RECYCLING GUIDE SYSTEM PRESENTATION

Student Name	Student ID
CHIN ZHEN HO	1221102540
ERIC TEOH WEI XIANG	1221102007
BERNARD RYAN SIM KANG XUAN	1221101777
HO YU HANG	1211111437

By Group Grey

SYSTEM OBJECTIVES AND DELIVERABLES

- Core Objective: To develop a software/solution aligned with Sustainable Development Goals (SDG) 10, 11, or 12, demonstrating fundamental software design concepts and principles.
- Design Pattern Application: To identify and effectively apply relevant software design patterns to solve defined requirements and enhance software attributes.
- Functional Prototype: To deliver a functional prototype (80-95% complete) that showcases the practical implementation of our design choices.
- Problem Resolution: To provide a practical, user-centric solution that addresses a specific problem within the chosen SDG framework.

SYSTEM SCOPE

TARGET USERS: Users, Admins, Recycle Center Admins

KEY FEATURES:

USERS:

- Sign Up
- Log In and Logout
- Search items to dispose
- Profile
- View Recycle Tips
- Participate in Survey
- View FAQ
- Submit Recycled Proof
- View Leaderboard
- View Material Prices from Recycle Center
- View Nearby Recycle Center

ADMIN:

- Log In and Logout
- Manage Categories
- Manage Categories Tips
- View Total User and Details
- View Survey Results
- View Recycling Category Statistics
- View Total Amount of Material received from Recycle Center Admin

RECYCLE CENTER ADMINS:

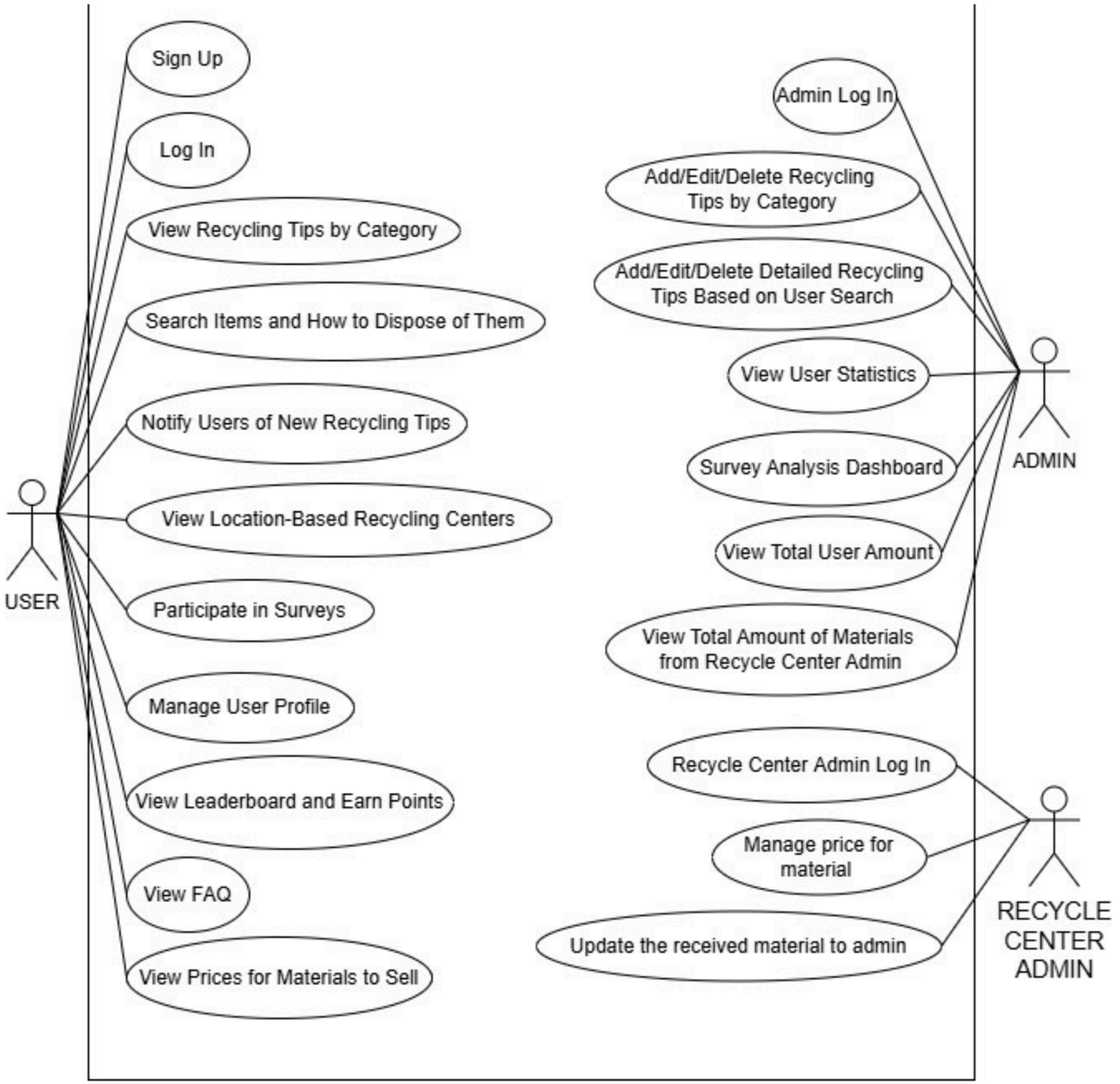
- Log In and Logout
- Material Management (add, update and view material name, price, recycle center)
- Find Nearby Recycle Centers
- Submit Material Reception from Users

