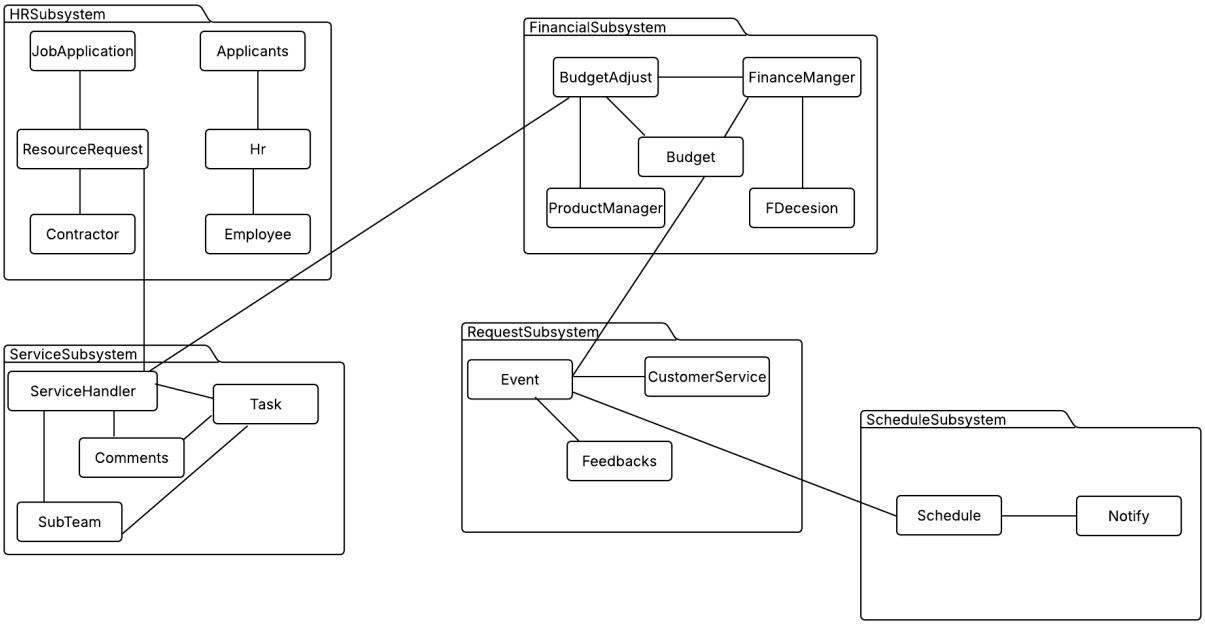
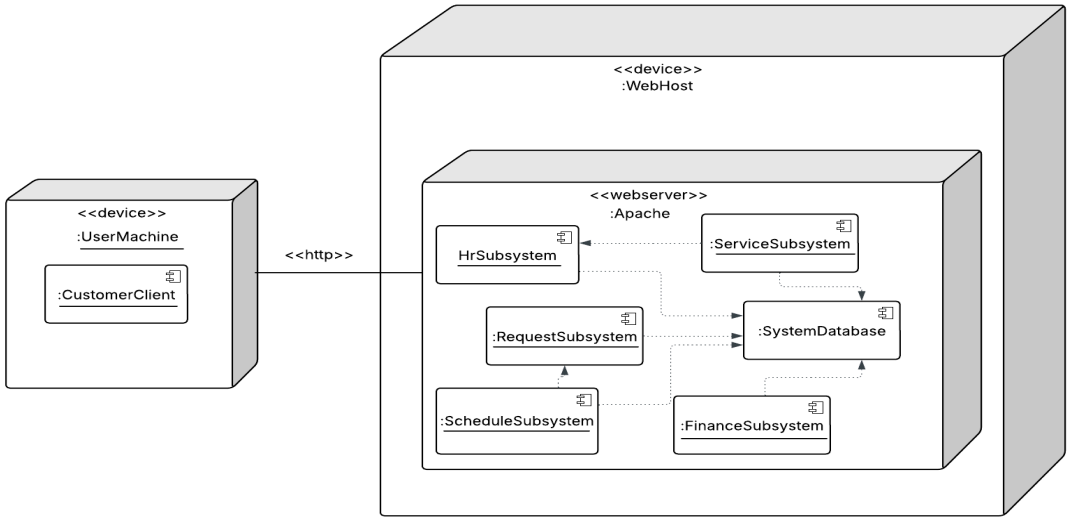
Homework - 4
Group -13

Subsystem decomposition structure (using class diagrams)



Mapping subsystems to processors and components (hardware/software mapping) using UML deployment diagrams.



