previously. Provide the flow of events for Collect data use case. The flow of

events is a description of the events needed to accomplish the required behaviour of the use case. The flow of events is written in terms of what the system should do, not how the system does it. Link the flow of events document to the use cases in your Rose model. (20%)

Exp 2 Solution:

Implementation of Use case View:

Use Case Modeling

1.The following actors may be identified from the requirements: Instrument, Weather Instrument, Pollution Instrument, Operator, Digitized map database and Consumer.

Actor documentation:

- Operator: a person who is responsible for the maintenance of the weather mapping system.
- Consumer: a person who uses or needs the weather information produced by the system.
- Weather Instrument: weather sensors related to weather stations used to collect raw weather data.
- Pollution Instrument: air data sensors used to collect air data for pollution monitoring.
- Instrument: generalization of weather and pollution sensors.
- Digitized map database: an existing database containing mapping information.

2.The following needs must be addressed by the system:

- The Consumer actor needs to use the system to view or print weather maps.
- Weather sources collect weather data and send them on request to the area computer.
- The area computer validates, integrates and archives the weather data collected, and then processes weather maps.
- The digitized map database provides the mapping information needed to process local weather maps.
- The Operator actor manages and maintains the system.
- The area computer monitors the weather sources, for start-up, shutdown, and for instrument testing and calibration.

Needs:

Based on these needs, the following use cases may be identified:

- Startup
- Shutdown
- Collect data
- Calibrate
- Test
- Validate
- Integrate
- Archive
- Process map
- Display
- Print

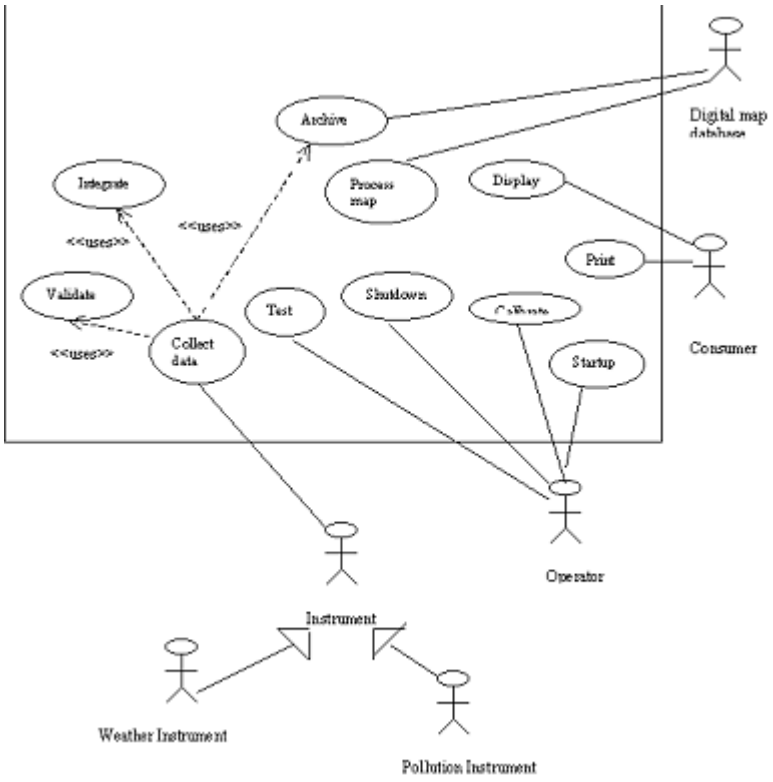
You may identify different use cases (or the same use cases with different names). In all cases the sum of the use cases identified must cover all the key functionalities mentioned in the needs. That should appear through the documentation that you will provide.

Use Case Documentation:

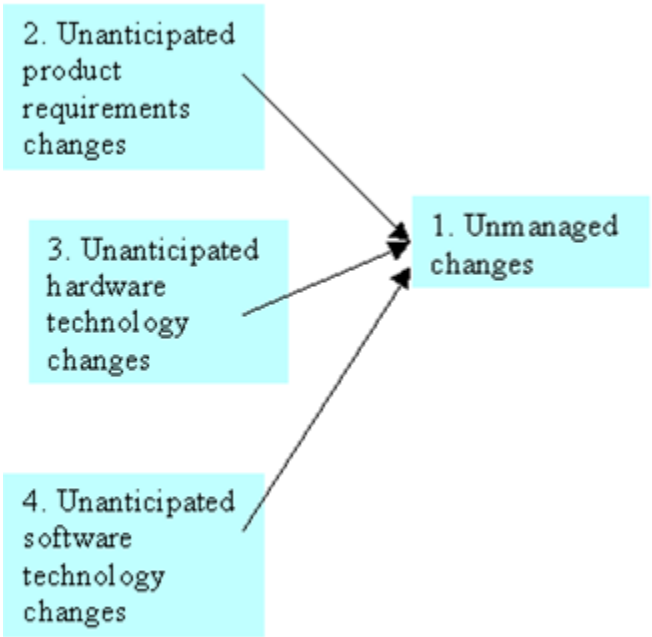
The documentation provided for a use case consists of a brief description of the purpose of the use case in a few sentences. For instance, a brief description of the Collect data use case may be:

The operator or the system starts this use case. It consists of sending a request for data collection to the weather sources, which collect and summarize weather data from the instruments and send them to the area computer.

3. Main Use Case Diagram for WMS



C. Risk Analysis



Risk Registry

Risk	Description	Causes	Source of uncertainty	Nature	Probability	Impact	Reduction Measures
1	Unmanaged changes: failure to cope with the changes occurring in the system development or operation; excessive cost or effort required modifying, updating, evolving, or repairing the system during its operation.	Project fails	Event	Some of the components may remain stable and other may change over time.	L	E	S1:Identify, and Isolates functionalities and components, which are likely to change. S2: Make it easy to add or remove components (modular design, separation of policy and implementation, separation of interface and implementation etc.) S3: Portable design (use open standards and technologies; use platform independent technologies and standards such as Java and CORBA)
2	Unanticipated product requirements changes: product requirements such as the frequency or procedures of weather data reporting	1	Event	Complex design; Design lack of flexibility.	L	E	S1, S2

	byweather stations may change in the future.						
3	Unanticipated hardware technology changes: physical characteristics of the weather data acquisition instruments may change in the future.	**1**	Event	Complexity, lack of portability.	L	**E**	S1, S3
4	Unanticipated software technology changes: upgrade of COTS components, standard changes (e.g. file formats, communication of weather information etc.)	**1**	Event	Complex design; design lack of flexibility	L	**E**	S1, S2, S3

2.

The collect data use case is related to the change management risk in several respect: weather data acquisition instruments change, weather information communication procedure and standard change etc.

Flow of Events Description

The flow of events should include:

- When and how the use case starts and ends
- What interaction the use case has with the actors
- What data is needed by the use case
- The normal sequence of events for the use case

- The description of any alternate or exceptional flows

The following description may be given for the Collect data use case:

1.0Flow of Events for the Collect Data Use Case

1.1Preconditions

The main flow of the Startup use case needs to complete before this use can start.

1.2 Main Flow

This use case begins when the weather data collection system establishes a modem link with the weather station and requests transmission of the data. The weather station sends an acknowledgment to the collection system (E-1), and then collects and summarizes the data from the weather (S-1) and air (S-2) data instruments. The summarized data is sent to the weather data collection system.

1.3 Sub flows

S-1: Report weather data

Weather data readings are collected and reported from weather data instruments. The data sent are the maximum, minimum and average ground and air temperatures, the maximum, minimum and average air pressures, the maximum, minimum, and average wind speeds.

S-2: Report air data

The pollution readings are collected and transmitted at the same time as the weather data.

The pollution data collected are NO, smoke and benzene rates

1.4 Alternative Flows

E-1: The collection system hasn't received any acknowledgment after a certain deadline. The main flow of Test use case is started. The use case is terminated and an error message is displayed.

Exp 3: Logical View

Design of the Logical View of the Weather Mapping System (WMS).

A. Use Case Realization

The realization of a use case is called collaboration. A use case is a collection of scenarios, each describing specific aspect of the use case. We consider two scenarios of the Collect data use case called Report Weather Readings and Report Air Quality:

- Report Weather Readings: encompasses functionality for reading weather data from remote sources. More specifically weather stations transmit the data collected from weather instruments to the area computer when they receive a request from that machine.

- Report Air Quality Readings: the pollution readings are automatically collected when requested by a weather station and transmitted at the same time as the weather data.

1. The flow of events for a use case is captured in text, whereas scenarios are captured in interaction diagrams. There are two types of interaction diagrams: sequence diagrams that show object interactions arranged in time sequence, and collaboration diagrams, which show object interactions organized around the objects and their links to each other. Both represent an alternate way of describing a scenario.

a. Give using Rational Rose the sequence diagram corresponding to the Report Weather Readings and the Report Air Quality scenarios. (30%)

b. Give an alternate representation for the same scenarios using collaboration diagrams. (10%)

2.

a. Identify the boundary, entity and control classes involved in these scenarios. Create these classes using Rational Rose and add if applicable appropriate stereotypes. (10%)

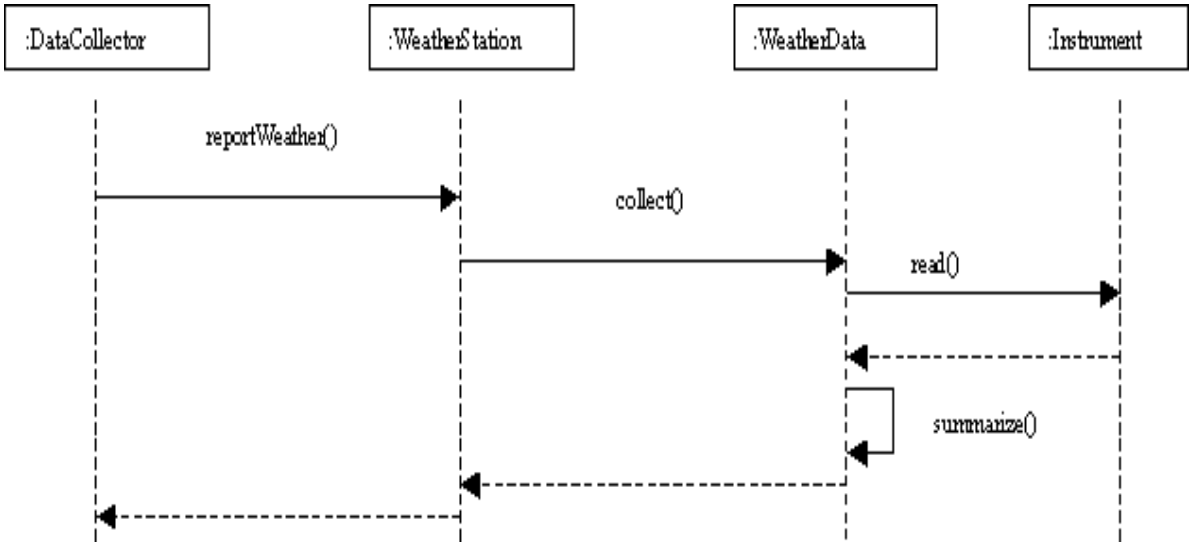
b. Class diagrams may also be attached to use cases and contain a view of the classes participating in the use case. Create using Rose a view of participating classes to the Collect data use case. (10%)

c. Identify the relationships among the classes involved in the scenarios, and create them in the corresponding class diagram using Rose. (10%)

3. Create the attributes and operations involved in the classes participating in the Collect data use case and document them using Rose. The documentation for an operation should state briefly the functionality performed by the operation. It should also state any input values needed by the operation along with the return value of the operation. This information may not be known initially, in which case it should be added later when more is known about the class. The ocumentation for an attribute should state concisely the purpose of the attribute, not the structure of the attribute. (30%)

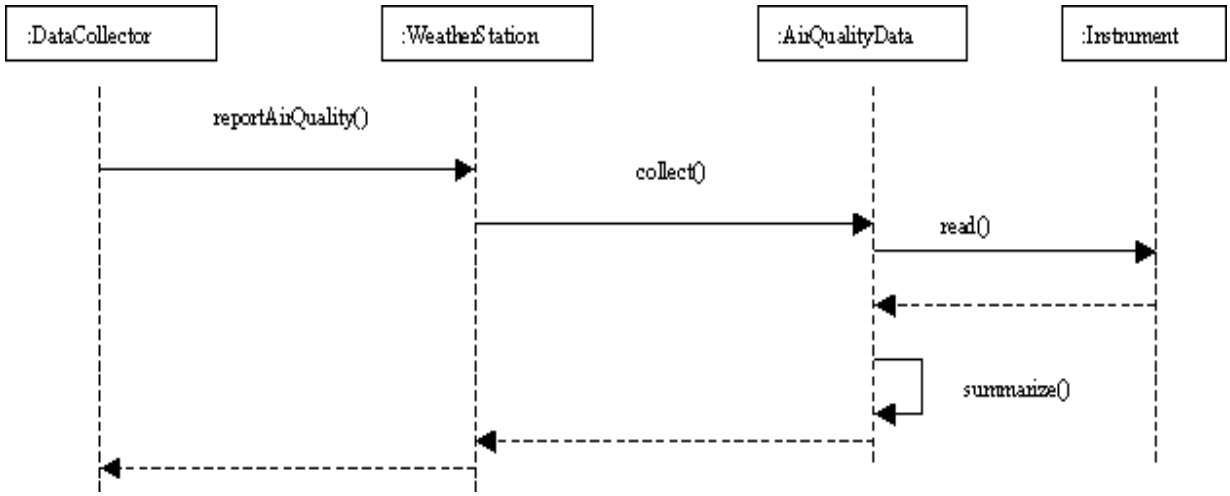
Exp 3: Solution

- Report Weather Readings Scenario



- Report Air Quality readings Scenario

Both diagrams are similar: you may describe both scenarios using one diagram, in which case you should describe the concurrency aspects.



b. Collaboration and sequence diagrams are dual; using Rose, you can generate either diagram from the other automatically. In any case, you must find the same objects, messages, and communication paths.

