TERM 2510

CSE 6234 Software Design

Title:

Waste Management and Recycle

Group Green:

Name	Student ID
Cheeng Tze Yuan	1211109439
Desmond Goh Liarg Jiing	1211110498
Leong Zong Xin	1231301891
Sam Jun Xiang	1211110401

1. Introduction

1.1. Abstract

This report documents the design and partial implementation of a waste management and recycling system, providing two key functions: an educational slide feature and the ability to browse nearby recycling centers. The system applies five software design patterns—Factory, Bridge, Strategy, Facade, and Observer—to ensure modularity, scalability, and maintainability. This report uses use case diagrams, object diagrams, sequence diagrams, class diagrams, and workflow diagrams to show the interactions within the system between three user roles: admin, member, and guest. The implemented features include slide creation, removal, searching, and management of recycling center addresses. This report also focused on how design patterns can be effectively applied in an application to produce a flexible and extensible software architecture.

1.2. Problem Statement

In many areas of Malaysia, waste management is still not effective. People often throw rubbish in the wrong places, even under signs that warn about illegal dumping rubbish will be fined. They also do not separate waste properly with lack of awareness and recycling habits. This leads to pollution of the surrounding. The current waste management system does not strongly encourage or educate the public about the importance of recycling and how to do it. Therefore, this project aims to design a waste management and recycling system that increases public awareness, educate user about proper waste separation, and help the user to locate the nearest recycling center to improve waste tracking and recycling efforts.

1.3. Project Objectives

The primary objective of this waste management and recycling project is to develop a user-friendly application that helps individuals locate nearby recycling centers and promotes awareness through educational content on sustainable waste management practices, in support of SDG Goal 11: Sustainable Cities and Communities.

To achieve this, the project will focus on the following sub-objectives:

- To develop a digital platform that enables users to find nearby recycling centers.
- To educate users by integrating informative slides, infographics, and best practices on waste reduction and recycling.

1.4. Literature Review

Introduction

With increasing urbanization and the growing environmental impact of waste, recycling has become a key focus for sustainable development in Malaysia. However, despite efforts by local governments and organizations to improve recycling practices, many Malaysians still face barriers to effective recycling. These barriers include a lack of awareness of recycling facilities and proper waste segregation methods, alongside limited access to educational resources on effective waste reduction. This literature review explores the potential for a digital platform that helps users find nearby recycling centers, and integrates educational content, such as slides for recycling and waste reduction.

1.4.1. Digital Platforms for Recycling Centers

A significant barrier to recycling in Malaysia is the inaccessibility of recycling centers. To overcome this, digital platforms can provide users with up-to-date information on nearby recycling locations and details about what materials can be processed there. Research suggests that mobile apps and online platforms can effectively guide users to these facilities and inform them about proper recycling methods. An example is Recircle, a Malaysian platform designed to link users with local recyclers, offering comprehensive details about accepted materials and even providing pickup services (VulcanPost, 2023). This kind of functionality is especially beneficial in urban environments, where the fast-paced nature of daily life can make finding recycling centers more challenging.

1.4.2. Educational Content on Waste Reduction and Recycling

In addition to accessible recycling options, it’s vital that users receive clear guidance on how to recycle correctly. Educational programs have consistently been an essential part of effective waste management, and studies demonstrate that visual tools, such as infographics and detailed guides, can significantly enhance comprehension and boost recycling participation. Infographics are effective at simplifying recycling procedures and correcting common misunderstandings about what materials can be processed (ResearchGate, 2022).

1.4.3. Waste Reduction Practices Beyond Recycling

While recycling is a crucial aspect of waste management, it is only one part of a broader strategy. Effective waste reduction involves a comprehensive approach, including minimizing waste production, reusing materials, and promoting composting. Research indicates that educating the public on zero-waste practices, such as limiting single-use plastics can complement recycling initiatives and create a more integrated waste management system. Furthermore, governmental bodies and non-governmental organizations (NGOs) play a vital role in driving these efforts by implementing national campaigns that encourage both education and actionable solutions to reduce waste (MDPI, 2023).

1.4.4. Behavioral Insights and Educational Strategies

Integrating behavioral science principles into recycling and waste reduction programs is crucial for achieving lasting change. Behavioral studies reveal that habit formation plays a central role in encouraging ongoing recycling activities. Digital platforms that incorporate nudges, such as reminders or geographically based notifications about local recycling options can support users in developing long-term, sustainable recycling habits (MDPI, 2023). Moreover, research suggests that social influence where users observe their peers engaging in recycling can significantly motivate individuals to adopt similar behaviors (ScienceDirect, 2022).

Conclusion

Combining a digital platform for finding recycling centers with educational resources and behavioral strategies presents a promising solution to the waste management issues in Malaysia. These platforms can increase access to recycling facilities and deliver informative, engaging content, which would likely boost public involvement in recycling and waste reduction efforts. Additionally, incorporating interactive elements, like automated reminders, could further promote consistent recycling habits, leading to a more sustainable and efficient waste management system.

1.5. Project and System Scope

Project Scope:

The goal of this project is to design and develop a waste management and recycling system by helping users find nearby recycling centers and learn about sustainable waste management practices.

Main Objectives:

- To provide a location-based search for recycling centers.
- To provide educational content in the form of slides, infographics, and best practices

Deliverables:

- A web application that includes location-based services.
- Educational content on waste management practices.

Inclusions:

- Integration with maps to show recycling center locations.
- Basic user interface for browsing and searching.

Timeline:

1. Gather requirements and research information
2. System design and planning
3. Prototype development
4. Testing and review

System Scope:

The Waste Management and Recycling system will serve as a digital platform for users to access information about recycling and sustainable waste management. The system will provide a search service to help users find nearby recycling centers and educational resources.

Main Functionalities:

- Locate nearby recycling centers using map integration.
- Educate users with informative slides, infographics, and best practices.

Inclusions:

- Clean, user-friendly interface with basic navigation.
- Location search using manually entered addresses.

Assumptions:

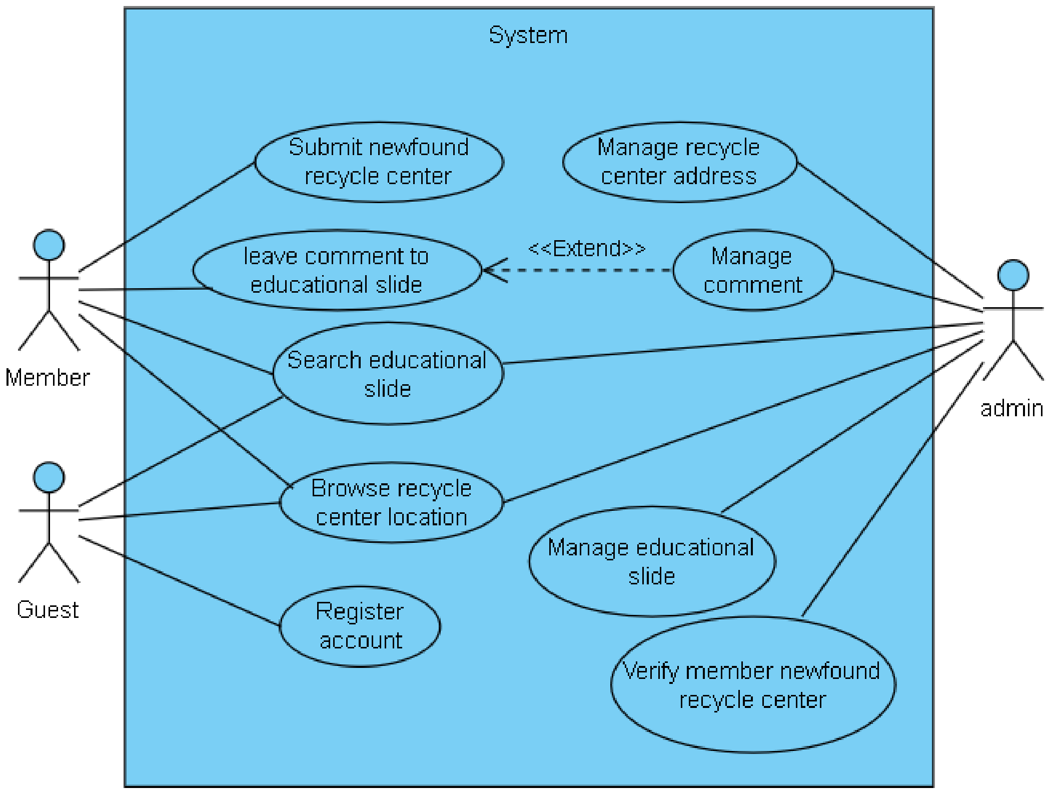
- Users have internet access.
- Recycling center data is manually updated by the admin.

2. System Overview

2.1. Generic Use Case

This use case diagram models the interactions between different types of users and a system designed for managing recycling centers and educational slides. It identifies three key actors: **Guest**, **Member**, and **Admin**, each with distinct permissions and responsibilities.

- **Guests** can browse recycle center locations to gain information as well as register an account to become a member for extended functionality.
- **Members** inherit all guest capabilities, can submit newfound recycling centers to the system, search educational slides for environmental or recycling education and leave comments on the educational slides.
- **Admins** have full control over system content and user submissions. Additionally manages, approves, modifies, or removes recycling center addresses. Other than that, they can manage educational slides by creating or editing them as well as managing the comments left by members.


System
Submit newfound recycle center
Manage recycle center address
leave comment to educational slide
<<Extend>>
Manage comment
Search educational slide
Browse recycle center location
Manage educational slide
Register account
Verify member newfound recycle center
Member
Guest
admin

2.2. Class Diagrams

Figure 2.2.1 shows the class diagram of the Waste Management and Recycling system. It includes thirty classes that provide structure to the whole system. The classes are "User", "Member", "Guest", "Admin", "SlideFacade", "RecycleCenterFacade", "Comment", "SearchSlide", "RemoveSlide", "CreateSlide", "SlideFactory", "plasticSlide", "paperSlide", "canSlide", "Slide", "slideFunctionality", "SlideManagement", "RemoveStrategy", "ModifyStrategy", "RC_ManageAddresses", "RC_ManageAddressesStrategy", "RC_RemoveAddressStrategy", "RC_ModifyAddressesStrategy", "NewFoundRecycleCenter", "VerifyRecycleCenter", "Observer", "memberObserver", "Subject", "RecycleCenterNotifier", "Observer Interface".

