Group 20

Assngnment Report - Homework 4

ID2207 HT25 Modern Methods in Software Engineering

Lecturer: Mihhail Matskin

1 Subsystem Decomposition

Class Diagram

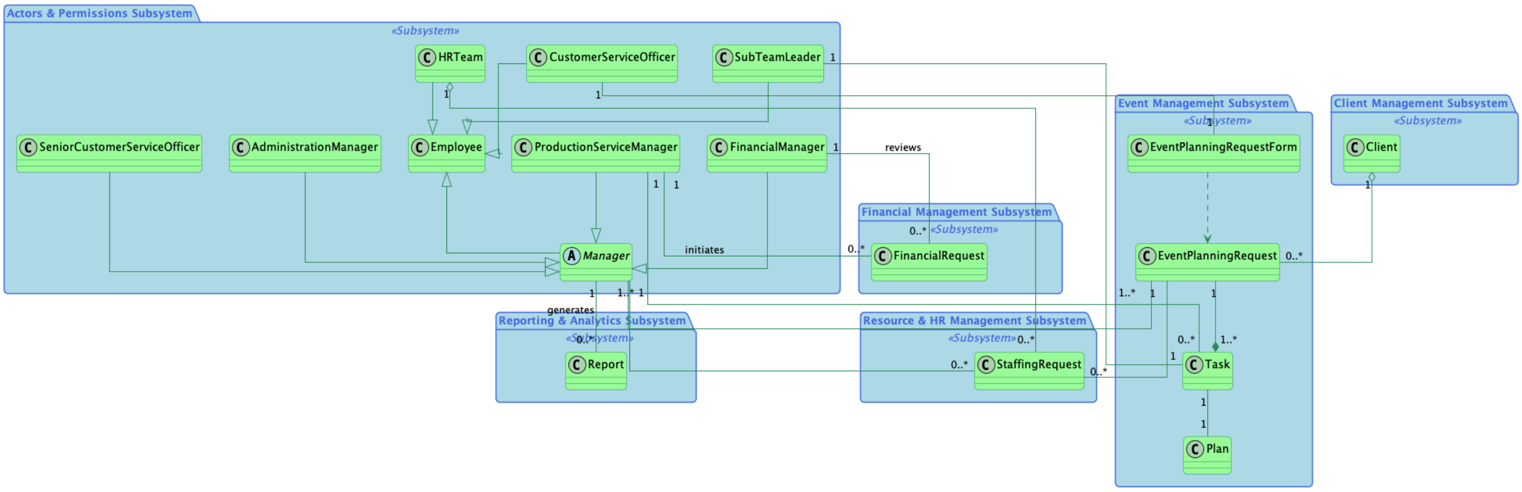

Figure 1: Subsystems of `SEP Internal System`

Subsystem Description	
Actors & Permissions Subsystem	Defines all user roles and their hierarchy, from a general `Employee` to specific roles like `FinancialManager` and `HRTeam`. This subsystem forms the foundation for system-wide access control and permissions.
Client Management Subsystem	Responsible for managing all information related to the `Client` entity. It handles the creation, retrieval, and updating of client records, providing this essential data to other subsystems when an event is planned.
Event Management Subsystem	This is the core functional subsystem that orchestrates the entire event lifecycle. It manages the `EventPlanningRequest` from its creation via the `EventPlanningRequestForm` to the assignment and completion of its constituent `Task`s and `Plan`s.
Resource & HR Management Subsystem	Handles the management of human resources in the context of event planning. Its primary responsibility is to process `StaffingRequest`s that are initiated by a `Manager` when an event requires additional personnel.
Financial Management Subsystem	Manages all financial aspects of an event. This subsystem is responsible for processing a `FinancialRequest` for budget adjustments, which is typically reviewed by a `FinancialManager`.
Reporting & Analytics Subsystem	Responsible for generating various `Report` objects to provide business intelligence. It synthesizes data from other subsystems to create summaries on client statistics, event profitability, and employee utilization for decision-makers like the `Manager`.