2.

a. Entity class models information and associated behaviour that is generally long lived. This type of class is typically independent of their surroundings; that is they are not sensitive to how the surroundings communicate with the system. Boundary classes handle the communication

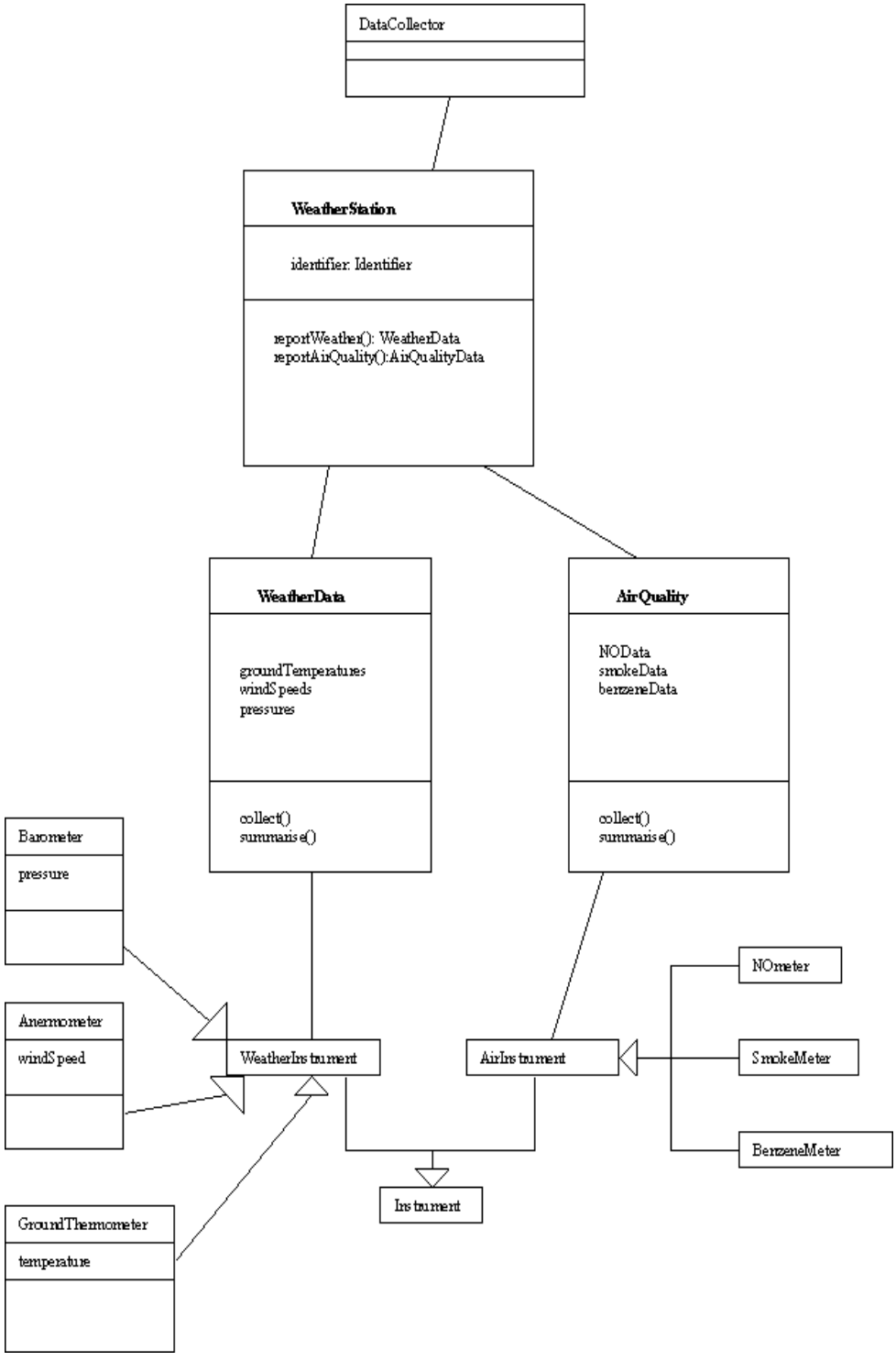
between the system surroundings and the inside of the system. They can provide the interface to a user or another system. Control classes model sequencing behaviour specific to one or more use cases: they represent the dynamics of the use case.

- **Boundary classes**: this use case interacts only with the data collection instruments. Hence, the boundary classes would be the classes encapsulating the instruments (weather and air data collection instruments: GroundThermometer, Anemometer, Barometer,Nometer, Smokemeter and Benzenmeter).

- **Entity classes**: this scenario deals with weather and air data collection. We can identify two entity classes: WeatherData and AirQualityData. AirQualityData class represents the air quality data.

- **Control classes**: there are two control classes that handle the flow of events for the use case: WeatherStation and DataCollector.

b. **and c.** We derive the classes participating to the use case from the interaction diagrams. We shall find in the class diagram the same relationships (correspond to the message flows) and the same operations.


DataCollector
WeatherStation
identifier: Identifier
reportWeather(): WeatherData
reportAirQuality():AirQualityData
WeatherData
groundTemperatures
windSpeeds
pressures
collect()
summarise()
AirQuality
NOData
smokeData
benzeneData
collect()
summarise()
Barometer
pressure
Anemometer
windSpeed
GroundThermometer
temperature
WeatherInstrument
AirInstrument
Instrument
NOmeter
SmokeMeter
BenzeneMeter

3.

Operations:

Operations may be identified using interaction diagrams: operations are associated to the messages exchanged.

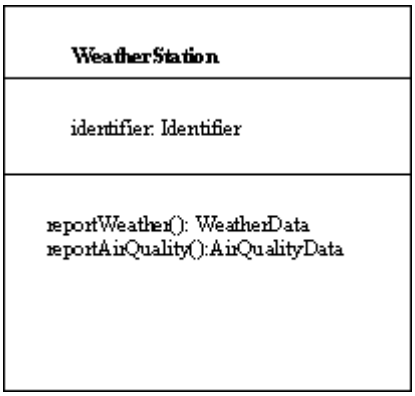
Documentation for the collect() operation of the WeatherData class:

Collect weather information from the weather instruments. Inputs: request from the weather data collection subsystem

Outputs: summarised weather data sent to the weather collection subsystem.

Attributes:

Example: attributes and operations for WeatherStation class:



Documentation for the identifier attribute of the WeatherSation class:

The unique reference used to distinguish any weather station in the WMS system.

The documentation should state briefly the purpose of the attribute, not the structure of the attribute. For instance a bad definition for the identifier attribute would be “a character string of length 15”.

Exp:4

Integrating Patterns in the Architecture

Integration of selected architectural and design patterns in the logical view obtained previously

Figure 1 depicts a class diagram describing the static structure of the classes participating to the Report weather data scenario of the Collect data use case. The diagram consists of eight classes. Classes WeatherStation, WeatherData, Instrument, GroundThermometer, Barometer, and Anemometer describe the data collection functions carried by the weather stations and associated instruments. Classes DataCollector and ArchivedData describe data collection functions carried by the area computer.

- WeatherStation: provides the basic interface of a weather station with its environment.
- WeatherData: encapsulates the summarised data from the different instruments in a weather station.
- Instrument, GroundThermometer, Barometer, and Anemometer: represent corresponding weather instruments in the system.
- DataCollector: sends data collection requests to Weather stations, and stores the received data.
- ArchivedData: represents the data collected by the area computer from a weather station; it encapsulates the collected data, the collection date and time, and the identifier of the source station.
- Identifier, Date, Time, and Readings are user-defined types: Identifier is a user-defined type consisting of a sequence of 4 digits; Readings is a record type whose fields correspond to maximum, minimum, and average weather readings (e.g. temperature, pressure etc); Date (day,month,year) and Time (hour,minute,second) are also defined as record types.

Some of the patterns selected by the architecture team as structuring mechanisms for the software architecture, include the pipes-and-filters architectural pattern and the factory design pattern.

1. Based on the risk factors identified during the risk analysis (conducted in Lab1) explain (briefly) why these patterns are appropriate for handling some of the design issues underlying the development of the weather mapping system. (20%)
2. Re-organize the class diagram given in Figure 1 around the Pipes-and-filter architectural pattern: specify the filters, the data source and sink, and the model of pipes used for interconnection. Precisely, you must provide (in your report) the following information: (50%)
    a. Filters: indicate for each filter whether it is active or passive; specify corresponding classes (a filter may match one or several classes).
    b. Data sink and source: indicate whether they are active or passive; specify corresponding classes.
    c. Pipes: specify for each pair of filters (including data source and subsequent filter, or data sink and preceding filter) the model of pipes used for interconnection, either as direct call or synchronization pipeline. For synchronization pipelines, specify whether they use a push, a pull, or a mix model.
3. Use the Factory design pattern to re-design the class structure of the various classes corresponding to weather instruments, and update the class diagram accordingly. (10%)
4. Reusability, extensibility and maintainability are some the main quality factors that characterize the weather mapping system. Organizing the system into loosely coupled and highly cohesive subsystems (e.g. modules) is one of the strategies used to achieve these quality goals. Decomposition in three subsystems named Sensors, Station, and Data Collection is considered. Two possible grouping strategies are proposed in Tables 1 and 2. Select the most suitable one, and motivate your choice by computing a coupling metric such as CBO (Coupling Between Object Classes). Create the subsystems using Rational Rose, and relocate the classes in the Rose Browser. (20%)