The "User" class contains attributes which are "accName" and "accPassword". This "User" class is used as the base class for specific types of users such as "Admin", "Member" and "Guest" class which are inherited from "User" class. "Admin" class has permissions to manage recycle center addresses, slides and comments. "Member" class can comment on slides, submit newfound recycling center locations and receive updates notifications when a recycle center is verified. "Guest" class has limited access and is used for users without accounts.

The "Slide" class contains attributes which are "type", "description", "title" and "slideID". This "Slide" class is an abstract base class for slide content. There are three concrete subclasses which are "plasticSlide" class, "paperSlide" class and "canSlide" class. The "SlideFactory" class use the Factory Design Pattern. It uses the "get_Type()" method to return an object based on the slide type. Operations like "CreateSlide", "RemoveSlide" and "SearchSlide" are wrapped using the "SlideFacade" class, which use the Facade Design Patterns and provides methods like "create_slide()", "remove_slide()" and "search_slide()". The system also uses the Bridge Design Pattern with "SlideManagement" class and the "slideFunctionality" interface to let user search slides.

The "Comment" class contains attributes which are "commentID", "content" and "timestamp". This "Comment" class allows "Member" class to interact with slides and "Admin" class to manage it. It contains methods like "createComment()", "editComment()"

and "deleteComment()". These comment actions are also accessed through the "SlideFacade" class for better abstractions.

The "RecycleCenterFacade" class handles operation like adding, removing and modifying addresses. The "RC_ManageAddresses" class uses the Strategy Design Pattern to switch between address related operation dynamically which is "RC_RemoveAddressStrategy" class and "RC_ModifyAddressStrategy" class. Both implement the "RC_ManageAddressStrategy" interface.

The "Observer" class uses the Observer Design Pattern to notify members when a new recycle is verified and added. "VerifyRecycleCenter" class is responsible for the verification process. It uses the "RecycleCenterNotifier" class to manage and notify all observers. The "Observer" interface has a method called "notify()". The "memberObserver" class implements the "Observer" interface and gets notified when a center is verified.

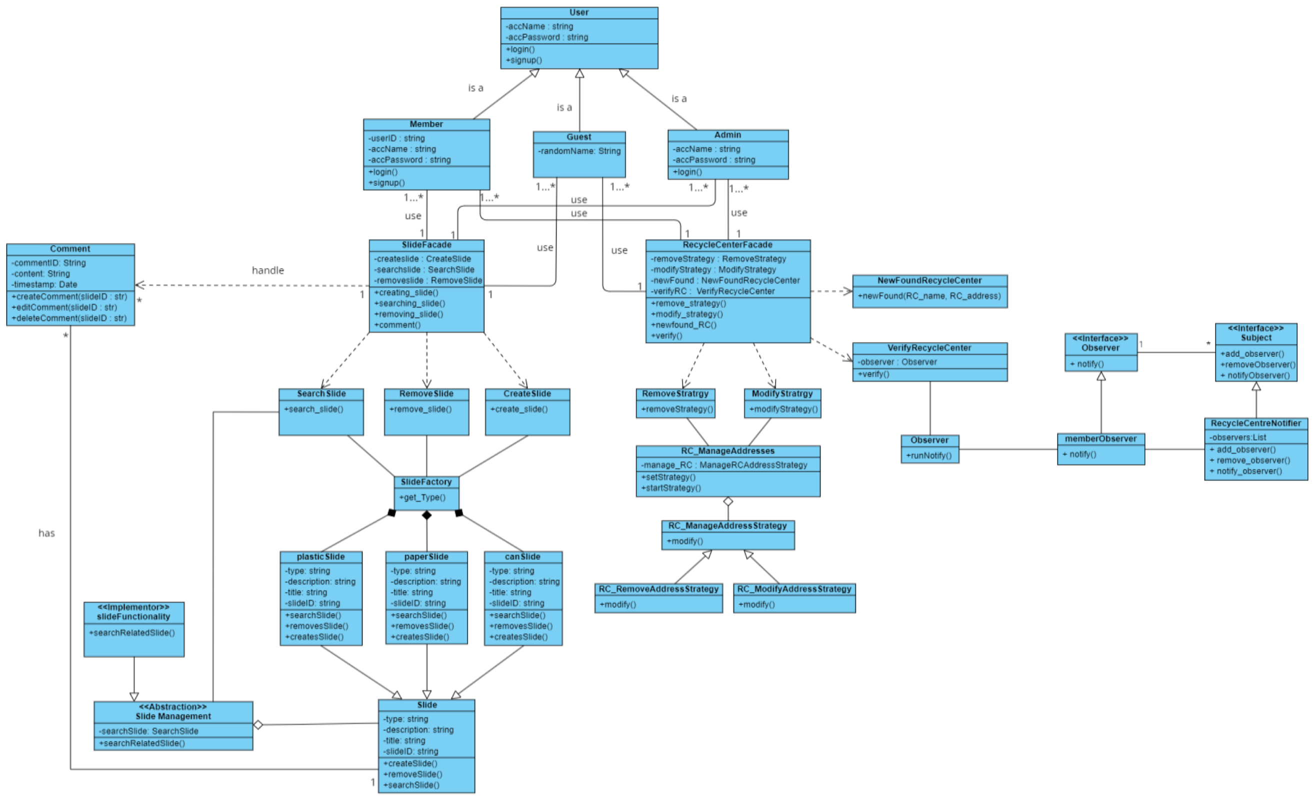


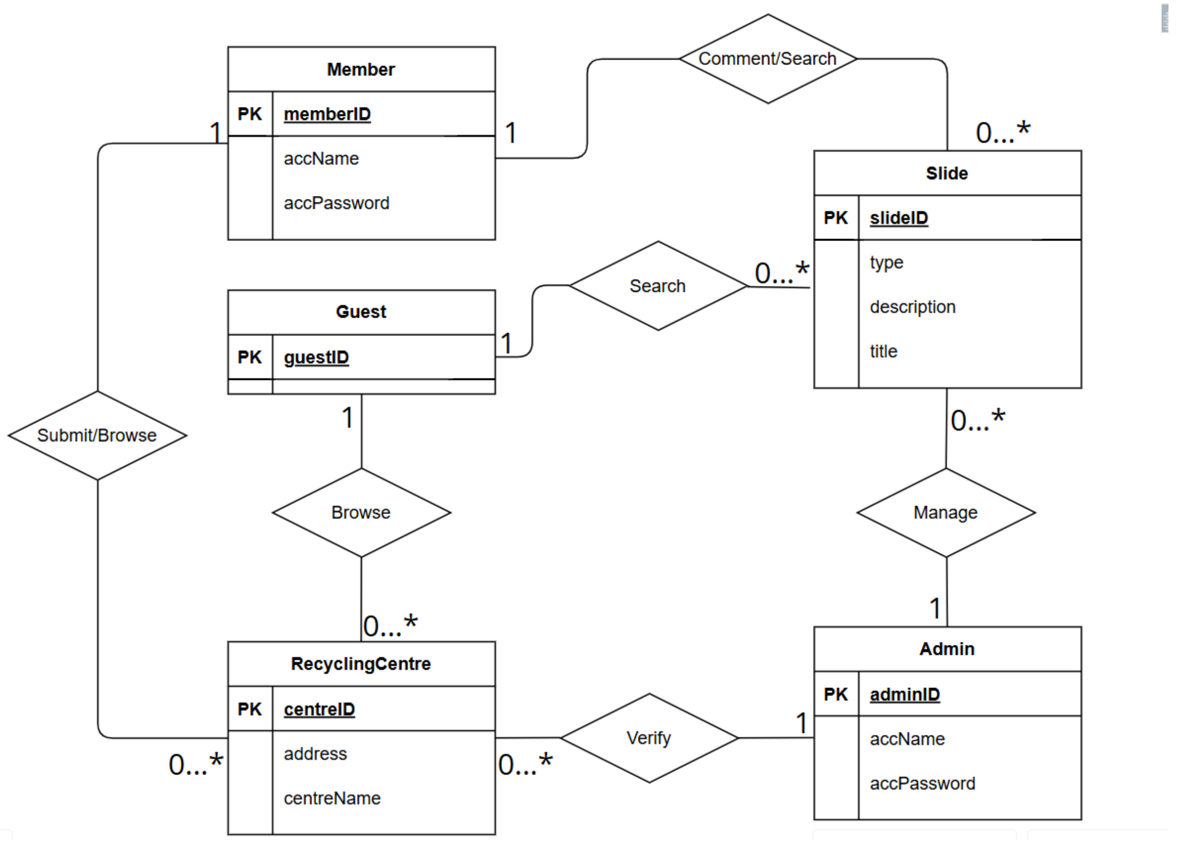
Figure 2.2.1 Class Diagram

2.3. Entity Relationship Model

This Entity-Relationship Diagram (ERD) models a recycling education system. It includes four main entities: **Member**, **Guest**, **Admin**, **Slide**, and **RecyclingCentre**.

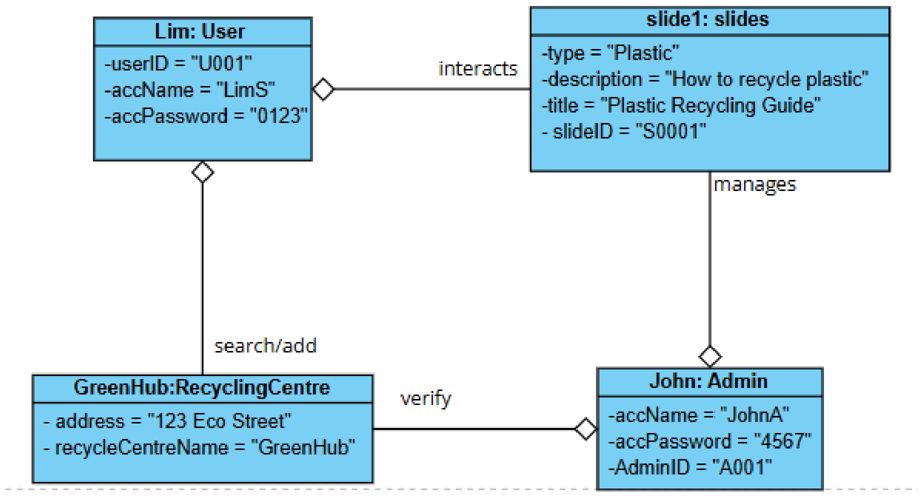
- **Member** can search and submit new locations for recycling centers and comment as well as search for educational slides.
- **Guest** can browse for recycling centers' locations as well as search for educational slides
- **Admin** manage slides and verify recycling centers' locations provided by the members.
- **Slide** provide information about different recycling topics.
- **RecyclingCentre** store location and name details.