SYSTEM OVERVIEW

- Problem Statement:
  - Lack of clear information on what, where, and how to recycle.
  - Disconnection between awareness and actual recycling participation.
  - Recycling centers need consolidated data for planning.

- Our Solution: EcoRecycle
  - Web-based platform for recycling information and tools.
  - Bridges users to recycling centers and knowledge.
  - Empowers admins with insights for sustainable practices.

Use Case Diagram



COMPONENT LEVEL DIAGRAM

Frontend Component

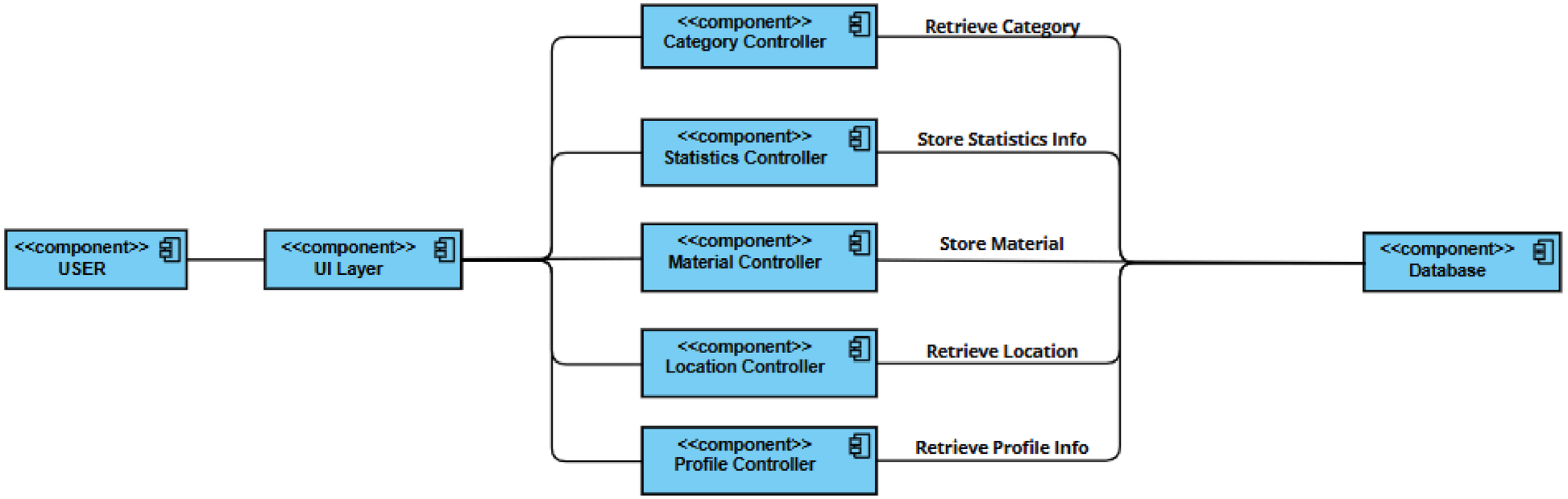
- Login/Sign Up Interface
- User Interface
- Admin Interface
- Recycling Center Admin Interface