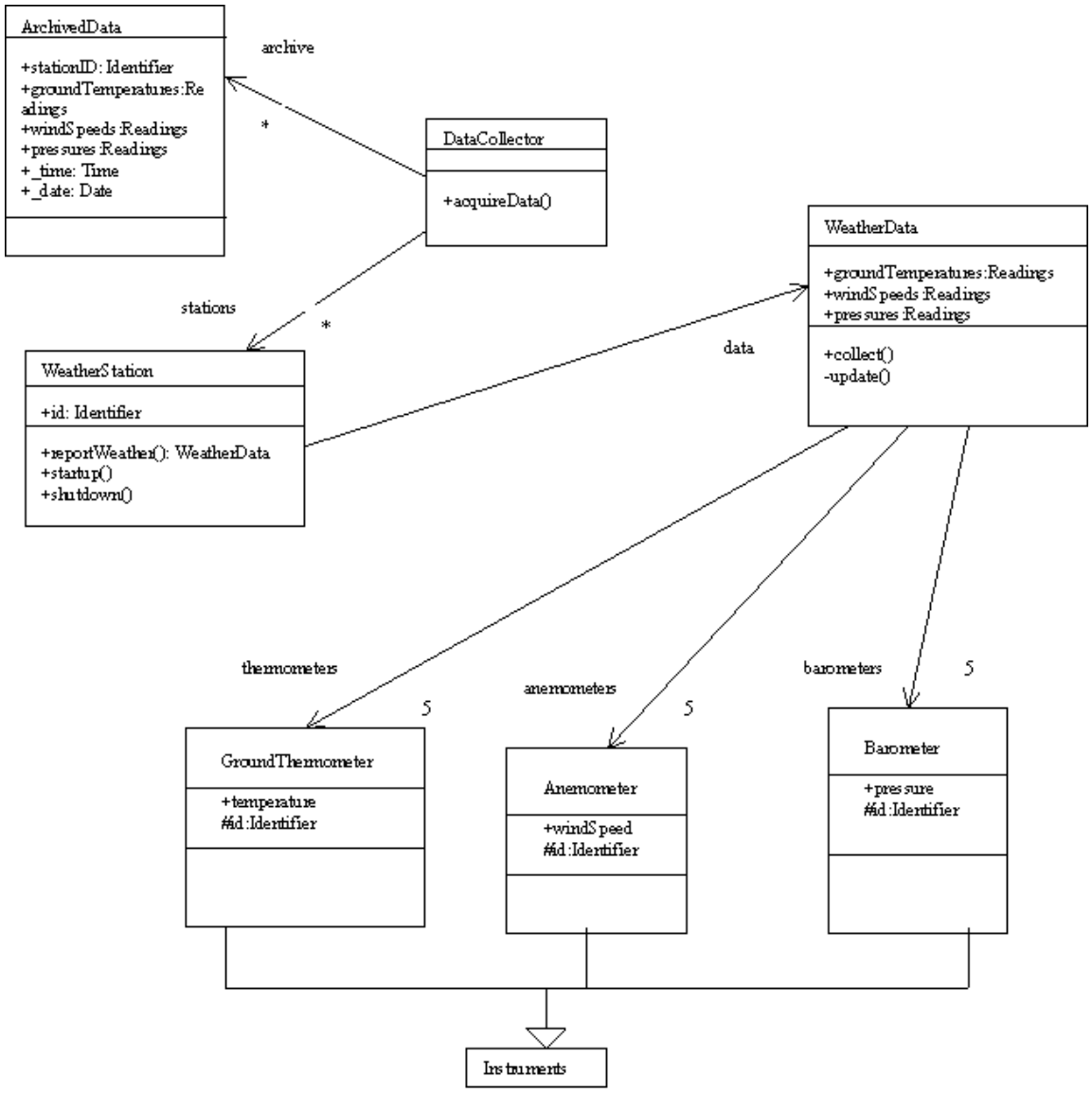


Figure 1: Class Diagram

Classes	Subsystems
Instrument, GroundThermometerBarometer, Anemometer	Sensor
WeatherStation, WeatherData	Station
DataCollector, ArchivedData	DataCollection

Table 1: Grouping 1

Classes	Subsystems
Instrument, GroundThermometer Barometer, Anemometer, WeatherData	Sensor
WeatherStation	Station
DataCollector, ArchivedData	DataCollection

Table 2: Grouping 2

Exp 4: Solution

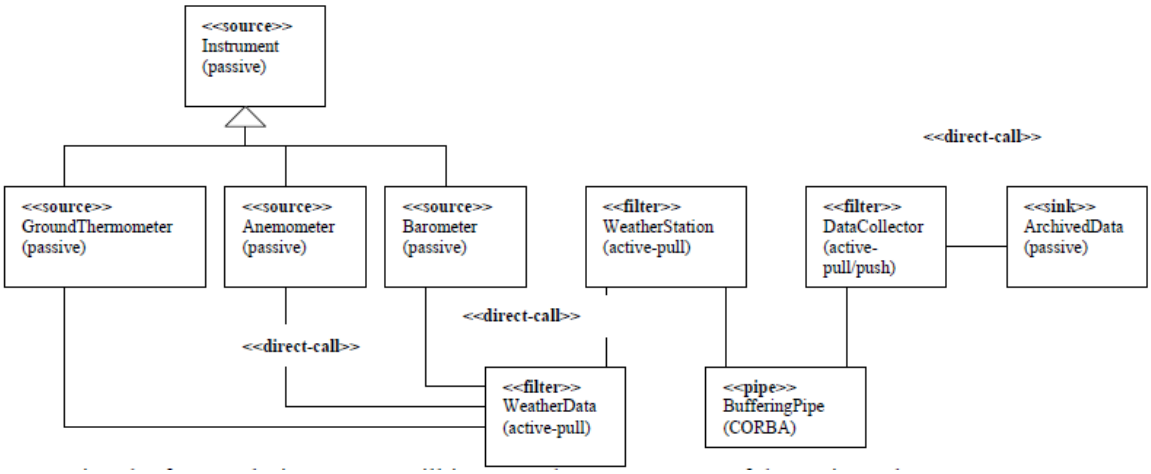
1. Extensibility, maintainability, and portability are among key requirements of the system. It is expected that the product requirements may change during development or operation. The physical characteristics, the kind and the number of the instruments involved may change.

An important design issue in that case consists of making easy removal and addition of components. That can be achieved using the pipes-and-filter pattern.

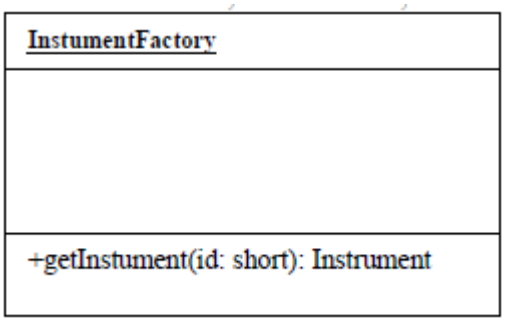
In order to handle the changes in the instruments, the factory design pattern is a mechanism that can introduce the desired level of decoupling.

2.The pipes-and-filter pattern can be applied to the given class structure in various ways. The instruments must be designed as data source, active or passive. WeatherData class may be designed either as a filter or as a pipe. WeatherStation and DataCollector must represent active filters. ArchivedData must be designed as a passive data sink.

According to the requirements, the area computer, through DataCollector class, polls weather stations. So DataCollector is an active filter based on push/pull mechanism. Since WeatherStation is also an active filter, a synchronization buffer may be included here; the buffer may be implemented later using an Interprocess Communication Mechanism such as CORBA. The DataCollector may also pull data directly from weather



3. Using the factory design pattern will improve the management of the various classes corresponding to weather instruments and to anticipate changes in the class structure. The factory design pattern is applied by integrating a factory class named InstrumentFactory in the class diagram that will be used to create instances of instruments. It provides the getInstrument() method, which is in charge of calling the appropriate constructors (GroundThermometer, Barometer, Anemometer).



4. The difference between the two grouping proposed is the allocation of class WeatherData.

So we only need to compute coupling metrics related to this class:

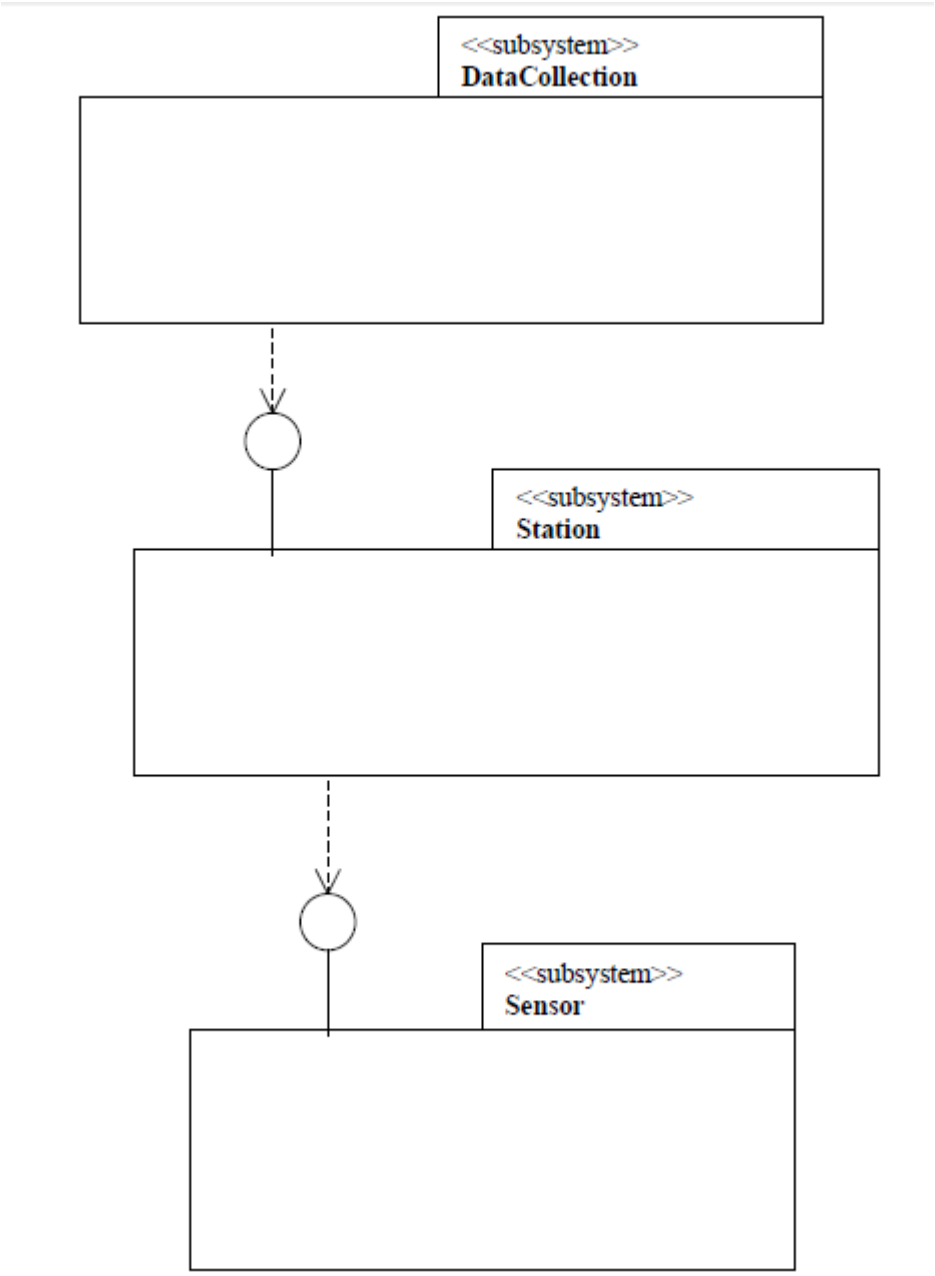
Grouping 2:

Coupling (station, sensor) = CBO(WeatherData, WeatherStation)= 1

Grouping 1:

Coupling(station,sensor) = CBO(WeatherData,Anemometer) + CBO(WeatherData, GroundThermometer) + CBO(Barometer) = 1+1+1=3.

The best grouping is the one that provides the lowest coupling between modules. We need of course to evaluate also cohesion and if necessary make a trade-off.



Exp 5: Implementation, Process, and Deployment Views for the Weather Mapping System

A. Process and Deployment Views

After studying the subsystems defined for the problem, examining existing hardware, and estimating the load on the system during normal operation, the architecture team decided that they will need seven processors for the system: two for the system operators to access the system, one for the database, one for the data processing activities, and three for data sources. The data source processors are located at the weather stations: one processor per station (e.g. we assume that there are three stations).

1. Identify the runtime entities (e.g. process, thread etc.) involved in the Collect data use case and provide a corresponding class diagram showing the relationships among them using Rose (use suitable stereotypes). 30%

2. The deployment view of architecture involves mapping software to processing nodes: it shows the configuration of run-time processing elements and the software processes living in them. Create using Rose the deployment diagram for the WMS system. 30%

B. Implementation View for WMS

The architecture team decided that the classes encapsulating the functionality of weather data acquisition instruments would be designed as DLL (Dynamic Loadable Library).