All relationships are simple associations, representing user interactions and admin management tasks within the system.



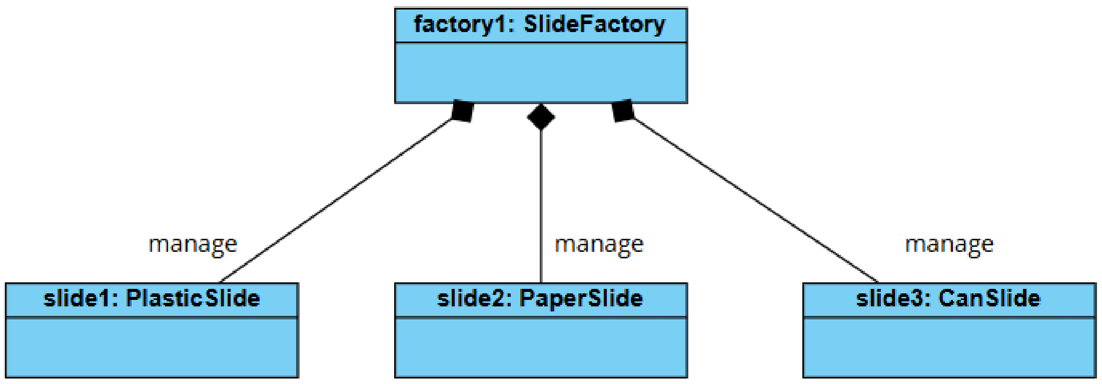
2.4. Object Diagrams

This object diagram illustrates a recycling management system with four main components: a user (Lim) who can search, add educational slides and verify recycling centers, an admin (John) who manages the system, a recycling center (GreenHub) with location details, and educational content about plastic recycling. The aggregation relationship in this diagram represents a “has-a” relationship where the user can access the slides, but the slides can exist independently. These relationships show user interactions with resources and administrative control over the system.

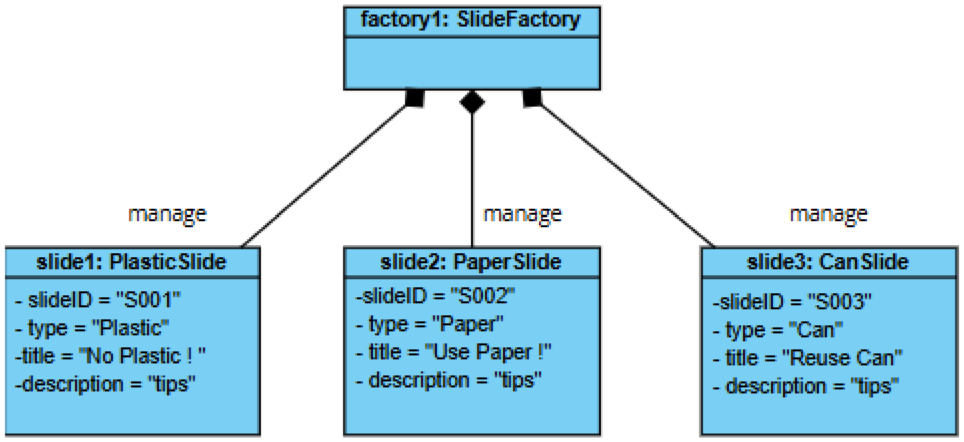


2.4.1 Main system’s object diagram

This diagram shows a SlideFactory that creates and manages specialized recycling slides (PlasticSlide, PaperSlide, CanSlide) through composition. This factory pattern controls these object’s lifecycle to ensure its scalability when adding new slide types.

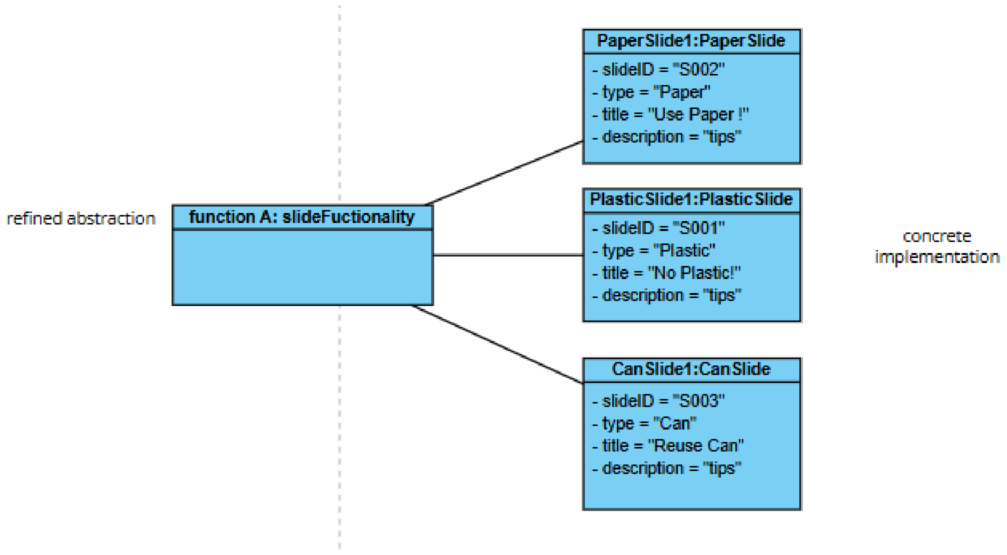


2.4.2.1 Factory pattern object diagram



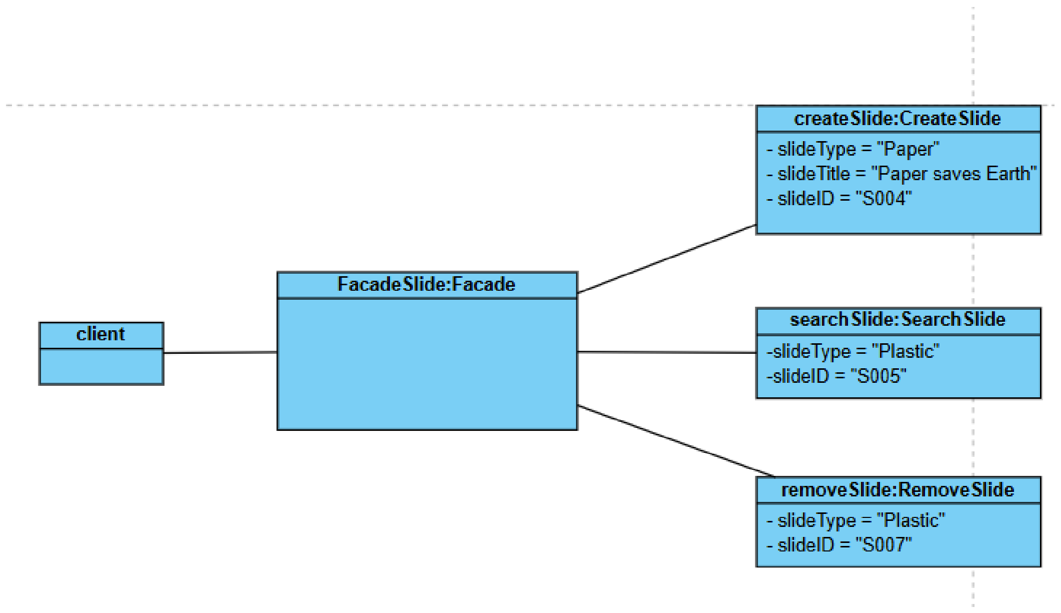
2.4.2.2 Factory pattern example

This diagram shows a Bridge pattern implementation where refined abstractions (like "slideFunctionality") are separated from concrete implementations ("PaperSlide1", "PlasticSlide1", "CanSlide1"). This structure allows different types of recycling slides to share common functionality while maintaining flexibility for future extensions or modifications without altering the core system.



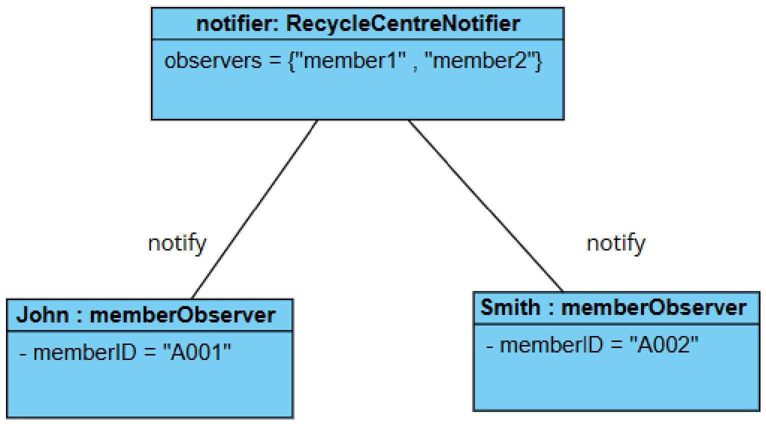
2.4.3.2 Bridge pattern example 1

This diagram shows the Facade pattern, where “FacadeSlide” provides a simplified interface (with methods like “createSlide”, “searchSlide”, and “removeSlide”) to a complex subsystem of slide operations. This interface hides implementation details, making the system easier to use while maintaining its full functionality.



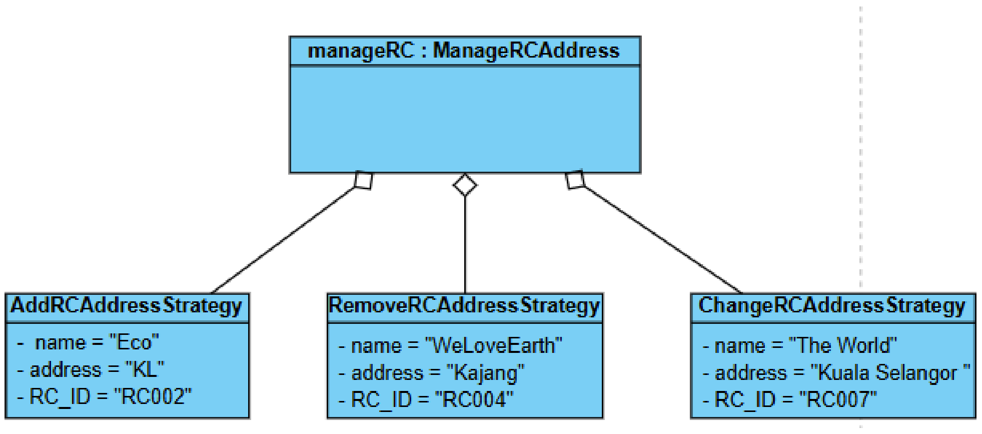
2.4.4.2 Facade design pattern object diagram

This diagram shows the Observer pattern, where “RecycleCentreNotifier” (subject) maintains a list of observers ("member1", "member2") and automatically notifies them (like John: “memberObserver”) of changes. This decouples the notification system from observers, enabling dynamic updates without tight dependencies.



