Advanced Programming Concepts

Lecture 4

Outline

- Modularization techniques.
- SOLID principles for code organisation.

Resources

- Chapter 3 of "Clean Code: A Handbook of Agile Software Craftsmanship“
- “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma et al.
- “Introduction to Algorithms” by Cormen et al.

Modularization

- The process of breaking down a complex software system into smaller, manageable modules or components.
- Enhances maintainability, reusability, and collaboration.

Advantages of Modularization

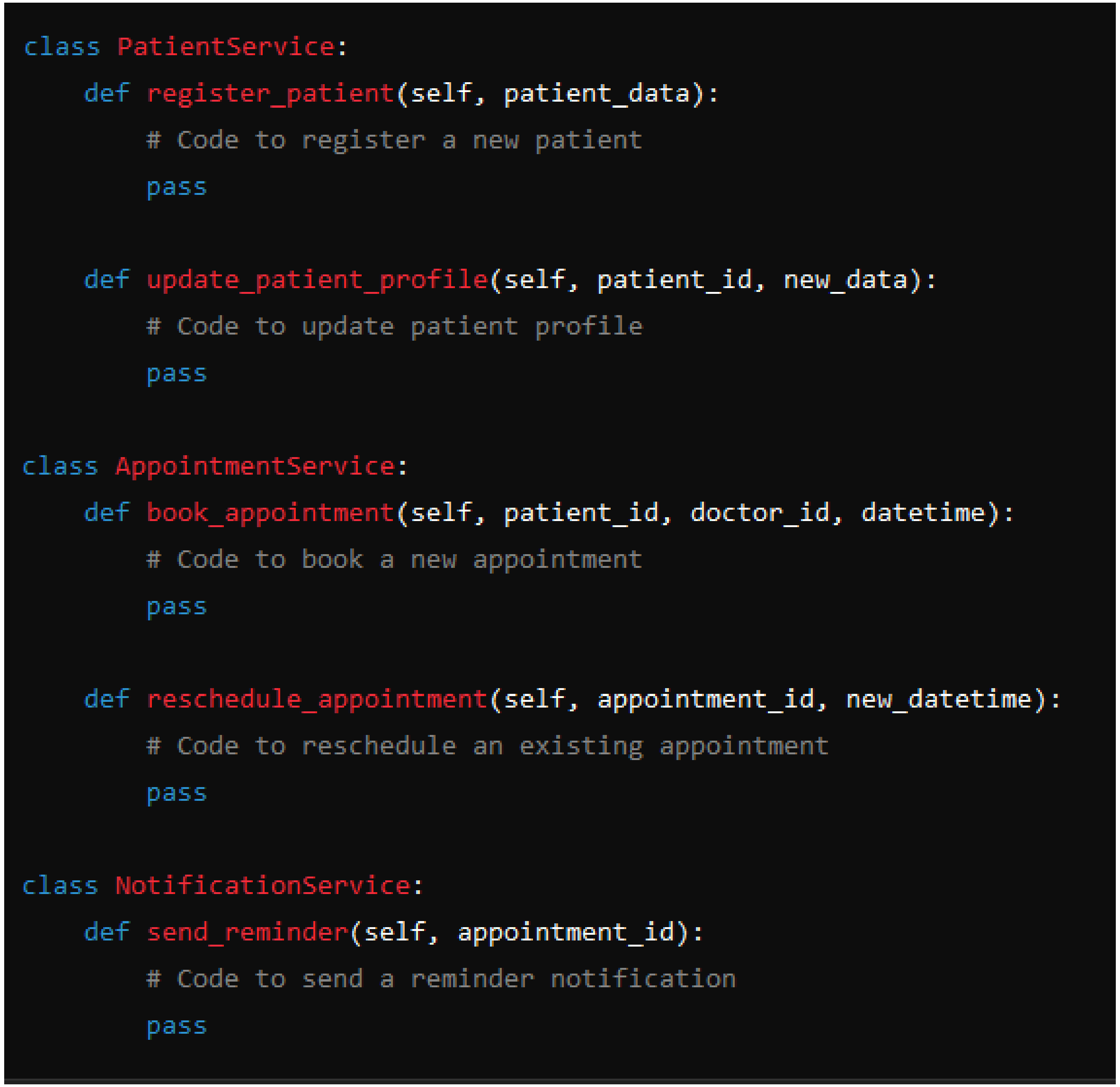
- Improved code organization.
- Easier debugging and testing.
- Encourages code reuse.
- Simplifies collaboration among developers.

Techniques for Modularization:

- **Functional Decomposition:** Divide the system into functional units or modules.
- **Object-Oriented Decomposition**: Organize code around objects and classes.
- **Layered Architecture:** Divide the application into logical layers (e.g., presentation, business logic, data access).
- **Component-Based Design:** Build reusable components or libraries.
- **Service-Oriented Architecture (SOA):** Expose functionality as services.

