**Project Report**

|  |  |
| --- | --- |
| Project Title | Design and Develop a website portal using Liferay |
| Qualification Name (NICF) | Advanced Certificate in Web Development using Platforms |
| Product Name | Triple-A (AAA) Company |
| Module Name (NICF) | Web Development using Platforms |

|  |  |  |  |
| --- | --- | --- | --- |
| Student name | | Assessor name | |
| Syukur Sidiq Nur Alam | |  | |
| Date issued | Completion date | | Submitted on |
|  |  | |  |
|  | |  | |
| Project Title | Design and Develop a website portal using Liferay | | |

|  |
| --- |
| Learner declaration |
| I certify that the work submitted for this assignment is my own and research sources are fully acknowledged.  Student signature: Date: |

**Content**

1. Project Background
2. Project Objectives
3. Task 1
4. Task 2
5. Task 3
6. Task 4

**Project Scenario:**

Triple-A (AAA) company is the best web host solutions service company located in downtown of Indonesia. It provides all types of services which include domain name registration, Shared Hosting, Reseller Hosting, Cloud Hosting, VPS Hosting, Dedicated Hosting, and Colocation service, etc.

**Project Objective:**

The objectives of this project are:

• To create the AAA Services company site using Liferay features for site creation and content creation

• To assign 2 users, Administrator and site member to AAA Services company site and the permissions

• To develop the customer's data management application or portlet with OOP concepts using Liferay Framework

• To prepare a class diagram for the application structure using a UML tool like draw.io, lucidchart or creately.

• To identify and investigate the design pattern used in the customer's data management application or portlet

**Functional Requirements**

The AAA website consists of the following Key pages

1. Home Page

2. Registration Page

3. Login Page / Sign In

4. Our services Page

a. Domain name service page

b. Shared Hosting service page

c. Reseller Hosting service page

d. Cloud Hosting service page

e. VPS Hosting service page

f. Dedicated Hosting service page

g. Colocation service page

5. Contact us Page

6. About us Page

7. Terms and Conditions Page

Customers can purchase their services from AAA’s branch offices and needs to provide personal information such as name, email, address, national identity card number and contact number. Customers' data is a very important factor for AAA Company. To maintain and manage all of their customer information, AAA has been decided to develop an application with an object-oriented approach.

An application is required to keep track of their customer’sinfo and services given to customers. The well-organized customer data can help the company to select the correct recipients for promotions and new services.

The Scope of the Project is to build a Liferay Framework website with customer’s data management application.

The overview of the project is as below There are two types of users in this portal.

They are 1. Administrator 2. Site Member (Staff)

Administrator should be able to perform following functions in the portal,

1. Manage all of the site contents and pages.

2. Update the theme and layout.

3. Manage the customer data portlet and control permission.

4. Manage all user (Site Member and Customer) roles and permission

Site Member should be able to perform following functions in the portal,

1. Update AAA company site contents.

2. Add and Update the AAA’s customer data.

TOOLS

Liferay for build Design and Code

 Chrome for show the Result

Snipping Tool For take Picture

**Task 1**

**Program Paradigm and Design Pattern**

**1.** Identify and explain the characteristics of the object-orientated paradigm and relationship between the various classes from a given code scenario

1. **Explain briefly Object-Oriented Programming Paradigm.**

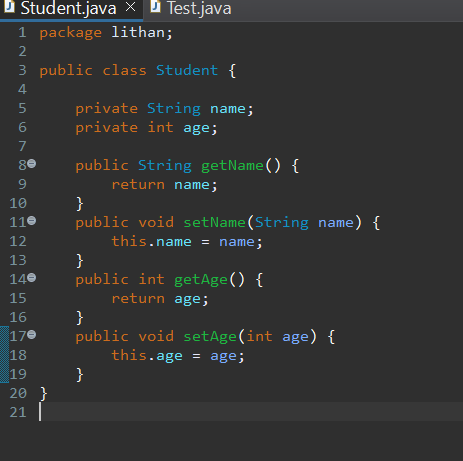
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code to manipulate that data. The key idea behind OOP is to organize code into reusable and modular units (objects), which can communicate with one another to achieve a specific goal.

**(b) Examine characteristics of Object-Oriented Programming Paradigm**

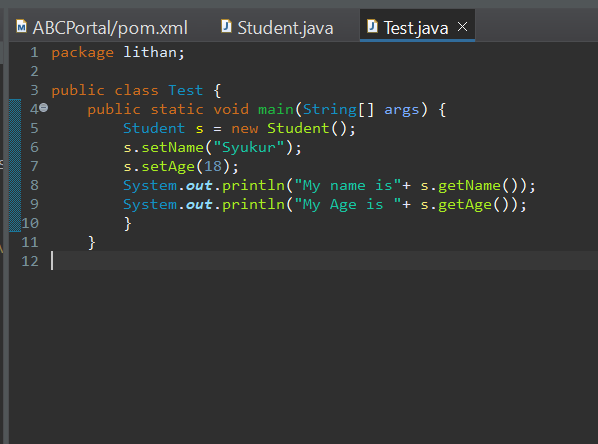
1. Encapsulation

Encapsulation is the wrapping of data or hiding personal data from an object so that it cannot be accessed from other objects. Can only access using the current class, getter and setter methods.

Make Student.java and make the object. Make it private



Make Test.java to test the encapsulation works or not



1. Polymorphism

Polymorphism allows different objects to be treated as if they were the same type of object, even if they are actually different. This can be useful when working with collections of objects, as it allows you to treat them all as if they were the same type of object, even if they have different behaviors.



1. Constructors/ Destructors

Constructor is a special method that is used for initialization when creating an object. The constructor is called as soon as the new object is created. The feature of the constructor is that the method name exactly matches the class name.

Remember when you creating constructor the name must same with the class. This is ConstructorExample.java with result

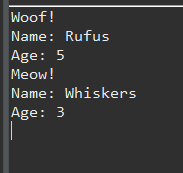


1. Abstract

An abstract class serves as a blueprint for other classes to follow. It defines a set of common methods and properties that its subclasses must implement. However, it cannot be instantiated on its own. Instead, it must be extended by a concrete class that provides implementations for all its abstract methods.

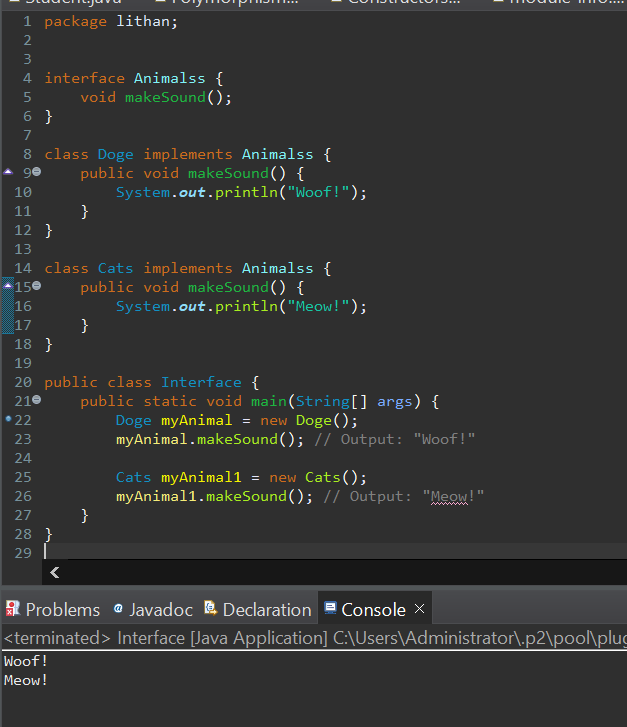


This is the result



1. Interface

interface is a programming construct that defines a set of methods or functions that a class must implement. It is a contract that specifies what the class should do but not how it should do it. Interfaces define a set of public methods that can be called by other objects without revealing the internal workings of the object. They allow for loose coupling between different objects, promoting code reusability. By programming to an interface instead of a specific class, you can change the implementation of an object without affecting other objects that depend on it.



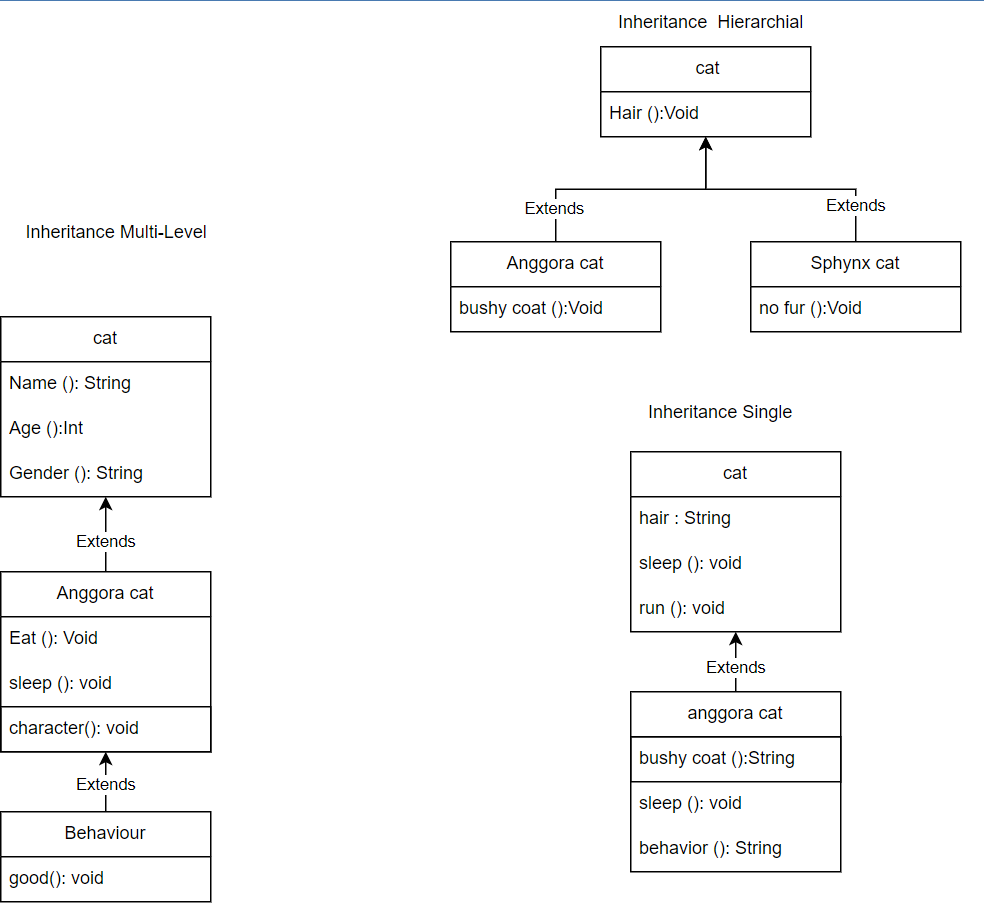
1. What are the differences between abstract classes and interfaces?

|  |  |
| --- | --- |
| Abstract | Interface |
| Use ‘abstract’ keyword to define as abstract class | Use ‘interface keyword to define as interface class |
| Can contain both abstract and non-abstract methods. | It can only contain abstract methods. |
| Allow data fields and constant to be defined | No data fields |
| Can declare constants and instance variables. | Can only declare constant. Implicitly the variables declared in the interface are public, static and final. |
| Can have constructor | Cannot have constructor |
| An abstract class can only extend one other abstract class. | An interface can extend one or more other interfaces. |
| Can have access modifiers. | Does not have access modifiers and can only be defined as public |

**(c) Examine Class Relationships using UML diagrams**

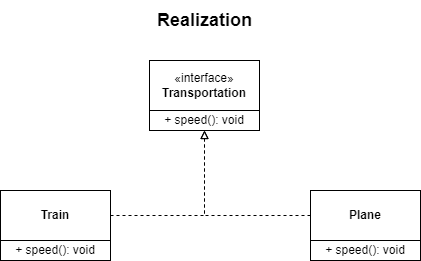
i. Inheritance

The "is-A" relationship between two classes, primarily the parent class and the child class, which inherits the parent's structure and behavior, is represented by inheritance, also known as generalization. A solid line arrow clearly illustrates the relationship (hollow arrowhead). The three inheritance types are demonstrated below:



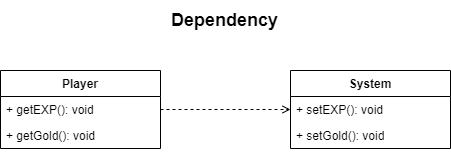
ii. Realization

Realization is seen in interfaces when the blueprint class is paired with an object to carry out each detail. A dashed line with a hollow arrowhead can be used to indicate a relationship.