2.4.5.2 Observer design example 1

This diagram shows the Strategy pattern, where “ManageRCAddress” can switch between different address management strategies (“AddRCAddressStrategy”, “RemoveRCAddressStrategy”, “ChangeRCAddressStrategy”). This design encapsulates each algorithm separately, making them interchangeable without modifying the core management logic.



2.4.6.2 strategy pattern example 1

3. Functional and Non-Functional Requirements

Functional Requirements:

1. User Management:

    - The system shall allow users to register as members or browse as guests
    - Members shall be able to submit new recycling center locations for admin verification.
    - Guests shall be able to browse recycling center locations and educational slides without registration.

2. Recycling Center Management:

    - The system shall allow admins to verify, add, remove, or modify recycling center locations.
    - Users shall be able to search for nearby recycling centers by type (e.g., plastic, paper, cans)
    - The system shall display recycling center details (name, address, accepted materials)

3. Educational Slides Management:

    - The system shall allow admins to create, update, or remove educational slides (e.g., infographics, best practices)
    - User shall be able to search for slides by type (e.g., plastic, paper, cans)
    - The system shall categorize slides by waste type for easy navigation.

4. Notification System:

    - The system shall notify admins when a new recycling center location is submitted by a member.

Non-Functional Requirements:

1. Usability:

    - The system shall be intuitive, with clear navigation and a minimal learning curve for users.
    - The interface shall support both desktop and mobile devices for accessibility.

2. Performance:

    - Search results for recycling centers or slides shall load within 3 seconds under normal conditions.
    - The system shall handle up to 50 concurrent users without significant latency.

3. Security:

    - User data (e.g., member details) shall be encrypted and stored securely.
    - Admin actions (e.g., modifying slides or centers) shall require authentication.

4. Reliability:

    - Data backups shall be performed daily to prevent loss.

5. Scalability:

    - The database shall support increasing volumes of user submissions and content.

6. Maintainability:

    - The codebase shall follow modular design principles (e.g., using Factory, Bridge, and Facade patterns) for easy updates.

    - Documentation shall be provided for all design patterns and system components.

7. Compatibility:

- The system shall work on major browsers (e.g., Chrome, Firefox, Safari).

3.1 Software Design Concepts and Design Principles

Concept/Principle	Explanation	How It Current and Future Use in This Project
Abstraction	Hide complicated operations behind simple methods.	Users only need to call “slidefacade.creating_slide()”, they don't know the internal factory logic.
Modularity	Separate a big module into small pieces.	Organized the code into different classes.Such as, “Slide”, “SlideFactory”, ”“SlideFacade, “RC_ManageAddresses”
Encapsulation	Each class bundles data and functions together	Other classes cannot directly modify the lists paperslide, “plasticslide” and “canslide” outside the class.
Pattern	A standardized way to address a recurring problem.	Factory, Bridge, Facade, Strategy, Facade design patterns.
Single Responsibility Principle (SRP)	Each class has one job only.	“CreateSlide” only creates slides, “RemoveSlide” only removes available slides.
Open/Closed Principle (OCP)	The system can add new slide types by creating a new class, without changing existing classes.	“Slide” class can be extended without modifying existing code.

3.1 Table of Used Software Design Concepts and Design Principles

4. Proposed Design Patterns

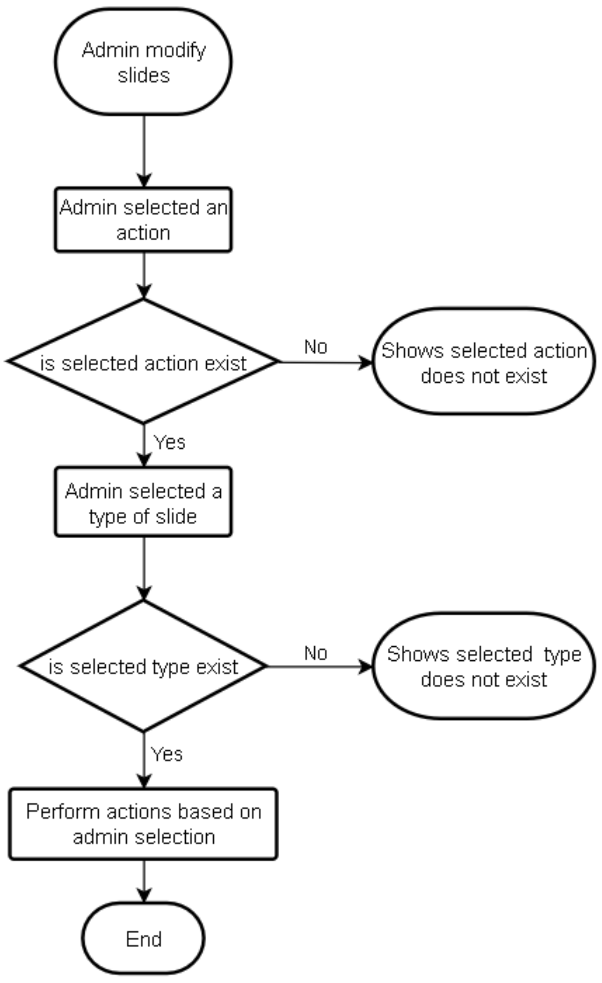
4.1. Factory Design Pattern

4.1.1. Function or Software Component Affected

The Factory Design Pattern affects the “SlideFactory” class, which is used for centralizing the object creation of different slide types in the waste management system. This pattern makes it easier to manage and extend different slide types without changing client code. All specific slide types, such as, “PlasticSlide”, “PaperSlide” and “CanSlide”, inherited from the abstract “Slide” class. It ensures all the slide types were implemented common methods like “createSlide()” and “removeSlide()”. As a result, this pattern reduces direct dependencies in the client code and allows new slide types to be added easily by only modifying the factory.

4.1.2. Description of Work Flow Diagram

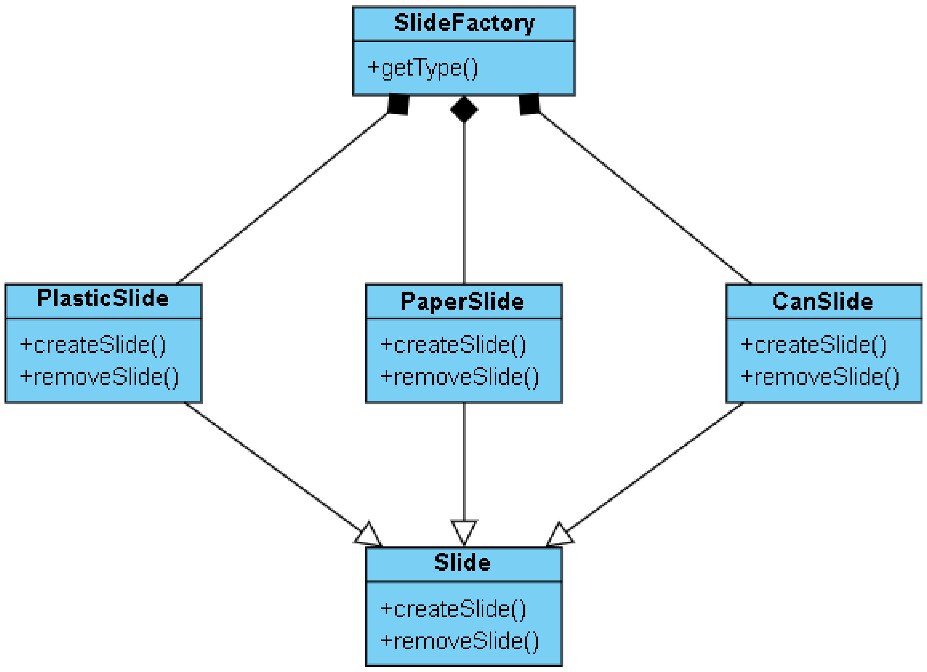
This work flow diagram shows the process by which an admin modifies different types of slides within a system. The process begins with the admin selecting an action (remove, create). If the action does not exist, an error message is displayed; otherwise, the admin proceeds to select a type of slide. Then, the system verifies whether the selected slide type exists. If it does not, an error message is displayed. After successful validation of both the action and the slide type, the system performs the specified action based on the admin's selection.


Admin modify slides
Admin selected an action
is selected action exist
No
Shows selected action does not exist
Yes
Admin selected a type of slide
is selected type exist
No
Shows selected type does not exist
Yes
Perform actions based on admin selection
End

4.1.2 Factory Design Pattern Workflow Diagram

4.1.3. Sample Class Diagram

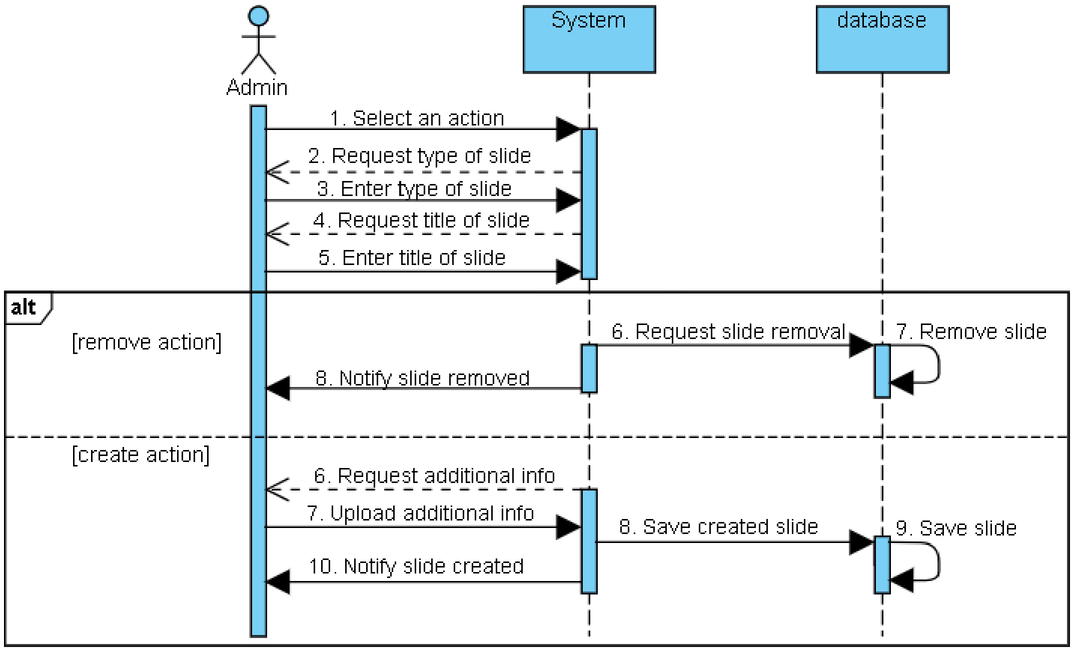
This class diagram shows a simple implementation of the Factory Design Pattern centered on creating different types of slides. In the core, there is a "Slide" class that defines two common methods: "createslide()" and "removeslide()", which is used for inherited by all slide types. To manage the creation and deletion process, there is a "SlideFactory" class, which uses a "getType()" method to determine the correct type of slide based on user input. The "SlideFactory" to each type of slide class indicates a strong composition relationship, meaning the "SlideFactory" is responsible for managing the lifecycle of the slide objects.