Functional Decomposition Example

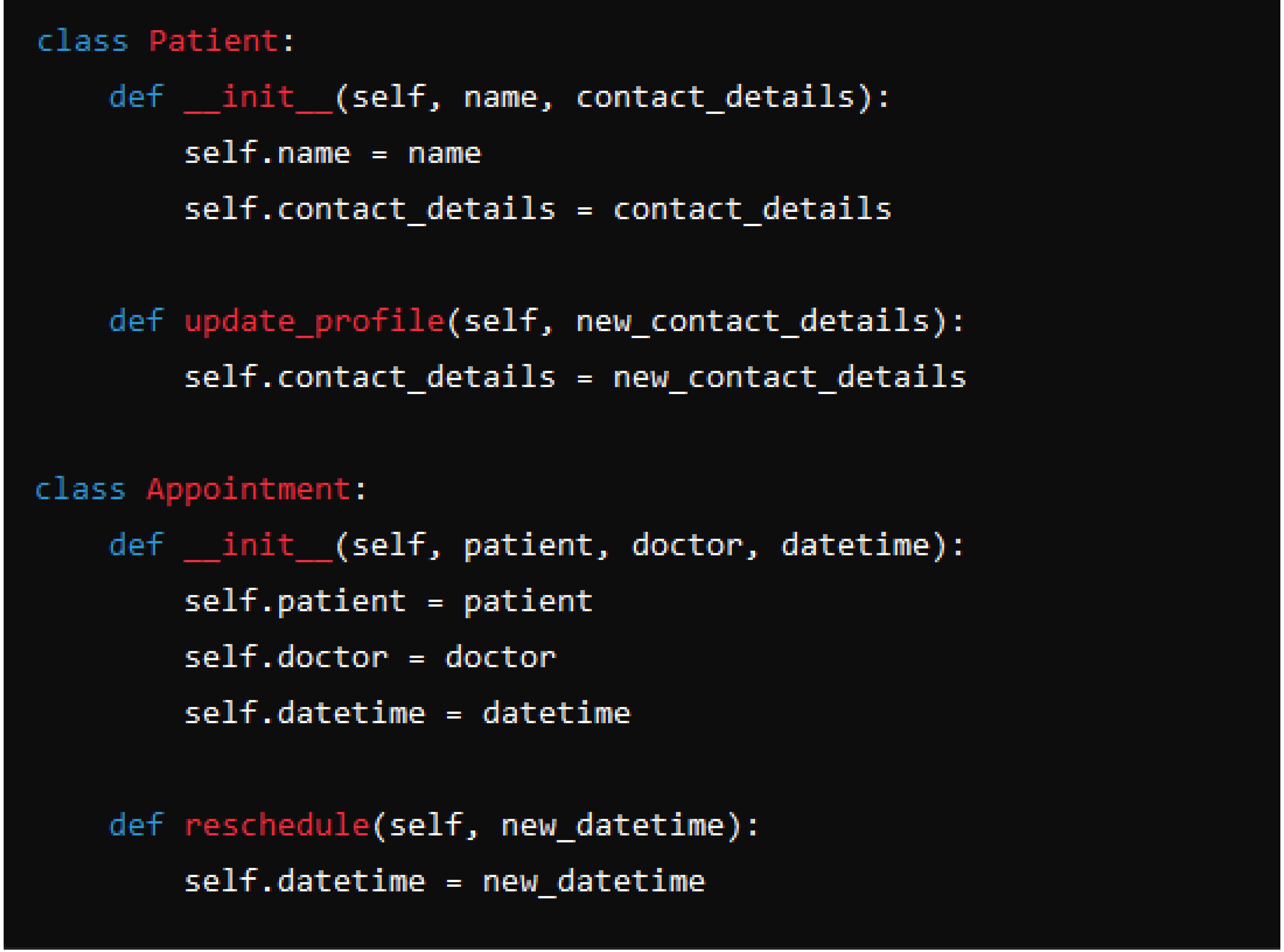
- Patient Management Module: Handles patient registration, login, and profile management.
- Appointment Scheduling Module: Manages the booking, rescheduling, and cancellation of appointments.
- Notification Module: Sends reminders and notifications to patients and staff about upcoming appointments.



```python
class PatientService:
    def register_patient(self, patient_data):
        # Code to register a new patient
        pass

    def update_patient_profile(self, patient_id, new_data):
        # Code to update patient profile
        pass

class AppointmentService:
    def book_appointment(self, patient_id, doctor_id, datetime):
        # Code to book a new appointment
        pass

    def reschedule_appointment(self, appointment_id, new_datetime):
        # Code to reschedule an existing appointment
        pass

class NotificationService:
    def send_reminder(self, appointment_id):
        # Code to send a reminder notification
        pass
```

Object-Oriented Decomposition

- Organize code around objects and classes.
- Object-Oriented Decomposition:
  - Patient Class: Represents a patient with attributes like name, contact details, and medical history. Methods include register(), login(), and updateProfile().
  - Appointment Class: Represents an appointment with attributes like date, time, and doctor. Methods include book(), reschedule(), and cancel().
  - Notification Class: Manages notifications with attributes like message content and recipients. Methods include sendReminder() and sendCancellationAlert().



```python
class Patient:
    def __init__(self, name, contact_details):
        self.name = name
        self.contact_details = contact_details

    def update_profile(self, new_contact_details):
        self.contact_details = new_contact_details

class Appointment:
    def __init__(self, patient, doctor, datetime):
        self.patient = patient
        self.doctor = doctor
        self.datetime = datetime

    def reschedule(self, new_datetime):
        self.datetime = new_datetime
```

Layered Architecture:

- A layered architecture divides the application into logical layers, each responsible for specific tasks. Here are suggested layers for the Student Registration Application:
- Example:
  - Presentation Layer: Handles user interactions such as viewing available time slots, booking appointments, and managing patient profiles.
  - Business Logic Layer: Manages the core functionality, including checking doctor availability, booking logic, and handling appointment conflicts.
  - Data Access Layer: Interacts with the database to store and retrieve patient information, appointment details, and notification logs.