For this software, there will be one user device which will be used by the customer care or other users to submit requests or update.
For our application we will have an Amazon server which is running Apache and also we will have our database in the same server.

PERSISTENT STORAGE OBJECTS :

Status {
Enum : Open/Progress/Archived
}
	Enumeration representing the state of an event (Open, In Progress, Archived).

Schedules{
ClientID:
Schedules:[]
}
	Stores scheduling information for clients, to have meetings with the team to finalise on the requirements.

ClientID:{
name:
phone:
}
	Contains client details such as ID, name, and contact information

Event{
ClientID:
Status:
Feedbacks:[],
}
	Represents an event, tied to a client, with status and feedback attached.

Budget{
amount:
BudgetAdjustmentRequests:[]
}
	Defines the allocated financial resources for an event, including the total amount

Subteam{
managerID:
name:
staffIDs:[]
tasks:[]
}
	A group of staff within the organization, with assigned manager, staff, and tasks.

Manager:{
name:
Role:
ID:
}
	Represents a manager with authority to assign tasks, oversee teams, and approve requests.

Task:{
taskDescription:
subteamID:
ManagerID:
Comments:[{
Comment:
staffID:
}]}
	A unit of work assigned to staff or subteams, including description and related comments.

ResourceRequest{
Role:
eventID:
NumberofPeople:
ManagerID:
}
	A request for additional staff/resources for an event, raised by a manager.

JobApplication:{ Role: Name: Phone:	Details of applications from potential candidates, including role, resume, and event applied to.

Resume: EventID: }	

Employee:{
Name:
Role:
Salary:
}
	Information on employed staff, including ID, role, and salary.

SELECTED STORAGE STRATEGY : MongoDB, document-oriented storage (NOSQL)

1. Supports hierarchical/nested data (comments, feedback, schedules).
2. Flexible schema allows changes without costly migrations.
3. Also supports indexing like RDBMS.
4. Unlike relational schemas, MongoDB collections let us add new fields or entire schemas quickly without migration overhead. This is important for evolving business requirements.

ACCESS CONTROL :

Access control matrix -

Actors	Event	Task	Budget	Sub-team	Resource Request	Comments	Employee data
HR	View	_	View	Update	View/Update	_	View/Update
Service/ Producti	View	Create/View/Upda	View	Create/View/Upda	Create/View	Create/View/Upda	_

on Manager		te		te		te	
Staff	View	View/Update	_	View	_	Create/View/Update	_
Finance Manager	View	_	Create/View/Update	_	View/Update	_	View
Customer Service	Create/View	_	_	_	_	_	_

- Security in the system is ensured through encrypted communication (TLS) and data encryption at rest.
- Authentication is handled via JWT tokens, providing secure and stateless login sessions.
- Authorization is enforced using MongoDB's built-in RBAC, restricting actions based on user roles.
- Confidentiality of sensitive data (employee details, budgets, resumes) is maintained through field-level encryption and controlled access.
- At the infrastructure level, firewalls, private subnets, and monitoring tools secure the network and detect intrusions

GLOBAL CONTROL FLOW :

User actions (task creation, budget requests, comments..) trigger corresponding service processes. A control layer manages coordination between boundary objects (UI/API endpoints) and entity objects (persistent data in MongoDB). Requests are authenticated via JWT and authorized using MongoDB's RBAC before database operations occur. This ensures that workflows such as scheduling, task assignment, and approvals follow a well-defined sequence of validation, processing, and persistence.

BOUNDARY USE CASES :

1.

