CSYE 6200 Final Project

Team Peterborough

Han-Sheng Chen, Bo-Lin Lin, Yuan-Chang Lee, Feiyu Chen

Catalog

01

Introduction

speaker:Feiyu Chen

Introduction

- **Time Management Challenges**
  - Managing time is challenging for students and professionals.
  - Balancing coursework, work commitments, and social life is often overwhelming.

- **Problem Example:**
  - John, a graduate student, struggles to manage his classes, lab work, and meetings effectively, often missing important deadlines.

Limitations of Existing Solutions

- **Current Tools Overview:**
  - **Google Calendar**: Great for scheduling but lacks advanced task management features.
  - **Microsoft To-Do**: Effective for organizing tasks, but lacks calendar integration.

- **Key Gaps:**
  - Lack of integration between task management and scheduling.
  - Difficulty in prioritizing and categorizing tasks effectively.

Why Integration Matters

- **Research Insights:** Studies suggest combining task prioritization with scheduling can significantly reduce stress and improve efficiency.

- **Key Takeaway**: Users need a single tool to manage both tasks and time in an intuitive way.

Our Integrated Solution

- **Introducing Our App**
  - Combines task management with calendar functionality to create a seamless user experience.
  - Supports assigning priorities, adding tags, and managing recurring tasks easily.

- **What Sets Us Apart**
  - A comprehensive tool that allows users to handle schedules and tasks all in one place.
  - Reduces the risk of missed deadlines and makes daily planning more efficient.

02

Class Overview

Class Overview

ToDoItem - The Building Block:
- Properties: title, description, dueDate, priority, tag, completion status
- Enums for Classification:
  * Priority: HIGH, MEDIUM, LOW
  * Tag: ERRAND, HOME, OFFICE, IMPORTANT, PENDING
- Provides rich task representation

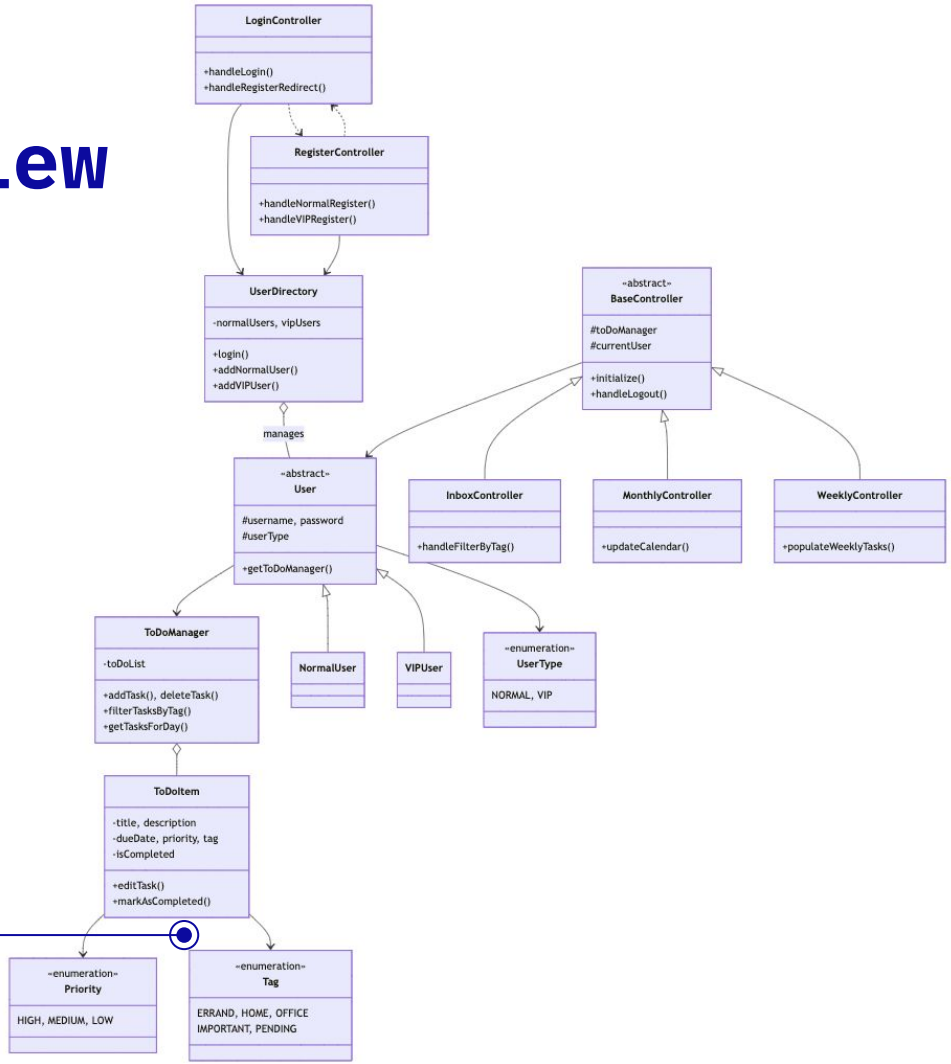
ToDoManager - The Task Engine:
- Central task management system
- Key Features:
  * Task CRUD operations
  * Filtering by tags
  * Date-based task retrieval
  * Task completion tracking

Class Overview

User Management:
- Abstract User class with NormalUser and VIPUser implementations
- UserType enum: NORMAL, VIP
- Each user has their own ToDoManager and task list

Controller Hierarchy:
Specialized Views:
1. InboxController: List-based task view with filtering
2. TodayController: Daily task management
3. WeeklyController: Week-based calendar view
4. MonthlyController: Month calendar with task distribution

Class Overview



03

Important features

Inheritance/Polymorphism

Inheritance:

`NormalUser` and `VIPUser` inherit from the abstract class `User`, sharing common attributes and methods to improve code reusability.