Component-Based Design

- Component-based design focuses on creating reusable components or libraries. Here are suggested components for the Student Registration Application:
- Authentication Component: Manages patient and staff login and registration. This can be reused in other healthcare-related applications.
- Appointment Management Component: Handles all operations related to booking and managing appointments, which could be reused in similar scheduling systems.
- Notification Component: Sends notifications and reminders, which could be integrated into other applications that require communication with users.

Service-Oriented Architecture (SOA):

- SOA involves exposing functionality as services that can be accessed independently. In the context of the Student Registration Application:

1. **User Authentication Service:**
    - Provides user authentication and account management as a service.
    - Can be consumed by various applications within the institution.
2. **Student Profile Service:**
    - Manages student profiles and personal information.
    - Offers RESTful APIs for data retrieval and updates.
3. **Course Management Service:**
    - Exposes course-related functionalities through APIs.
    - Supports integration with other educational platforms.
4. **Payment Service:**
    - Acts as a standalone payment processing service.
    - Can be used by multiple applications across the institution.

SOLID principles:

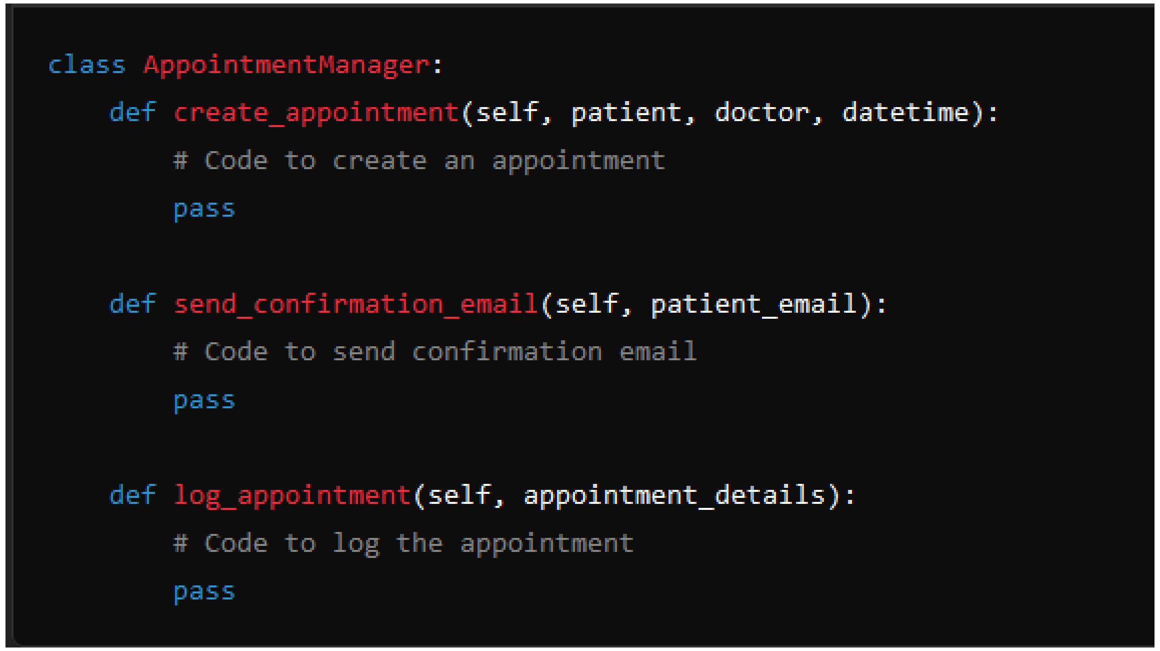
- SOLID is a set of five principles aimed at improving the organization and maintainability of software.
  - **S**ingle Responsibility Principle (SRP),
  - **O**pen-Closed Principle (OCP),
  - **L**iskov Substitution Principle (LSP),
  - **I**nterface Segregation Principle (ISP),
  - **D**ependency Inversion Principle (DIP).

Single Responsibility Principle (SRP)