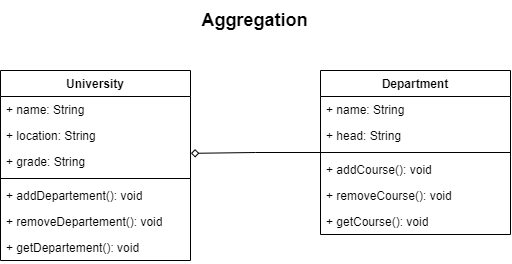
iii. Dependency

Dependency is when class A is dependent on class B. Therefore, when class B changes, class A will also be affected by the change. A dashed line with an open arrow can represent this relationship.



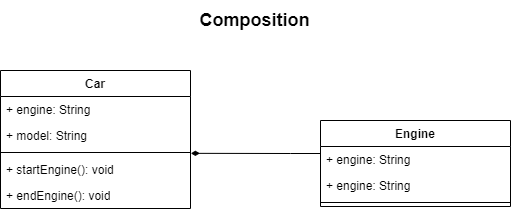
iv. Aggregation

Aggregation is the single point of control where it aggregates or assembles a group of objects that is part of the object which aggregate. The part of the relationship can be represented by a solid line with an unfilled diamond at the parent end connected to the child class. In this case, if the parent is deleted, the child class can still exist independently.



v. Composition

Composition is the stronger form of aggregation. In this case, if the source or the aggregator is destroyed, then part of it is also destroyed with it and cannot stand by itself. The relationship is displayed with a solid line and the filled diamond at the associated/source end connected to the composite class.



**(d) Examine and list out class relationships which are used in project scenario.**

**1. Inheritance:**

Administrators, site members, and customer classes are

All common variables and methods use inheritance from User.

A new superclass with generic variables and methods.

* + - 1. **Association :**

Administrators can only manage customers and site members, and site members can only manage customers, so the relationships between administrators and customers, administrators and site members, and site members and customers are one-way.

* + - 1. **Composition :**This class is used because there is a relationship between the Customer and Service classes. In other words, when something changes in the Service class, the Customer class changes.

**2. Identity the suitable design patterns based on the project scenario.**

1. What are Creative Design Patterns?

The process of creating objects is at the heart of creative design patterns. These design patterns are used when a choice needs to be made during class instantiation (that is, create an object of the class).

• Factory method pattern

To create an object, simply define an interface or abstract class and let subclasses choose which class to instantiate.

• Abstract factory pattern

Defines an interface or abstract class for assembling a family of related (or dependent) objects without mentioning a specific concrete subclass. In other words, an Abstract Factory allows a class to return the factory of the class.

• Singleton pattern

Create a single instance class with a single global access point. In other words, a class should allow only a single instance to be created and all other classes to use a single object.

• Prototype pattern

Duplicating an existing object allows customization and eliminates the need to create a new object.

• Builder pattern

Build complex objects from simple objects using a step-by-step approach

• Object pool pattern

Contains a certain number of articles. When an object is taken out of the pool, it cannot be used in the pool until it is put back in. The lifetime of each object in the pool consists of creation, validation, and destruction.

1. Identify suitable design pattern for “AAA Portlet”

The Most suitable design pattern for AAA Portlet is **The Factory Method Patern.**

1. What is structural design pattern?

**structural design pattern is a pattern that deals with object** composition to form larger structures or relationships between objects. The main goal of these patterns is to provide a way to compose objects to obtain new functionality and to provide flexibility and maintainability to the software system. Structural design patterns typically involve class and object composition, inheritance, and interface usage to provide solutions to common design problems.

Some common examples of structural design patterns include:

* Adapter pattern: This pattern allows the interface of an existing class to be used as another interface.
* Bridge pattern: This pattern decouples an abstraction from its implementation so that the two can vary independently.
* Composite pattern: This pattern allows you to treat a group of objects as a single object.
* Decorator pattern: This pattern adds new functionality to an existing object without altering its structure.
* Facade pattern: This pattern provides a simplified interface to a complex subsystem.
* Flyweight pattern: This pattern allows sharing of objects to reduce memory consumption.
* Proxy pattern: This pattern provides a surrogate or placeholder for another object to control access to it.

1. Identify suitable design pattern for “AAA Portlet”

The Most suitable design pattern for AAA Portlet is **The Facade Patern.**

1. What is behavioral design pattern?

Behavioral design patterns are a set of software design patterns that focus on how different objects in a system interact and communicate with each other to achieve a specific behavior or task. These patterns delegate responsibilities and promote collaboration among objects to accomplish a common goal.

Some common examples of behavioral design patterns include the Observer Pattern, Strategy Pattern, Command Pattern, Iterator Pattern, and State Pattern. These patterns help to make software systems more flexible, maintainable, and extensible by reducing the complexity and coupling between objects.

The Observer Pattern involves an object keeping track of its dependents and notifying them automatically of any state changes. The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable, enabling the selection of an algorithm at runtime based on the context. The Command Pattern encapsulates a request as an object, allowing clients with different requests to be parameterized, supporting undoable operations, and enabling requests to be queued or logged. The Iterator Pattern provides a sequential way to access the elements of an aggregate object without exposing its underlying representation. The State Pattern allows objects to alter their behavior when their internal state changes.

By implementing these behavioral design patterns, developers can create software systems that are more flexible, maintainable, and extensible, while also reducing the coupling and complexity between objects.

f) Identify suitable design pattern for “AAA Portlet”

The Most suitable design pattern for AAA Portlet is **The Template Patern.**

**3. Examine how the object-orientated paradigm and its key principle is identified in each of the design patterns.**

Object-oriented programming (OOP) is a programming style characterized by identifying classes of objects closely linked to the methods (functions) with which they are associated. On the other side, design patterns are reusable fixes for typical issues in software design. The foundation of many design patterns is the object-oriented paradigm.

Each design pattern references the object-oriented paradigm and its fundamental ideas of encapsulation, inheritance, polymorphism, and composition. Encapsulation is used by the Singleton Pattern, inheritance is used by the Factory Method Pattern, the Adapter Pattern uses polymorphism, the Decorator Pattern uses composition, and abstraction and encapsulation are used by the Observer Pattern.

Overall, these design patterns demonstrate how the object-oriented paradigm is focused on using classes and objects to organize and encapsulate code, while also emphasizing principles like inheritance, composition, polymorphism, and abstraction to create flexible and extensible systems. The relationship between the object-oriented paradigm and design patterns is one of interdependence. The object-oriented paradigm provides the foundation for design patterns, while design patterns help to demonstrate the usefulness of object-oriented concepts and highlight areas where improvements can be made.

**Task 2**

**System Design**

**1. Design and build class diagrams based on project scenarios using a UML tool.**

1. **Draw Class Diagrams showing Class-relationships**

**What is Class Diagram**

Class diagrams are an effective tool for describing various design patterns. In many cases, a class diagram is actually required to fully represent the key elements of a particular pattern. This is because class diagrams provide a complete representation of the static structure of the system while emphasizing key connections between classes, objects, traits and activities. By providing a conceptual model of the system in terms of entities and their relationships, class diagrams help designers better understand the overall architecture of the system and identify key areas for optimization and development. Incorporating classes, interfaces, relationships, collaborations, and constraints into well-designed class diagrams can withstand even the most difficult and complex real-world scenarios.

Modifiers are used to indicate visibility of attributes and operations.

1. ‘+’ is used to denote Public visibility (everyone)

2. ‘#’ is used to denote Protected visibility (friends and derived)

3. ‘-’ is used to denote Private visibility (no one)

By default, attributes are hidden and operations are visible.

**(b) Explain relationships between classes in No.(a)**

Guidance

The inheritance link between the two subclasses of the superclass Users, administrator and site member, is indicated by a solid line with a hollow point.

This is true since site members and administrators at AAA Company are two distinct categories of people.

Both of the two classes are subclasses of the super class Users as a result of their shared attributes of name and email as well as similarity in maintaining contents, registering, logging in, and logging out via the Generalization connection.

A single solid line is used to illustrate the relationship between the administrator and the customer because the administrator is in charge of managing the customers. Users of the website engage with clients since they are in charge of looking after them.

**Classes**

o **Admin class:** Manage everything in the portal

o **Site Member class:** Manage everything within the site (Triple-A AAA).

o **Customer class:** Write what customer class can do

o **Service class:** Write what service class can do

**Classes and Attributes**

o **Admin Attributes:** adminId, adminUsername, adminEmail, adminPassword

o **Site Member Attributes:** siteMemberId, siteMemberUsername, siteMemberEmail, siteMemberPassword

o **Customer Class Attributes:** Write customer attributes in Customer Entity

o **Service Class Attributes:** Write service attributes in Service Entity

**Classes and Methods**

o **Admin Methods:** register (), login (), …….

o **Site Member Methods:** addCustomer (), updateCustomer(), addContent(), updateContent(),….

o **Service Methods:** addService (), editService(),…………

o **Customer Methods:** ……

 **Class Diagram**

**2. Identify possible situations where design patterns would be beneficial and then develop the UML diagrams reflecting the design patterns.**

The factory method has the advantage of serving as the knowledge hub for creating

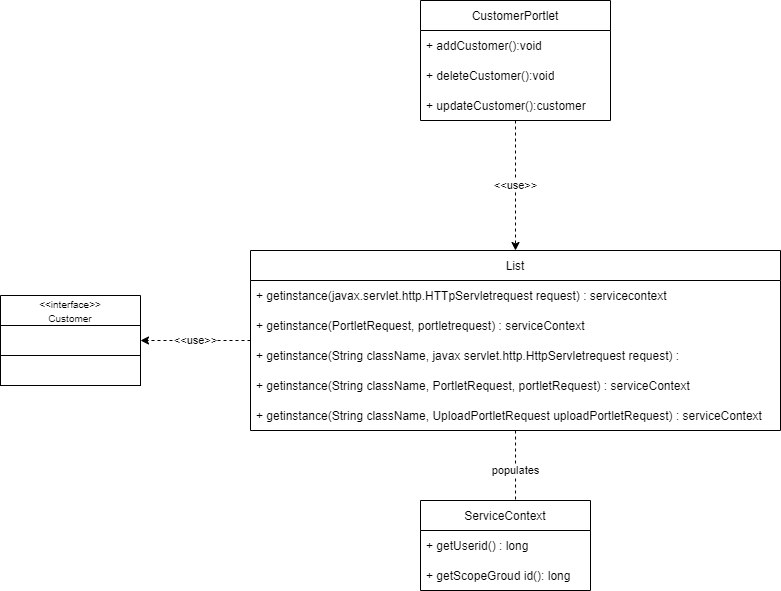
the necessary objects. Depending on the serviceContext object for Customer required

to obtain the appropriate user id to update the customer, the getInstance() function in