Backend Component

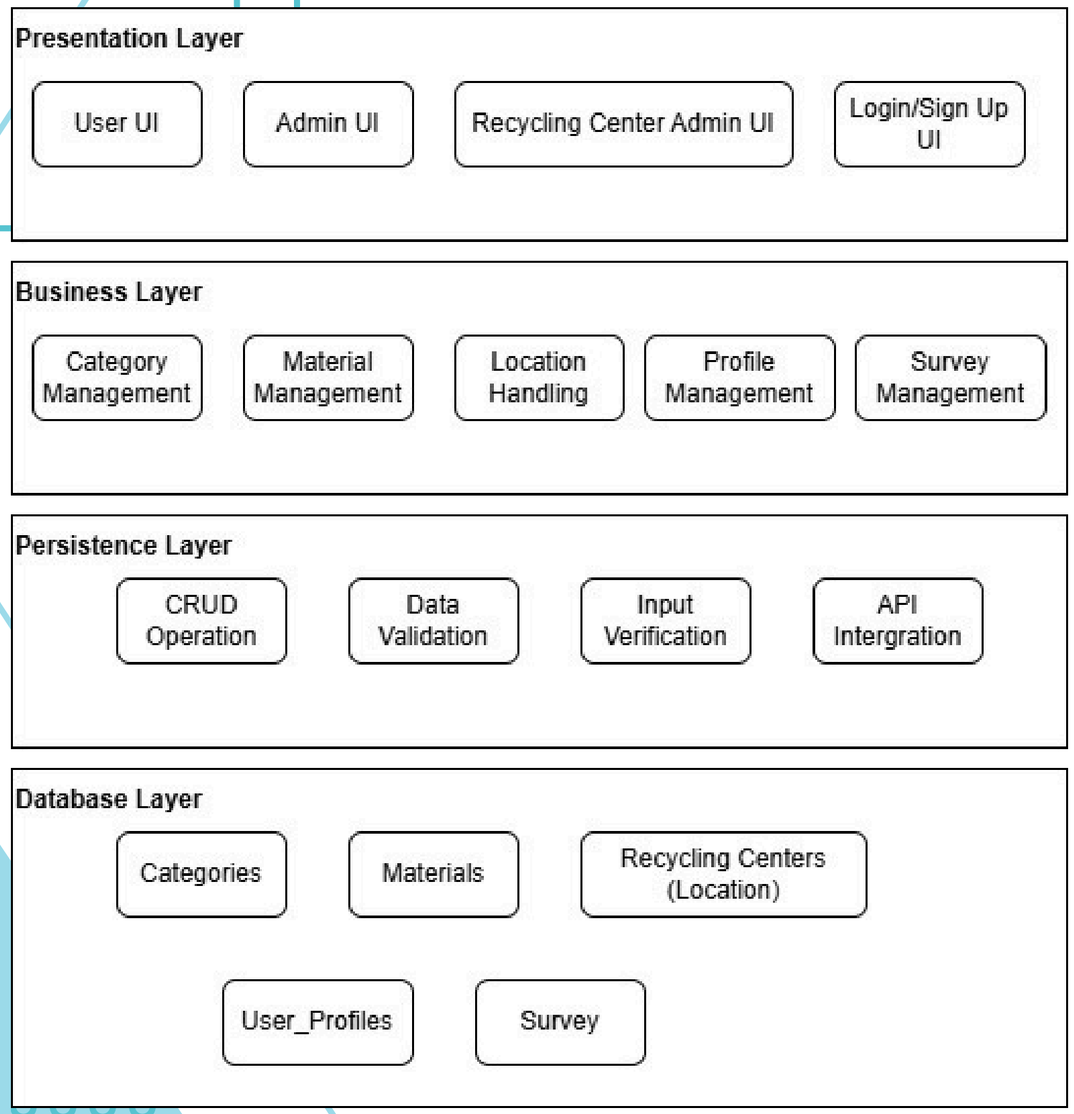
- Admin functions management
- Recycling Center management
- User functions management

Database Component

- Categories
- Materials
- Location
- Survey
- User Info



SOFTWARE ARCHITECTURE



This layered design promotes modularity, scalability, and maintainability, allowing each layer to be independently developed and tested. It also supports the use of key design patterns such as Singleton (for database access) and Factory (for action instantiation), ensuring a robust and extensible system architecture.