2 Mapping Subsystems to Processors and Components

UML Deployment Diagram and Brief Description

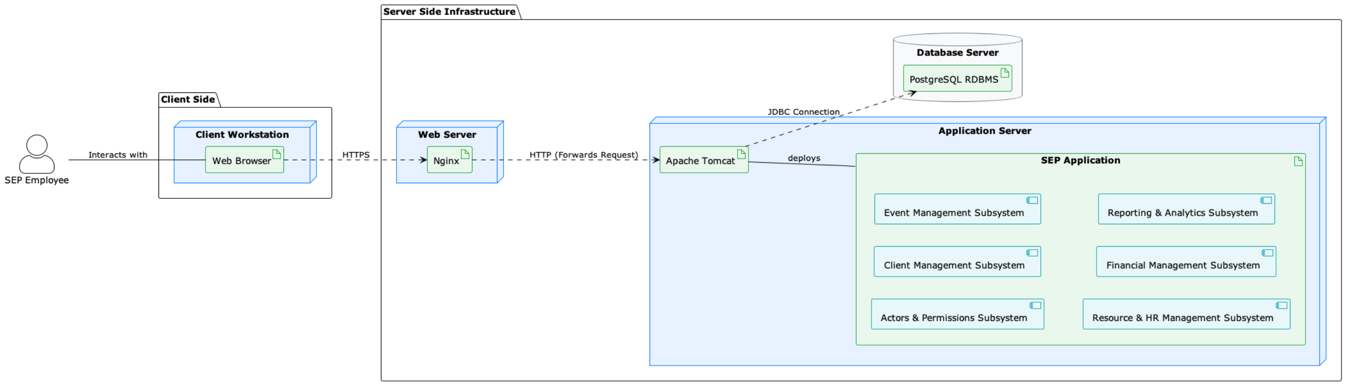

Figure 2: UML Deployment Diagram of `SEP Internal System`

Components Descriptions

Hardware Components	Client Workstation	This represents an end-user's computer, such as a desktop or laptop, located within the SEP office. Its purpose is to run the `Web Browser`, which acts as the client interface to the `SEP Application`.
	Web Server	A dedicated server machine (physical or virtual) responsible for handling all incoming network traffic from clients. It is optimized for network I/O and runs the `Nginx` software to securely route requests to the Application Server.
	Application Server	The primary workhorse of the system's hardware infrastructure. This is a powerful server (physical or virtual) equipped with significant CPU and RAM resources, as it is responsible for executing the entire `SEP Application`'s business logic.
	Database Server	A server dedicated to running the `PostgreSQL RDBMS`. This hardware is optimized for high-speed disk I/O and data-intensive operations, equipped with robust backup and recovery mechanisms.
Software Components	Web Browser	The client-side user interface that runs on an employee's workstation. It is responsible for rendering the system's front-end and sending secure `HTTPS` requests to the server.
	Nginx	Acts as a high-performance reverse proxy. It serves as the single entry point for all client requests, enhancing security by masking the internal application server and efficiently handling static content.
	Apache Tomcat	A robust Java Servlet container that functions as the runtime environment for the core application. Its primary role is to host, manage, and execute the business logic contained within the deployed `SEP Application`.
	SEP Application	This is the central, custom-developed software artifact containing all business logic. It is composed of all defined subsystems (e.g., `Event Management`) and is responsible for handling all functional requirements.
	PostgreSQL RDBMS	The relational database management system that provides persistent storage for all business-critical data like `Client` records and `EventPlanningRequest` details, ensuring data integrity.

3 Persistent Storage Solution

This section identifies the data that must be persistent and outlines the strategy for its storage management.

List of Persistent Objects

To ensure the system can recover its state after a shutdown or crash and maintain a historical record, the following objects, which represent the core business entities, must be persistent:

- `Client`: Contains customer records, which are a primary asset of the company.
- `EventPlanningRequest`: As the central object for any event, its status and details must survive across user sessions and system restarts.
- `Task`: Represents specific work items assigned to sub-teams; their status and details are crucial for project execution.
- `Plan`: The detailed plan submitted for a `Task`, which includes resource and budget needs, must be saved.
- `Employee`: Records of all staff members, including their assignment history, must be kept.
- `StaffingRequest`: These are formal requests for personnel that need to be tracked throughout their approval lifecycle.
- `FinancialRequest`: Formal requests for budget adjustments must be persisted for review and auditing.
- `Report`: Generated reports may need to be archived for historical analysis and tracking of business performance over time.