4.1.3 Factory Design Pattern Class Diagram

4.1.4. Sequence Diagram

This sequence diagram shows the interaction between an admin, system and slide storage during the slide modification process. The process begins with the admin selecting an action, after the system requests and receives the type and title of the slide. Depending on the selected action, two situations occur: If the action is to remove a slide, the system removes the slide and notifies the admin once the operation is complete; If the action is to create a slide, the system requests additional information, receives the uploaded data, saves the new slide in slide storage and notifies the admin of the successful creation.



4.1.4 Factory Design Pattern Sequence Diagram

4.1.5. Sample Potential Code

```python
class Slide:
    def createSlide(self, Title):
        pass
    def removeSlide(self, Title):
        pass

paperslide = ["Slide Title 3","Slide Title 4"]
class paperSlide(Slide):
    def createSlide(self, Title):
        print(f"Title is {Title}")
        paperslide.append(Title)
    def removeSlide(self, Title):
        print(f"Removing...: {Title}")
        paperslide.remove(Title)

plasticslide = ["Slide Title 1","Slide Title 2"]
class plasticSlide(Slide):
    def createSlide(self, Title):
        print(f"Title is {Title}")
        plasticslide.append(Title)
    def removeSlide(self, Title):
        print(f"Removing...: {Title}")
        plasticslide.remove(Title)

canslide = ["Slide Title 5","Slide Title 6"]
class canSlide(Slide):
    def createSlide(self, Title):
        print(f"Title is {Title}")
        canslide.append(Title)
    def removeSlide(self, Title):
        print(f"Removing...: {Title}")
        canslide.remove(Title)

#Factory
class SlideFactory:
    def get_Type(self, slide_type: str):
        if slide_type is None:
            return None
        slide_type = slide_type.lower()
        if slide_type == "paper":
            return paperSlide()
        elif slide_type == "plastic":
            return plasticSlide()
        elif slide_type == "can":
            return canSlide()
        else:
            print("Selected type does not exist")
            return None

#client code
print("Removing slide...")
factory = SlideFactory()
slideType = input("Enter a type (paper/plastic): ")
slide = factory.get_Type(slideType)
slideTitle = input("Enter a slide Title to remove the slide: ")
slide.removeSlide(slideTitle)
print("Slide Removed \n")
```

4.1.6. Benefit

Flexibility to Add New Slide Types:

The Factory Design Pattern allows the system to easily add new slide types, such as "GlassSlide", "MetalSlide", without modifying existing client code. Only the factory class needs to be updated.

Encapsulation of Object Creation:

The details of how each specific slide type is created are hidden inside the factory, reducing duplication and complexity in the client code.

4.1.7. Limitations

Limited to Defined Types:

The factory must know all types beforehand. If new types of slides need to be added, modifications to the factory may still be required.

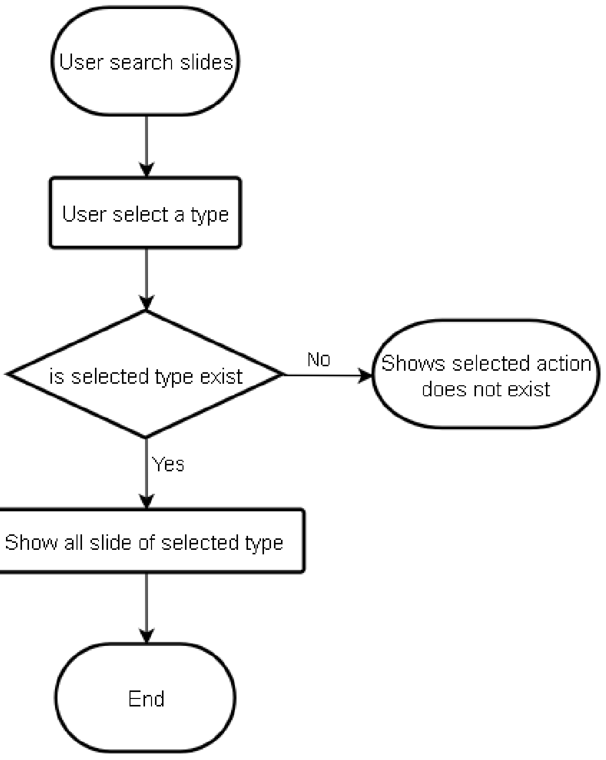
4.2. Bridge Design Pattern

4.2.1. Function or Software Component Affected

The Bridge Design Pattern in this system is applied between “SlideManagement” and “Slide” classes. Instead of linking the management logic to any one slide type, the “SlideManagement” class interacts with the abstract “Slide” object, allowing flexibility to work with any type of slide. This pattern ensures that if new types of slides are added, the core management logic does not need to be modified. Only the specific slide types need to be extended. Thus, this pattern improves the system's flexibility and maintainability by independently evolving both the management side and the specific slide types.

4.2.2. Description of Work Flow Diagram

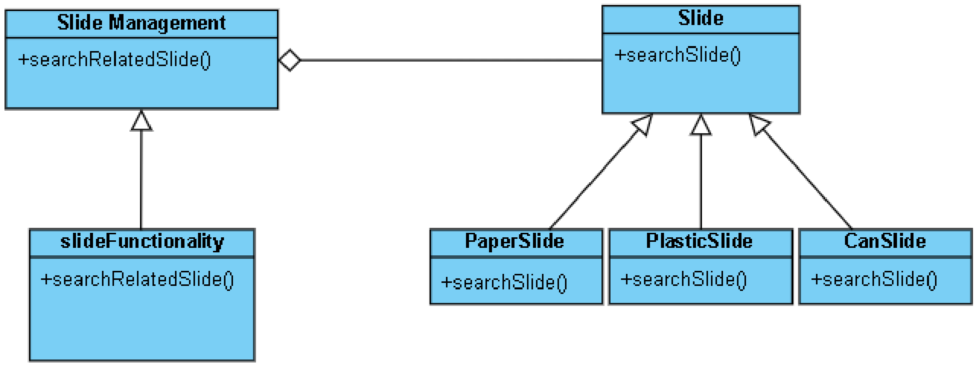
This work flow diagram shows the process of how a user searches for slides based on type. First, the user is required to select a type, and the system checks whether the selected type exists. If it exists, the system displays all slides of that type. If it does not, the system notifies the user that the selected type does not exist. The process ends either when all related slides are displayed or the selected type does not exist.


User search slides
User select a type
is selected type exist
No
Shows selected action does not exist
Yes
Show all slide of selected type
End

4.2.2. Bridge Design Pattern Work Flow Diagram

4.2.3. Sample Class Diagram

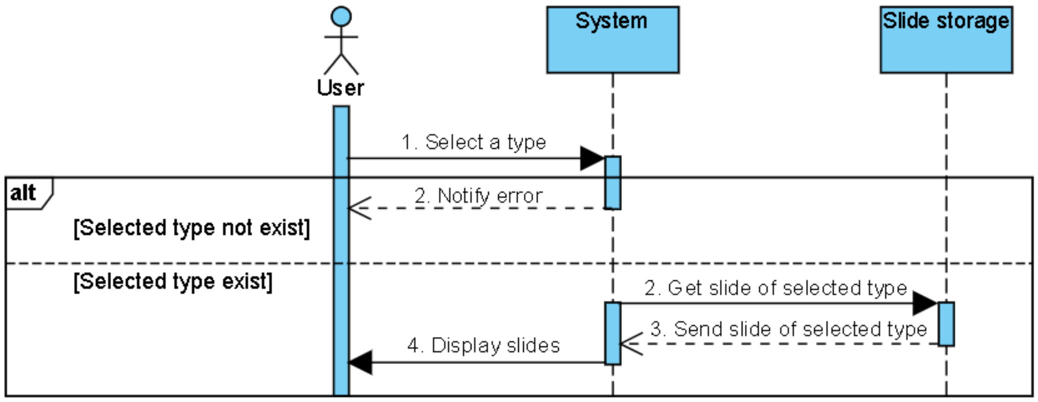
This class diagram shows the Bridge Design Pattern to separate slide management from specific slide types. The “SlideManagement” class defines the abstraction for managing slides and connects to the abstract “Slide” class, while “SlideFunctionality” extends “SlideManagement” to implement the search behavior. On the implementation side, the “Slide” class is inherited by “PaperSlide”, “PlasticSlide”, and “CanSlide”, each providing their own version of the “searchSlide()” method. This pattern allows slide management and specific slide types to vary independently, making the system more flexible and easier to extend when adding new slide types.



4.2.3. Bridge Design Pattern Class Diagram

4.2.4. Sequence Diagram

This sequence diagram shows the interaction between a user, system and slide storage during the slide search process. The process starts with the user selecting a type of slide. The system checks if the selected type exists. If the type does not exist, the system notifies the user of the error. If the type exists, the system gets the slides of the selected type from slide storage and displays it to the user.



