**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)