Storage Management Strategy

For the SEP system, the selected storage management strategy is a **Relational Database Management System (RDBMS)**.
This strategy was chosen for the following key reasons:

- **Structured & Relational Data**: The system's data model is inherently relational. For example, a `Client` can have multiple `EventPlanningRequest`s, and an `EventPlanningRequest` is composed of multiple `Task`s. An RDBMS is expertly designed to model and enforce these complex relationships using foreign keys, ensuring data consistency.
- **Data Integrity and Consistency**: Business applications require a high degree of reliability. An RDBMS guarantees **ACID properties** (Atomicity, Consistency, Isolation, Durability) for all transactions. This means that operations like creating a new event request or updating its budget are handled reliably, preventing data corruption.
- **Powerful Querying Capabilities**: The system must generate various complex reports, such as summaries of employee utilization and event statistics. The **SQL** language, native to relational databases, provides a powerful and standardized way to perform the complex queries and data aggregation needed for this reporting.
- **Maturity and Industry Standard**: RDBMS technology (as exemplified by `PostgreSQL` in our deployment diagram) is a mature, secure, and well-supported industry standard for applications of this nature.

4 Access Control, Global Control Flow, and Boundary Conditions

Access Control

The system applies role-based access control (RBAC). Actors are only allowed to perform operations that correspond to their responsibilities. The following access matrices summarize the mapping between actors and system objects.

Table 1: Access Matrix – Part 1

Actor / Object	EventRequest	ClientRecord
Customer Service Officer	`createEventRequest()`	—

Senior Customer Service Officer	`reviewEventRequest()`
	`rejectEventRequest()`
	`forwardRequest()`
		`createClient()`
		`updateClient()`

Administration Manager	`approveEventRequest()` `finalizeDecision()`	`viewClientRecords()`

Financial Manager	`reviewBudget()`
	`negotiateBudget()`
	`approveBudgetRequest()`
		`viewClientRecords()`

Marketing Team	—	`viewClientRecords()`

Table 2: Access Matrix – Part 2

Task	Plan	StaffingRequest
`createTask()`	—	`createRequest()`
`createTask()`	—	`createRequest()`
`reviewTask()` `updateProgress()`	`submitPlan()`	—

—	—	`reviewRequest()`
		`updateStatus()`
		`processRequest()`

Table 3: Access Matrix – Part3

EmployeeRecord	Report
`manageEmployee()`	`generateAdminReport()`
`viewEmployee()`	`generateFinanceReport()`
`viewSchedules()`	—
`viewSchedules()`	—
`manageEmployee()`	—
—	`generateMarketingReport()`
—	`generateSummaryReport()`

Security measures include centralized authentication and authorization through an `AuthenticationController` (singleton), TLS for all communication, encryption of sensitive client, employee, and financial data, and audit logging of critical actions.

Global Control Flow

The global control flow of the SEP system follows an **event-driven paradigm**. Each `EventRequest` serves as the central event that initiates the workflow. When a Customer Service Officer submits a new request, it triggers a sequence of events handled by different subsystems: the Senior Customer Service Officer may reject or forward it, the Financial Manager reviews the budget, the Administration Manager makes the final approval, and the Production and Service Managers assign tasks to sub-teams. Additional events such as budget adjustments or recruitment requests may be generated during execution. By using an event-driven paradigm, the system ensures loose coupling between subsystems, supports asynchronous approvals, and remains flexible in handling concurrent requests.

Boundary Conditions

Here are three boundary use cases that describe the system's behavior at its operational boundaries, such as startup, shutdown, and critical failure modes.

Table 4: Boundary Use Case: Start System

Use case name	**Start System**
Entry condition	1. The System Administrator is logged into the Application Server's operating system. 2. The `Apache Tomcat` and `PostgreSQL` services are currently not running.
Flow of events	1. The System Administrator starts the `PostgreSQL` database service. 2. The System Administrator executes the command to start the `Apache Tomcat` service. 3. Upon startup, `Tomcat` deploys the `SEP Application`. 4. The `SEP Application` initializes its components, establishes a connection pool to the database, and loads all necessary configurations. 5. The system performs a quick consistency check to ensure all subsystems are operational.
Exit condition	1. The `SEP Application` is fully available and accessible to employees via their web browsers. 2. The system is now waiting for incoming user requests.

Table 5: Boundary Use Case: Shutdown System

Use case name	**Shutdown System**
Entry condition	1. The `SEP Application` is currently running and operational. 2. The System Administrator is logged into the Application Server, intending to perform maintenance.
Flow of events	1. The System Administrator executes the command to gracefully stop the `Apache Tomcat` service. 2. The system immediately stops accepting any new incoming connections from users. 3. The `SEP Application` waits for any currently processing requests to complete to prevent data inconsistency. 4. The application's shutdown sequence is initiated, which cleanly closes all database connections and releases system resources.
Exit condition	1. The `Apache Tomcat` service is fully stopped. 2. The `SEP Application` is no longer running and is inaccessible to users.