Use case name	UserLoginField

Entry conditions	User should have opened the webpage
Flow of events	1 . User enters the username and the password 2. The credentials reach the server and are validated . The server returns a JWT with the user info .
Exit conditions	1. User is logged in 2. Invalid credentials

2.

Use case name	TaskAssignForm
Entry conditions	The user is logged in and has a role of a manager
Flow of events	1 . Creates a new task and inputs the description 2. Chooses an event 3. Assigns a sub-team from the list
Exit conditions	1 .Task has been created and assigned

3.

Use case name	ResourceRequestForm
Entry conditions	The user is logged in and has a role of a manager
Flow of events	1 . Creates a new request , and the requirements (can be budget or resource) 2 . Links it to the event that needs it 3 . Includes additional comments
Exit conditions	1 . The managers have agreed and come to a conclusion. 2. The request has not been approved .

Design Patterns to Designing Object Model:

Handle Service Request:

Command Pattern : Encapsulate each service action as a command object so the UI and controllers invoke a uniform execute() entry point for audit and reversals. ServiceCtrl acts as the invoker, creating commands such as AssignTaskCmd, AddCommentCmd, RequestBudgetCmd, and SetStatusCmd, each targeting the appropriate receiver (EventApplication, EventTask, or BudgetAdjustmentRequest).

How it applies :

- **Invoker:** ServiceCtrl builds and executes commands.
  Command interface: ServiceCommand - execute()
- **Concrete commands:** AssignTaskCmd, AddCommentCmd, RequestBudgetCmd, SetStatusCmd.
- **Receivers:** EventApplication, EventTask, BudgetAdjustmentRequest.
- **Composite structure:** EventTask (component), AtomicTask (leaf), TaskGroup (composite) with add/remove, markComplete(), estimateCost().
- **Flow:** UI → ServiceCtrl executes the command → the receiver updates state → the command logs the outcome.

Composite Pattern：Model the work breakdown as a task tree so code treats single tasks and groups uniformly. Define EventTask as the component interface, with AtomicTask as leaves and TaskGroup as composites; operations like markComplete() and estimateCost() apply at any level. Managers decompose work for sub-teams, while roll-ups compute progress and cost from children to parents.

How it applies:

- **Invoker:** ServiceCtrl builds and executes commands.
- **Command interface:** ServiceCommand - execute()
- **Concrete commands:** AssignTaskCmd, AddCommentCmd, RequestBudgetCmd, SetStatusCmd.
- **Receivers:** EventApplication, EventTask, BudgetAdjustmentRequest.
- **Composite structure:** EventTask (component), AtomicTask (leaf), TaskGroup (composite) with add/remove, markComplete(), estimateCost().

- **Flow:** UI → ServiceCtrl → command execute() → update receivers and totals roll up via TaskGroup.estimateCost().

Handle Client Request

Facade Pattern: Provide a single entry point for client-request operations so controllers don't depend on multiple subsystems. ClientRequestFacade exposes createRequest(), reviseRequest(), getStatus(), and scheduleMeeting(), while internally coordinating validation, account lookup, workflow, calendar, and persistence (ClientRecord, RequestWorkflow, Calendar, repositories). Controllers like CreateRequestCtrl, ReviewRequestCtrl, and ScheduleMeetingCtrl call the facade, which reduces coupling, centralizes policy and access control, and keeps controllers thin.

How it applies:

- **Invoker:** CreateRequestController, ReviewController, SetMeetingController.
- **Facade interface:** ClientRequestFacade - high-level API that hides subsystems - returns DTOs
- **Subsystems:** ClientRecord, RequestWorkflow, CalendarService, EventRequestRepo (persistence), Auth/AccessControl, ValidationService.
- **Key ops:** createRequest(), reviseRequest(), getStatus(), redirectToFinance(), redirectToAdmin(), scheduleMeeting().
- **Flow:** UI → Controller → ClientRequest() → validate + authorize → call subsystems → aggregate result/side effects (persist, schedule, log) → return DTO to controller.

Observer Pattern : Publish - Subscribe for request decisions. The client request (subject) raises events when its state changes (submitted, reviewed, approved, rejected, meetingScheduled). Registered observers (Senior CSO, Finance Manager, Admin Manager, scheduling/notification services) receive updates immediately, so each role acts without polling and UIs stay consistent.