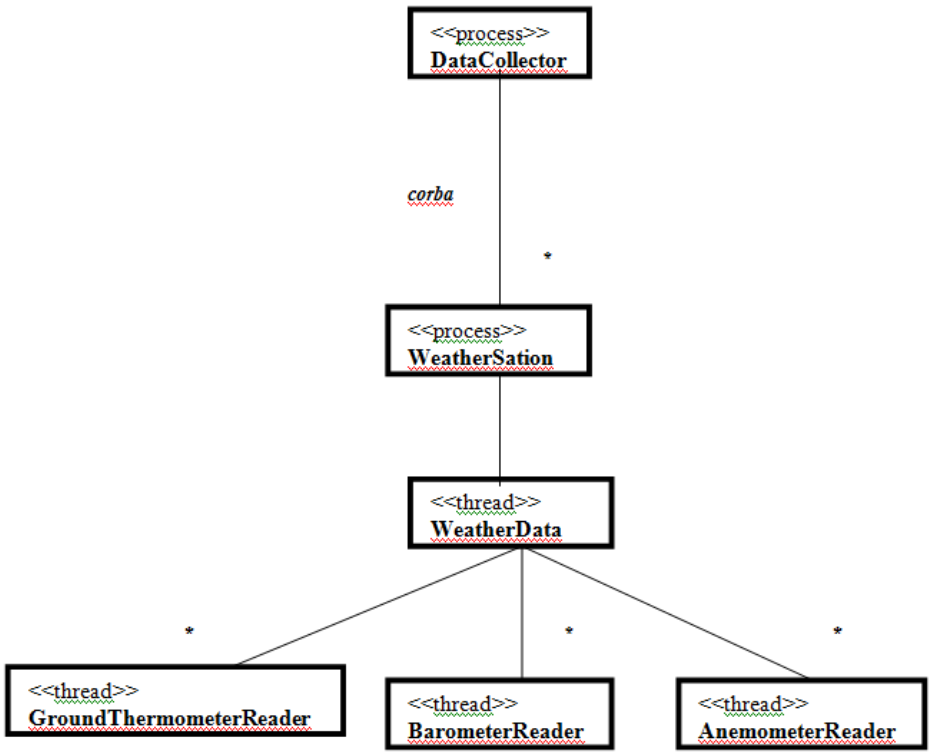
This allocation was chosen in order to anticipate future change of the weather instruments. By making them libraries, only the libraries would have to be replaced in case of change.

1. Create using Rose a UML component diagram depicting the executable release for the classes participating in the Collect data use case. A release is a relatively complete and consistent set of artifacts delivered to an internal or external user. A release focuses on the parts necessary to deliver a running system. 40%

Exp 5:Solutions

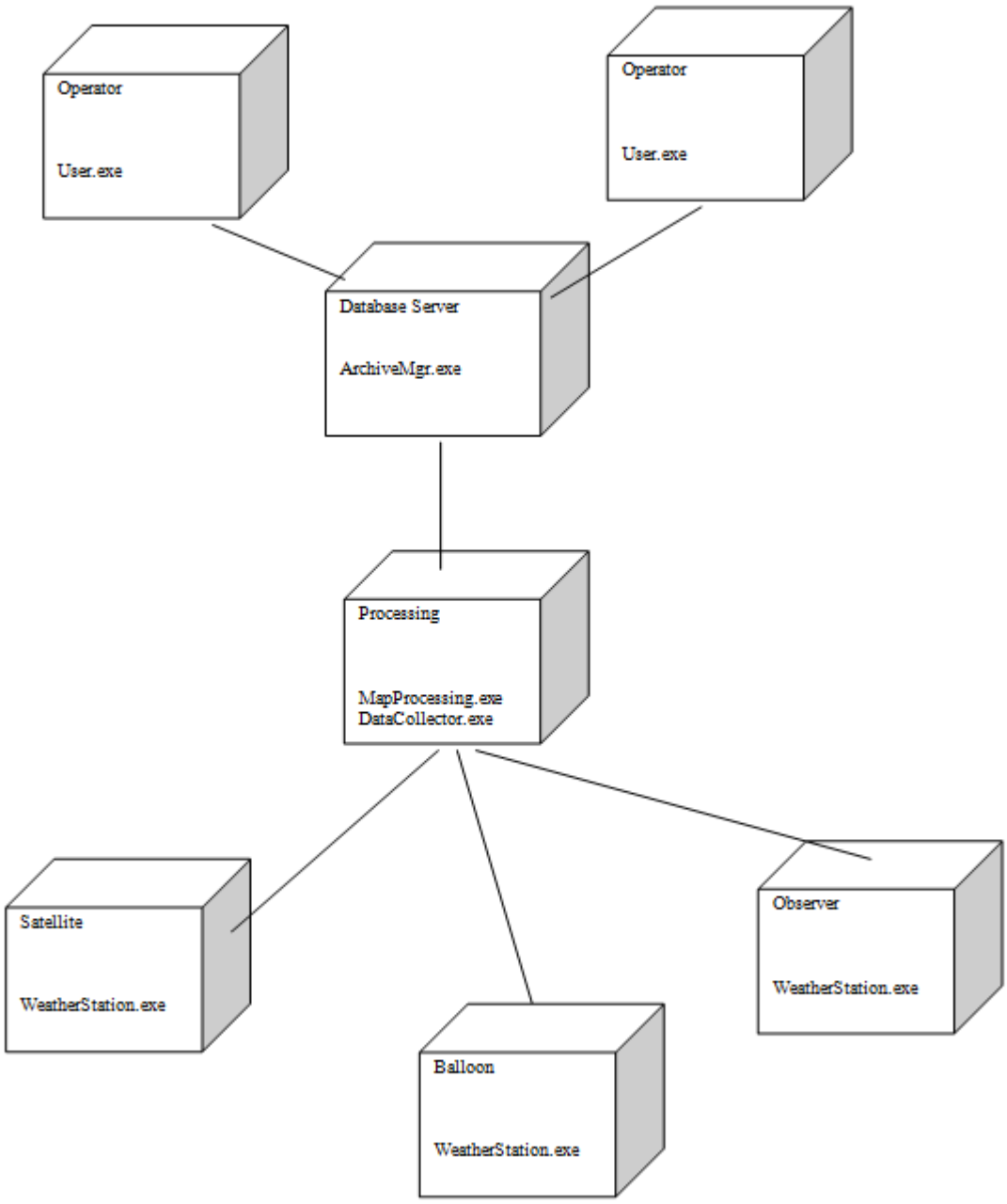
A.Process and Deployment Views

1. The runtime entities are the processes and threads involved in the system. They correspond directly to the active classes that can be identified from the design (class) diagram. You may decide to convert each active class in one process, which requires designing appropriate interprocess mechanisms for each pair of related processes, and is more expensive. You may integrate some threads, in which case you'll have to design appropriate synchronization mechanism. We adopt the following strategy (see figure):



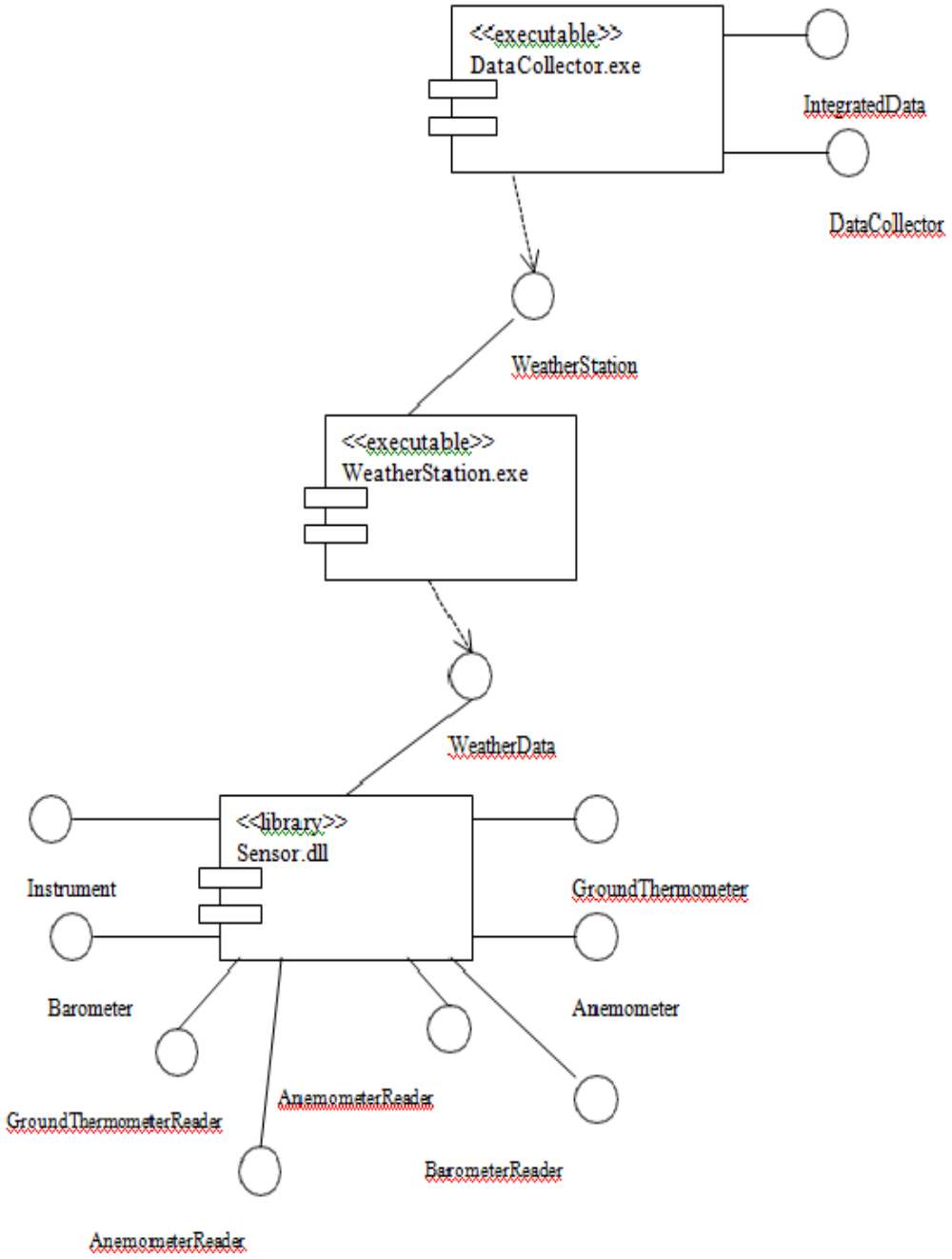
Data collection service (DataCollector) and weather stations (WeatherStation) are designed as independent processes. Every WeatherStation process carries the main thread of control, which is dedicated for receiving incoming requests. These requests are then spawned as specific threads for actual data collection from the instruments, via an independent WeatherData thread. For every kind of instrument XXX, there is a specific XXXReader thread that is spawned. The XXXReader thread will collect readings from all the related instruments and then return the minimum, maximum, and average values.

2. Deployment diagram:



A. Implementation View

1. **Executable release:** below is an overview of the executable release.



Exp-6: Component and Interprocess Communication Design for WMS

The architecture team has decided to use the CORBA standard for the design of the interprocess communication mechanisms and the software components involved in WMS.

Figure 1 depicts the new class diagram describing the static structure of the classes participating to the Report weather data scenario of the Collect data use case.

We assume that each weather station is connected to 3 instances of each kind of instruments (e.g. 3 barometers, 3 ground thermometers etc.). We assume also for the sake of simplicity that there are only two weather stations.

1. Refine the class diagram in order to provide a new diagram consisting only of corresponding CORBA interfaces and types definitions. More specifically your diagram must include:

    - CORBA Interfaces

    - Data types definitions for the parameters and arguments involved in the operations using Rose-CORBA stereotyped classes (see also Lab 3).

    - Definition and documentation when needed (using Rose) of the relationships among the different elements involved.

2. Based on the modularization strategy adopted in previous Labs, generate using Rose the IDL code corresponding to the refined class diagram obtained previously. The generated IDL code should consist of at least three modules corresponding to the different subsystems (for example Sensor, Station, and DataCollection).