INPUTS AND OUTPUTS

INPUTS

User

- login credentials
- registration personal info
- search queries.

Admin

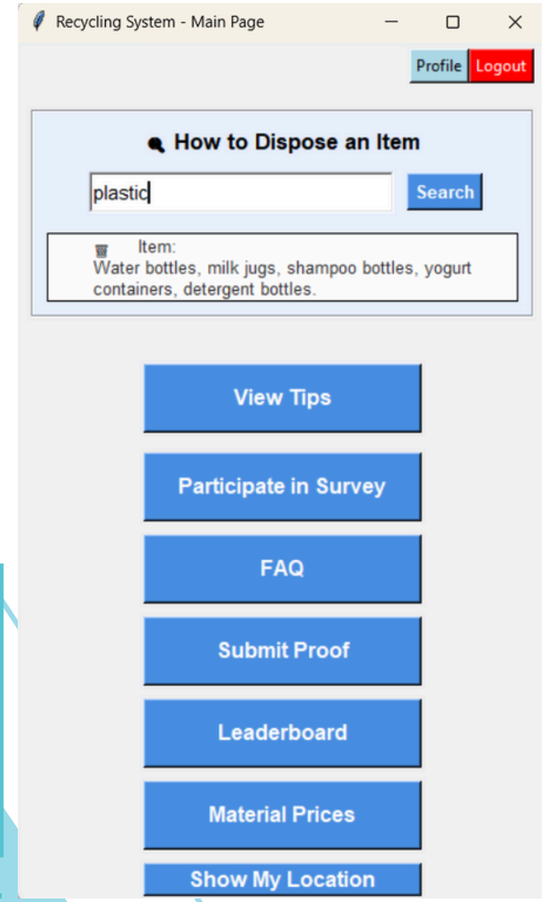
- Categories Info
- User Info
- Surveys
- Recycling Tips

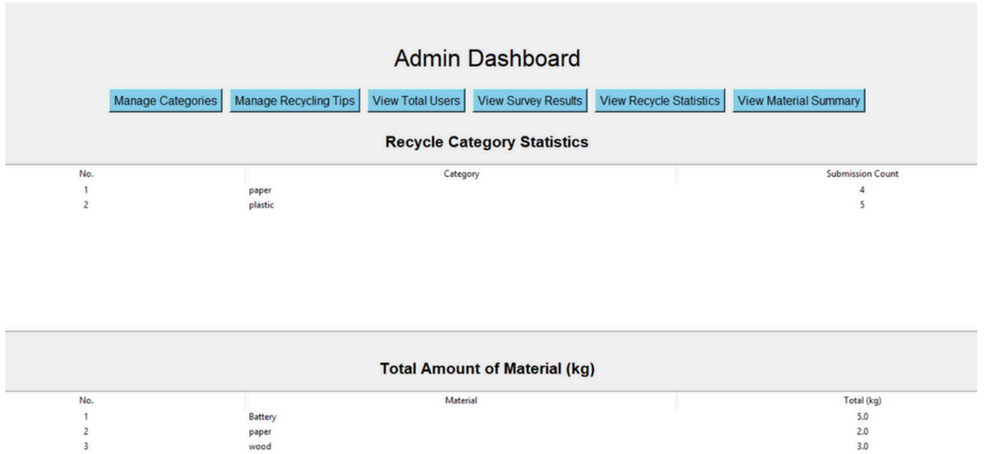
Recycling Center Admin

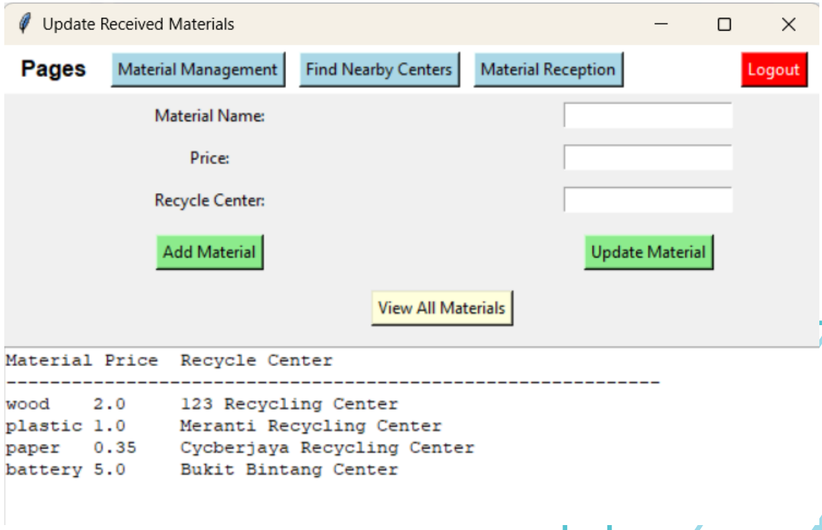
- Material Info
- Recycling centers location