4.2.3. Bridge Design Pattern Sequence Diagram

4.2.5. Sample Potential Code

```python
class Slide:
    def searchSlide(self):
        Pass

paperslide = ["Slide Title 3","Slide Title 4"]
class paperSlide(Slide):
    def searchSlide(self):
        print(f"The slide about paper : ")
        for i in paperslide:
            print(i)

plasticslide = ["Slide Title 1","Slide Title 2"]
class plasticSlide(Slide):
    def searchSlide(self):
        print(f"The slide about plastic : ")
        for i in self.plasticslide:
            print(i)

canslide = ["Slide Title 5","Slide Title 6"]
class canSlide(Slide):
    def searchSlide(self):
        print(f"The slide about plastic : ")
        for i in self.canslide:
            print(i)

class SlideFactory:
    def get_Type(self, slide_type: str):
        if slide_type is None:
            return None
        slide_type = slide_type.lower()
        if slide_type == "paper":
            return paperSlide()
        elif slide_type == "plastic":
            return plasticSlide()
        elif slide_type == "can":
            return canSlide()
        else:
            print("Selected type does not exist")
            return None

class SlideManagement:
    def __init__(self, searchSlide):
        self.searchSlide = searchSlide

    def searchRelatedSlide(self):
        pass

class SlideFunctionality(SlideManagement):
    def __init__(self, searchSlide):
        super().__init__(searchSlide)
    def searchRelatedSlide(self):
        self.searchSlide.searchSlide()

print("Searching slide...")
factory = SlideFactory()
findType = input("Enter a type (paper/plastic/can): ")
slide = factory.get_Type(findType)
searchingSlide = SlideFunctionality(slide)
searchingSlide.searchRelatedSlide()
```

4.2.6. Benefit

Scalability:

New types of slides or new management features can be added independently without modifying existing code.

Flexibility:

Different slide types can be handled uniformly without the management code needing to know the specific details.

4.2.7. Limitations

Requires Consistent Interface Implementation:

The Bridge Pattern relies on all concrete slide types must correctly implementing the methods defined in the "Slide" interface. If any subclass fails to implement the expected behavior, it could lead to runtime issues during slide management.

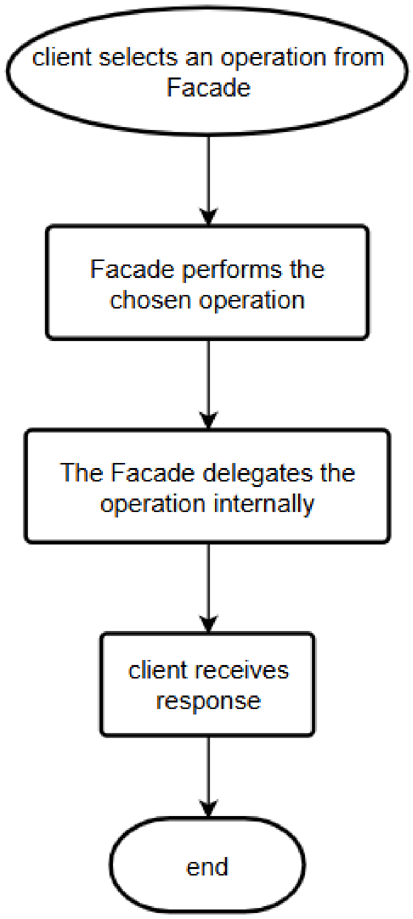
4.3 Facade Design Pattern

4.3.1. Function or Software Component Affected

The Facade Design Pattern affects the "FactorySlide" class, which acts as a unified interface to simplify interactions with the subsystem components (CreateSlide , SearchSlide , RemoveSlide) in the waste management system. This pattern ensures clients interact only with the FacadeSlide interface, hiding the internal subsystem complexity. Clients can call simplified methods like creating_slide(), removing_slide(), and searching_slide(), while the Facade delegates these tasks to the appropriate backend components. As a result, it reduces direct dependencies in client code and improves slide management.

4.3.2. Description of work flow or data flow

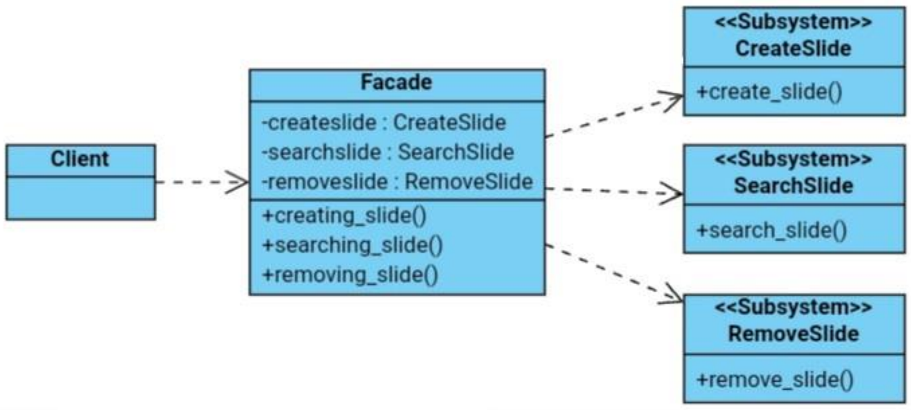
This workflow illustrates how a client interacts with the waste management system through the Facade to manage slides. The process begins when the client calls a high-level method on the FacadeSlide interface (e.g., “create_slide()”, “remove_slide()”, or “search_slide()”). The Facade then internally delegates the request to the appropriate subsystem component (CreateSlide, RemoveSlide, or SearchSlide) without exposing these details to the client.


client selects an operation from Facade
Facade performs the chosen operation
The Facade delegates the operation internally
client receives response
end

4.3.2.1 Facade Design Pattern Work Flow Diagram

4.3.3. Sample Class Diagram

This class diagram illustrates a simple implementation of the Facade Design Pattern focused on performing various slide-related operations. At the center is the “FacadeSlide” class, which provides three primary methods: “creating_slide()”, “searching_slide()”, and removing_slide(). These methods internally use attributes such as “CreateSlide()”, “SearchSlide()”, and “RemoveSlide()” to invoke the appropriate operations in their respective subsystems. The subsystems contain specific methods “create_slide()”, “search_slide()”, and “remove_slide()”—that carry out the actual operations. There is a dependency relationship between the client and the “FacadeSlide” class, as well as between the “FacadeSlide” class and the subsystem classes. The purpose of the “FacadeSlide” class is to offer the client the desired operations while hiding the underlying complexity of the subsystems.



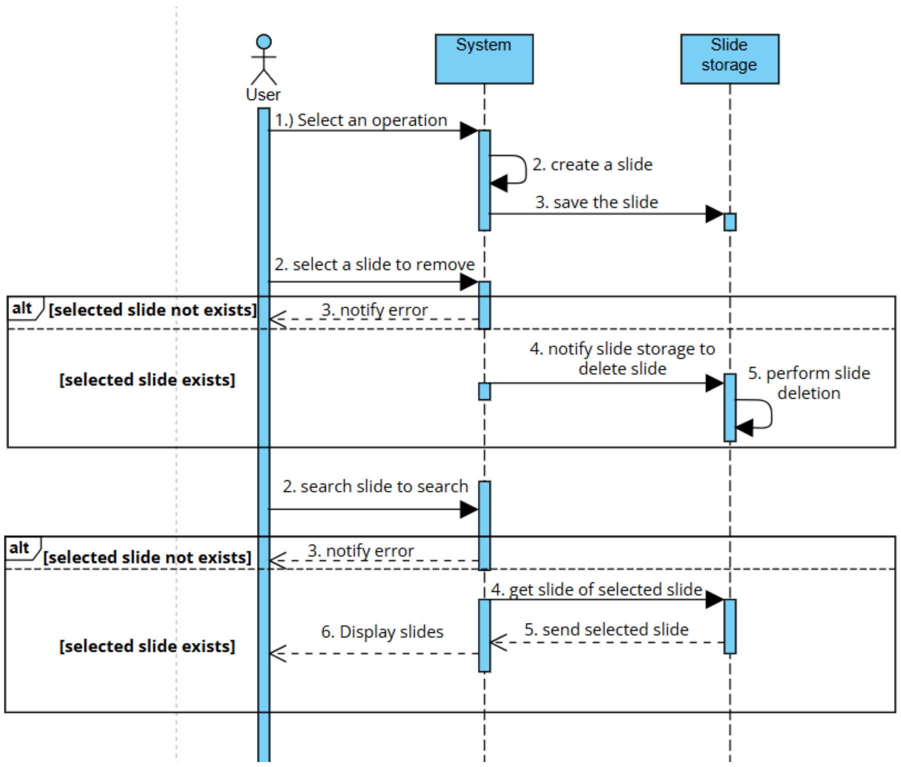
4.3.3.1 Facade design Pattern Class Diagram

4.3.4. Sequence Diagram

This sequence diagram illustrates the interaction between a user, the system, and slide storage during a search operation. The process begins when the user selects an operation. If the user chooses the "create" operation, a new slide is created and stored in slide storage.

If the user selects the "remove" operation, the system checks whether the selected slide type exists. If the specified slide does not exist, the system notifies the user of the error. If the slide exists, the system notifies slide storage, and the slide is then deleted.

If the user selects the "search" operation, the system again checks if the selected slide type exists. If it does not, the user is notified of the error. If the slide exists, the system retrieves the slides of the selected type from slide storage and displays them to the user.


User
System
Slide storage
1.) Select an operation
2. create a slide
3. save the slide
2. select a slide to remove
alt [selected slide not exists]
3. notify error
[selected slide exists]
4. notify slide storage to delete slide
5. perform slide deletion
2. search slide to search
alt [selected slide not exists]
3. notify error
[selected slide exists]
4. get slide of selected slide
5. send selected slide
6. Display slides

4.3.4.1 Facade Design Pattern Sequence Diagram

4.3.5. Sample Potential Code

```
#Facade design pattern
class CreateSlide:
    def create_slide(self):
```

```python
        print("Creating slide...")
        factory = SlideFactory()
        slideType = input("Enter a type (paper/plastic): ")
        slide = factory.get_Type(slideType)
        slideTitle = input("Enter a slide Title for the slide: ")
        slide.createSlide(slideTitle)
        print("Slide Created \n")


class RemoveSlide:
    def remove_slide(self):
        print("Removing slide...")
        factory = SlideFactory()
        slideType = input("Enter a type (paper/plastic): ")
        slide = factory.get_Type(slideType)
        slideTitle = input("Enter a slide Title to remove the slide: ")
        slide.removeSlide(slideTitle)
        print("Slide Removed \n")


class SearchSlide:
    def search_slide(self):
        print("Searching slide...")
        factory = SlideFactory()
        findType = input("Enter a type (paper/plastic): ")
        slide = factory.get_Type(findType)
        searchingSlide = SlideFunctionality(slide)
        searchingSlide.searchRelatedSlide()

class SlideFacade:
    def __init__(self):
        self.createslide = CreateSlide()
        self.searchslide = SearchSlide()
        self.removeslide = RemoveSlide()

    def creating_slide(self):
        self.createslide.create_slide()

    def removing_slide(self):
        self.removeslide.remove_slide()

    def searching_slide(self):
        self.searchslide.search_slide()
```

```python
#client code
slidefacade = SlideFacade()
action_slide = input("Choose an action to modify slides: ")
action_slide.lower()
if action_slide == "create":
    slidefacade.creating_slide()
elif action_slide == "search":
    slidefacade.searching_slide()
elif action_slide == "remove":
    slidefacade.removing_slide()
else:
    print("Selected action does not exist")
```

4.3.6. Benefit

Simplified Access to Complex Subsystems

The Facade Design Pattern hides the complexities of slide creation, search, and removal. The client interacts only with the “FacadeSlide” class, which internally uses a factory to instantiate the correct slide type (e.g., “GlassSlide”, “MetalSlide”), making the interface simpler and cleaner.