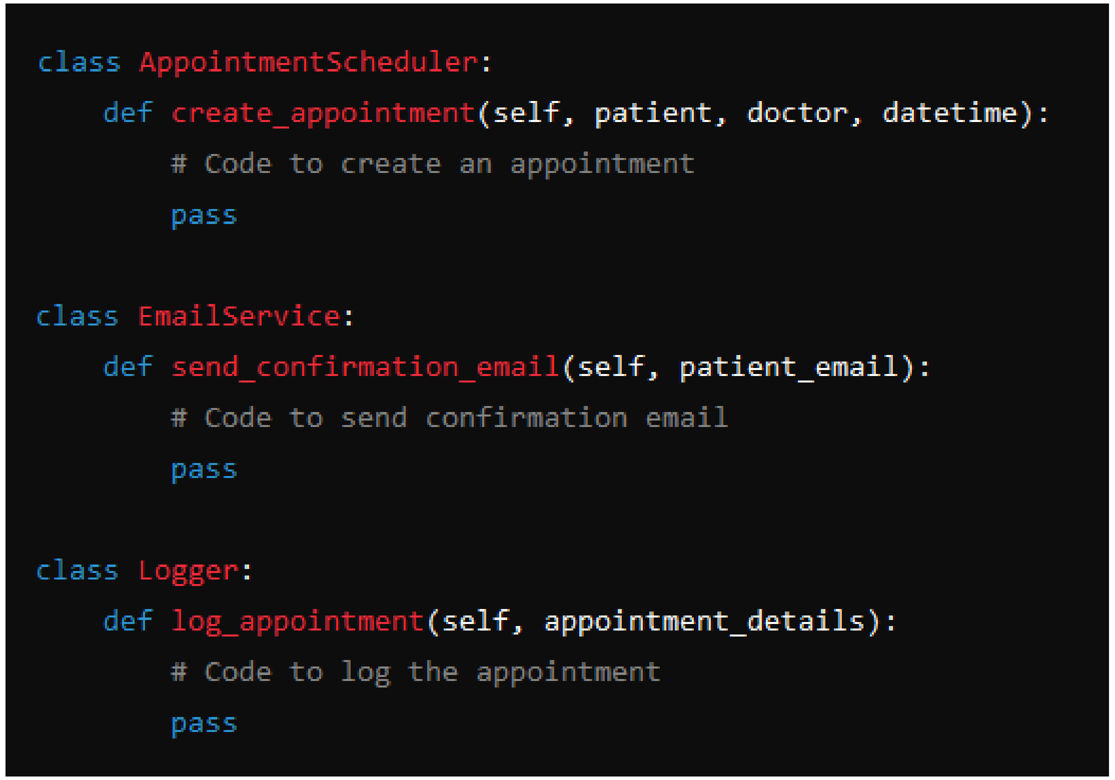
- A module should have only one reason to change.
- A class or module should have a single responsibility.
- Avoid "God objects" that do too much.
- Applying Single Responsibility Principle (SRP):
  - Each module or class should have a single responsibility.
  - Avoid functions or classes that do too much.
  - Refactor code to split responsibilities into smaller modules.

Single Responsibility Principle (SRP)

- A class should have only one reason to change, meaning it should have only one responsibility. In the context of the Appointment Booking App, we should separate concerns to ensure maintainability and clarity.



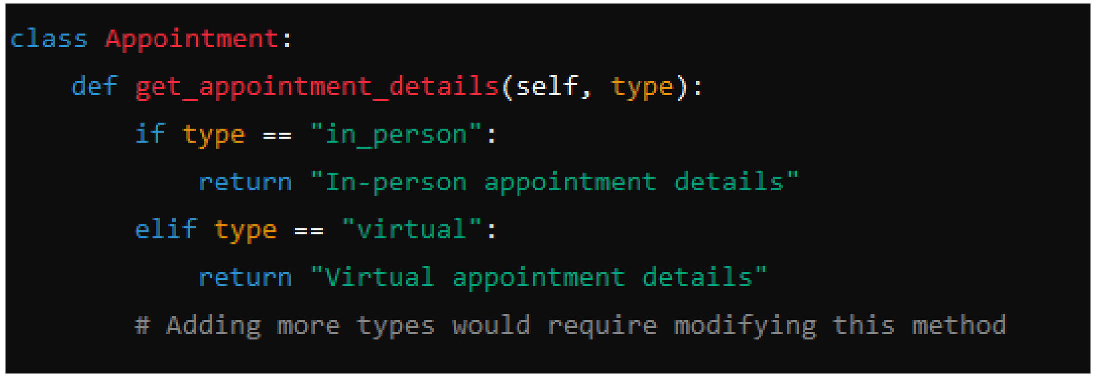
```
class AppointmentManager:
    def create_appointment(self, patient, doctor, datetime):
        # Code to create an appointment
        pass

    def send_confirmation_email(self, patient_email):
        # Code to send confirmation email
        pass

    def log_appointment(self, appointment_details):
        # Code to log the appointment
        pass
```