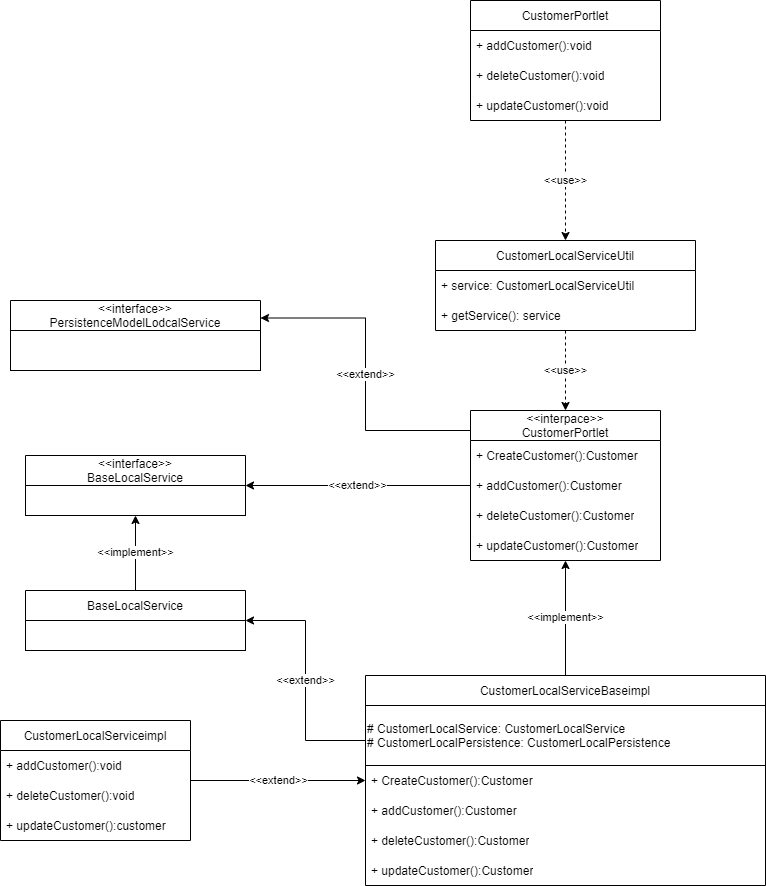
the serviceContext Factory class is reusable.

When creating the serviceContext object, the user that uses the Customer Portlet will

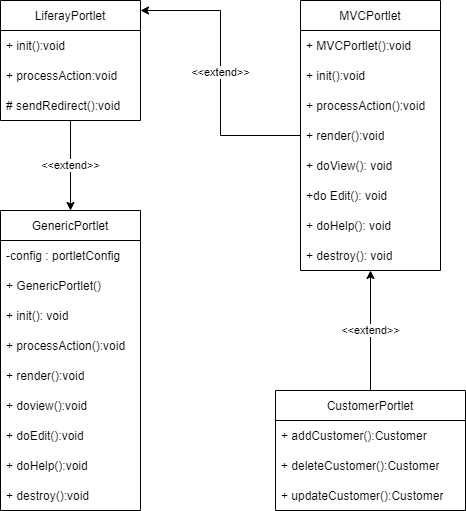
refer to the factory instead of understanding the convoluted construction logic.



the facade style is advantageous in terms of hiding or decreasing the complexity of the subclasses, The CustomerLocalServiceUtil class is the only point of access for the user to create the business logic for the firm using the CRUD operation, The mentioned class is thought of as a façade class that serves as a common interface that anyone using the CustomerPortlet can use, hence reducing overall complexity.



Using a template The same object structure can be created in a variety of different ways by reusing the code. Customer Portlet's structure resembles that of Abstract GenericPortlet, making it possible to implement some of its methods. This makes it possible to create different portlets, like greeting Portlet and thankyouPortlet.



Benefits of Design Patterns

1. They can be used in numerous projects.

2. They offer the answers that contribute to defining the system architecture.

3. They document the experiences with software engineering.

4. They make an application's design more transparent.

**3. Observe how the class diagrams are reflected from a given scenario by using the UML tool.**

Based on the UML class diagram for the factory design pattern, users can easily acquire different serviceContext objects for Customers by utilizing the serviceContextFactory or factory class. This is facilitated by the reusable factory method getInstance(), which can be used to obtain the associated id for a particular model or object, enabling the update of customer information.

Graphical user interface, text, application

Description automatically generated

The ServiceContext contains essential information, such as the method to return the current user ID, which makes it possible to implement the following functionalities. The Aggregation relationship can be observed in the ServiceContextFactory.getInstance(..) methods, which fill the ServiceContext with relevant data from a request object, as described in Liferay documentation.

|  |
| --- |
| Diagram  Description automatically generated |

The class diagram for the facade design pattern shows that the CustomerLocalServiceUtil class is extensively utilized to perform essential operations, such as adding, deleting, and updating customers, as well as other services. This class is dependent on the CustomerLocalService interface, which contains the necessary operations for the administrator. The Util class, also referred to as the facade, serves to conceal the complexity beyond this class, thereby simplifying the administrative methods. Furthermore, the class diagram identifies a Generalization relationship between the interface and CustomerLocalServicelmpl class, which implements the interface methods. Additionally, the diagram shows the inheritance relationship, where the Impl class extends the super class BaseLocalService and PersistedModeLocalService.

Graphical user interface, application, Teams

Description automatically generated

The template design pattern showcases the generic relationship between the abstract class GenericPortlet and its subclasses. The subclasses override the method implementation while keeping the abstract class structure intact.

**Task 3**

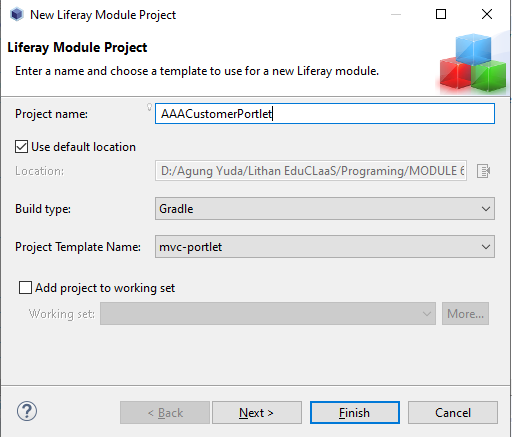
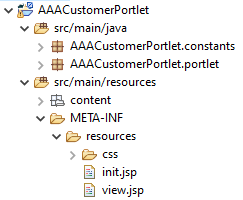
**System Development**

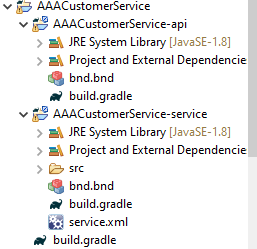
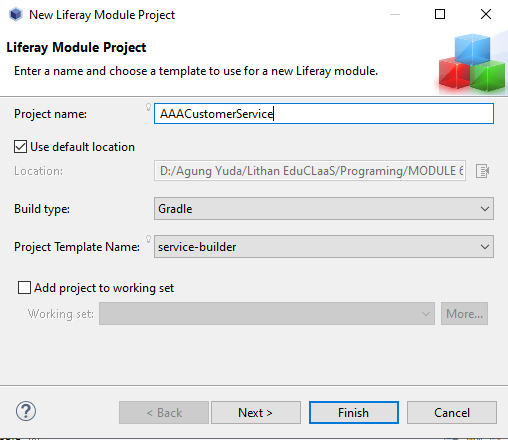
**1. Set up a project, build an application based on your derived UML class diagrams and design patterns. Provide screenshots as evidence.**

1. Create MVC Portlets (Customer and Service) to manage AAA customer’s

data

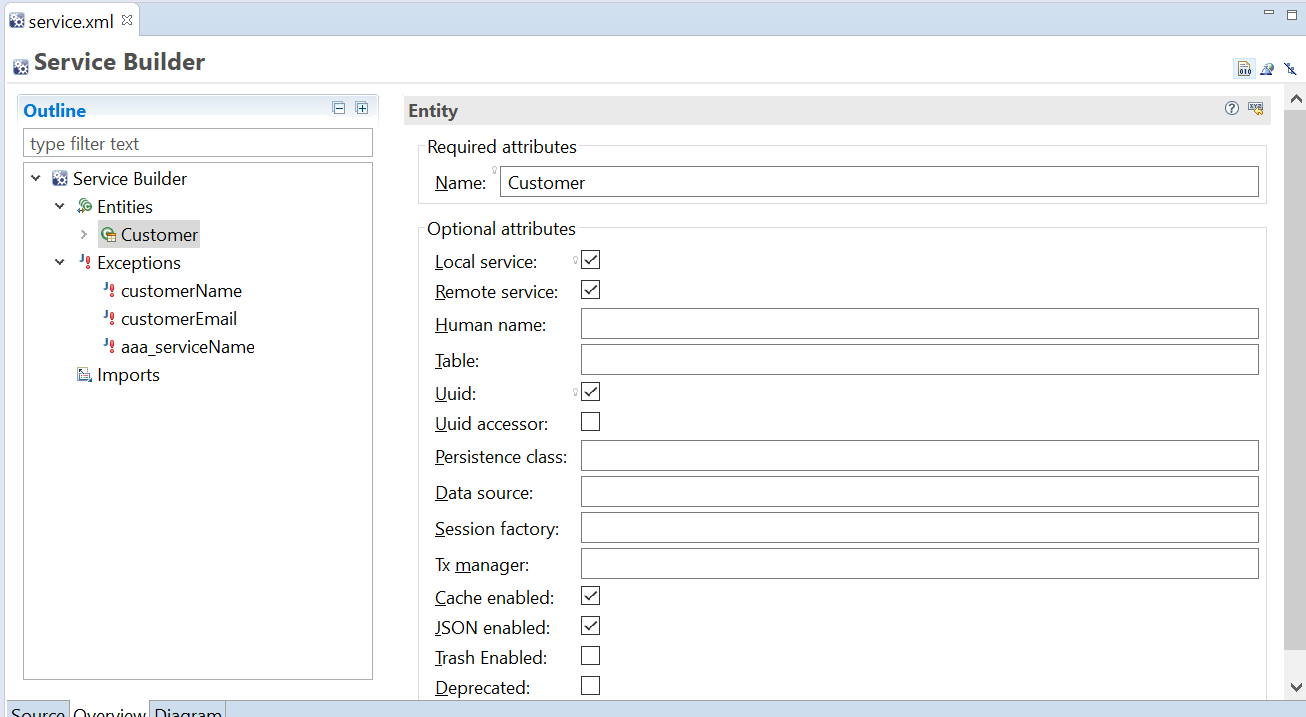
Create liferay module project and set like this





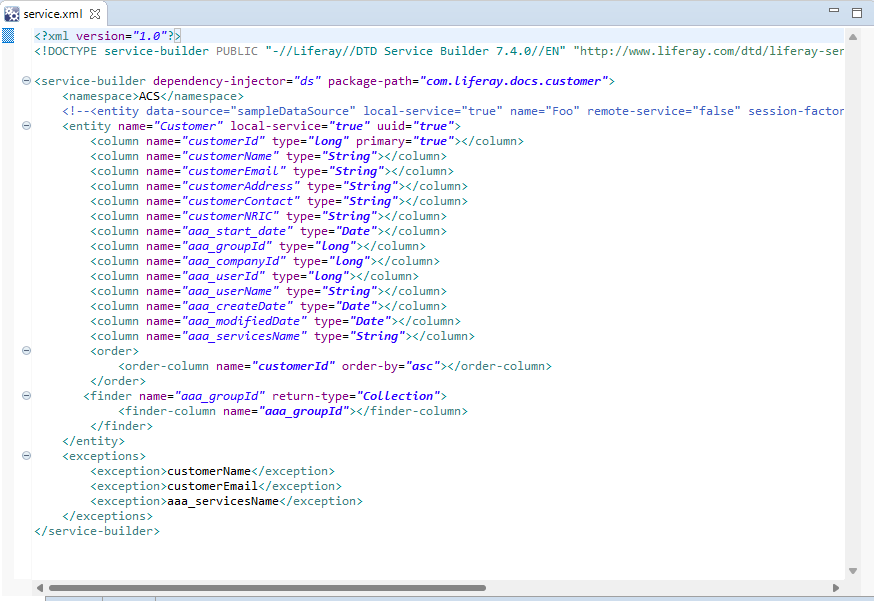
2. Build required entities using service builder.