Improved Maintainability and Loose Coupling

By using a facade over a factory-based system, changes in the internal slide creation logic (handled by the Factory) do not affect the client. This separation of concerns improves maintainability and promotes loose coupling between the client and subsystems.

4.3.7. Limitations

Reduced Transparency of Underlying Logic

Since the Facade hides subsystem details, developers may have less visibility into how slides are created (e.g., via a factory). This can make debugging or extending functionality more challenging if internal behavior needs to be understood.

Increased Responsibility in Facade Class

When the Facade class integrates both factory logic and coordination of multiple operations (create, search, remove), it can become overly complex or bloated as the number of subsystems increases.

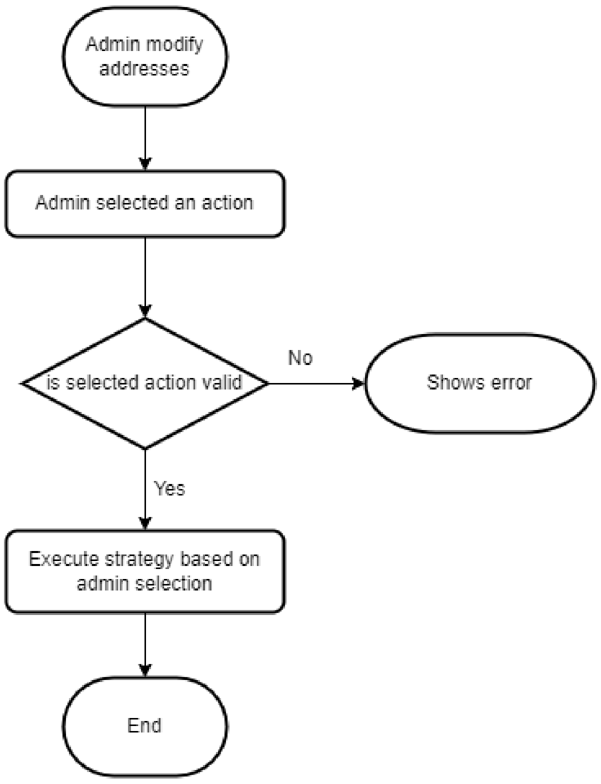
4.4. Strategy Design Pattern

4.4.1. Function or Software Component Affected

The Strategy Design Pattern affects the “RC_ManageAddresses” class, which is used for centralizing and dynamically switching between different algorithms to manage recycle center addresses. This pattern makes it easier to handle operations such as adding the addresses, removing and modifying. Strategies such as“RC_RemoveAddressStrategy”, and “RC_ModifyAddressStrategy” implement a common interface that defines methods like “modify()”. As a result, this will allow the system to adopt different strategies without affecting the core implementation and allow new strategies to be added easily by only modifying the strategy class.

4.4.2. Description of workflow or data flow

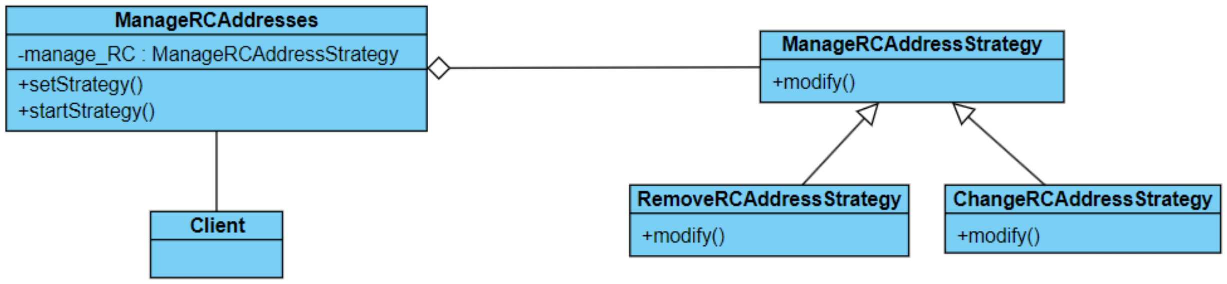
This workflow diagram shows the process by which an admin modifies recycle center addresses by using different strategies. The process begins with the admin selecting an action (remove, modify). If the actions do not exist, an error message is displayed; otherwise, the system will execute the specified strategy action based on the admin’s selection.



4.4.2.1 Strategy Design Pattern Workflow Diagram

4.4.3. Sample Class Diagram

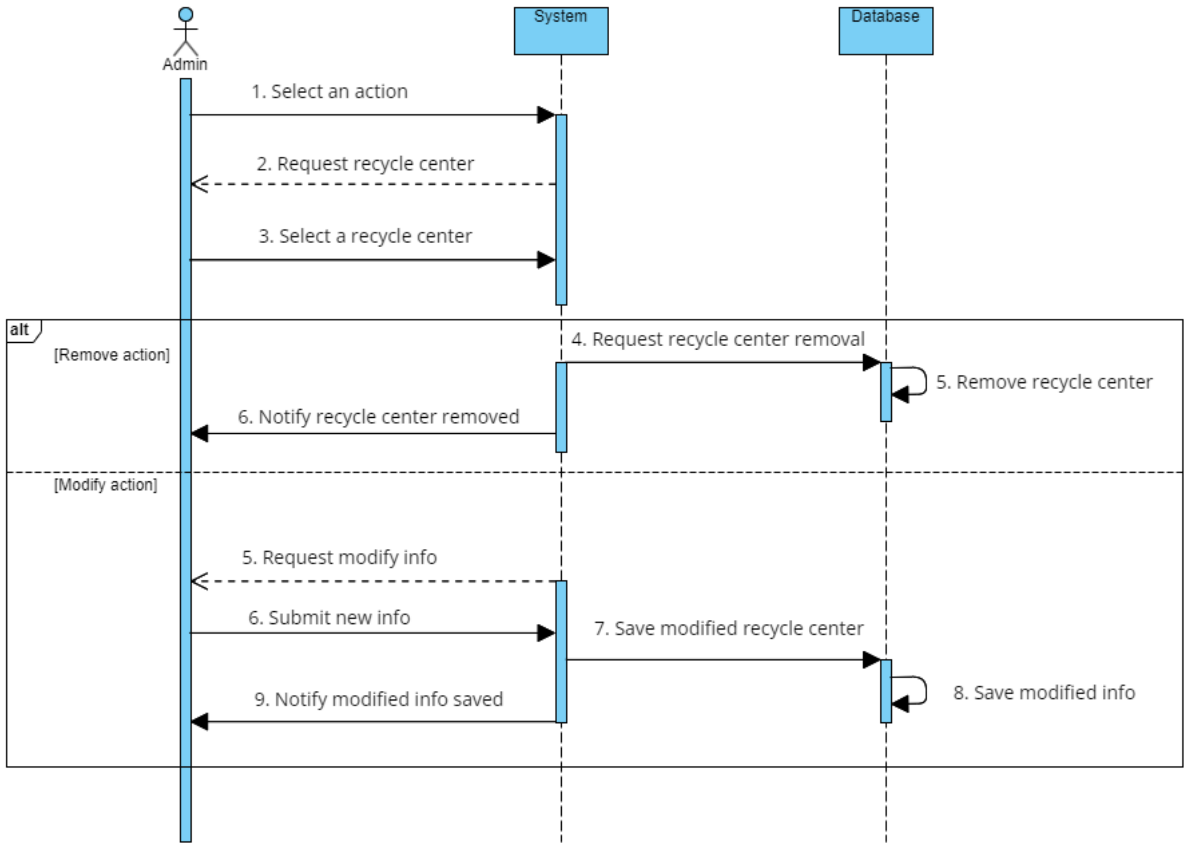
This class diagram shows a simple implementation of the Strategy Design Pattern, designed to be dynamically manage recycle addresses. In the core, the "RC_ManageAddressStrategy" interface defines a common method, which is "modify()". The common method will be implemented by three concrete strategy classes: "RC_RemoveAddressStrategy" and "RC_ModifyAddressStrategy". The "ManageRCAddressStrategy" is responsible for managing the lifecycle of the strategy objects to ensure that the appropriate strategy is created dynamically.



4.4.3.1 Strategy Design Pattern Class Diagram

4.4.4. Sequence Diagram

This sequence diagram shows the interaction between an admin, system and database during the recycle center modification process. The process begins with the admin selecting and action. Then the system will request a recycle center . After the admin selects it and system has received the recycle center, depending on the selected action, two situations will occur. The first one is if the action is to remove the recycle center, the system will remove the recycle center and notify the admin once the operation is complete. The second situation is if the action is to modify the recycle center, the system will request modified information. After the admin submit the information and the system receive it, it will save the new information in database and notifies the admin.


Admin
System
Database
1. Select an action
2. Request recycle center
3. Select a recycle center
alt
[Remove action]
4. Request recycle center removal
5. Remove recycle center
6. Notify recycle center removed
[Modify action]
5. Request modify info
6. Submit new info
7. Save modified recycle center
8. Save modified info
9. Notify modified info saved

4.4.4.1 Strategy Design Pattern Sequence Diagram

4.4.5. Sample Potential Code

```python
class RC_ManageAddressStrategy:
    def modify(self, RC_name ,address):
        pass

class RC_ModifyAddressStrategy(RC_ManageAddressStrategy):
    def modify(self, RC_name ,address):
        RC_dict[RC_name] = address

class RC_RemoveAddressStrategy(RC_ManageAddressStrategy):
    def modify(self, RC_name, address):
        if RC_name in RC_dict:
            RC_dict.pop(RC_name)
            print(f"{RC_name} has been removed from the platform.")
        else:
            print("Not found.")

class RC_ManageAddresses:
    def __init__(self, strategy):
        self.strategy = strategy

    def setStrategy(self, strategy):
        self.strategy = strategy

    def startStrategy(self, RC_name, address):
        self.strategy.modify(RC_name,address)
```

4.4.6. Benefit

Extensibility:

The Strategy Design Patterns allows new strategies can be added in the future without modifying existing code

Improve Code Readability:

The Strategy Design Patterns make the code cleaner and easier to update by isolating each behavior in separate classes.