Table 6: Boundary Use Case: Handle Database Failure

Use case name	**Handle Database Failure**
Entry condition	1. The `SEP Application` is running. 2. An `Employee` is performing an operation that requires a database transaction. 3. The connection to the `PostgreSQL` database is unexpectedly lost.
Flow of events	1. The `SEP Application` attempts to execute a transaction but receives a critical database connection exception. 2. The system's global exception handler catches the error. 3. The current operation is immediately halted and rolled back to prevent data corruption. 4. The system displays a user-friendly error page to the `Employee` (e.g., "System is temporarily unavailable. The IT department has been notified."). 5. A high-priority alert with error details is automatically sent to the System Administrator.
Exit condition	1. The user is informed of the temporary failure. 2. The System Administrator is notified and can begin recovery procedures. 3. The system enters a degraded state where it may block further database operations until the connection is restored.

5 Design Patterns and Why

In the design of the SEP internal management system, several well-established design patterns were applied to ensure modularity, flexibility, and maintainability.

ModelViewController (MVC) The system is divided into models, views, and controllers. The models correspond to the core entity objects such as `Client`, `Event`, `Employee`, `Task`, `BudgetRequest`, and `RecruitmentRequest`. The views represent the user interfaces, including `EventRequestForm`, `TaskAssignmentView`, and `ReportGenerationView`. The controllers include `EventRequestController`, `TaskController`, and `BudgetController`, which coordinate the workflows and business logic. By applying the `MVC` pattern, we ensure a clear separation of concerns and allow independent evolution of the user interface and the business logic.

Observer This pattern is used to handle automatic notifications across the workflow. When a manager creates a new `Task`, the relevant sub-team members are notified immediately. When a `BudgetRequest` changes status, both the `Financial Manager` and the client receive updates. Similarly, when the state of an `EventRequest` changes, the related departments are informed without requiring direct coupling between components. The `Observer` pattern thus enables event-driven communication while reducing dependencies.

Singleton Certain components must exist only once in the entire system to guarantee consistency. `AuthenticationController` is implemented as a singleton to centralize user login and authorization. `ReportController` is also a singleton to avoid conflicts in report generation. Applying the `Singleton` pattern guarantees system-wide consistency and avoids redundant instances.

Strategy The system needs flexible mechanisms for calculating discounts for clients. The `DiscountPolicy` class defines a strategy interface and supports multiple implementations, such as percentage-based discounts, loyalty-based discounts, or seasonal promotions. By using the `Strategy` pattern, the system can dynamically switch or extend discount rules without changing the financial logic.

Facade Finally, the `Facade` pattern is applied in the reporting subsystem. `ReportingService` acts as a facade that integrates data from Finance, HR, and Event subsystems to provide top management with summary reports. This pattern hides the internal complexity and gives the `Vice President` a simple and unified access point.

6 Contracts for Noteworthy Classes

Invariant for `Client` Class
Class: `Client`
Constraint: Invariant
Description: A client must always have a valid, non-empty email.

```
context Client inv:
  self.email <> ''
```

Precondition for `Client::submitEventRequest`
Class: `Client`
Constraint: Precondition
Description: A client must be registered before submitting an event request.

```
context Client::submitEventRequest(req: EventPlanningRequest) pre:
  self.isRegistered = true
```

Postcondition for `Client::submitEventRequest`
Class: `Client`
Constraint: Postcondition
Description: After submission, the event request status becomes "Submitted".

```
context Client::submitEventRequest(req: EventPlanningRequest) post:
  req.status = 'Submitted'
```

Precondition for `Manager::approveFinancialRequest`
Class: `Manager`
Constraint: Precondition
Description: A manager can only approve a financial request if assigned as its reviewer.

```
context Manager::approveFinancialRequest(req: FinancialRequest) pre:
  req.reviewer = self
```

Postcondition for `HRTeam::processStaffingRequest`
Class: `HRTeam`
Constraint: Postcondition
Description: After processing, a staffing request must be either "Approved" or "Rejected".

```
context HRTeam::processStaffingRequest(req: StaffingRequest) post:
  req.status = 'Approved' or req.status = 'Rejected'
```

Bibliography

[1] ID2207 HT25 Modern Methods in Software Engineering Course Staff. (2025). *Business Case Description For Homework 2, 3 and 4* [Course material]. KTH Royal Institute of Technology, ID2207 HT25 Modern Methods in Software Engineering.