INPUTS AND OUTPUTS

OUTPUTS







SOLUTION ACHIEVED

- All functions, including user registration, login, recycling tips, disposal searches, location services, notifications, surveys, leaderboards, admin features and recycle center admin features, work 100% as intended.

- Every function strictly follows the documented use cases, ensuring pre-conditions, workflows, and post-conditions are fully met.

- All design patterns—Singleton, Observer, Factory, Strategy, and Command—are correctly implemented for efficiency, scalability, and modularity.

SINGLETON PATTERN

```python
#database.py

import mysql.connector

class Database:
    _instance = None

    def __init__(self):
        # Real constructor
        self.db_connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="",
            database="user_system"
        )
        self.db_cursor = self.db_connection.cursor()

    @classmethod
    def get_instance(cls):
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance

    def execute(self, query, params=None):
        if params:
            self.db_cursor.execute(query, params)
        else:
            self.db_cursor.execute(query)
        self.db_connection.commit()

    def fetch_all(self, query, params=None):
        if params:
            self.db_cursor.execute(query, params)
        else:
            self.db_cursor.execute(query)
        return self.db_cursor.fetchall()

    def fetch_one(self, query, params=None):
        if params:
            self.db_cursor.execute(query, params)
        else:
            self.db_cursor.execute(query)
        return self.db_cursor.fetchone()
```

The Singleton Pattern ensures only one database connection instance exists, reused across the system via Database.get_instance(), preventing multiple connections and optimizing performance.

OBSERVER PATTERN

```python
from abc import ABC, abstractmethod
from database import Database

# 1. User Interface (formerly Subscriber)
class User(ABC):
    @abstractmethod
    def update(self, admin, data):
        pass

# 2. Admin Interface (formerly Publisher)
class Admin(ABC):
    def __init__(self):
        self._users = []

    def add_user(self, user: User):
        self._users.append(user)

    def remove_user(self, user: User):
        self._users.remove(user)

    def notify_users(self, data):
        for user in self._users:
            user.update(self, data)

# 3. Concrete Admin (CategoriesAdmin)
class CategoriesAdmin(Admin):
    def __init__(self):
        super().__init__()
        self.db = Database.get_instance()
        self.last_tip = None  # Store the last tip checked

    def check_new_tips(self):
        query = "SELECT category_name, category_type, description FROM categories ORDER BY created_at DESC LIMIT 1"
        result = self.db.fetch_one(query)

        if result and result != self.last_tip:
            self.last_tip = result
            self.notify_users(result)

# 4. Concrete User (UserNotification)
class UserNotification(User):
    def __init__(self, app):
        self.app = app  # Tkinter root or frame

    def update(self, admin, data):
        from tkinter import messagebox
        category_name, category_type, description = data
        messagebox.showinfo(
            "New Recycling Tip Available!",
            f"Category Name: {category_name}\nCategory Type: {category_type}\nDescription: {description}"
        )
```

The Observer Pattern notifies users instantly when new recycling tips are added, with CategoriesAdmin as the publisher sending updates to subscribed UserNotification observers.

FACTORY PATTERN

```python
class MaterialAction:
    def execute(self, material_name, price):
        raise NotImplementedError("Subclasses must implement execute()")

class AddMaterial:
    def execute(self, material_name, price, center_name):
        db = Database()
        query = "INSERT INTO materials (material_name, price, center_name) VALUES (%s, %s, %s)"
        params = (material_name, price, center_name)
        db.execute(query, params)

class UpdateMaterial:
    def execute(self, material_name, price, center_name):
        db = Database()
        query = "UPDATE materials SET price = %s WHERE material_name = %s AND center_name = %s"
        params = (price, material_name, center_name)
        db.execute(query, params)

class MaterialActionFactory:
    @staticmethod
    def get_action(action_type):
        if action_type == "add":
            return AddMaterial()
        elif action_type == "update":
            return UpdateMaterial()
        else:
            raise ValueError("Invalid material action type.")

class MaterialReceptionHandler:
    def __init__(self):
        self.db = Database()

    def update_received_material(self, center_name, material_name, amount_kg):
        query = """
            INSERT INTO recycle_center_materials (center_name, material_name, amount_kg)
```

The Factory Pattern dynamically creates material management actions (like AddMaterial or UpdateMaterial) through a MaterialActionFactory, allowing flexible object creation without exposing logic to client code.