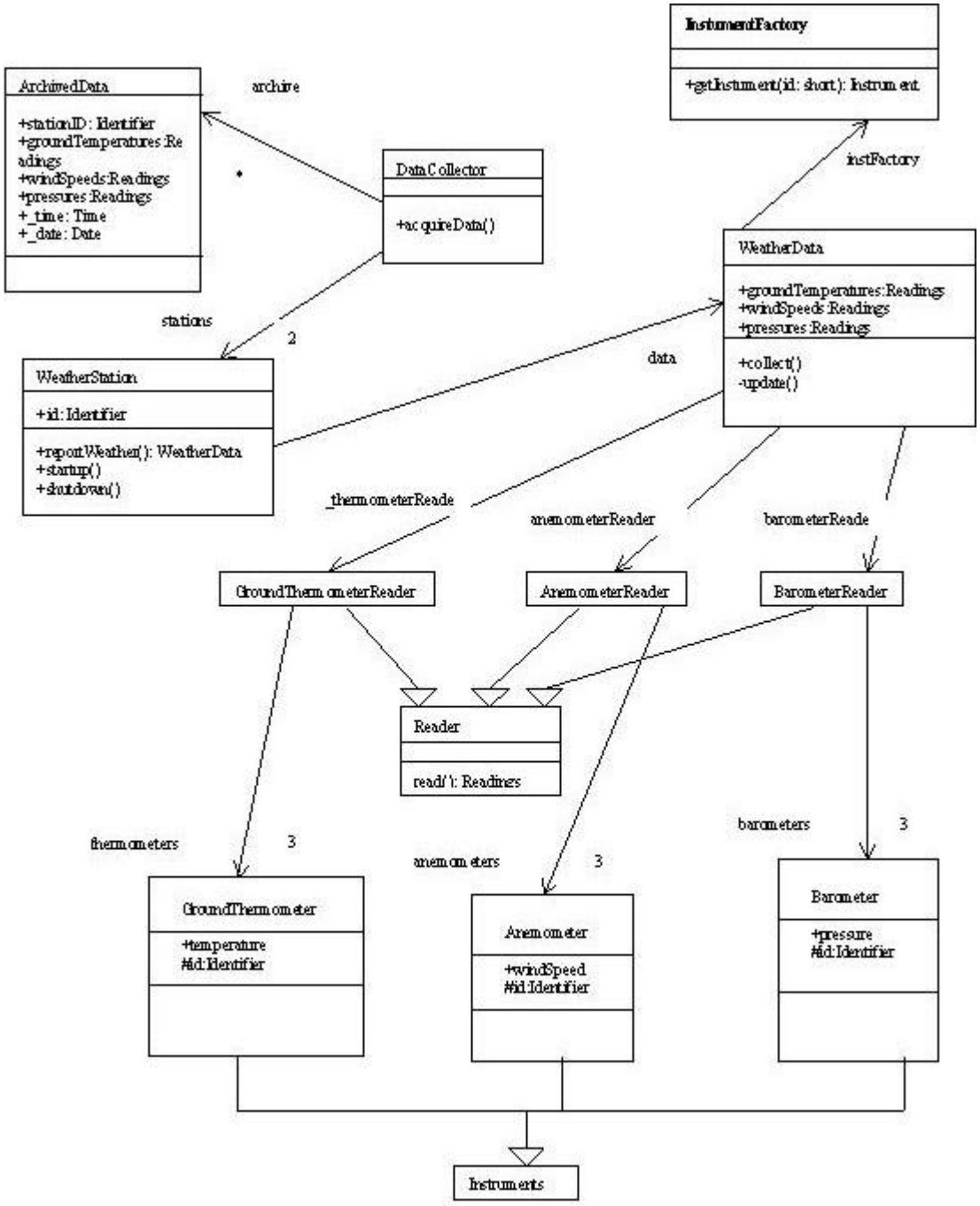


Figure 1: Class Diagram

Exp 7

Implentation of weather Mapping System

The project consists essentially of implementing in Java or C++ some of the CORBA objects identified for the Weather Mapping System, deploying them, and then testing them using a Java or C++ Client. The implementation will be based on the software architecture designed during the lab sessions.

1. Based on the IDL codes generated in Lab 5, implement in C++ and/or Java. You may implement weather instruments as simple random number generators. Use the following range for the weather readings:

- Temperature: [-80oC, +70oC] (oC: Degree Celsius)
- Pressure: [800,1100mbar] (mbar: millibars)
- Wind Speed: [0, 250mph] (e.g. mph: miles per hour).

2. Deploy copies of your packages on two different machines. Each server runs on a separate machine, and represents a remote weather station.

3. Implement the data collection client; the data collection client is in principle deployed on a central and separate computer. The data collection client is a CORBA client that will request and store (in a file or database) weather data from the two remote weather stations. The data collection client may access a remote station by first obtaining a reference to a Station instance from a corresponding server. For testing purpose, the data collection client must provide a user interface through which viewers can access collected weather data. For example, the user can select maximum, minimum, or average of selected weather parameters (e.g. temperature, pressure etc.) for a given location (e.g., weather station). Deploy the data collection client on a third machine.

4. Deploy and test your application. In order to simulate the different weather stations, you'll deploy copies of your components on two different machines (each running on different machine). You'll also deploy your client on a separate computer (e.g., a third one).

Notes: You must provide with your report (a hard copy) of the source code of the components and the client programs. You don't need to provide the source code generated automatically (e.g., skeleton, stubs etc.) by the IDL compiler.

Marking Guidelines

1. Report (30%)

1) Overview of your application (2 or more pages): purpose of the application, how it works (architecture, design), difficulties encountered during development and solutions adopted (10%).

2) Sensor component source code (5%).

3) Station component source code (5%).

4) Data collection client source code (10%).

2. Demo (70%) [15 Minutes]

1) System Works properly, including the connections between the servers and the CORBA client (40%).

2) Interface Design: present the minimum functionalities of client side (5%).

3) Instruments implementation (follow the guidelines given) (5%).

4) Proper data storage at the client (e.g. file or database) (5%).

5) Five-minute presentation (15%).

Design Patterns Lab

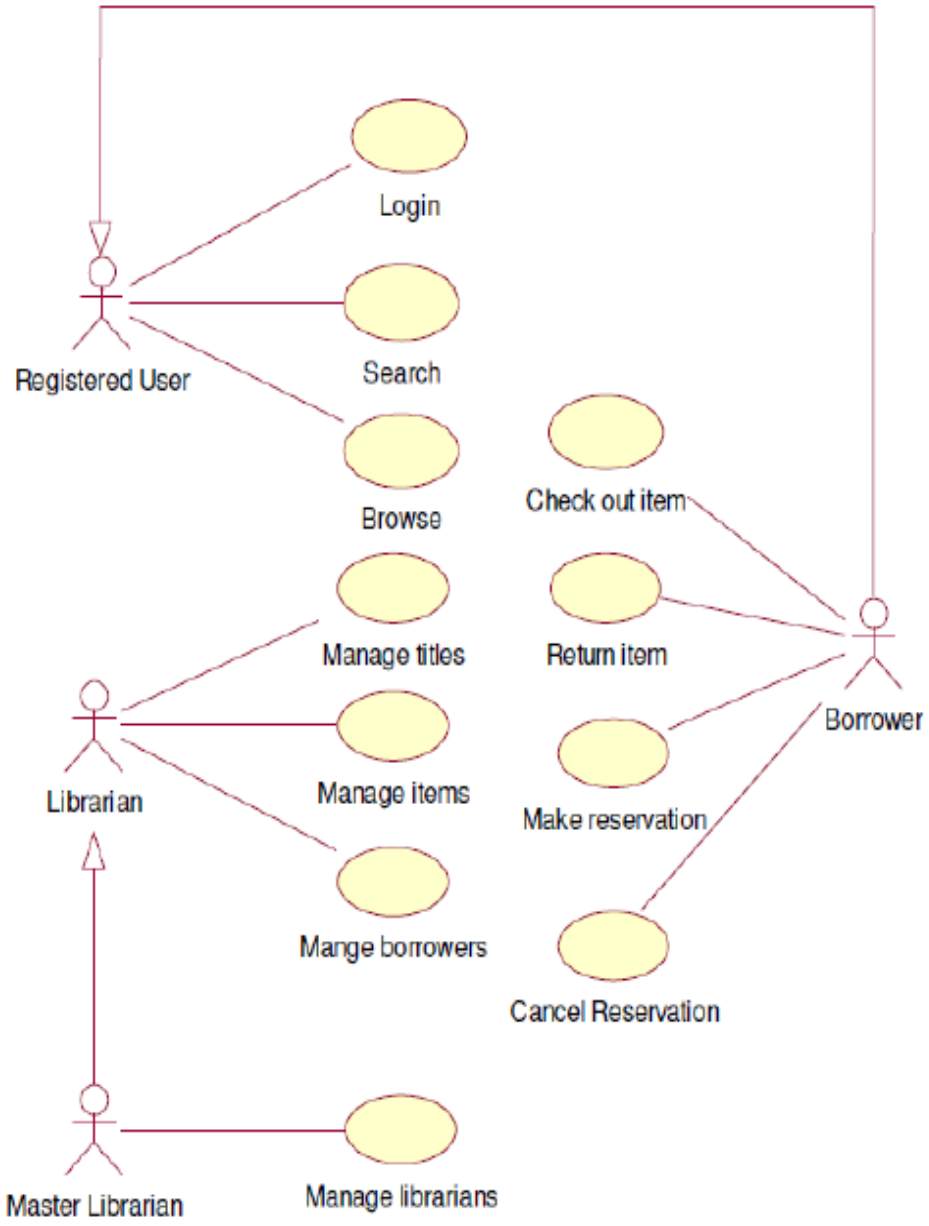
Exp 1:

Use case Diagram for Librarian Scenario

Solution:

Library Management System (LMS)

LMS Login Use Case Diagram:



Exp 2:

a)Using UML implement Factory pattern.

Theory:

- Defines an interface for creating objects but let sub-classes decide which of those instantiate.
- Enables the creator to defer Product creation to a sub-class.
- Factory pattern is one of the most used design pattern in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Intent:

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As:

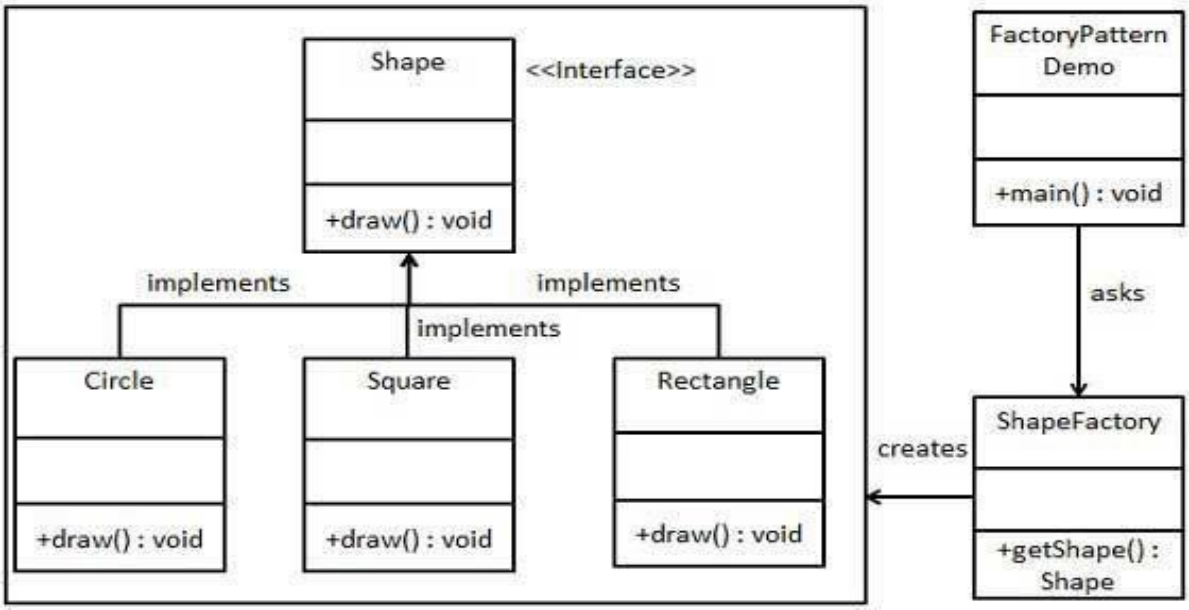
- Virtual Constructor .