How it applies:

- **Subject:** EventRequest and/or Decision aggregate exposes attach(observer), detach(observer), notify(event, payload).
- **Observers:** SeniorCSO, FinanceMgr, AdminMgr, ScheduleService, NotificationService implement update(event, payload).
- **Events:** submitted, redirectedToFinance, financeFeedbackAdded, approved, rejected, meetingScheduled.
- **Registration:** Controllers register their department observer at session start or workflow entry to system services register at boot.

- **Notification:** When EventRequest.status or Decision changes, call notify(). Observers react (SCSO contacts client, ScheduleService proposes slots).
- **Flow:** Client submits → EventRequest fires submitted → SCSO observer queues review → Finance/Admin add feedback → approved/rejected (fired) → observers perform next steps (schedule meeting, send message, or close).

Handle Finance

Strategy Pattern : Encapsulate budgeting, adjustment, and discount rules as interchangeable strategies. ManageFinanceCtrl selects the appropriate algorithm per client tier, event type, and negotiation stage, so policies evolve without changing controllers or domain objects.

How it applies :

- **Strategy interface:** CostingStrategy with estimate(), adjust(), applyDiscount().
- **Concrete strategies:** InitialEstimateStrategy, NegotiationStrategy, LoyalClientDiscountStrategy, EventTypeCostStrategy.
- **Context:** ManageFinanceCtrl chooses a strategy using inputs from ClientRecord, EventApplication, and RateCard.
- **Receivers/data:** BudgetRequest, Budget, BudgetDecision.
- **Flow:** Controller calls selected strategy → strategy computes figures → controller persists updates and returns results to UI.

Observer Pattern : Publish - Subscribe for finance decisions. When a budget request changes state (feedback added, approved, needs revision, rejected), the subject emits an event. Subscribed departments receive updates instantly and act without polling.

How it applies:

- **Subject:** BudgetRequest / BudgetDecision exposes attach(), detach(), notify(event, payload).
- **Observers:** ProductionMgr, ServiceMgr, SeniorCSO, NotificationService, ReportingService implement update(event, payload).
- **Events:** feedbackAdded, approved, needsRevision, rejected, mandated.

- **Registration:** Observers subscribe at workflow start or app boot; controllers can add/remove observers per case.
- **Flow:** Finance updates decision → subject calls notify() → observers update dashboards, adjust plans, or contact client → UI reflects the new state.

Handle Staff Recruitment

Abstract Factory Pattern : Create a consistent family of HR artifacts for two recruitment paths (internal hire and outsourcing) without scattering conditionals. The factory encapsulates object creation so the HR flow switches "families" by choosing a factory, not by branching everywhere.

How it applies:

- **Abstract factory:** RecruitmentFactory with createAdvert(), createApplication(), createAgreement() (employment contract or vendor order), createOnboarding().
- **Concrete factories:** InternalHireFactory, OutsourcingFactory.
- **Outcome:** JobAdvert, CandidateApplication, EmploymentContract / VendorOrder, OnboardingPackage.
- **Client:** HRPortal / HRController picks the factory based on StaffRequest type.
- **Flow:** HR selects path → factory creates artifacts → controller persists and advances workflow (screening or vendor dispatch).

Adapter Pattern : Integrate external job boards and vendor systems behind stable interfaces. Adapters translate your calls to each provider's API so HR features stay unchanged when platforms vary.

How it applies :

- **Targets:** ExternalJobPlatform (postJob(), fetchCandidates()), VendorService (submitOrder(), checkAvailability()).

- **Adapters:** LinkedInAdapter, IndeedAdapter, VendorAPIAdapter implement targets.
- **Adaptees:** External job-board APIs, vendor APIs.
- **Client:** HRPortal calls target interfaces only.
- **Flow:** HR action → call target → adapter translates to provider API → response normalized → portal updates applications or vendor orders.