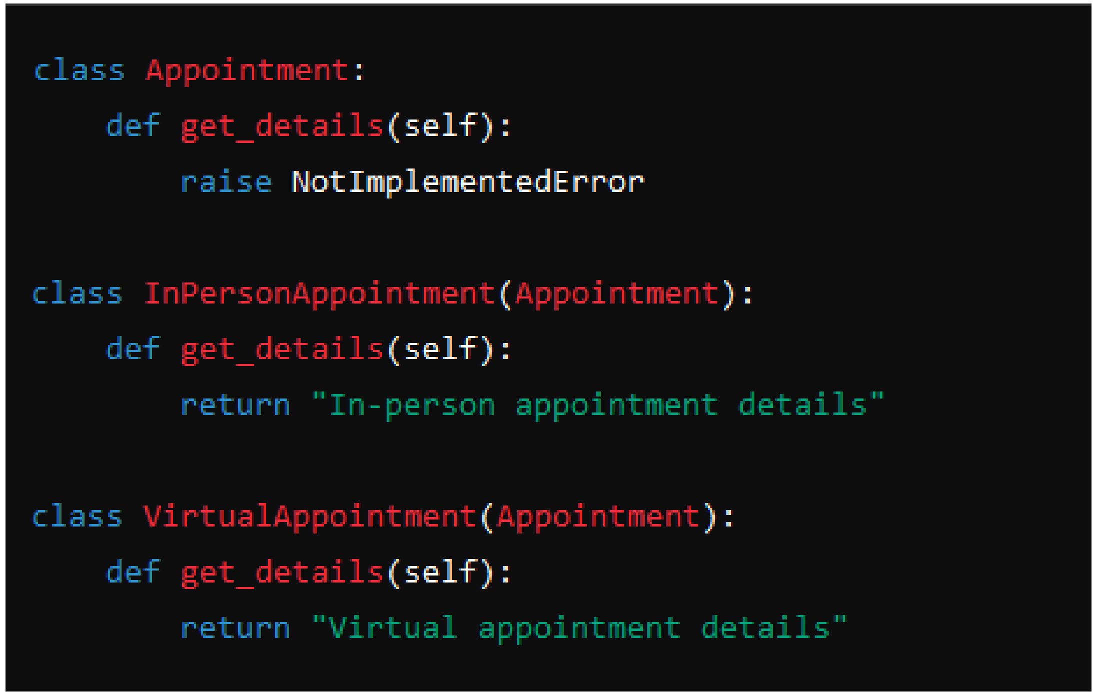
```
class AppointmentScheduler:
    def create_appointment(self, patient, doctor, datetime):
        # Code to create an appointment
        pass

class EmailService:
    def send_confirmation_email(self, patient_email):
        # Code to send confirmation email
        pass

class Logger:
    def log_appointment(self, appointment_details):
        # Code to log the appointment
        pass
```

Open-Closed Principle (OCP): Software entities should be open for extension but closed for modification.

- Software entities should be open for extension but closed for modification. We can achieve this by using inheritance or composition to extend the functionality without altering existing code.



```python
class Appointment:
    def get_appointment_details(self, type):
        if type == "in_person":
            return "In-person appointment details"
        elif type == "virtual":
            return "Virtual appointment details"
        # Adding more types would require modifying this method
```



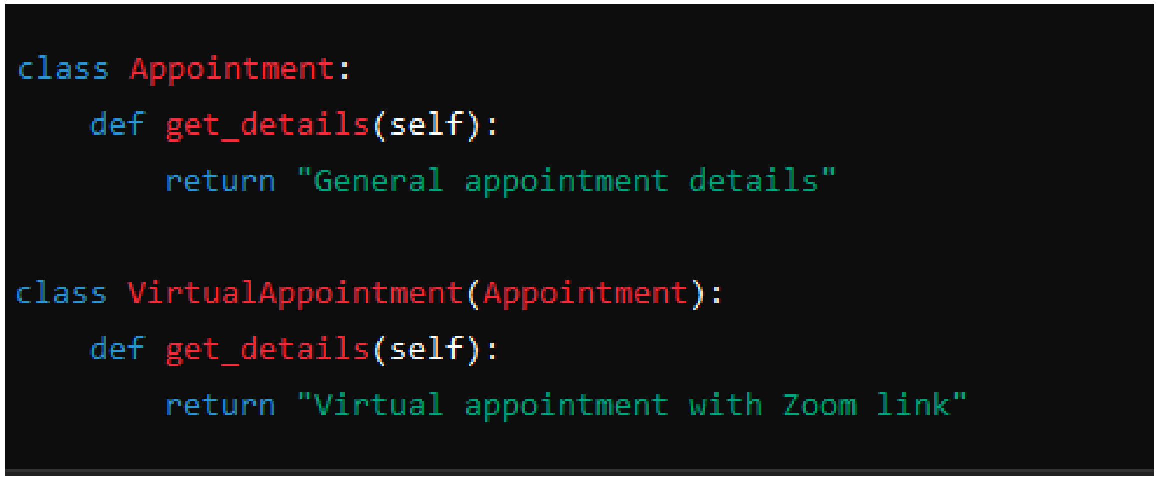
```python
class Appointment:
    def get_details(self):
        raise NotImplementedError

class InPersonAppointment(Appointment):
    def get_details(self):
        return "In-person appointment details"

class VirtualAppointment(Appointment):
    def get_details(self):
        return "Virtual appointment details"
```

Liskov Substitution Principle (LSP):