Applicability:

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

CLASS DIAGRAM :



Exp 2:

b)Write a program to implement abstract factory.

Theory:

- Create instances of classes belonging to different families.
- Abstract Factory patterns works 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.

Intent:

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also Known As: Kit

Applicability:

Use the Abstract Factory pattern when

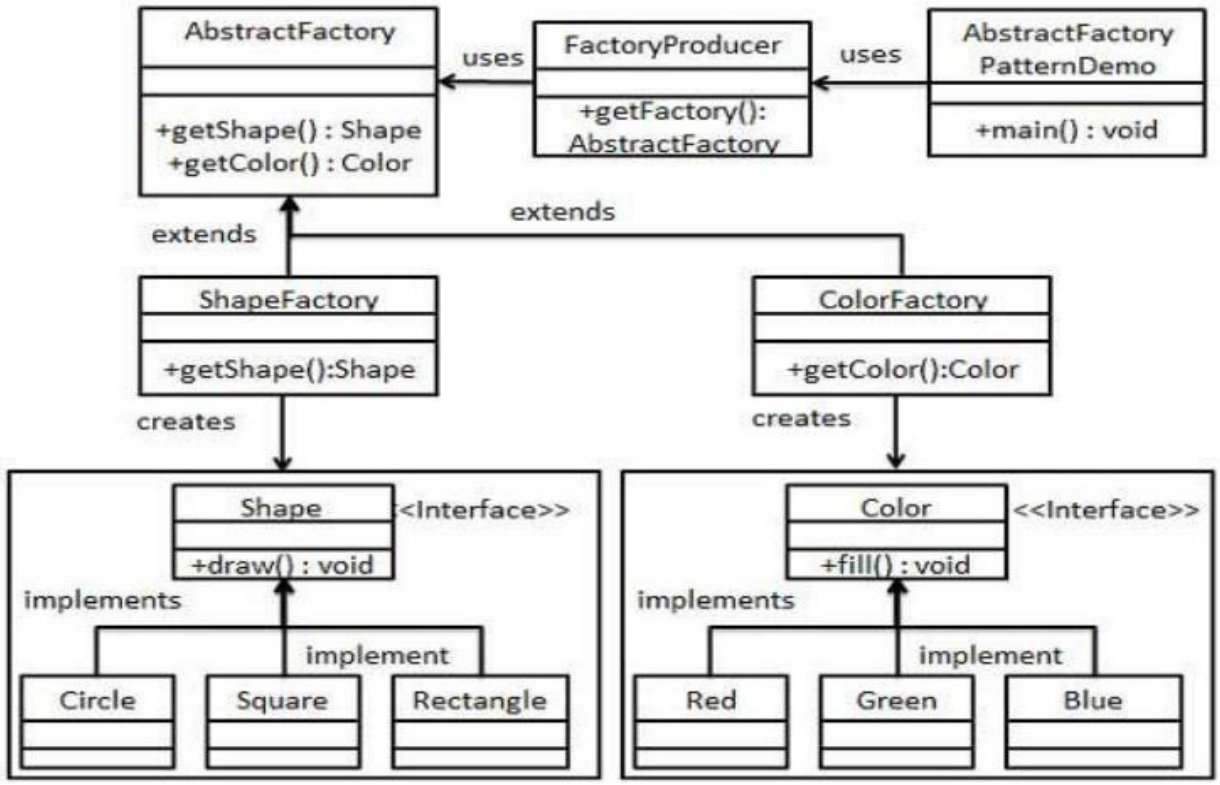
· a system should be independent of how its products are created, composed, and represented.

· a system should be configured with one of multiple families of products.

· a family of related product objects is designed to be used together, and you need to enforce this constraint.

· you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Class Diagram:



Exp 3:

Using UML design Adapter-class Design pattern

Solution:

This pattern is easy to understand as the real world is full of adapters. For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

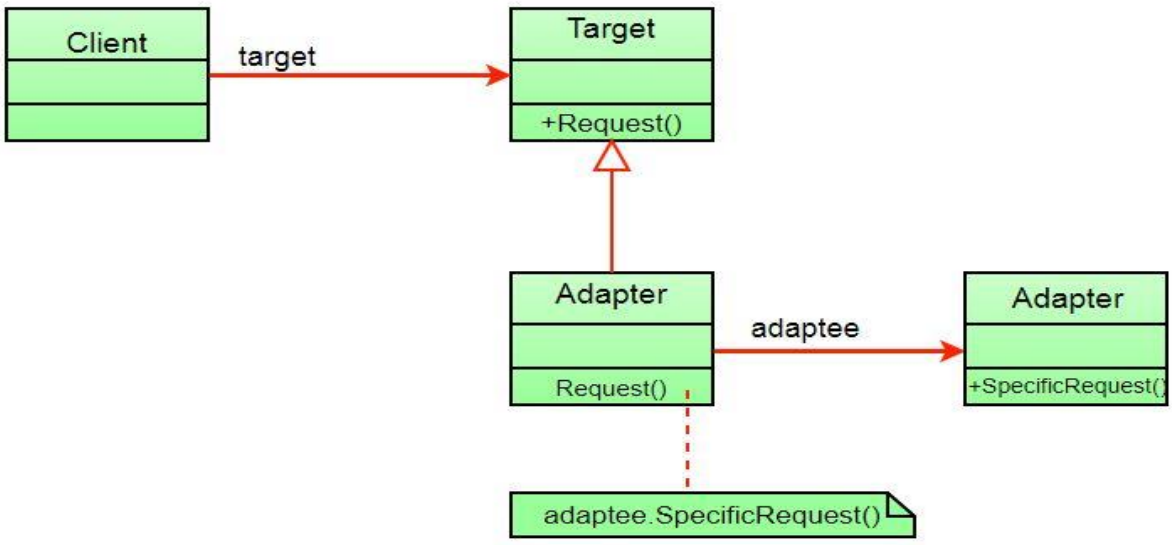
To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter's presence.

Definition:

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Class Diagram:



Exp 4:

Using UML design Adapter-object Design pattern

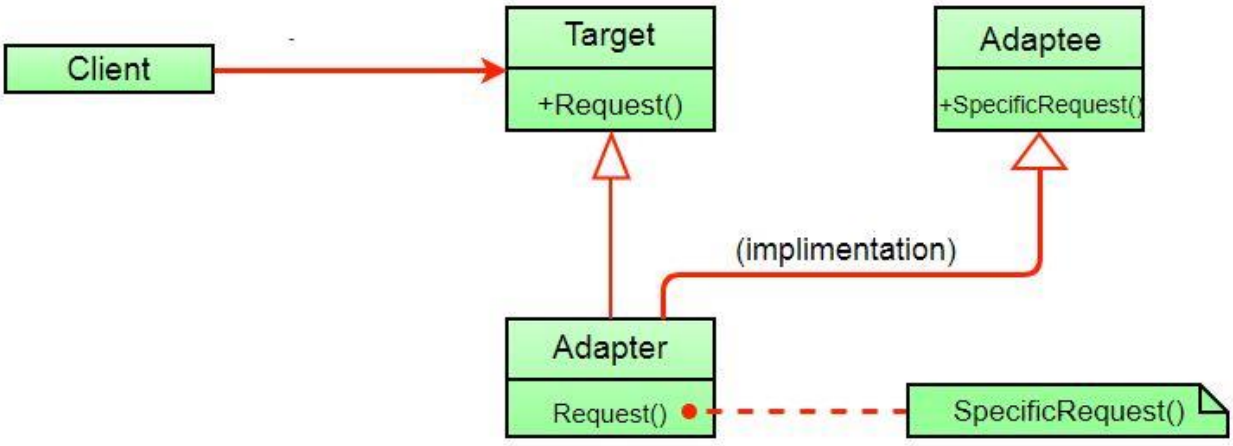
Solution:

The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

ObjectAdapterVsClassAdapter:

The adapter pattern we have implemented above is called Object Adapter Pattern because the adapter holds an instance of adaptee. There is also another type called Class Adapter Pattern which use inheritance instead of composition but you require multiple inheritance to implement it.

Class diagram of Class Adapter Pattern:



Here instead of having an adaptee object inside adapter (composition) to make use of its functionality adapter inherits the adaptee.

Since multiple inheritance is not supported by many languages including java and is associated with many problems we have not shown implementation using class adapter pattern.

The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

Exp 5:

Using UML design Strategy Design pattern

Solution:

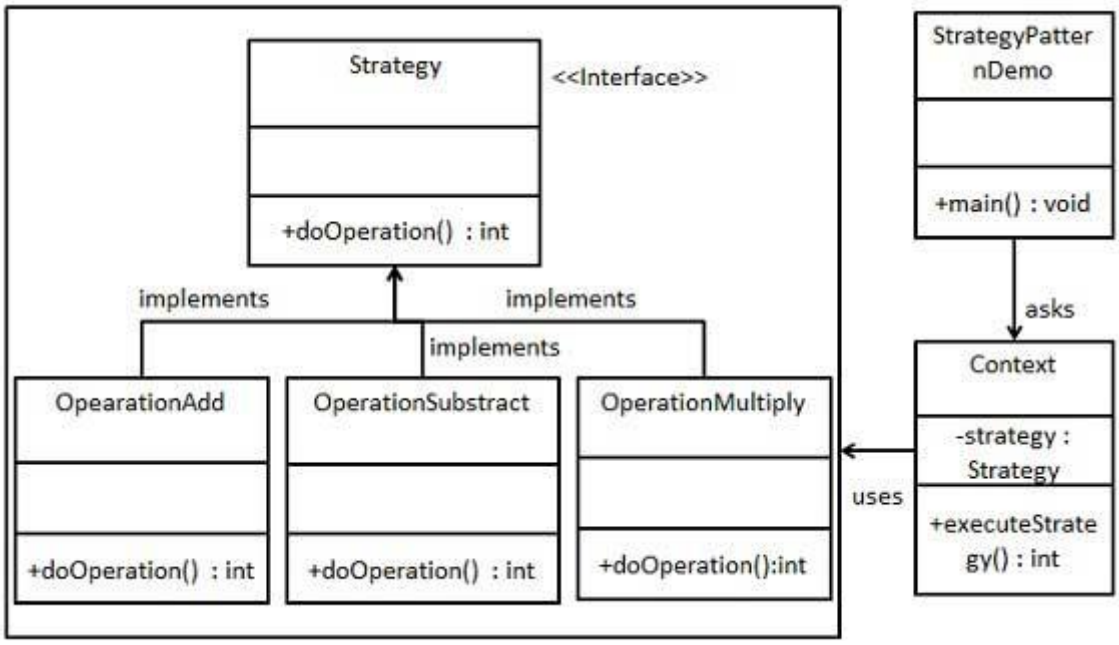
In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

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.

Strategy Pattern Demo, our demo class, will use Context and strategy objects to demonstrate change in Context behavior based on strategy it deploys or uses.