After the service has created, you can go to the service.xml in service service portlet. Double click on that, go to overview, and create the entity

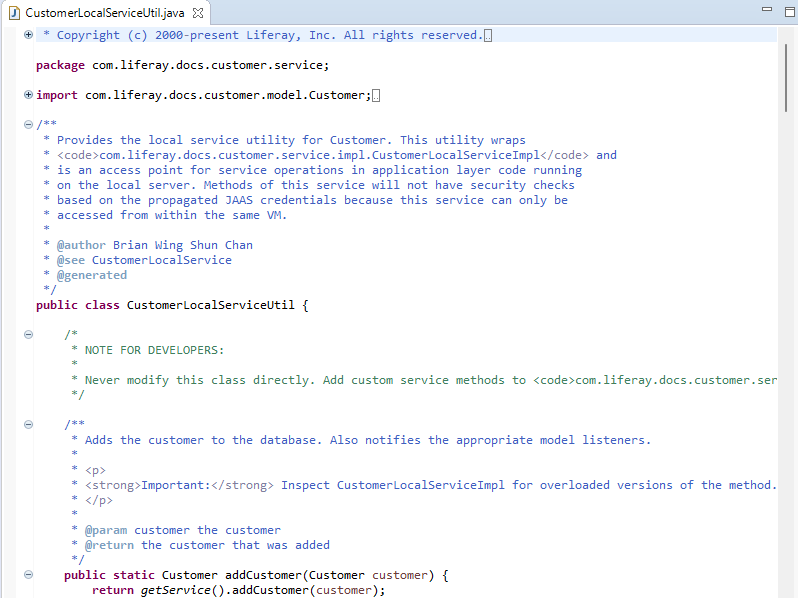


3. Provide the screen capture of service.xml

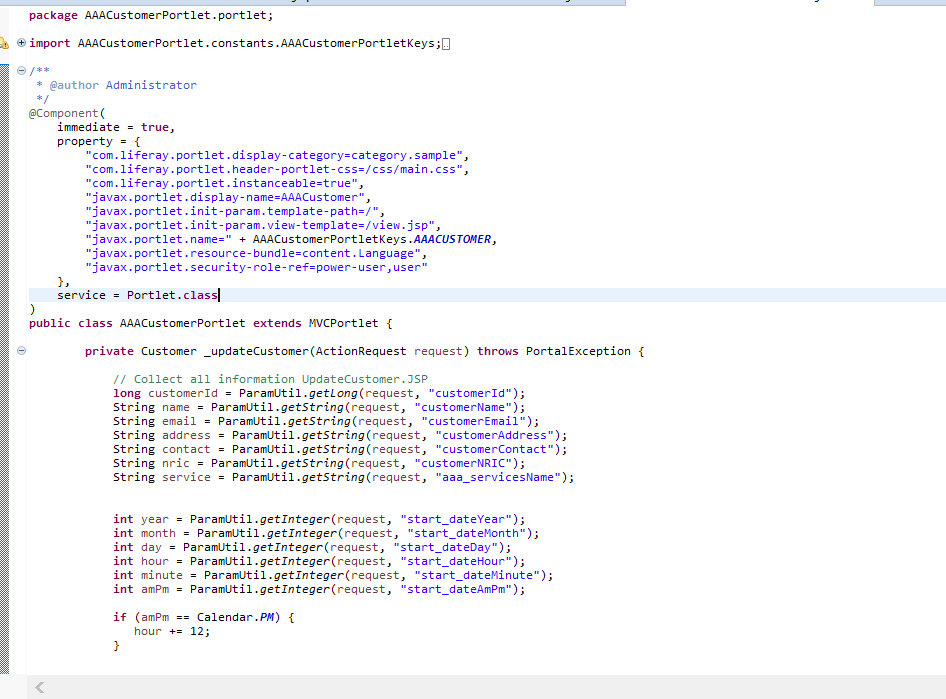
Go to source and you can see the code



4. Provide screen capture of Service Layer



5. Provide screen capture of Controller Class





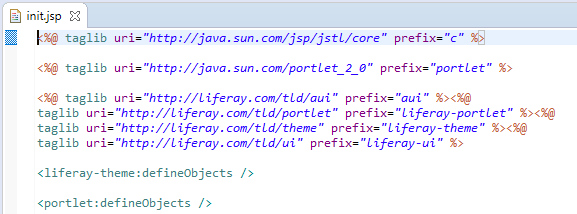
6. Provide screen capture of required jsp files for “AAA Customer and Service

Portlet”.

