6-1 Journal: Explore Object-Oriented Programming

Justice Williamson

Southern New Hampshire University

CS-500: Introduction to Programming

Instructor: Ronak Nouri

October 5, 2025

Practical Application and Rationale

The mobile task manager application I selected for this journal is designed to support parents and students who need help managing their daily responsibilities. Users can create tasks and organize them into projects such as “School,” “Home,” or “Work.” Each task can include deadlines, priority levels, and reminders through alert notifications.

This application is a strong choice because it reflects real-world usage, serves a broad range of users, and clearly demonstrates object-oriented programming (OOP) principles. It models tasks as objects, organizes them through project management, delivers reminders via notification systems, and secures user data through encapsulation. Its design provides an excellent opportunity to show how OOP improves program maintainability, reusability, and scalability.

Key OOP Concepts in the App

A. Classes and Objects

The application uses classes as blueprints—such as Task, Project, UserSettings, and Notifier—while objects represent concrete instances, like a specific “Submit Week 4 Journal” task.

- The Task class contains properties such as title, description, due date, priority, and completion status.
- The Project class manages collections of tasks, with methods for adding, removing, and filtering.
- UserSettings stores notification preferences, themes, and time zone information.

- The Notifier class acts as an abstraction for reminders, enabling delivery through push notifications, email, or SMS.

B. Inheritance

Inheritance defines “is-a” relationships between classes. The base Notifier class provides a shared interface, while child classes extend it with specialized delivery methods:

- PushNotifier (mobile push)
- EmailNotifier
- SMSNotifier

Each subclass inherits shared behavior, such as message formatting, but customizes how notifications are delivered.

C. Encapsulation

Encapsulation hides internal details and provides safe methods for interaction.

- The Task class protects its internal _completed state while offering methods like mark_complete() and reschedule(date).
- UserSettings secures its private fields and uses setters with validation to prevent errors and unauthorized access.

D. Polymorphism

Polymorphism allows different notifier types to be used interchangeably through a shared interface. The scheduling system can call notifier.send(task) without knowing which subclass is being used. This design also supports future expansion: new classes like WebhookNotifier or InAppBannerNotifier can be added without modifying existing code.

Illustrative Python Snippets:

```
from datetime import datetime
from typing import List, Protocol

class Task:
    def __init__(self, title: str, due: datetime, priority: int = 3):
        self.title = title
        self.due = due
        self.priority = priority
        self._completed = False

    def mark_complete(self):
        self._completed = True

    def is_overdue(self, now: datetime) -> bool:
        return (not self._completed) and now > self.due

class Project:
    def __init__(self, name: str):
        self.name = name
        self._tasks: List[Task] = []

    def add_task(self, task: Task):
        self._tasks.append(task)

    def pending(self) -> List[Task]:
        return [t for t in self._tasks if not t._completed]

class Notifier(Protocol):
    def send(self, task: Task) -> None: ...

class PushNotifier:
    def send(self, task: Task) -> None:
        print(f"[PUSH] Reminder: '{task.title}' is due {task.due:%Y-%m-%d %H:%M}")

class EmailNotifier:
    def __init__(self, address: str):
        self._address = address
    def send(self, task: Task) -> None:
        print(f"[EMAIL→{self._address}] Subject: Task due | {task.title}")

def remind(tasks: List[Task], notifier: Notifier, now: datetime):
    for t in tasks:
        if t.is_overdue(now):
            notifier.send(t)
```

Benefits of OOP in This Application

A. Effect on the End User

- Encapsulation and clear interfaces increase system reliability by preventing accidental changes.
- Polymorphism creates a consistent user experience and allows new reminder options to be added seamlessly.
- The structure supports recurring tasks, subtasks, and tags without disrupting existing features.

B. Effect on the Developer

- Classes divide responsibilities clearly, making the codebase easier to maintain.
- Inheritance reduces duplication by centralizing shared functionality.
- Polymorphic interfaces simplify testing by making it easy to create mock objects.
- New developers can quickly understand the system because its structure maps directly to real-world concepts.

C. Effect on Program Structure and Performance

- OOP enforces modular boundaries, keeping components well-organized.
- Clear code allows developers to identify and optimize performance-critical areas.
- The design supports extensibility, enabling new storage or notification features without rewriting core logic.

Conclusion

A mobile task manager application aligns naturally with OOP principles. Classes and objects model real-world entities, inheritance and polymorphism enable flexible extension, and encapsulation safeguards data integrity. These design choices improve user confidence and usability, simplify developer operations, and ensure the system remains adaptable and scalable for future growth.