Polymorphism:

The abstract method `accessFeatures()` in `User` is implemented by subclasses, dynamically executed based on the actual type (`NormalUser` or `VIPUser`) at runtime.

```java
package application.User;

public class NormalUser extends User {

    public NormalUser(String username, String password) {
        super(username, password, UserType.NORMAL);
    }

    @Override
    public void accessFeatures() {
        System.out.println("Some feature of NormalUser");
    }
}
```

```java
public abstract class User {
    protected String username;
    protected String password;
    protected UserType userType;
    private List<ToDoItem> toDoList;
    private ToDoManager toDoManager; // ToDoManager for each user

    // constructor
    public User(String username, String password, UserType userType) {
        this.username = username;
        this.password = password;
        this.userType = userType;
        this.toDoList = new ArrayList<>(); // Initialize task list
        this.toDoManager = new ToDoManager(this); // Pass this user to the manager
    }

    // abstract method just prepare for probably future use
    public abstract void accessFeatures();
```

Abstract Classes/Interfaces

Purpose of Abstract Class:

Defines common attributes (`username`, `password`, `userType`) and enforces implementation of `accessFeatures()` in subclasses.

Design Scalability:

New user types, like `AdminUser`, can be added by extending `User` and implementing `accessFeatures()`, following the Open-Closed Principle.

```java
public abstract class User {
    protected String username;
    protected String password;
    protected UserType userType;
    private List<ToDoItem> toDoList;
    private ToDoManager toDoManager; // ToDoManager for each user

    // constructor
    public User(String username, String password, UserType userType) {
        this.username = username;
        this.password = password;
        this.userType = userType;
        this.toDoList = new ArrayList<>(); // Initialize task list
        this.toDoManager = new ToDoManager(this); // Pass this user to the manager
    }

    // abstract method just prepare for probably future use
    public abstract void accessFeatures();
```

Generics/Collections:

In our project, we chose to use **generics** and **collections** because they make our code more flexible, organized, and easier to maintain. Here's how we applied them and why:

We used `ArrayList<ToDoItem>` in the `ToDoManager` class to store tasks, leveraging generics for type safety and cleaner code by ensuring the list handles only `ToDoItem` objects.

We used `ArrayList` to store tasks because it provides a dynamic array that adjusts its size automatically, offering flexibility for adding, removing, or accessing tasks, and efficient access through direct indexing, which aligns with our requirements for managing and filtering tasks.

```java
    // Constructor (initialize to-do list)
    public ToDoManager(User user) {
        this.toDoList = user.getToDoList(); // Initialize with user's to-do list
    }

    public void addTask(ToDoItem task) {
        toDoList.add(task);
    }

    public void deleteTask(ToDoItem task) {
        toDoList.remove(task);
    }
```

Generics/Collections:

In our project, we also used `Iterator` in the `handleDeleteTask` method to safely traverse and modify the task list.

With methods like `hasNext()`, `next()`, and `remove()`, `Iterator` allowed us to delete tasks that met specific conditions without errors like `ConcurrentModificationException`.

This made our task management more reliable and efficient, especially for batch deletions.

```
Iterator<ToDoItem> iterator = currentUser.getToDoList().iterator();
while(iterator.hasNext()) {
    ToDoItem todo = iterator.next();
    if(todo.getTitle().equals(selectedTask.getTitle()) && todo.getTag().equals(selectedTask.getTag())) {
        iterator.remove();
    }
}
```

Lists:

In `ToDoManager`, we designed a series of methods to manipulate `List`, including filtering and categorization. These implementations demonstrate the flexibility and powerful functionality of lists in real-world business logic.

List Storage Application:
All to-do items are stored in `toDoList` (`List<ToDoItem>`), which is a list-based collection.
The list is used to store and manage each to-do item and allows operations such as adding, deleting, filtering, and categorizing based on specific needs.

List Filtering Application:
In `ToDoManager`, filtering methods are typical examples of list manipulation.
Each filtering method iterates through all items in the list, checks them against specific conditions, and adds the matching items to a new list, which is then returned.

```java
    // choose a specific tag
    public List<ToDoItem> filterTasksByTag(ToDoItem.Tag tag) {
        List<ToDoItem> filteredTasks = new ArrayList<>();
        for (int i = 0; i < toDoList.size(); i++) {
            ToDoItem task = toDoList.get(i);
            if (task.getTag() == tag) {
                filteredTasks.add(task);
            }
        }
        return filteredTasks;
    }
```

04

Live Demo

Thanks for listening.