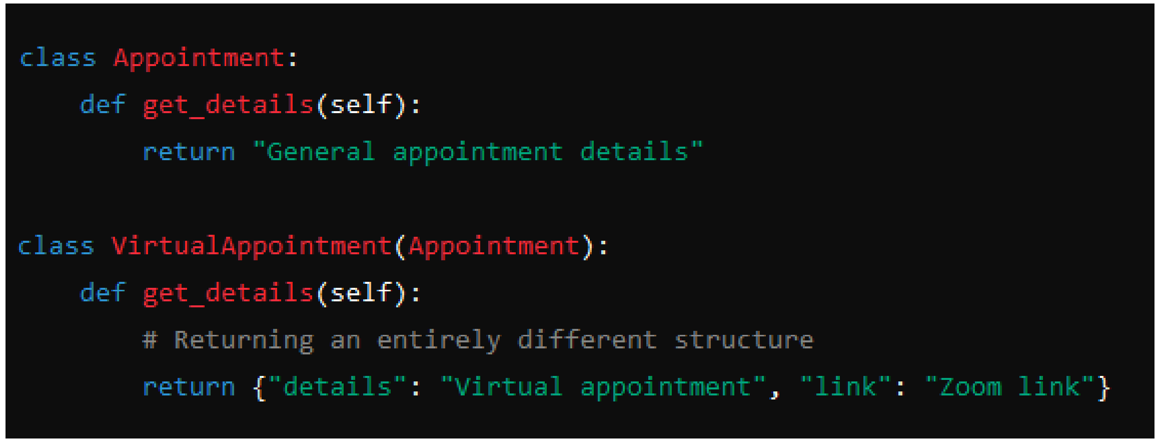
- Subtypes must be substitutable for their base types.
- Inheritance should not violate the behavior expected of the base class.
- Ensures that derived classes don't break the contract of the base class.
- Applying Liskov Substitution Principle (LSP):
  - Ensure that derived classes adhere to the contract of the base class.
  - Validate that inherited methods don't change the behavior expected by clients.
  - Thoroughly test subtypes for correct behavior.

Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types.

- The LSP states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In an appointment booking context, this means all types of appointments should behave consistently.



```python
class Appointment:
    def get_details(self):
        return "General appointment details"

class VirtualAppointment(Appointment):
    def get_details(self):
        return "Virtual appointment with Zoom link"
```



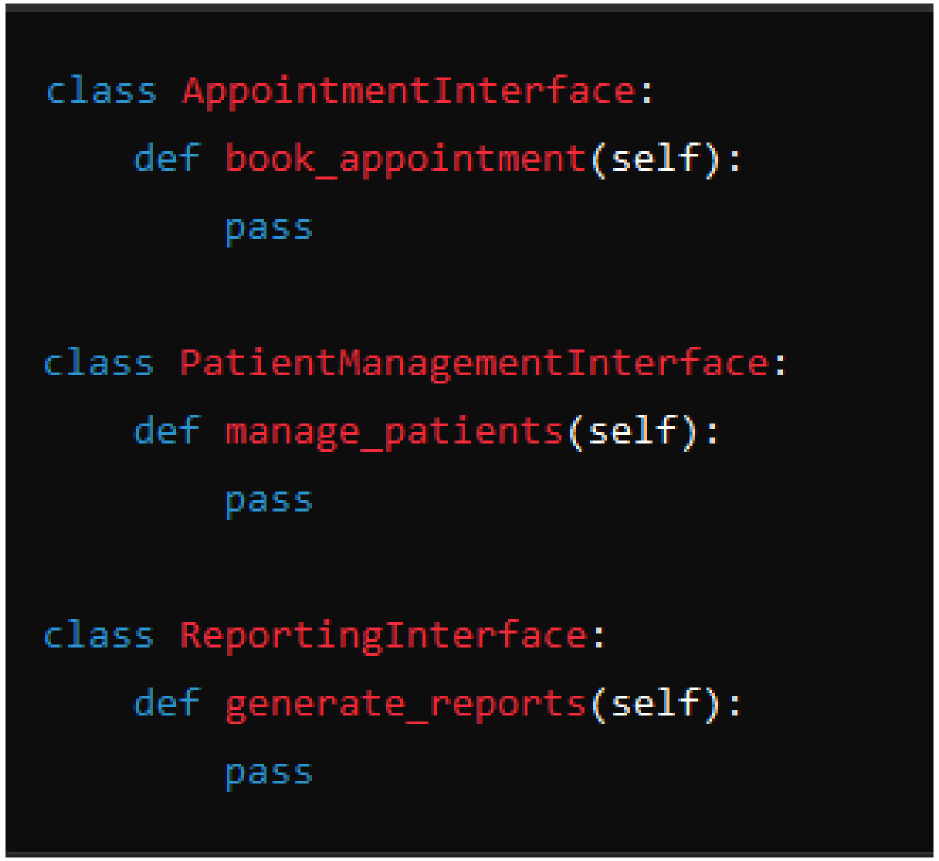
```python
class Appointment:
    def get_details(self):
        return "General appointment details"

class VirtualAppointment(Appointment):
    def get_details(self):
        # Returning an entirely different structure
        return {"details": "Virtual appointment", "link": "Zoom link"}
```

Interface Segregation Principle (ISP):