Exp 6:

Using UML design Builder Design pattern

Solution:

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

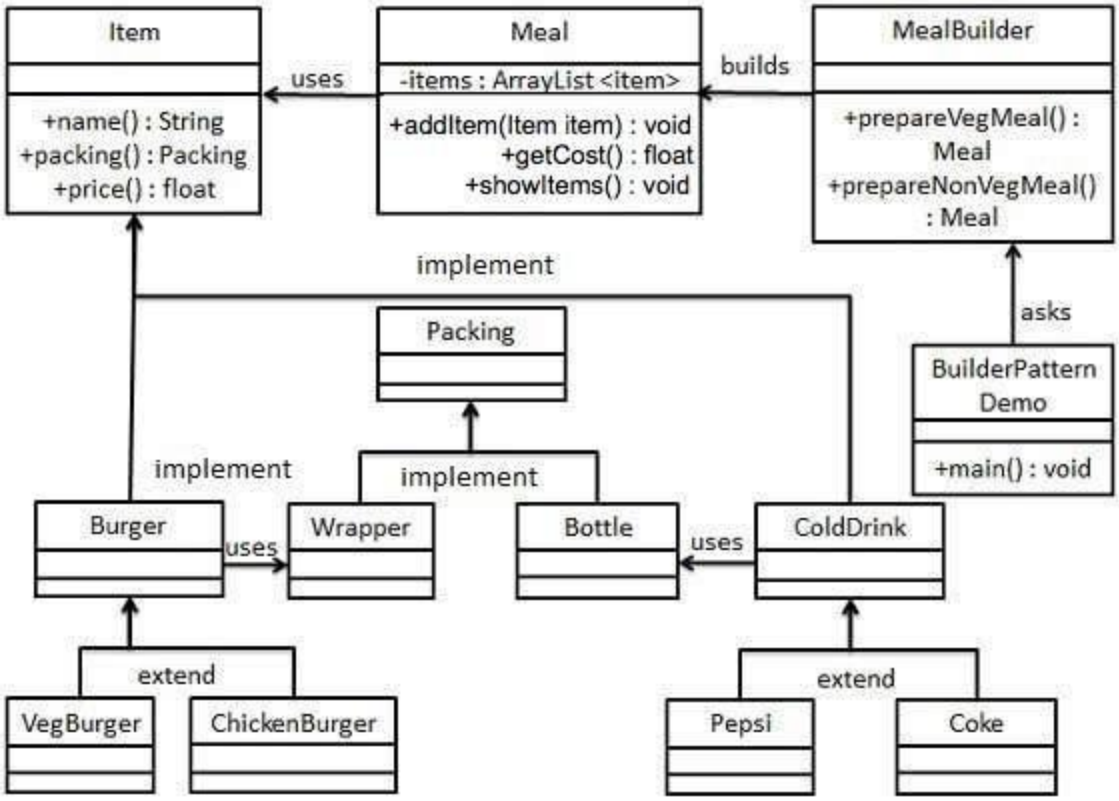
A Builder class builds the final object step by step. This builder is independent of other objects.

Implementation

We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.

We are going to create an Item interface representing food items such as burgers and cold drinks and concrete classes implementing the Item interface and a Packing interface representing packaging of food items and concrete classes implementing the Packing interface as burger would be packed in wrapper and cold drink would be packed as bottle.

We then create a Meal class having ArrayList of Item and a MealBuilder to build different types of Meal objects by combining Item. BuilderPatternDemo, our demo class will use MealBuilder to build a Meal.


Item
+name() : String
+packing() : Packing
+price() : float
uses
Meal
-items : ArrayList <item>
+addItem(Item item) : void
+getCost() : float
+showItems() : void
builds
MealBuilder
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
implement
asks
Packing
BuilderPattern Demo
+main() : void
implement
implement
Burger
uses
Wrapper
Bottle
uses
ColdDrink
extend
extend
VegBurger
ChickenBurger
Pepsi
Coke

Exp 7:

Using UML design Bridge Design pattern

Solution:

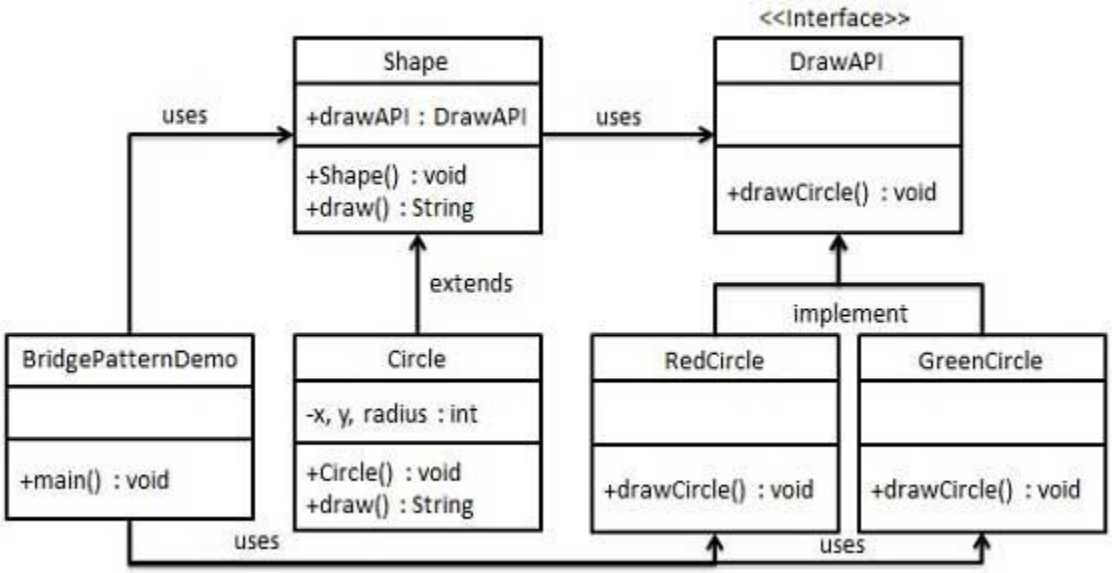
Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes.

Implementation

We have a DrawAPI interface which is acting as a bridge implementer and concrete classes RedCircle, GreenCircle implementing the DrawAPI interface. Shape is an abstract class and will use object of DrawAPI. BridgePatternDemo, our demo class will use Shape class to draw different colored circle.



Exp 8:

Aim: Using UML design Decorator Design pattern

Theory:

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

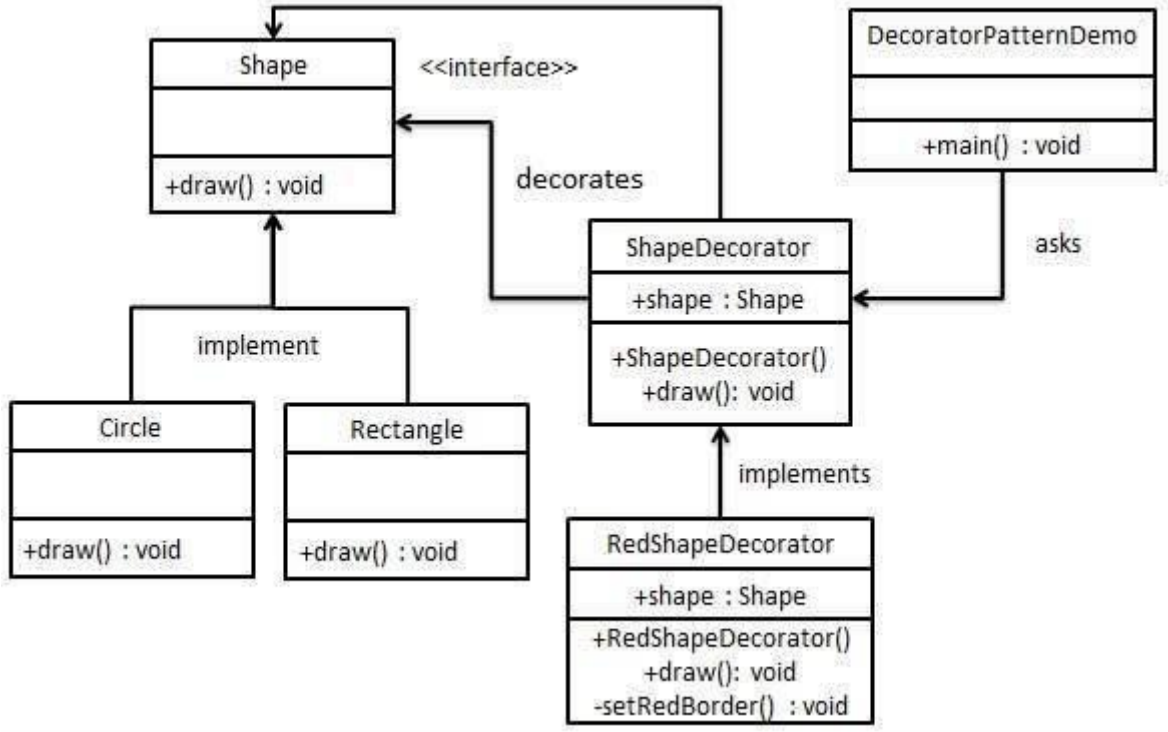
Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Also Known As Wrapper **Applicability** Use Decorator

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by sub classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for sub classing.

Class Diagram:



Exp 9:
Aim: Write a Program to design chain of responsibility pattern.
Theory:

As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.

In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Example: ATM(rupees of 1000,500,100 etc)

Intent

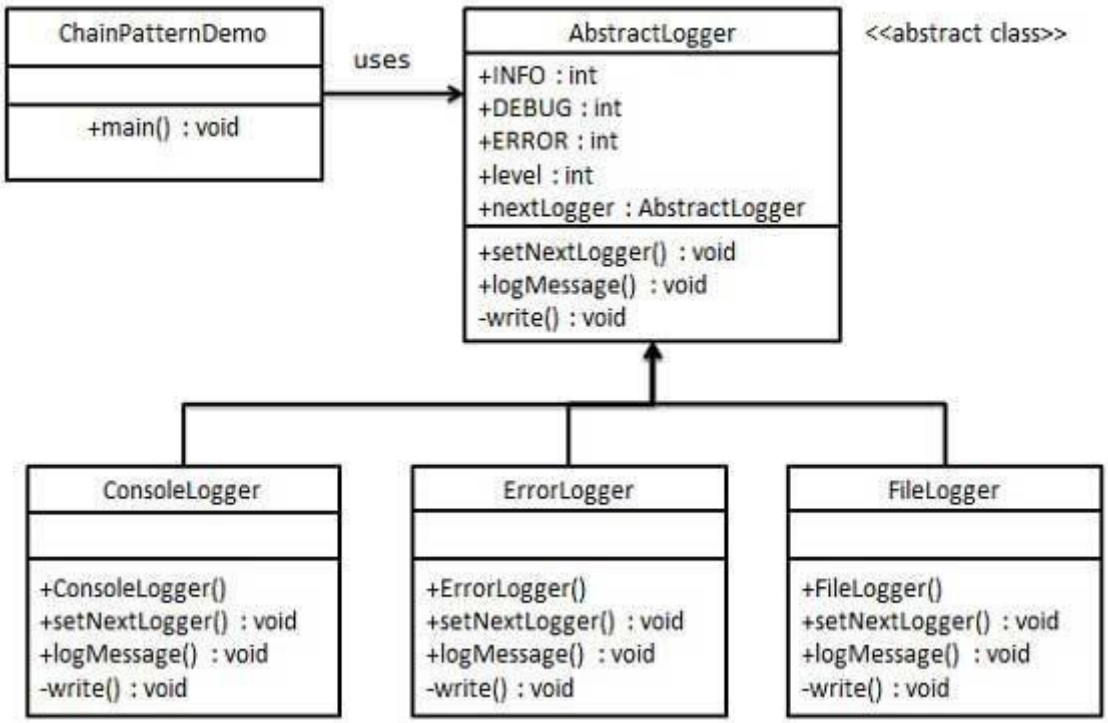
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Applicability

Use Chain of Responsibility when

- more than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

Class Diagram:



Exp 10:

Using UML implement Flyweight Design Pattern

Solution:

Flyweight pattern is one of the structural design patterns as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 10^5). One important feature of flyweight objects is that they are immutable. This means that they cannot be modified once they have been constructed.

Why do we care for number of objects in our program?

- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
- Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

In Flyweight pattern we use a HashMap that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.

To understand Intrinsic and Extrinsic state, let us consider an example.

Suppose in a text editor when we enter a character, an object of Character class is created, the attributes of the Character class are {name, font, size}. We do not need to create an object every time client enters a character since letter ‘B’ is no different from another ‘B’ . If client again types a ‘B’ we simply return the object which we have already created before. Now all these are intrinsic states (name, font, size), since they can be shared among the different objects as they are similar to each other.

Now we add to more attributes to the Character class, they are row and column. They specify the position of a character in the document. Now these attributes will not be similar even for same characters, since no two characters will have the same position in a document, these states are termed as extrinsic states, and they can’t be shared among objects.