4.4.7. Limitations

Increased Number of Class:

Using Strategy Design Patterns will increase the number of classes by introducing multiple strategies classes. This may lead to over-engineering.

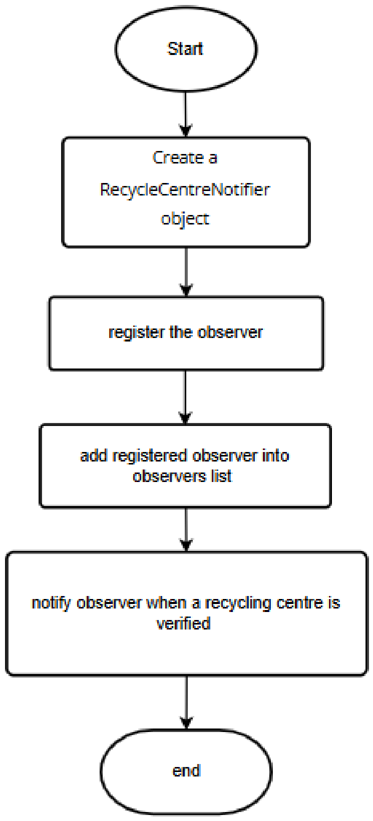
4.5. Observer Design Pattern

4.5.1. Function or Software Component Affected

The Observer Design Pattern affects the “RecycleCentreNotifier” class and “memberObserver” class. The “RecycleCentreNotifier” sets up the subject and “memberObserver” would register member as observers. When one subject changes state, its dependents or observers are notified. In our program it is used as a notification system to notify the members when a new verified recycling center's name and address is introduced.

4.5.2. Description of work flow or data flow

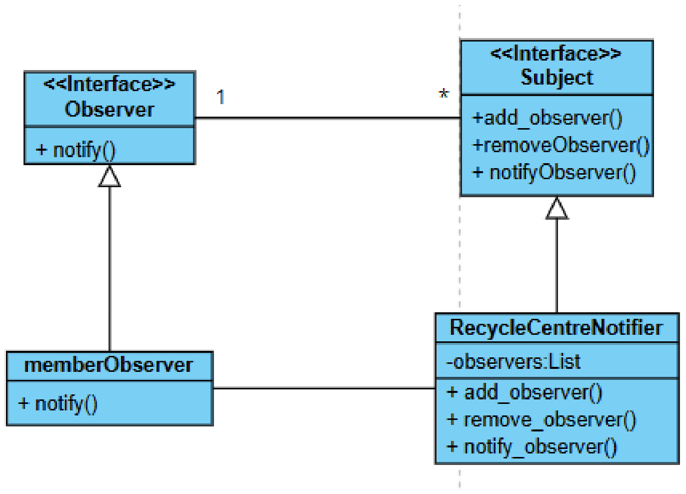
This workflow diagram shows the process of how the subject object is created and how observers are added into the notifier’s list. Once it detects a new verified recycling center has been passed into “observer” class, it will notify all the observers in the notifier’s list.


Start
Create a RecycleCentreNotifier object
register the observer
add registered observer into observers list
notify observer when a recycling centre is verified
end

4.5.2 Observer Design Pattern Workflow Diagram

4.5.3. Sample Class Diagram

This class diagram shows a simple implementation of the Observer Design Pattern centered around notifying administrative users when a new verified recycling center is added. At the core is the Observer interface, which defines a common method “notify()” that must be implemented by all concrete observers. The “memberObserver” class implements this interface and provides a specific behavior for handling notifications, such as printing a message. The “RecycleCentreNotifier” class acts as the concrete subject inherited from “Subject” interface in the pattern. It maintains a list of observer instances (“adminObserver” objects) and provides methods to “add_observer()”, “remove_observer()”, and “notify_observer()”—which calls “notify()” on each registered observer. The relationship between “RecycleCentreNotifier” and “memberObserver” is an association relationship, indicating that one notifier can notify many observers. The triangle-headed arrow from “memberObserver” to “Observer” interface represents an implementation (realization) relationship, showing that “memberObserver” implements the Observer interface.



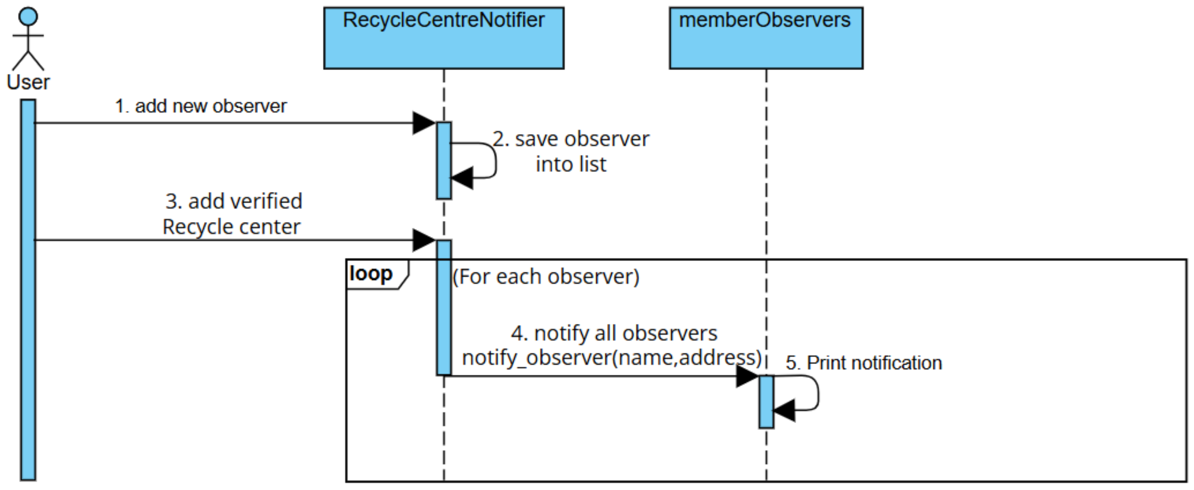
4.5.3 Observer Design Pattern Class Diagram

4.5.4. Sequence Diagram

This sequence diagram illustrates the interaction between a user, the notification system ("RecycleCentreNotifier"), and multiple admin observers during the recycling center submission process. The process begins when the user adds new admin observers to the notification system. These observers are stored internally by the "RecycleCentreNotifier".

Once the user submits the name and address of a new recycling center, the notifier triggers a loop that iterates through all registered observers. For each observer, the "notify()" method is invoked, sending the relevant details. The observers, in turn, handle the notification—typically by displaying a confirmation message acknowledging the new submission.

The observers remain passive and unaware of each other, and the subject (the notifier) is responsible for broadcasting updates to all observers without needing to know their specific implementations.


User
RecycleCentreNotifier
memberObservers
1. add new observer
2. save observer into list
3. add verified Recycle center
loop
(For each observer)
4. notify all observers notify_observer(name,address)
5. Print notification

4.5.4 Sequence Design Pattern Sequence Diagram

4.5.5. Sample Potential Code

```
class Observer():# Observer
    def notify(self , recycle_centre , address):

class Subject(): #Subject
    def add_observer(self,observer):
        pass
    def remove_observer(self,observer):
        pass
    def notify_observer(self , recycle_centre_name, address):
        pass

class RecycleCentreNotifier(Subject): # concrete subject
    def __init__(self):
        self.observers = []

    def add_observer(self,observer):
        self.observers.append(observer)

    def remove_observer(self,observer):
        self.observers.remove(observer)

    def notify_observer(self, recycle_centre_name, address):
        for observer in self.observers:
            observer.notify(recycle_centre_name, address)

class memberObserver(Observer): # concrete observer
    def __init__(self , member_name):
        self.member_name = member_name

    def notify(self, recycle_centre_name , address):
        print(f"[{self.member_name}] Notification: New Recycle Centre Submitted -
{recycle_centre_name} at {address} verified by {admin}")



#Observer client code
notifier = RecycleCentreNotifier()

for member in members:
    add_member=memberObserver(member)
    notifier.add_observer(add_member)
```

```
RC_name = "abc" # temporary data (works as a new verified Recycle Center)
address = "Kajang"

notifier.notify_observer(RC_name,address)
```

4.5.6. Benefit

Real-time communication between component:

The Observer Design Pattern enables real-time updates by automatically notifying all subscribed observers when a subject's state changes.

Loose coupling between object:

Observers and the subject (notifier) are loosely coupled, meaning changes to one do not require changes to the other. This improves modularity and makes the system easier to maintain and scale.

4.5.7. Limitations

Notification to all observers without filtering:

By default, all registered observers are notified, regardless of whether the update is relevant to them. This may lead to unnecessary notifications unless additional logic is introduced.

5. Conclusion and Suggestion

Conclusion:

The waste management and recycling system designed in this project addresses some key challenges in Malaysia's waste management, such as poor recycling habits, lack of awareness and hard to access to recycling facilities. The system allows users to find nearby recycling centers and provide education slides and content.

By using software design patterns such as factory, strategy, facade, bridge and observer to ensure a modular, scalable and maintainable architecture. These patterns use design principles like the Open/Closed principle and Single Responsible principle, enhancing code readability.

Future work:

1. Develop an 80–90% functional prototype of the system
2. Design and implement a more user-friendly interface for better usability
3. Extend the categories of educational slides by adding new types, such as glass, metal or electronic waste.
4. Enhancing the commenting features like adding the ability to upvote or downvote which can increase engagement.
5. Rating/review system for the recycling centers which will allow members to talk about their experience visiting the center.

6. Bibliography/Reference

- MDPI. (2023). *Assessing Residents' Acceptance of Mobile Apps for Household Waste Management*. *Sustainability*, 14(17), 2023. https://doi.org/10.3390/su141710874
- ResearchGate. (2022). *Assessing Students' Municipal Solid Waste Recycling Behavior in Putrajaya, Malaysia*. Retrieved from https://doi.org/10.47405/mjssh.v7i9.1716
- ScienceDirect. (2022). *Trash to Treasure: Gamification and Informed Recycling Behavior*. *Resources, Conservation and Recycling*, 100, 2022. https://doi.org/10.1016/j.resconrec.2024.108108
- VulcanPost. (2023). *Recircle: M'sian Online Recycling App Digitalising the Industry*. Retrieved from vulcanpost.com