OCL contracts pack for SEP

EventRequest

Invariants :

context EventRequest **inv** ValidDates:
startDate < endDate

context EventRequest **inv** AllowedStatus:
status ∈ Set{#Draft, #UnderReview, #Approved, #InProgress, #Closed, #Archived}

context EventRequest **inv** ApprovalRequiresReview:
status = #Approved implies (hasFinanceFeedback = true and hasAdminDecision = true)

context EventRequest **inv** MeetingAfterApproval:
meetingScheduled = true implies status = #Approved

Operation Contracts :

context EventRequest::redirectToFinance()
pre: status = #UnderReview

post: routedTo = #Finance and status = status@pre

context EventRequest::approve()
pre: status = #UnderReview and hasFinanceFeedback and hasAdminDecision
post: status = #Approved and approvedAt <> null

Budget / BudgetRequest

Invariants :

context Budget **inv** NonNegative:
estimated >= 0 and allocated >= 0 and spent >= 0

context Budget **inv** ApprovedConsistent:
approved = true implies allocated >= spent

context BudgetRequest **inv** LinkedToEvent:
eventRequest <> null

Operation Contracts :

context BudgetRequest::applyFinanceFeedback(newAllocated: Real)
pre: newAllocated >= spent
post: allocated = newAllocated

BudgetAdjustmentRequest

Invariants :

context BudgetAdjustmentRequest **inv** PositiveDelta:
deltaAmount > 0

context BudgetAdjustmentRequest **inv** OnlyWhenEventActive:
status = #Open implies eventRequest.status ∈ Set{#UnderReview, #Approved, #InProgress}

Operation Contracts :

context BudgetAdjustmentRequest::approve()
pre: status = #Open
post: status = #Approved and
eventRequest.budget.allocated =
eventRequest.budget.allocated@pre + deltaAmount

EventTask

Invariants :

context AtomicTask **inv** LeafHasNoChildren:
children->isEmpty()

context TaskGroup inv GroupHasChildren:
children->size() > 0

context TaskGroup **inv** CostRollup:
cost = children.cost->sum()

context TaskGroup **inv** CompletionRollup:
isComplete = children->forAll(c | c.isComplete = true)

Operation Contracts :

context EventTask::markComplete()
pre: self.oclIsTypeOf(AtomicTask) or
(self.oclIsTypeOf(TaskGroup) and children->forAll(c | c.isComplete))
post: isComplete = true

Meeting / Schedule

Invariants :

context Meeting **inv** MeetingPreconditions:

request.status = #Approved and startTime > now()

Operation Contracts :

context ClientRequestFacade::scheduleMeeting(request: EventRequest, slot: DateTime)
pre: request.status = #Approved and slot > now()
post: request.meetingScheduled = true and result.slot = slot

StaffRequest (HR)

Invariants :

context StaffRequest **inv** PositiveQuantity:
quantity > 0

context StaffRequest inv **InternalSatisfies**:
resolution = #Internal implies assignedCount >= quantity

context StaffRequest **inv** OutsourcingNeedsVendor:
resolution = #Outsourced implies vendor <> null

Operation Contracts :

context StaffRequest::approve()
pre: status = #Open and quantity > 0
post: status = #Approved
and assignedCount >= quantity

context StaffRequest::outsource(v: Vendor)
pre: status = #Open and resolution = #Outsourced
post: vendor = v and status = #Approved

JobAdvert / Application

Invariants :

context JobAdvert **inv** OpenToAcceptApps:
applications->size() > 0 implies status = #Open

context Application **inv** TargetsOpenAdvert:
advert.status = #Open

Operation Contracts :

context Application::advance(next: ApplicationStatus)
pre: next ∈ Set{#Screening, #Interview, #Offer, #Rejected}
post: status = next

Contract (Employment or Vendor)

Invariants :

context Contract **inv** RoleConsistency:
(type = #Employment implies employee <> null) and
(type = #Vendor implies vendor <> null)

context Contract **inv** ValidDates:
startDate < endDate

Operation Contracts :

context Contract::assignParty(e: Employee, v: Vendor)
pre: (type = #Employment implies e <> null) or (type = #Vendor implies v <> null)
post: (type = #Employment implies employee = e) and
(type = #Vendor implies vendor = v)