Software Engineering – Design Exercise 2

1. Personio is a Software as a Service company offering an all-in-one HR management and recruiting platform targeted at small and medium-sized businesses (SMBs).

Their software covers the entire employee lifecycle:

1. Core HR
    - Employee Management
    - Digital Employee Files
    - Organizational Chart
2. Recruiting
    - Applicant Tracking
    - Job Postings
    - Interview Scheduling
3. Onboarding
    - Task Management
    - Onboarding Templates
4. Payroll
    - Salary Management
    - Integrations (z. B. DATEV)
5. Time Tracking & Absences
    - Attendance Tracking
    - Leave Requests
6. Performance & Development
    - Goals & Feedback
    - Reviews
7. Integrations & APIs
    - Slack, Microsoft Teams, DATEV usw.

My Model:
1. Layered Architecture
The system is clearly organized into logical layers:

Presentation Layer: GUIClient with components ManagerGUI, HRGUI, EmployeeGUI

Application/Business Logic Layer: personioServer services handling core HR and payroll logic

Integration Layer: IntegrationServer exposing services via APIs

Data Layer: DBServer with dedicated databases

Security Layer: New AuthService component handling authentication

2. Component-Based Architecture
Each functional unit is broken down into components (denoted by <<component>>), encouraging high cohesion and low coupling. For example:

PayrollService, WorkflowAutomationService in personioServer

AuditLoggingService and EmployeeManagementService are separate components for clarity and reuse.

3. Service-Oriented Architecture (SOA)
IntegrationServer exposes business logic as services through APIs like PayrollProviderAPI, EmployeeManagementAPI, and CommunicationAPI.

AuthService appears to be centralized for handling authentication needs.

Services are loosely coupled and likely communicate over a network via HTTP or messaging protocols.

personioServer (Business Logic)
WorkflowAutomationService: Triggers automated HR workflows.

PayrollService: Handles salary calculations and payouts.

EmployeeManagementService: Manages employee data and interactions.

AuditLoggingService: Sends logs to the AuditLogsDB.

IntegrationServer (API Exposure)
PayrollProviderAPI: External access point for payroll processing.

EmployeeManagementAPI: Access point for employee data.

CommunicationAPI: Facilitates internal/external messaging.

GUIClient (Presentation Layer)
UIs tailored to role-specific workflows: HR, Managers, and Employees.

Direct or proxied interaction with APIs (secured via AuthService).

DBServer (Persistence Layer)
UserDB: Stores user credentials and roles (linked to AuthService).

AuditLogsDB: Stores audit trail and logs.

DocumentsDB: Stores documents like contracts or certifications.

AuthService (Security Layer)
Handles authentication (possibly via OAuth2 or JWT).

Interfaces with UserDB.

Secures GUI access and potentially APIs.

Strengths of the Architecture
Scalable: Components and layers can be scaled independently.

Maintainable: Each service is modular and focused on a single responsibility.

Secure: Introduction of AuthService separates concerns for access control.

Extensible: Adding new APIs, UI clients, or services requires minimal changes to the existing system.

2. The architectural style described in the Yahyavi paper is a Service-Oriented Architecture (SOA) tailored for Unmanned Aerial Systems (UAS) operations in a networked environment. This style emphasizes distributed, loosely coupled services that can be independently developed, deployed, and scaled. The architecture is particularly suited to the dynamic and heterogeneous nature of UAS applications, where multiple UAVs and ground stations must coordinate and interact flexibly.

Yahyavi's Architectural Style:

- Service Orientation: Components are exposed as services that communicate through well-defined interfaces. These services are reusable, discoverable, and interoperable, allowing for flexibility in system composition.
- Middleware-Centric: A publish-subscribe middleware layer (such as DDS - Data Distribution Service) underpins communication between components, supporting decoupling in time, space, and synchronization.
- Modularity and Scalability: The architecture supports modular deployment across UAVs and ground stations. Services can be added, removed, or updated without affecting the whole system.
- Real-Time and Reliability Focus: The system is designed with real-time constraints and reliable communication in mind, crucial for mission-critical UAV applications.

Comparison with the 4+1 View Model

1. Logical View (Abstractions and object classes)

- *Yahyavi*: The logical architecture centers around services such as navigation, mission planning, and telemetry, encapsulated as independent modules. Each module provides clearly defined functionalities through interfaces.
- *4+1*: Focuses more generally on object models or class structures; in Yahyavi's case, services can be seen as higher-level abstractions replacing objects.

2. Process View (Run-time behavior and interactions)

- *Yahyavi*: Highlights message-based interactions between services, mediated by the middleware. Services may be distributed across physical nodes, and communicate via a publish-subscribe model that facilitates scalability and fault tolerance.
- *4+1*: Similar emphasis on concurrency and communication, but the Yahyavi architecture is more explicit in using middleware to handle real-time constraints and network transparency.

3. Development View (Software decomposition for development)

- *Yahyavi*: The system is broken into independently developed service modules, each with a defined interface. This encourages team specialization and independent lifecycles for components.
- *4+1*: Also emphasizes modular decomposition, typically into subsystems or packages. Yahyavi's approach fits well within this view, albeit with a service-oriented twist.

4. Physical View (Mapping to hardware)

- *Yahyavi*: Explicitly maps services to UAVs or ground station platforms. The middleware ensures services can be migrated or distributed efficiently, even during operation.
- *4+1*: Physical view is concerned with system topology. Yahyavi extends this with real-time deployment concerns, reflecting the operational demands of UAS.

5. Use Case View (+1)

- *Yahyavi*: Use cases such as “multi-UAV coordination” or “real-time mission updates” drive system requirements and influence the selection and orchestration of services.
- *4+1*: Serves the same role as the user-driven, requirement-capturing perspective; Yahyavi adheres closely to this by designing around specific mission scenarios.

4. a) A whistleblowing system on the internet

Recommended Pattern: Microkernel (Plug-in) Architecture

Justification:

- A whistleblowing system must be highly secure, anonymous, and extendable.
- The microkernel pattern provides a core system for essential services like encryption, identity protection, and submission storage, while allowing additional features (e.g., case management, reporting, analytics) to be added as plug-ins.
- It supports modularity and isolation, minimizing risk exposure.
- This separation of concerns enhances security, maintainability, and customization for different legal or organizational requirements.

b) A video conferencing system

Recommended Pattern: Client-Server Architecture with Elements of Peer-to-Peer (P2P)

Justification:

- A hybrid of client-server and P2P is common in modern video conferencing (e.g., Zoom, WebRTC-based tools).
- The server handles authentication, session control, signaling, and NAT traversal.
- Once a session is established, P2P connections can handle the media streams, improving latency and scalability.
- This structure ensures a reliable user experience, supports real-time communication, and reduces server load.

c) A GPS tracker for cats

Recommended Pattern: Event-Driven Architecture

Justification:

- The system involves sensors (GPS units) that send location data at intervals or when motion is detected.
- An event-driven architecture allows the system to react to GPS signals and status changes efficiently.
- It supports asynchronous communication, which is suitable for low-power, battery-operated devices.
- Backend services can subscribe to events for tracking, alerting, or geo-fencing, making the system responsive and scalable.