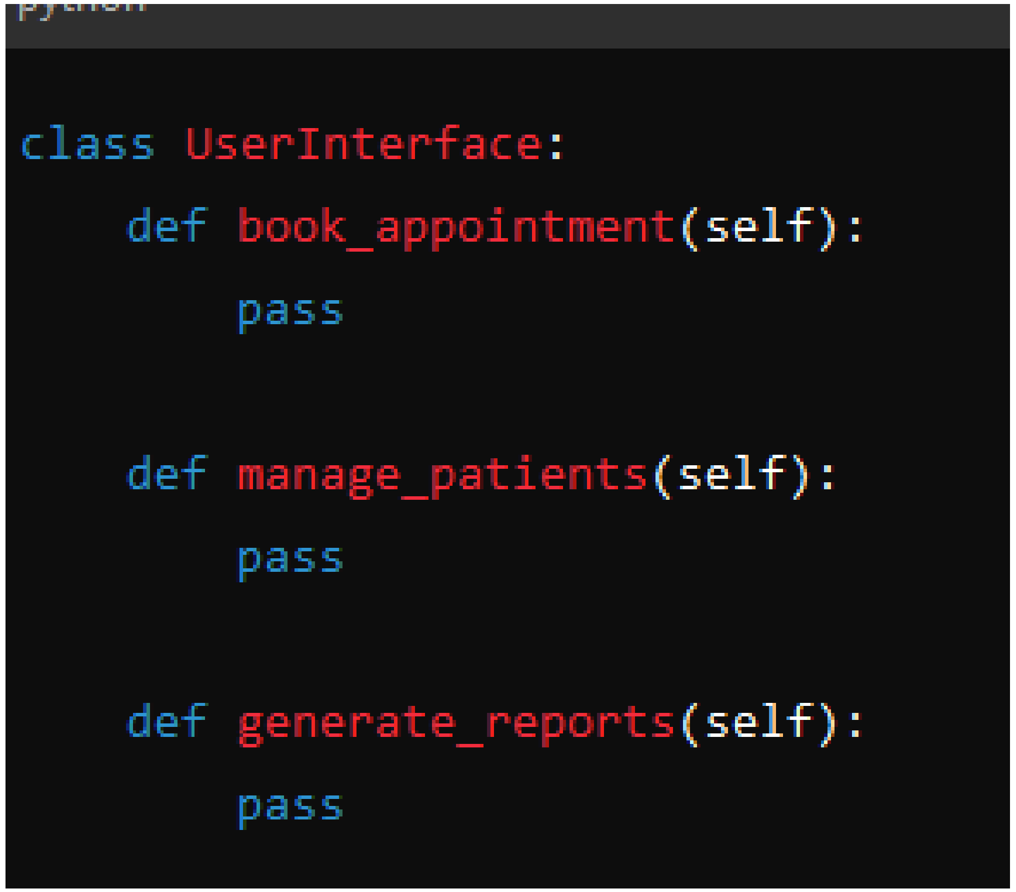
- Clients should not be forced to depend on interfaces they don't use.
- Large, monolithic interfaces should be split into smaller, specific ones.
- Promotes cohesion and avoids "fat" interfaces.
- Applying Interface Segregation Principle (ISP):
  - Create smaller, focused interfaces.
  - Avoid large, monolithic interfaces that force clients to implement unnecessary methods.
  - Use composition to fulfill interface requirements.

Interface Segregation Principle (ISP)

- The ISP states that no client should be forced to depend on methods it does not use. In our app, this means creating specific interfaces for different roles or functionalities.



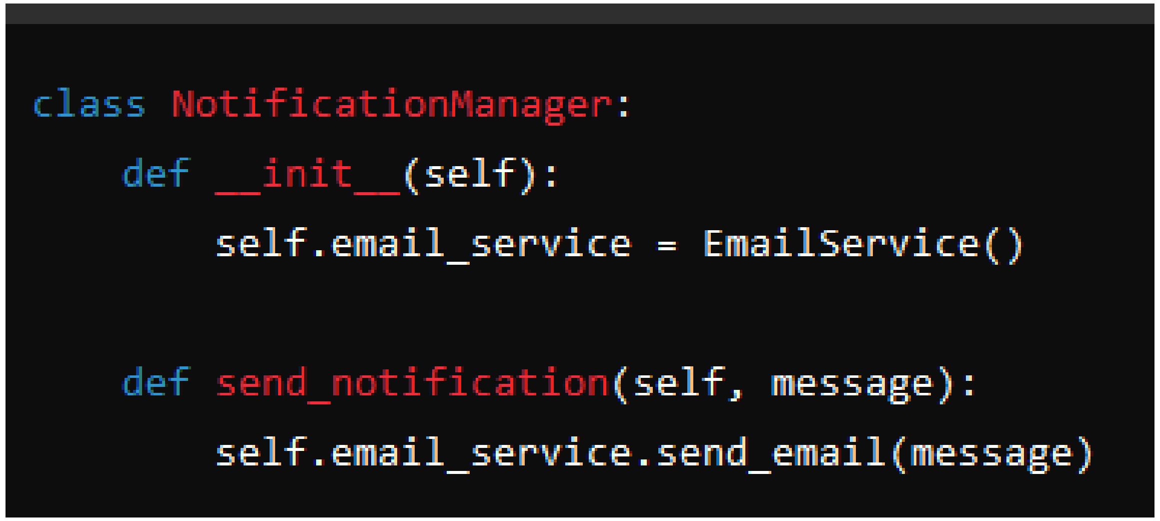
```python
class AppointmentInterface:
    def book_appointment(self):
        pass

class PatientManagementInterface:
    def manage_patients(self):
        pass

class ReportingInterface:
    def generate_reports(self):
        pass
```



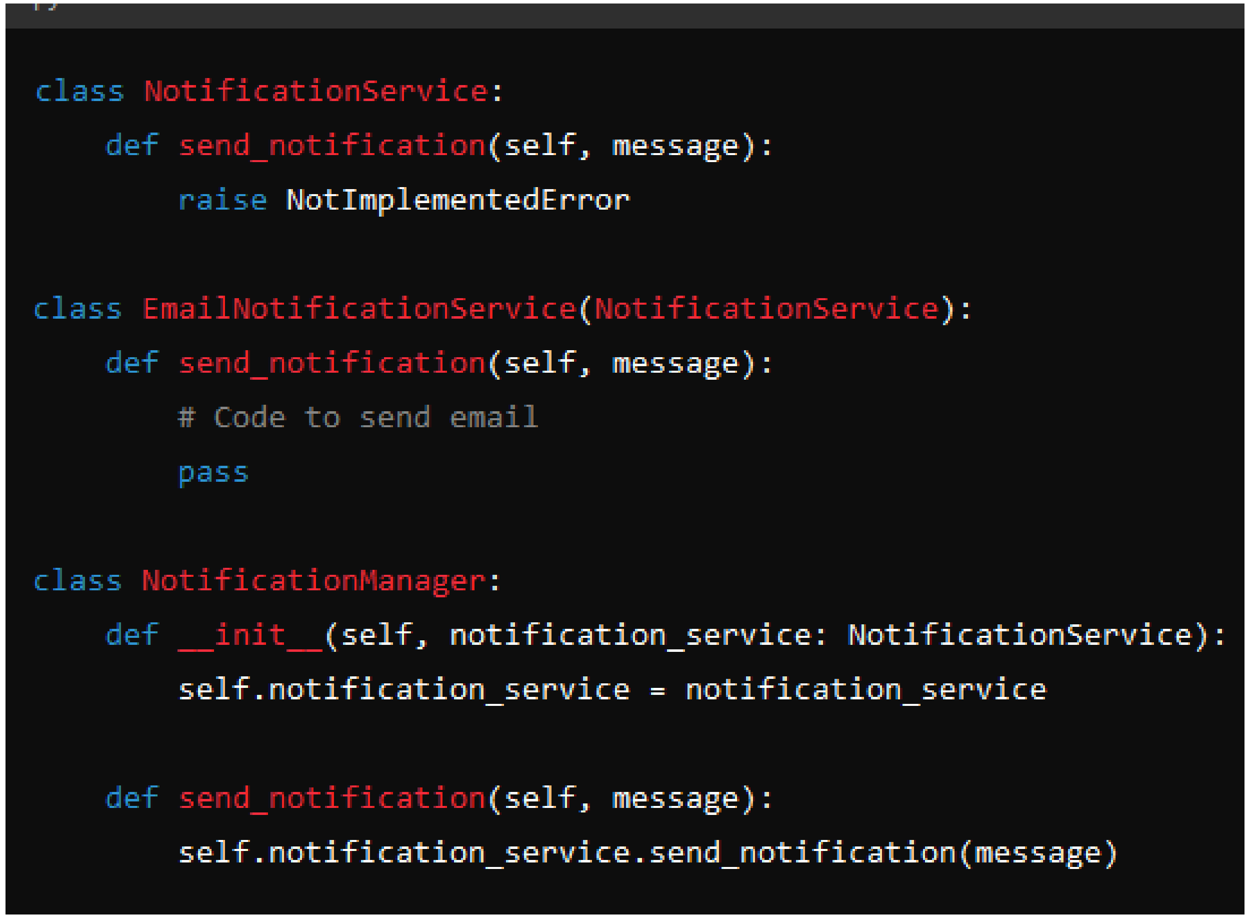
```python
class UserInterface:
    def book_appointment(self):
        pass

    def manage_patients(self):
        pass

    def generate_reports(self):
        pass
```

Dependency Inversion Principle (DIP)

- The DIP states that high-level modules should not depend on low-level modules but rather on abstractions. This promotes loose coupling and flexibility.



```python
class NotificationManager:
    def __init__(self):
        self.email_service = EmailService()

    def send_notification(self, message):
        self.email_service.send_email(message)
```



```python
class NotificationService:
    def send_notification(self, message):
        raise NotImplementedError

class EmailNotificationService(NotificationService):
    def send_notification(self, message):
        # Code to send email
        pass

class NotificationManager:
    def __init__(self, notification_service: NotificationService):
        self.notification_service = notification_service

    def send_notification(self, message):
        self.notification_service.send_notification(message)
```

Benefits of SOLID Principles

- Improved Code Quality: Cleaner, more maintainable code.
- Enhanced Flexibility: Easier to extend and modify.
- Better Collaboration: Clear interfaces and responsibilities.
- Code Reusability: Promotes reusable components.
- Reduced Bugs: Smaller, focused modules are easier to test.

Capstone Project in Detail

- In the context of the Appointment Booking App, the front-end development focuses on creating user interfaces that are intuitive, responsive, and well-integrated with the back end. Key components include:
  - **Booking Form:** Allows patients to book appointments by selecting available slots.
  - **Appointment Management Dashboard:** For clinic staff to view, manage, and reschedule appointments.
  - **Notification Center:** Displays reminders and updates for upcoming appointments.

Conclusion

- Understanding modularization techniques and SOLID principles is essential for writing clean, maintainable, and extensible code. Applying these principles can lead to better software design and development practices