a. CustomerAction.jsp



b.init.jsp

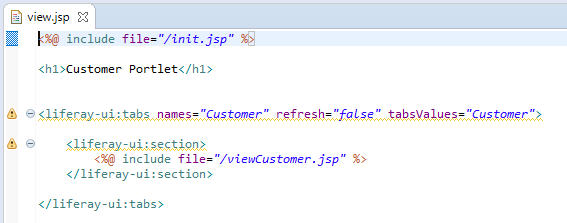


C .updateCustomer.jsp





d. view.jsp



e. viewCustomer.jsp

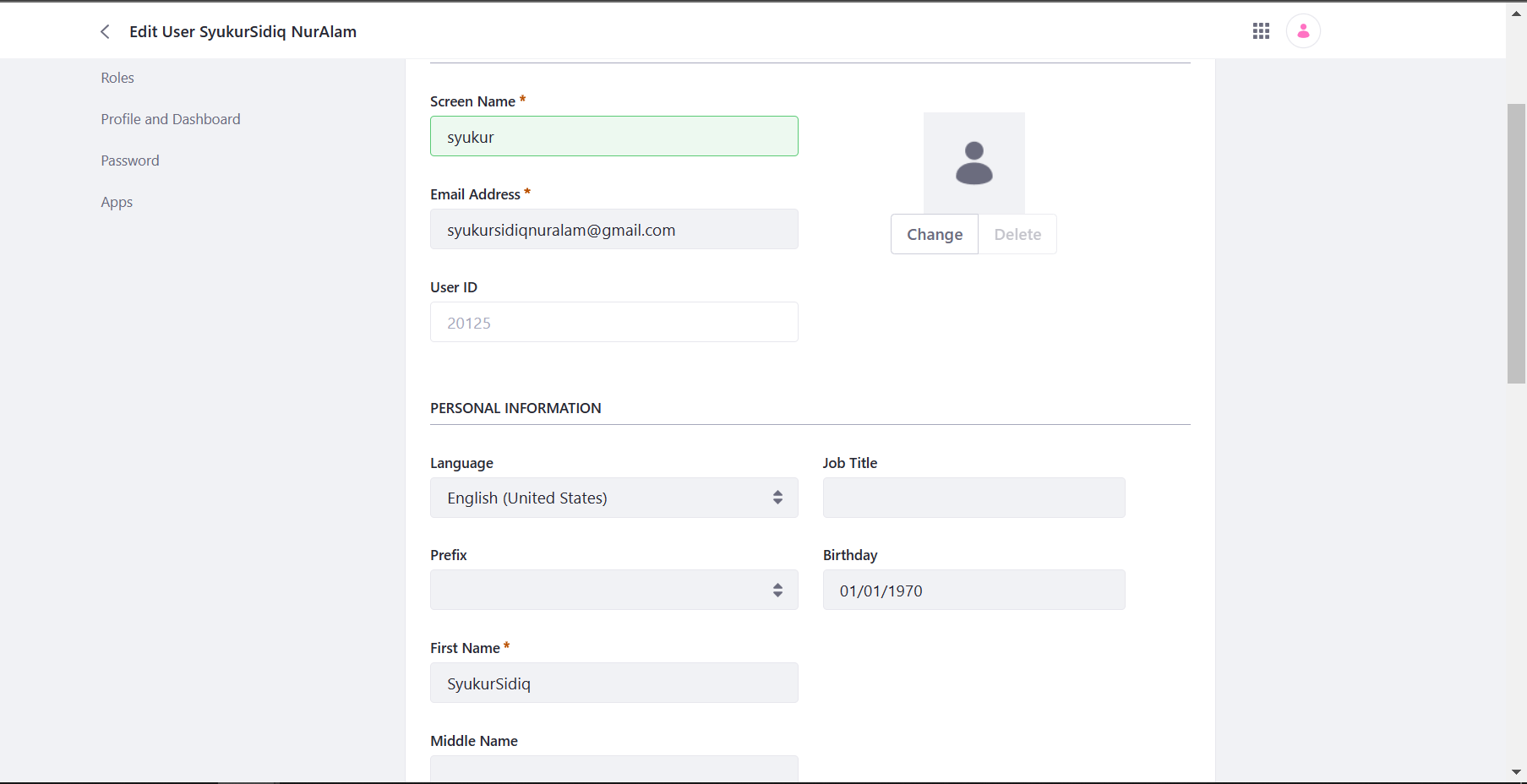




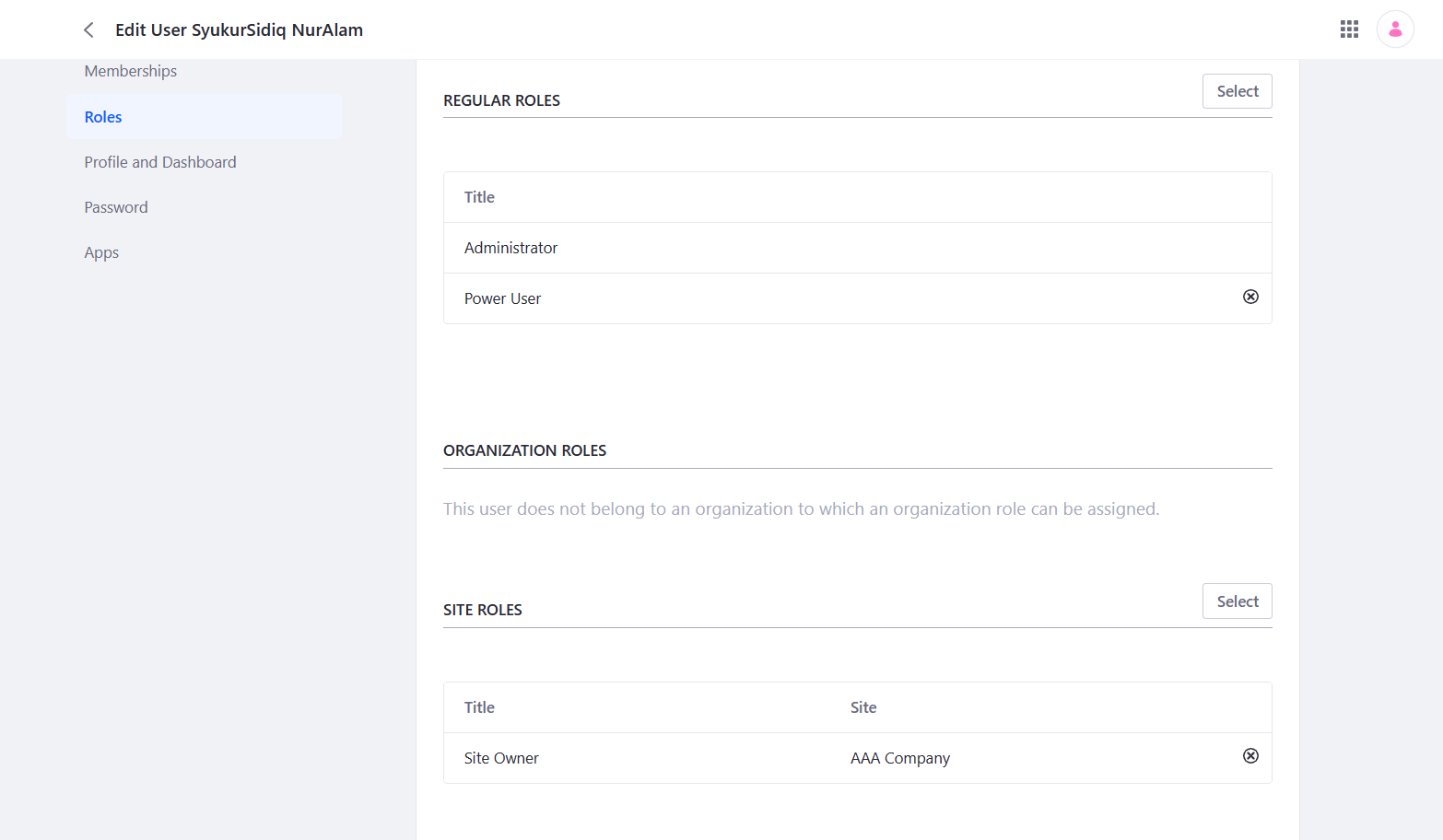


7. Create the Triple-A (AAA) company webpages. Take the screen captures of the following pages along with your Admin Account Name (Make sure that admin must be registered with Your Name)

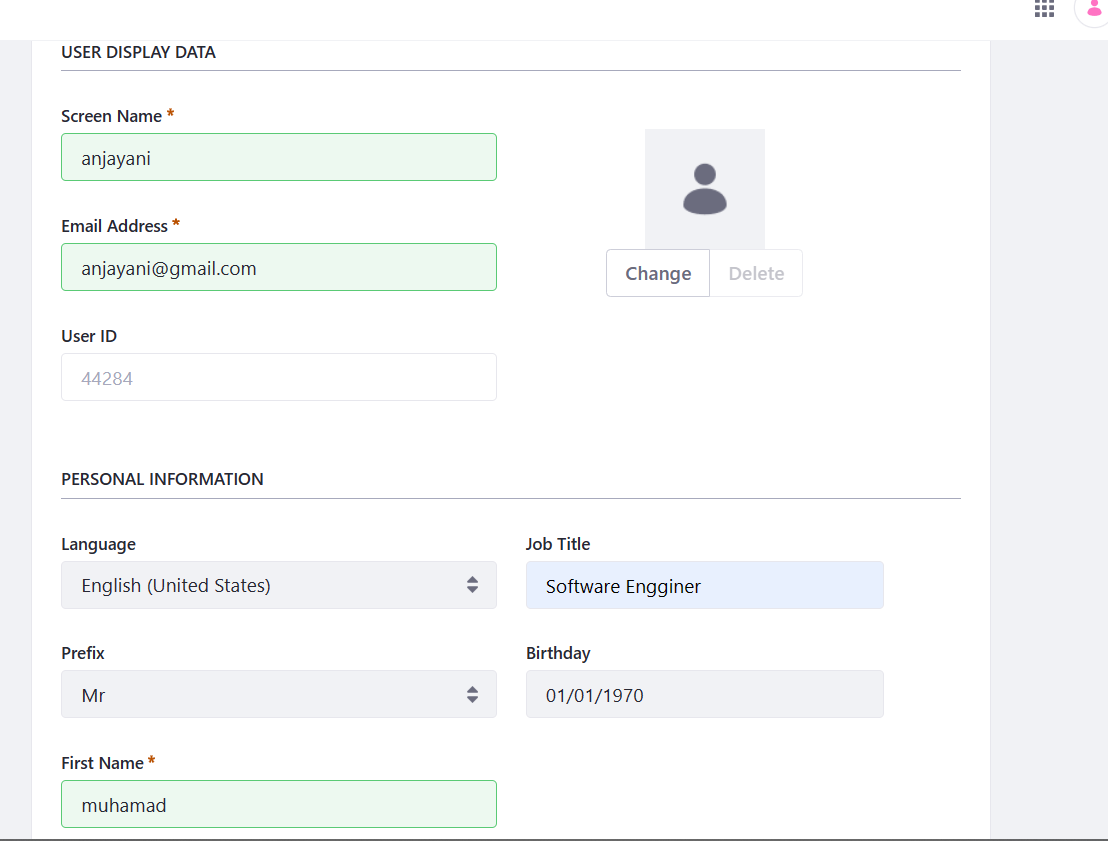
Administrator account



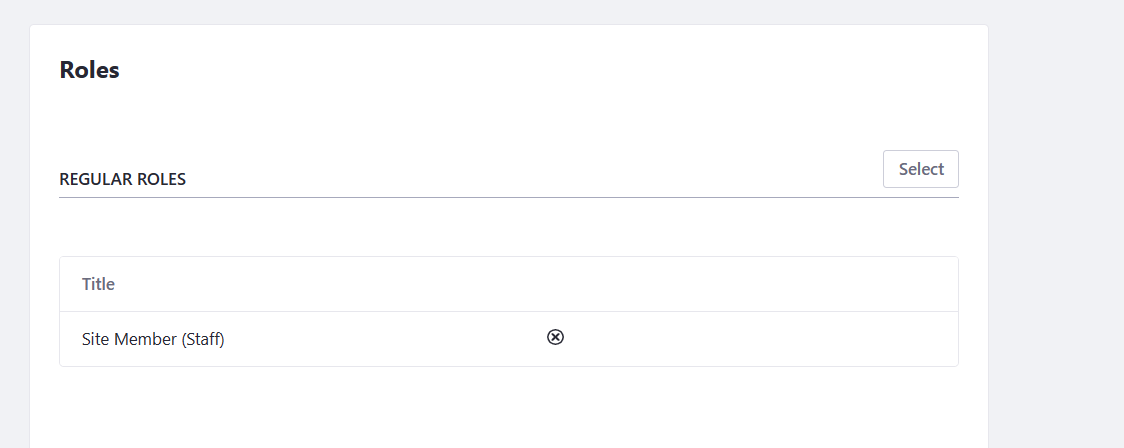
Administrator role



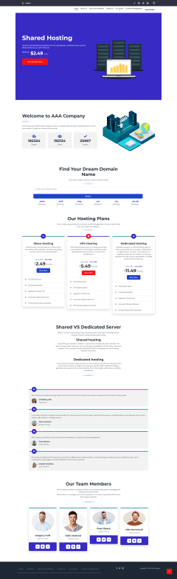
Site member(staff) Account Name



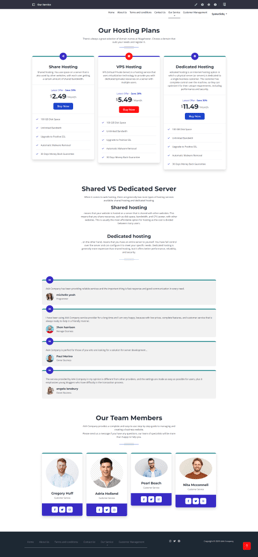
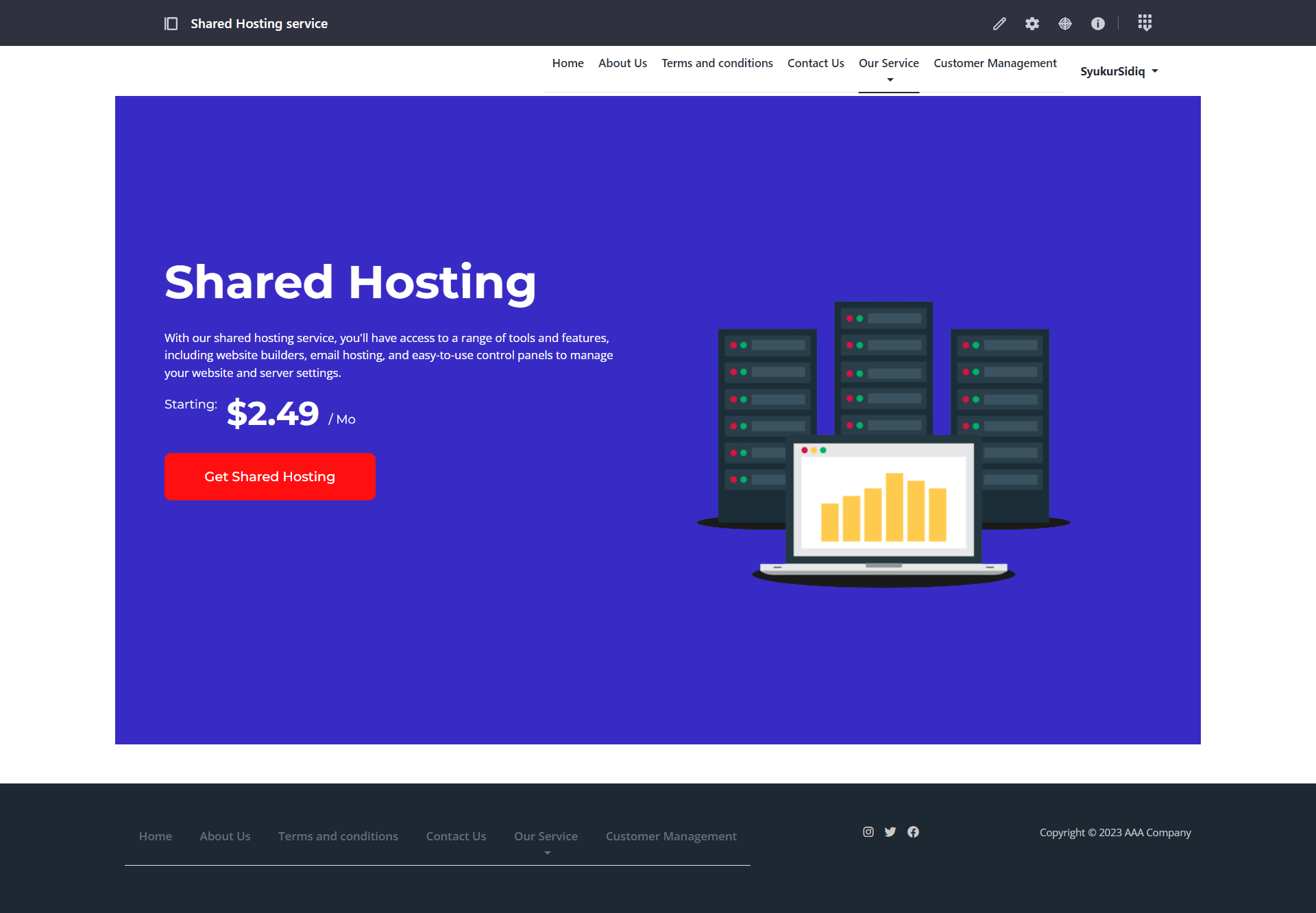
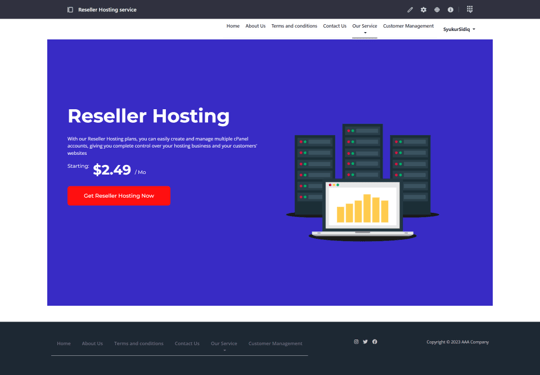
Site Member(Staff) Roles