STRATEGY PATTERN

```python
# strategy.py

from abc import ABC, abstractmethod
from database import Database  # Using your Database class

# 1. Strategy Interface
class DisposalStrategy(ABC):
    @abstractmethod
    def get_disposal_method(self, item_name):
        pass

# 2. Concrete Strategy for General Items
class GeneralDisposalStrategy(DisposalStrategy):
    def get_disposal_method(self, item_name):
        db = Database.get_instance()
        query = "SELECT description FROM categories WHERE category_name = %s"
        result = db.fetch_one(query, (item_name,))
        if result:
            return result[0]
        else:
            return "No disposal method found for this item."

# 3. Context
class DisposalContext:
    def __init__(self, strategy: DisposalStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: DisposalStrategy):
        self._strategy = strategy

    def get_disposal_instructions(self, item_name):
        return self._strategy.get_disposal_method(item_name)
```

The Strategy Pattern enables the search function to dynamically select the appropriate disposal method via a DisposalContext, ensuring users get accurate recycling instructions tailored to each item type they search for.

COMMAND PATTERN

```python
#commands.py
from abc import ABC, abstractmethod
from category_controller import CategoryController

class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

class AddCategoryCommand(Command):
    def __init__(self, controller: CategoryController, category_name, category_type, description):
        self.controller = controller
        self.category_name = category_name
        self.category_type = category_type
        self.description = description

    def execute(self):
        self.controller.add_category(self.category_name, self.category_type, self.description)

class EditCategoryCommand(Command):
    def __init__(self, controller: CategoryController, category_id, category_name, category_type, description):
        self.controller = controller
        self.category_id = category_id
        self.category_name = category_name
        self.category_type = category_type
        self.description = description

    def execute(self):
        self.controller.update_category(self.category_id, self.category_name, self.category_type, self.description)

class DeleteCategoryCommand(Command):
    def __init__(self, controller, category_id, backup_data):
        self.controller = controller
        self.category_id = category_id
        self.backup = backup_data

    def execute(self):
        self.controller.delete_category(self.category_id)

    def undo(self):
        name, type, desc = self.backup
        self.controller.add_category(name, type, desc)
```

The Command Pattern encapsulates admin actions (like AddCategoryCommand or DeleteCategoryCommand) as objects executed by a CommandInvoker, decoupling the admin interface from the actual operations while enabling undo/redo functionality.

WORK SEGREGATION

TEAM ROLES:

Team Roles:

- **Chin Zhen Ho:** Builds the Command Pattern for Admin functions, enabling category management (add/edit/delete) with undo/redo support.

- **Eric Tech Wei Xiang:** Applies the Singleton Pattern to user authentication (login/signup), profile management, and shared database connections.

- **Bernard Ryan Sim Kang Xuan:** Implements the Strategy Pattern (dynamic disposal search) and Observer Pattern (real-time notifications) for user-facing features like recycling tips by category.

- **Ho Yu Hang:** Develops the Factory Pattern for Recycle Center Admin functions, including material pricing and received-material tracking.

CONCLUSION

Development Success

- Addresses the evolving challenges of modern recycling efforts
- Combines a user-friendly interface with efficient material management
- Supports real-time tips and responsive interaction for users and admins
- Promotes environmental awareness through structured design
- Enables clear role separation for admins, recycling center staff, and end-users

Future Outlook

- Expanding need for sustainable practices offers room for system growth
- Potential integration of AI for personalized recycling tips and material categorization
- Use of advanced analytics to monitor recycling trends and behaviors
- Plans for mobile application support and gamified user engagement
- Continued emphasis on innovation, environmental impact, and user empowerment

DEMO TIME

By Group Grey

Q&A SECTION

By Group Grey

THANK YOU

By Group Grey