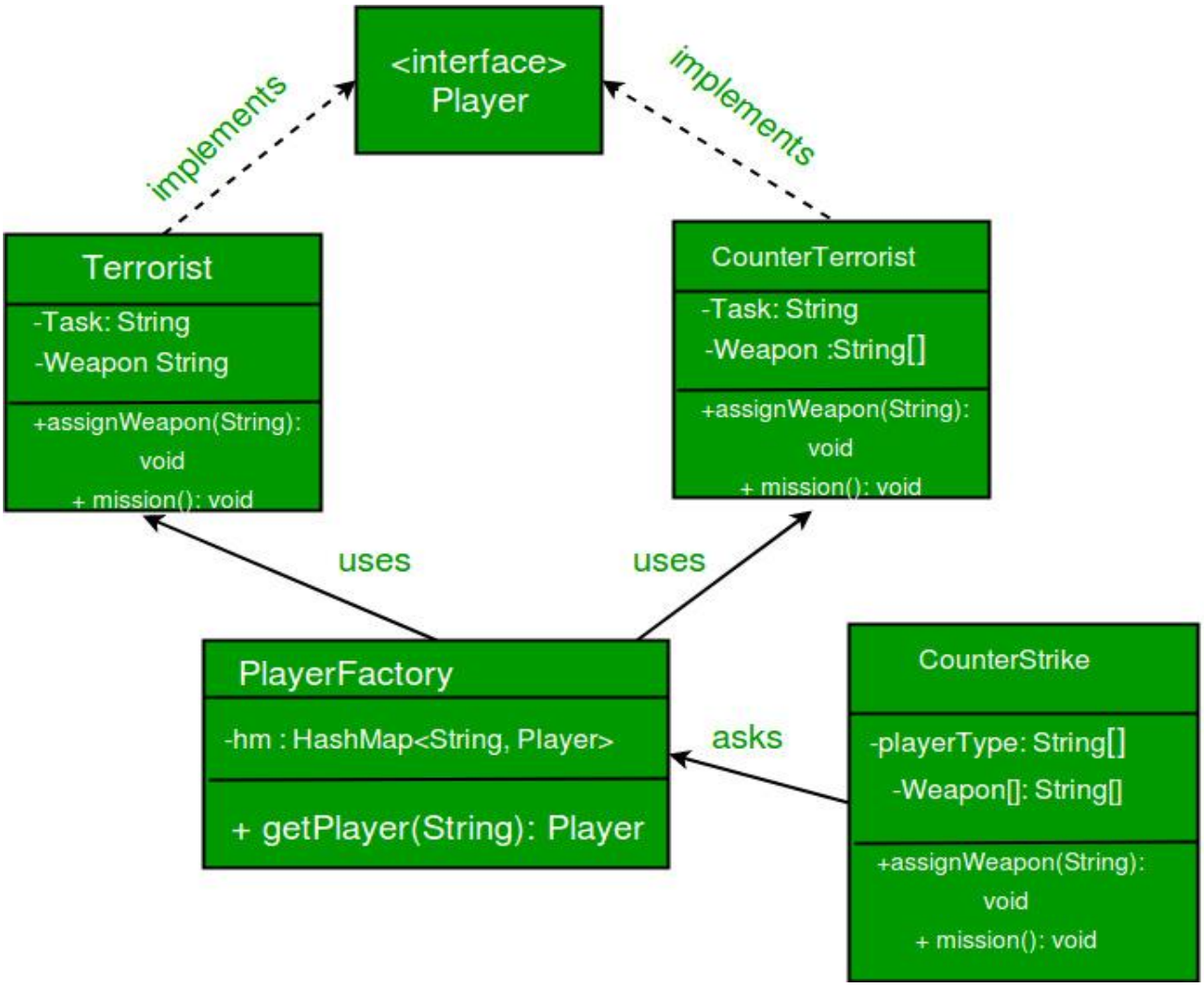
Implementation : We implement the creation of Terrorists and Counter Terrorists In the game of Counter Strike. So we have 2 classes one for Terrorist(T) and other for Counter Terrorist(CT). Whenever a player asks for a weapon we assign him the asked weapon. In the mission, terrorist's task is to plant a bomb while the counter terrorists have to diffuse the bomb.

Why to use Flyweight Design Pattern in this example? Here we use the Fly Weight design pattern, since here we need to reduce the object count for players. Now we have n number of players playing CS 1.6, if we do not follow the Fly Weight Design Pattern then we will have to create n number of objects, one for each player. But now we will only have to create 2 objects one for terrorists and other for counter terrorists, we will reuse then again and again whenever required.

Intrinsic State : Here 'task' is an intrinsic state for both types of players, since this is always same for T's/CT's. We can have some other states like their color or any other properties which are similar for all the Terrorists/Counter Terrorists in their respective Terrorists/Counter Terrorists class.

Extrinsic State : Weapon is an extrinsic state since each player can carry any weapon of his/her choice. Weapon need to be passed as a parameter by the client itself.

Class Diagram :



Exp 11:

Using UML implement FACADE PATTERN

Solution:

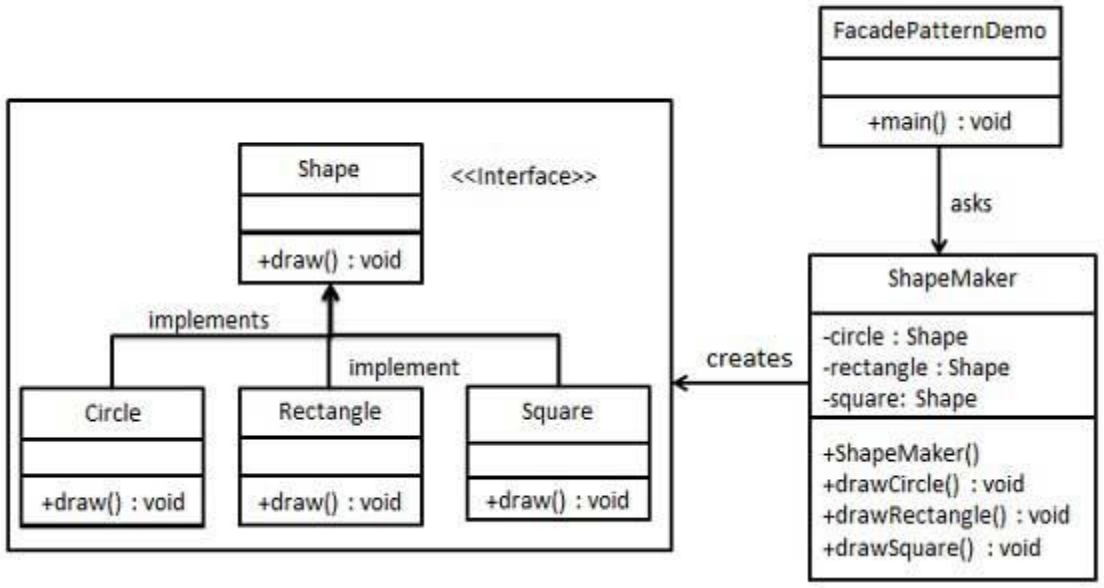
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Implementation

We are going to create a Shape interface and concrete classes implementing the Shape interface. A facade class ShapeMaker is defined as a next step.

Shape Maker class uses the concrete classes to delegate user calls to these classes. FacadePatternDemo, our demo class, will use ShapeMaker class to show the results.



Exp 12:

Using UML design Iterator Design pattern

Theory:

Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Iterator pattern falls under behavioral pattern category.

Intent

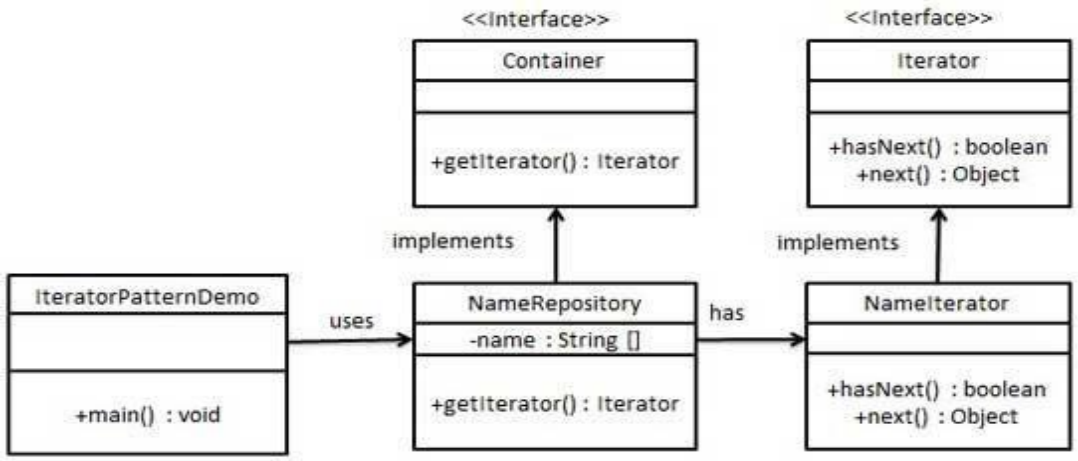
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also Known As Cursor Applicability

Use the Iterator pattern:

a. to access an aggregate object's contents without exposing its internal representation.
b. to support multiple traversals of aggregate objects.
c. to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Class Diagram:



Exp 13:

Using UML design mediator Design pattern.

Theory:

- Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling. Mediator pattern falls under behavioral pattern category

Intent

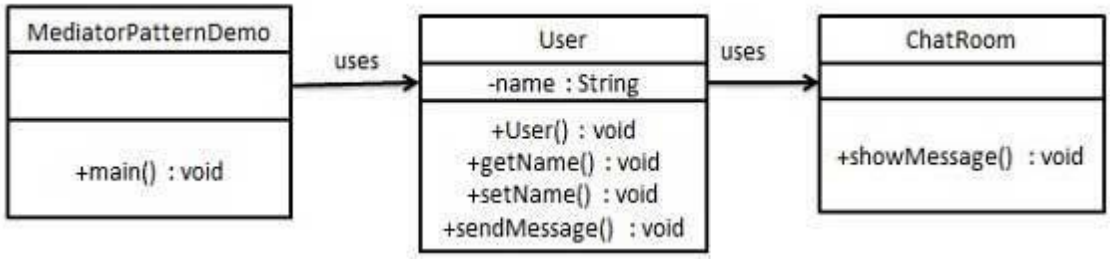
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

- Applicability

  Use the Mediator pattern when

  - a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
  - reusing an object is difficult because it refers to and communicates with manyother objects.

Class Diagram:



Exp 14:

Aim: Using UML design proxy design pattern.

Theory:

- In Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.
- In Proxy pattern, we create object having original object to interface its functionality to outer world

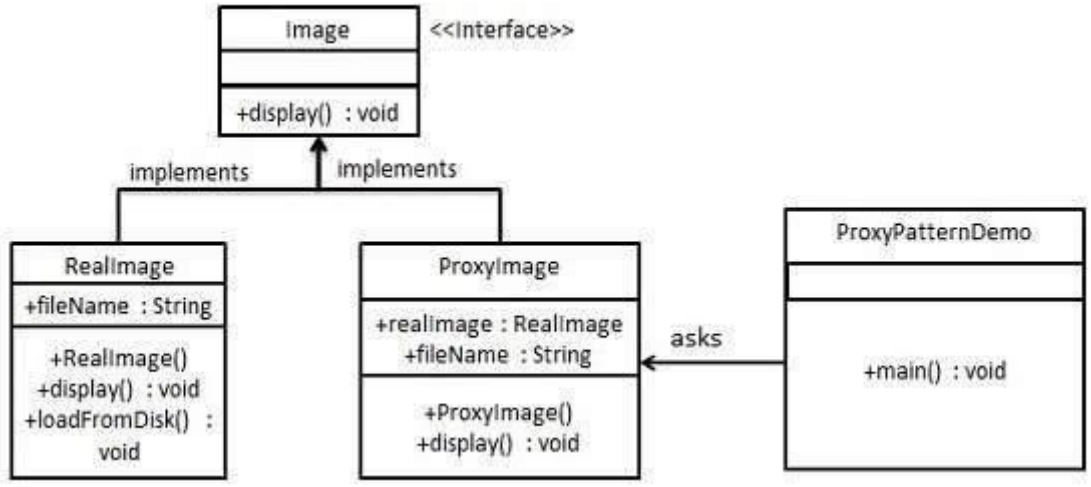
Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Also Known As:Surrogate

Applicability:

- A remote proxy provides a local representative for an object in a different address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose.
- virtual proxy creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.

Class Diagram:



Exp 15:

Aim: Using UML Design visitor Design pattern.

Theory:

In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. This pattern comes under behavior pattern category. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Applicability

Use the Visitor pattern when

a. an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that dependon their concrete classes.

b. many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.

c. the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

Class Diagram: