ISO 9001:2008 Certified Institute

Java Institute for Advanced Technology

Object Oriented Design Patterns II
JIAT/OODP II
JIAT/OODP II /EX/01

A.SANDEEP KAVINDA
200205300222
COLOMBO BRANCH

Part A: Patient Record Management

Question 01

Design pattern: **Chain of Responsibility**

Managing patient records which are secure and accessible - The Chain of Responsibility Design Pattern is most appropriate This is so multiple role access checks (eg. doctor, nurse, admin) can be easily extended but not hardcoded all in one spot.

Question 02

For patient records management, you could use the Chain of Responsibility in a way where you chain your access handlers as roles (Doctor, Nurse, Admin, Receptionist etc.).

- Users would try to retrieve or update patient records with the request simply propagating down the pipe.
- Each of the handlers checks the current user role (userBean(). get Role()) has access rights to perform the action requested (e.g., view records, update treatment).
- That access is authorized by handler - is allowed if not, request is forwarded to the next handler - until allowed or not allowed.

This ensures **fine-grained role-based security** while keeping the logic modular and reusable.

Question 03

a) Data Security

- Enforces Role-Based Access Control (RBAC) to prevent unauthorized access (e.g., patients) to view or modify sensitive medical information such as readings/results, doctors, nurses, and admins.
- Protects against unauthorized patient or receptionist access into restricted areas.

b) Accessibility

- We can easily manage multiple functionality without changing logic.

- New responsibilities (e.g., Pharmacist role) can be added into the chain by registering a new handler.
- Gives the ability to have different views (read, write, approve) depending on role.

c) Maintainability

- Separates access-checking from business logic, which also makes the system cleaner.
- The rules for a garden role are encapsulated in distinct handler classes. Separating roles by this kind of handler makes it easy to update its rules (e.g., if the hospital changes its policy).
- Facilitates scalability: as GlobeMed expands, further access rules or roles can be included without rewriting earlier implemented codes.

Part B: Appointment Scheduling

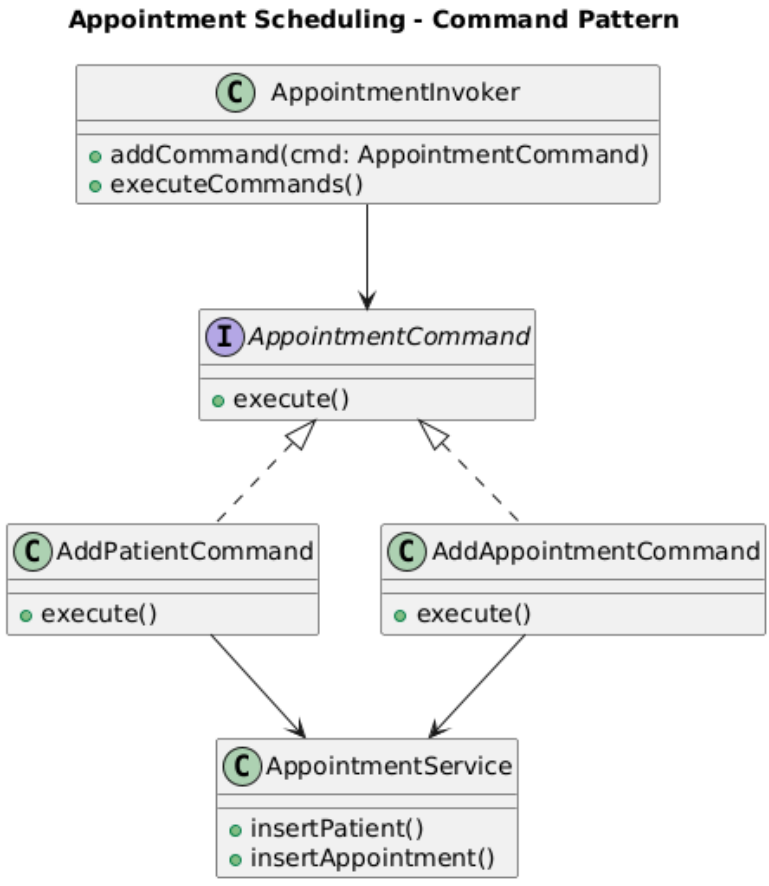
Question 01

The **Command Design Pattern** is the most suitable for appointment scheduling.

Reasoning:

- It takes too many steps to schedule an appointment (checking patient, adding patient, creating appointment).
- Each step can be **encapsulated as a command**, which can be executed, queued, or undone if needed.
- This helps in scheduling revisions easily, resolve conflicts and handle multi location, because each command can be run independently and sequentially.

Question 02



Question 03

Benefits:

- **Decoupling of Request and Execution:** The client code dec to DO does not execute DB operations. Operations are encapsulated in commands that is modularising the game.
- **Supports Multi-step Scheduling:** Each appointment action (adding patient, adding appointment) is a separate command. They can be executed sequentially, queued, or retried in case of failure.
- **Flexibility in Multi-location Scenarios:** It can send commands across services or locations (hospital branches, e.g.) without changing client logic.
- **Undo/Redo Support:** Command can be extended to include support for undo/rollback in case an appointment is to be cancelled or changed.
- **Maintainability and Extensibility:** Adding new kinds of actions for appointments (e.g., sending notifications) only involves adding a new command class; no existing code needs be touched.
- **Error Handling and Logging:** The commands themselves can use internal logging or exception handling, creating a more reliable system.

Part C: Billing and Insurance Claims

Question 01

The **Chain of Responsibility** Design Pattern is the most suitable for this.

Reasoning

- Billing and insurance claims involve multiple steps, such as:
    - Validating patient eligibility
    - Calculating billing amounts
    - Checking insurance coverage
    - Approving or rejecting claims
- Each step can be handled by a separate handler in a chain.
- Requests are passed down the chain until one of its handlers process it and being flexible and decoupled.

Question 02

Benefits of using Chain of Responsibility in billing

- Decoupling of responsibilities
    - All handlers (validation, billing calculation, insurance verification) are de-coupled.
    - The client code will send billing request only at first handler.
- Dynamic processing
    - Handlers can be added, removed, or reordered during runtime.
    - For example, some patients waive insurance verification if they are paying out of pocket.
- Reusable components
    - Standardized handlers that can be reused among various services (consults, treatments, meds).
- Simplifies complex logic
    - Instead of one large billing method with all kinds of nested conditions, CoR breaks down each step into another clean and manageble class.

Question 03

Role in insurance claiming process

- **Flexible Claim Approval Flow**
    - A claim may pass through several handler stages (e.g., eligibility → billing → insurance → final approval).
    - Each handler deals with only one task and is manageable.
- **Scalable for multi-step processing**
    - New steps of approval (such as specialist review and external verification) do not require existing handlers to be modified.
    - It is possible to add more services, insurers and rules to the system.
- **Supports Conditional Handling**
    - Different ways to process a claim based on type (eg, private pay vs insurance).
    - If they're not necessary, the chain can skip handlers dynamically.
- **Error Handling and Logging**
    - Each handler has the ability to log into what it does or handle its own errors itself, providing the benefit of reliability and audit.
- **Ease of Maintenance**
    - A change to one step (e.g., insurance calculation rules) doesn't affect the chain as a whole.

Part D: Managing Medical Staff Roles and Permissions

Question 01

This would best be achieved using the Chain of Responsibility Design Pattern, as it can facilitate the different access of different medical staff (Admin, Doctor, Nurse, Pharmacist, Receptionist) in a chain-like sequential order. Each handler has the choice of processing the request (allowing it), or to pass it to the next handler in the chain.

Question 02

- In a hospital, physicians, nurses, pharmacists, and receptionists have different privileges (e.g., only physicians can update medical reports, only pharmacists can modify medicine plans, only receptionists can add appointment slots.)
- Each role acts as a handler in the Chain of Responsibility and verifies if the current user has the role and the action to carry out the requested activity.

- The access is given if the handler knows the role and the action. Otherwise, it delegates the call to the next handler in the chain.
- An added benefit of this design is that it is very flexible and extensible if new roles or permissions gets added to the system, we can do it easily without modifying existing code just add another handler to the chain.

Question 03

```
package controller.permissions;
import gui.Signin;

public abstract class AccessHandler {
   protected AccessHandler nextHandler;
//Actions
   public void setNextHandler(AccessHandler nextHandler) {
      this.nextHandler = nextHandler;
   }
   public abstract boolean checkAccess(String action);
}

// Admin Permissions
```

```
class AdminAccessHandler extends AccessHandler {
    @Override
    public boolean checkAccess(String action) {
        if (Signin.userBean.getStaffRole().equals("ADMIN")) {
            if (action.equals(AccessControl.ADD_MEDICAL_RECORD)) {
                return true;
            }
        }
        return nextHandler != null && nextHandler.checkAccess(action);
    }
}

// Doctor Permissions
class DoctorAccessHandler extends AccessHandler {

    @Override
    public boolean checkAccess(String action) {
        if (Signin.userBean.getStaffRole().equals("DOCTOR")) {
            if (action.equals(AccessControl.ADD_MEDICAL_RECORD)) {
                return true;
            }
        }
        return nextHandler != null && nextHandler.checkAccess(action);
    }
}

// Nurse Permissions
class NurseAccessHandler extends AccessHandler {
    @Override
    public boolean checkAccess(String action) {
        if (Signin.userBean.getStaffRole().equals("NURSE")) {
        }
        return nextHandler != null && nextHandler.checkAccess(action);
    }
}

// Pharmacist Permissions
class PharmacistAccessHandler extends AccessHandler {
    @Override
    public boolean checkAccess(String action) {
```

```
        if (Signin.userBean.getStaffRole().equals("NURSE")) {

        }
        return nextHandler != null && nextHandler.checkAccess(action);
    }
}

// Receptionist Permissions
class ReceptionistAccessHandler extends AccessHandler {
    @Override
    public boolean checkAccess(String action) {
        if (Signin.userBean.getStaffRole().equals("NURSE")) {
        }
        return nextHandler != null && nextHandler.checkAccess(action);
    }
}
```

Part E: Generating Medical Reports

Question 01

The **Visitor Design Pattern** is the most suitable.
It breaks the report generating logic apart from the main medical data objects (Patient, Treatment, Diagnosis, Invoice). We can extract the necessary logic in a ReportVisitor to report the data instead of having to put the logic in every data class.

Question 02

- In the Visitor Pattern, each medical data object, for example, Patient, and Treatment, has a method accept (Visitor).
- The Visitor interface contains various methods like visit (Patient p), visit (Treatment t), and so forth.
- There is a ReportVisitor class which implements these methods and has the logic for generating the report.
- In this way the storing part of the medical components can remain relatively simple responsible for storing data, while the Visitor can generate a report, and in this way a system with various kinds of reports (like patient summary, diagnostics, financial reports) can easily be added to the mix (without adding fields to its associated data classes) vs just plain values/classes being reported on.

Question 03

The **Visitor Pattern** is useful for medical report generation

- **Separation of Concerns:** Medical Domain Classes (Patient, Diagnosis, etc.) are purely data objects with no tied in reporting logic held over in Visitor classes.
- **Flexibility:** Additional (FinancialReportVisitor, TreatmentSummaryVisitor) reports can be added without any modifications to existing classes for medical reports.
- **Maintainability:** Should report logic need to change only the visitor would need updating - no need to amend the medical objects and thus the duplication of functionality in code is decreased.
- **Extensibility:** We can implement various visitors to achieve other report formats (PDF, Excel, on-screen summary) for the same data.
- **Reusability:** The visitor can be used on different medical objects promoting reuse of reporting logic.

- **Compliance & Auditing:** Centralized reporting logic provides consistent reports as needed with healthcare systems

Part F: Security Considerations

Question 01

The **Decorator Design** Pattern is the most suitable pattern for adding additional security layers to the healthcare system.

By using decorators, sensitive operations on patient records, billing, and reports can be wrapped with security features without modifying the core logic.

Question 02

- **Unauthorized Data Access:** Decors that perform user role & permission checks protect sensitive data from unauthorized read & write operations.
- **Data Tampering:** A decorator with encryption means tampered data can be detected, since what is decrypted will not be what we expect.
- **Audit & Accountability:** Logging decorators log who accessed or modified data, helping with adherence to healthcare law.

Decorators help to make sure these security checks can be applied consistently without touching the underlying business logic, and without the risk of introducing bugs or opening holes.

Question 03

- Secure coding: Protect your software from SQL injection, XSS, and other risks by validating inputs and by using prepared statements.
- Encryption: Sensitive data should be encrypted both in transit (using HTTPS with TLS) and at rest (using AES-256 or equivalent) at the storage layer.
- Authentication & authorization: Use strong authentication and role-based access control (RBAC) to limit what user actions a user can perform.
- Logging & Monitoring: Log all user actions and monitor for anomalies.
- Regular Audits & Updates: Periodically check your security policy and update your software dependencies to remediate the known vulnerabilities.

By integrating best practices with Decorator Pattern in such a creative application, the GlobeMed system guarantees strong protection for confidential health data while providing flexibility and ease of maintenance.