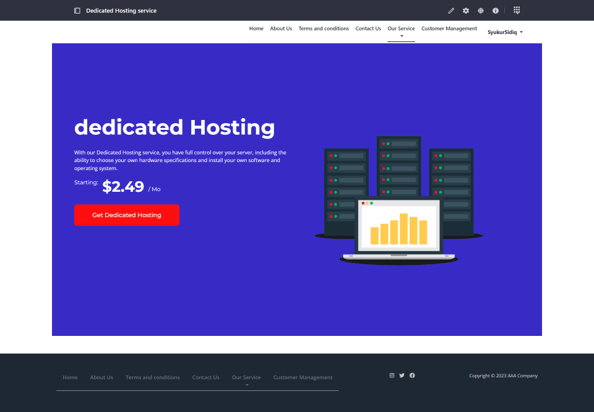
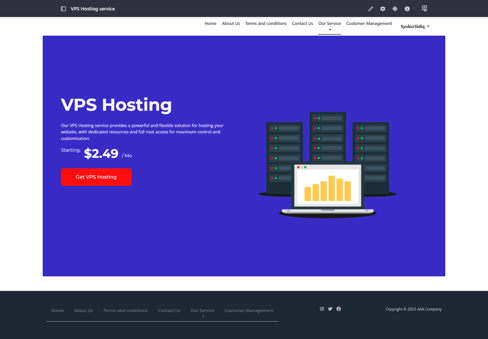
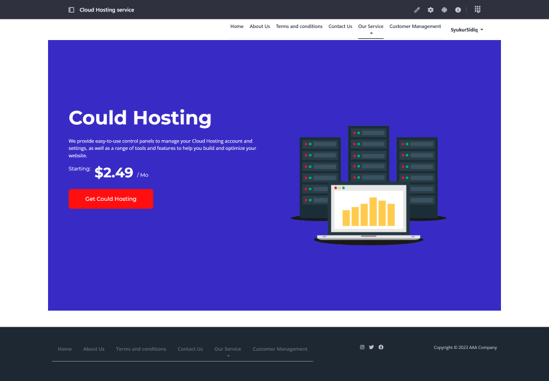


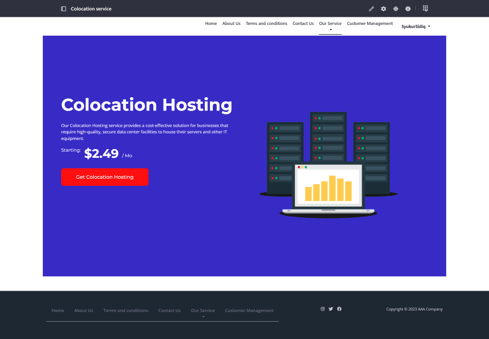
a) Home Page



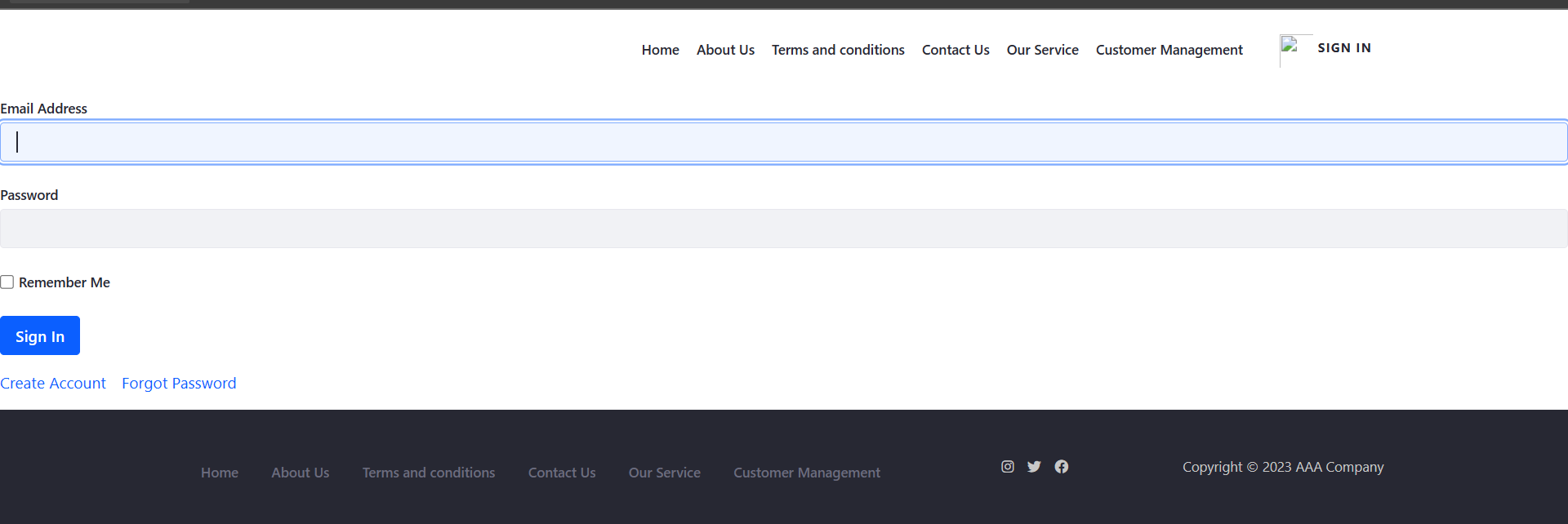
b) Our Services Page (including Customer and Service Portlet)



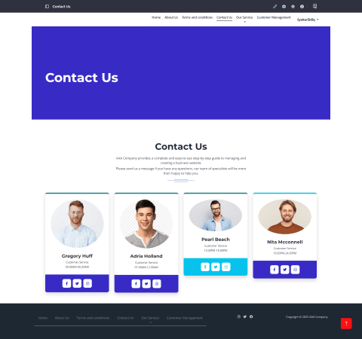




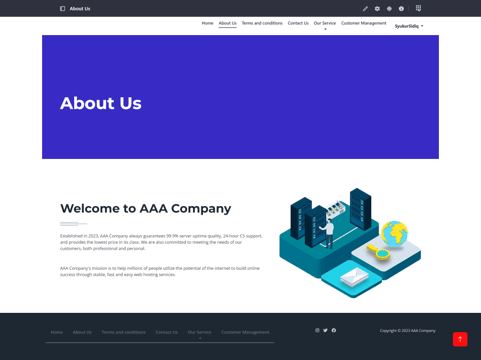
c) Register/Login Page



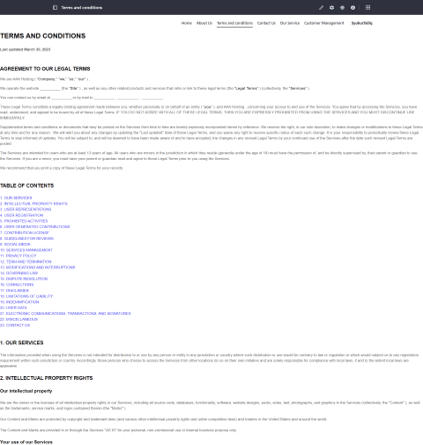
d) Contact Us Page



e) About Us Page

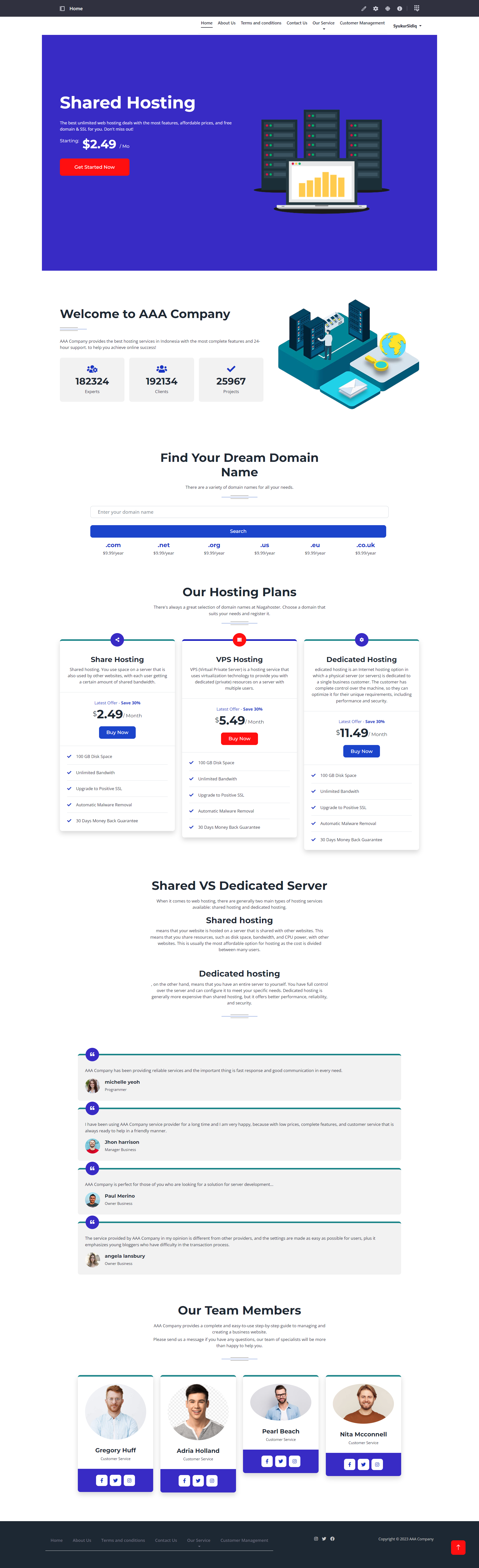


f) Terms and Condition Page

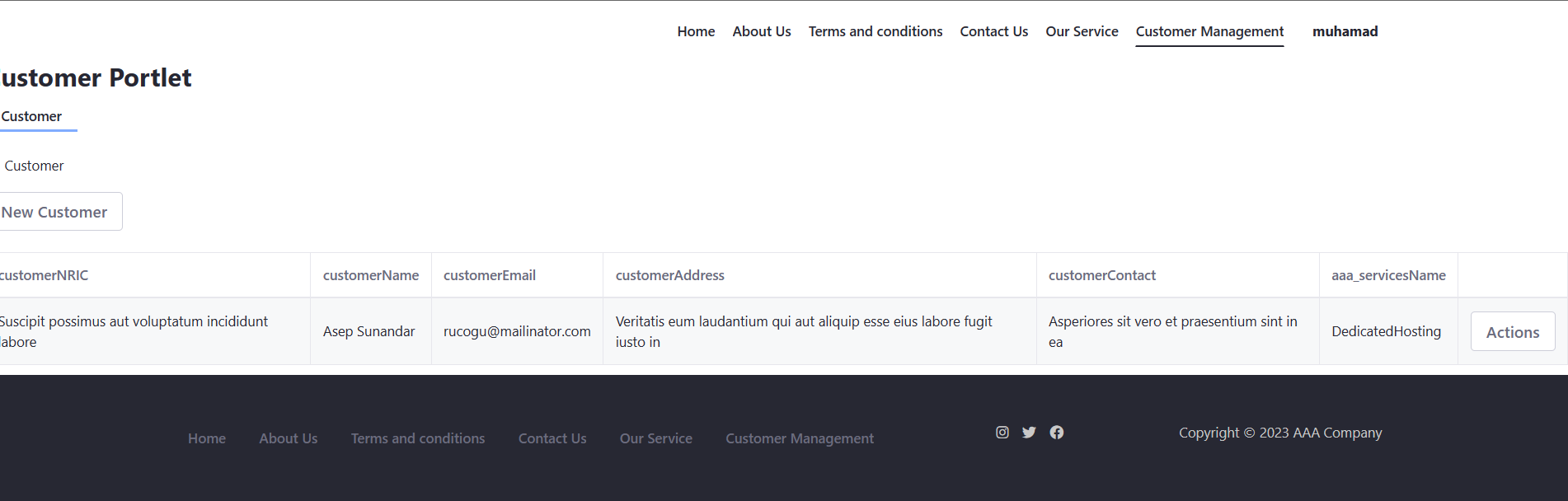


8. Provide screen capture logging in with two different users (Administrator, Site Member) showing Permission for Customer and Service Portlet.

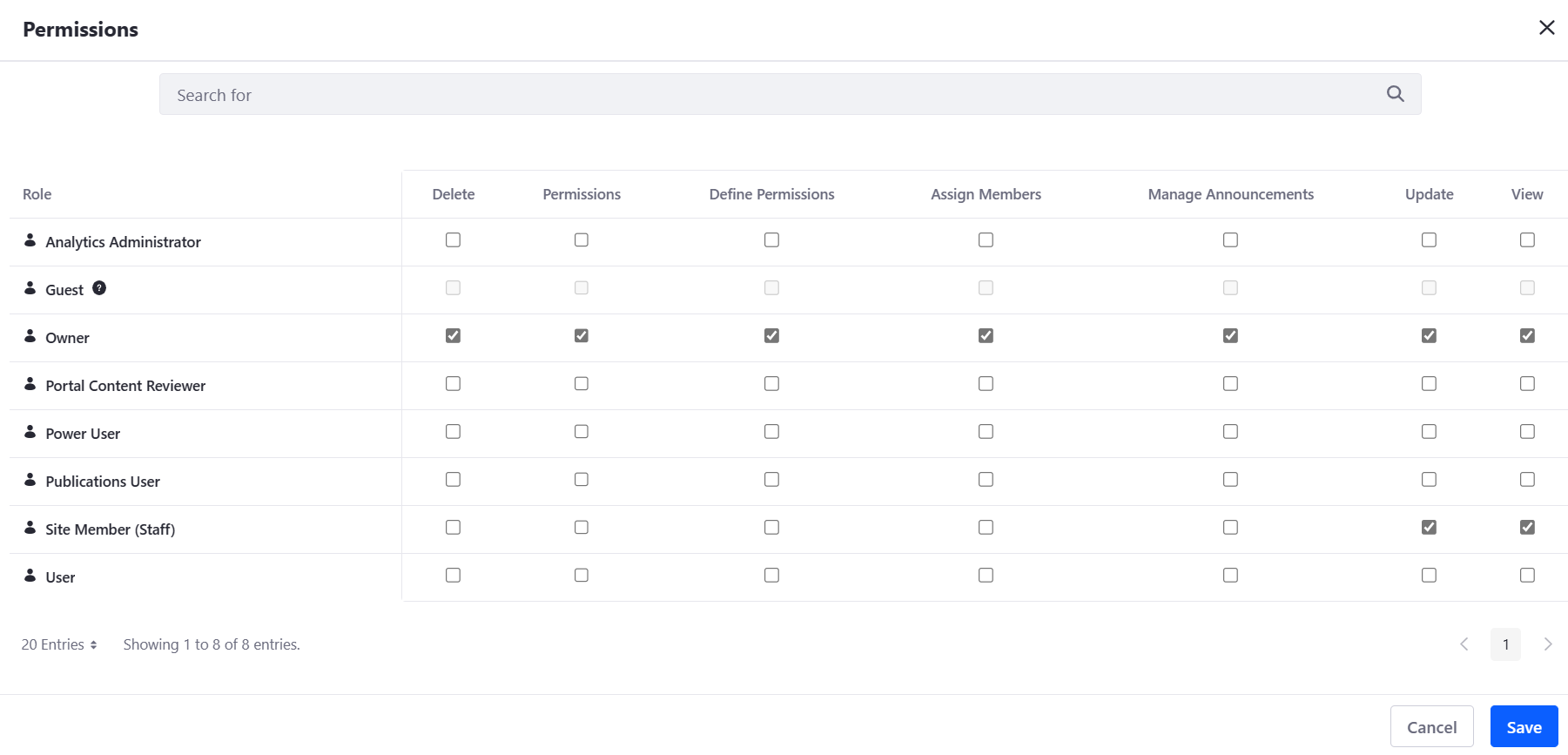
Administrator login



Site member (staff Login)



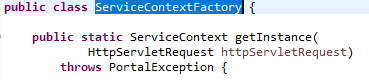
Permission



**8. Develop an application that implements design patterns and utilizes techniques to produce secure code. Provide the implemented code as evidence.**

The three potential design patterns that have been discovered are the industrial design pattern, facade, and template technique.

The factory design pattern makes use of polymorphism, as shown below:



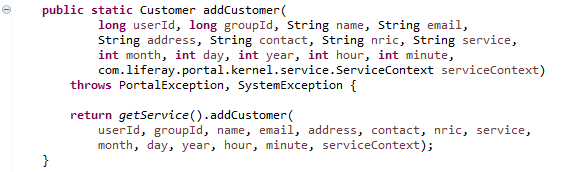
within the ServiceContextFactory class, the getInstance() method

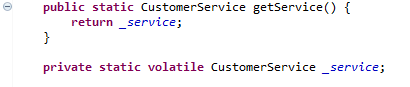
Method invocation statement from CustomerPortlet.java (controller class)



The getInstance() method of the serviceContext object above can instantiate it in more than one form depending on the request object, Customer, thanks to polymorphism. The benefit of polymorphism is that it allows for code reuse without modifying the original code.

Since it hides complexity outside it, CustomerLocalServiceUtil is the facade class in the facade design pattern. The code snippet below demonstrates how the facade class condenses the intricate system into a solitary interface object, the CustomerLocalService. Each method in the facade class returns the getService() function, which has a reference to the CustomerLocalService object, such as the addCustomer() method.





By using the facade design pattern, all of the data members of the CustomerLocalService are kept private and inaccessible to other classes by using encapsulation through data hiding. The restriction to the CustomerLocalService class strengthens security by protecting sensitive data. AddCustomer(), deleteCustomer(), and other functions are part of an interface for customers called CustomerLocalService. The interface offers total abstraction by providing all business logic or processes for the customer while segregating the implementation from the interface.

A parent class is this.













The implementing class, such the CustomerLocalServicelmpl class, which can define the method body, can inherit the parent class's methods and properties by implementing the interface.





Because the implementing class contains the same method as the parent class, such as the addCustomer() method, method overriding is permitted. By using the interface, the code is made to be loosely coupled, allowing other components to be changed without disrupting the structure as a whole. This increases the code's security.

Abstraction is also achieved using an abstract class, as shown in the template technique discussed previously, which makes use of the Generic Portlet abstract class. The class specifies the skeleton for constructing portlets as well as the default implementation of the Portlet interface. (Liferay, Upgrading a GenericPortlet. n.d.).

The general structure and algorithmic sequence of the parent class can be preserved while subclasses of the abstract class can override the method implementation as necessary. Additionally, duplicate code can be prevented. Subclasses can inherit the GenericPortlet class methods starting from the controller class CustomerPortlet, which uses the MVCPortlet build template, by using the extends keyword from the part of code below. As another class has already extended from another class, there is multi-level inheritance.



**9. Discuss the use of design patterns for the given purpose and consequences by applying design patterns**

Software developers frequently employ design patterns to tackle reoccurring issues and raise the caliber of their work. They offer tried-and-true answers to typical design issues, which can speed up software development, make it easier to maintain, and boost performance. The usage of design patterns can be especially helpful in the context of the stated goal, which is to manage site content and user roles in a web-based system.

The "Model-View-Controller" (MVC) design pattern is one that might be applied in this situation. Model, view, and controller are the three primary parts into which this architecture divides the application logic. The controller mediates communication between the model and the view, whereas the model represents the data and business logic and the view the user interface. Utilizing this pattern will allow the code to be modularized and the development process to be more organized, both of which will enhance the software's quality and maintainability.

The "Factory Method" design pattern is an additional option. With the help of this pattern, it is possible to build objects without identifying their precise class beforehand. This pattern could be used to produce many kinds of site content objects, including photos, videos, and text, depending on the context of the provided purpose. Utilizing this approach can make the code more adaptable and extendable, which helps speed up software development and enhance performance.

It's crucial to remember that applying design patterns might have negative effects as well. For instance, design patterns can make the code too complex and challenging to maintain if they are overused or improperly used. Furthermore, if the design patterns are not effectively applied, they can not have the desired effects or possibly cause new issues. Because of this, it's crucial to carefully analyze the application of design patterns in the context of the particular software development project and to make sure they're done so in a way that's appropriate and efficient.

**10. Investigate how different design patterns can work within a range of different scenarios.**

**(Non-BTEC Task 4 - 2)**

**Design Patterns**

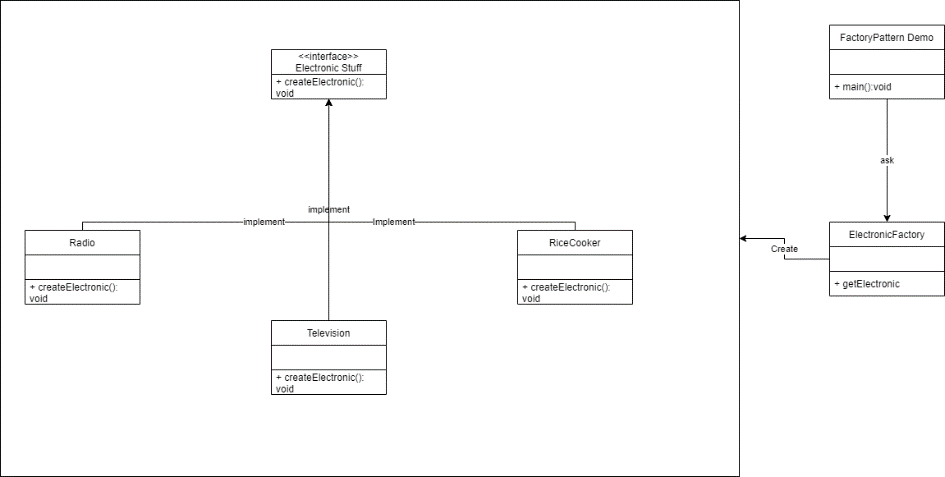
**1. Creational Design Pattern**

Factory Pattern

Scenario:

By transferring information (such as from a radio, television, or rice cooker) to the electronic factory, the factory pattern demo uses the electronic factory to produce various types of electronic goods. The three concrete classes (Radio, Television, and RiceCooker) implement the interface known as "Electronic," and by making reference to another interface, further types of Electronic can also be formed without the client being made aware of the creation logic.

Diagram:

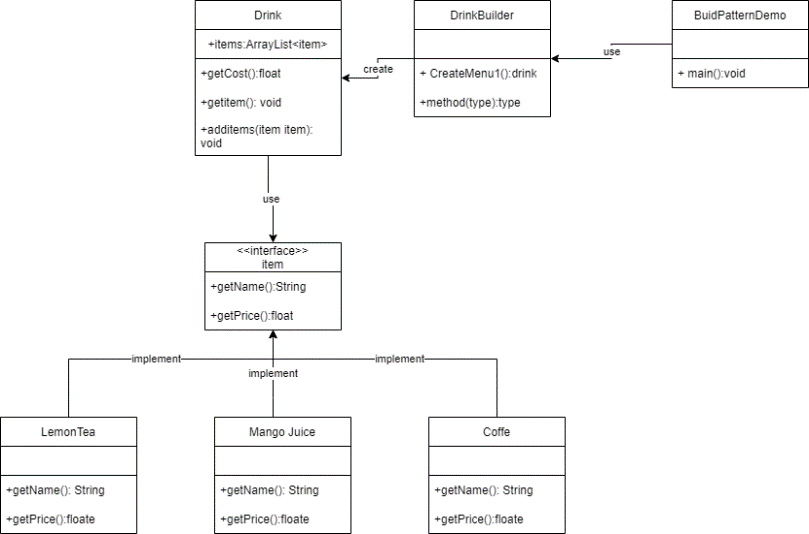


**Builder Pattern**

Scenario:

taking into account a restaurant where several menu options can be made. The builder pattern can demonstrate how to prepare a drink step by step. This is made possible via the Item interface, which implements the Item interface and represents all food products. After that, a drink can be made by adding items to an array list, and the drink builder will construct several drink combinations based on the restaurant's menu. The drink builder will be requested by the builder demo to create a drink that is ready to serve for the customer.

Diagram:



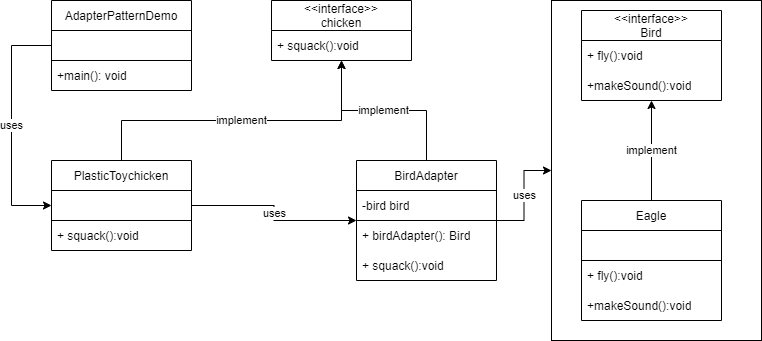
**2. Structural Design Pattern**

**Adapter Pattern:**

Scenario:

The diagram shows the BirdAdapter as a bridge between the incompatible Chikentoy and Bird interfaces. The functions of the interfaces must be connected by the BirdAdapter class in order for the PlasticToyChiken to produce a squeaking sound that is exact to that of a real bird, in this case the crow sound.

Diagram:

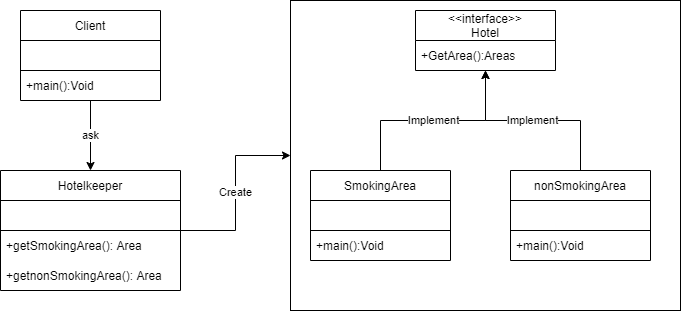


**Facade Pattern**

Scenario:

A client or customer is curious about the several restaurants that the hotel offers. If you ask the hotel manager, he will handle any challenging implementation so that the client receives what they want. As a result, the hotel manager acts as a front while the hotel system's complexity is hidden. This can be done using the hotel interface, which returns a location, or by using the adjacent eateries; for example, a restaurant that offers a smoking area will return a smoking area. The facade class or hotel keeper is the one who gathers and retains all the locations on hand whenever a customer asks for one.

Diagram:



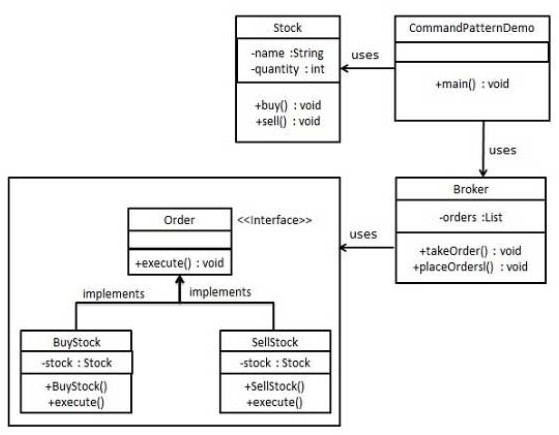
**3. Behavioral Design Pattern**

### Command Pattern

Scenario: In the command pattern, the Stock class acts as the request, and the Order interface acts as the command. Order Interface classes will handle the command itself during execution. The invoker, or Broker class, has the ability to accept orders, put orders, and select which command to execute.

.

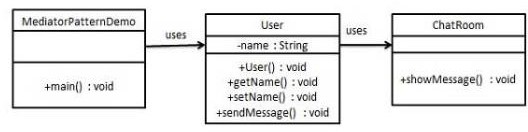
Diagram:



### Mediator pattern

Scenario: The chatRoom is in charge of showing messages to all users, as seen in the diagram. The User object will converse using the chatRoom function. The MediatorPattern Demo's User objects will be used to show how to communicate with any User.

Diagram:



**11. Identify the appropriate design pattern from the investigation.**

**(Non-BTEC Task 4 - 3)**

a. Among 2 different scenarios in creational design which you have analyzed in P4, reconcile most appropriate design pattern

According on the provided scenarios, the following design pattern would be most suitable:

The factory method would be the most suitable design pattern for the first scenario, where an electronic factory manufactures a variety of electronic goods. With the help of this pattern, objects can be generated without being specifically identified by their class. In this case, based on the information collected, the factory method would be in charge of producing the various sorts of electronic items (radio, television, rice cooker).

The Builder pattern would be the most suitable design pattern for the second situation, when a restaurant provides various menu items. This pattern allows alternative representations to be created using the same construction process by separating the construction of a complicated object from its representation. In this case, the client would ask the builder to make a drink that is ready to serve for the customer, and the builder ld be in charge of creating numerous drink combinations based on the restaurant's menu.

b. Among 2 different scenarios in structural design which you have analyzed in P4, reconcile most appropriate design pattern

The most suitable design patterns would be the following ones based on the scenarios provided:

The Adapter design, which enables the conversion of incompatible interfaces—in this case, the Chikentoy and Bird interfaces—is in fact the most suitable for the first scenario. The PlasticToyChicken is able to mimic the sound of a real bird thanks to the BirdAdapter, which serves as a bridge. When working with third-party libraries that have distinct interfaces or integrating old and new code, this design approach is ideal.

on the other hand, is best suited for the second scenario because it offers a straightforward interface to a complicated system—in this case, the hotel system. The Facade, in this case the hotel manager, conceals the system's complexity and gives the client a user-friendly interface. When dealing with complicated systems that need for streamlined user interfaces, this pattern is helpful since it enables clients to engage with the system without having to deal with its complexity.

c. Among 2 different scenarios in behavioral design which you have analyzed in P4, reconcile most appropriate design pattern

The behavioral design category's most suitable design pattern would be based on the circumstances provided:

The perfect illustration of the Command Pattern is the situation when the Order interface serves as the command and the Stock class serves as the request. The invoker class, Broker, takes and places orders and chooses which command to execute, while the Order interface classes handle command execution.

An illustration of the Mediator Pattern is the situation where the chatRoom serves as the mediator to enable communication between several User objects. The chatRoom serves as the primary hub, through which all conversation is routed. The User objects don't speak to one another directly; they only converse in the chatRoom.

**12. Evaluate and justify the design patterns that you had identify in each of the scenarios.**

**(Non-BTEC Task 4 - 4)**

Factory Pattern:

The Factory Pattern is an appropriate design pattern for the electronic factory scenario because it allows for the creation of multiple types of electronic goods without the client being aware of the creation logic. By using the Factory Pattern, the electronic factory can produce different types of electronic products based on the client's request. This makes it easier to maintain the codebase and update the production process of the electronic factory in the future.

Builder Pattern:

The Builder Pattern is an appropriate design pattern for the restaurant scenario as it allows for the creation of complex objects (in this case, a drink) in a step-by-step manner. The Item interface represents all food products, and the drink builder constructs several drink combinations based on the restaurant's menu. This makes it easier to modify and add new drinks to the menu without affecting other parts of the codebase.

Adapter Pattern:

The Adapter Pattern is an appropriate design pattern for the bird toy scenario because it allows the ChikenToy to produce a sound that is exact to that of a real bird. The BirdAdapter acts as a bridge between the incompatible ChikenToy and Bird interfaces, connecting their functions to produce the desired sound. The Adapter Pattern is useful in situations where two incompatible interfaces need to communicate with each other.

Facade Pattern:

The Facade Pattern is an appropriate design pattern for the hotel restaurant scenario as it simplifies the complexity of the hotel system and provides a simple interface for the client to access the information they need. The hotel keeper acts as a front while the hotel system's complexity is hidden, and the Facade class gathers and retains all the locations on hand whenever a customer asks for one. This makes it easier for the client to get the information they need without worrying about the underlying complexity of the hotel system.

Command Pattern:

The Command Pattern is an appropriate design pattern for the stock trading scenario as it allows for the decoupling of the requester and the executor of a command. The Stock class acts as the request, and the Order interface acts as the command. The invoker, or Broker class, has the ability to accept orders, put orders, and select which command to execute. This makes it easier to maintain the codebase and add new functionality to the stock trading system without affecting other parts of the codebase.

Mediator Pattern:

The Mediator Pattern is an appropriate design pattern for the chatroom scenario as it allows for the decoupling of the objects that need to communicate with each other. The chatRoom is in charge of showing messages to all users, and the User object will converse using the chatRoom function. The MediatorPattern Demo's User objects will be used to show how to communicate with any User. This makes it easier to modify and add new functionality to the chatroom system without affecting other parts of the codebase.