Mia Harang

Doyle Chism

Sam Gumm

Kenny Jia Hui Leong

Mani Raj Rejinthala

Computer Science 362

Project topic: Fashion

Iteration 01

Inception Paragraph

A fashion company wants to develop an electronic project management software. This company has many departments, including the Design, Marketing, Modeling, Recruitment, Manufacturing, Inventory, and Treasury Departments. Each department consists of a Head of Department, some department managers, and various employees under the managers. This product will allow the different teams within each department to track their design and supply chain flows. It allows the design department to retrieve product requests, create a blueprint of design, and transfer that design to other departments. The manufacturing department can take that design and put it into production.

It allows the head of inventory and logistics to create fashion product reports. Further, it allows the logistics and inventory manager to view orders, view the pending order requests, view the inventory, and initiate the order for shipment. It allows the Storage In charge to create, delete, and modify orders for retailers. The warehouse employee is able to update the quantity of the product after loading and unloading the products at storage.

The Head of Finance should be able to retrieve all employee information on all departments from Human Resources, enter employee information and retrieve a calculated payroll report, send payroll reports and payslips to Human Resources, make changes to employee salary, make new payroll reports with changed salaries, send payroll report with changed salaries to the head of departments and Human Resources for verification, receive approval for payroll report with changed salaries from the head of departments and Human Resources.

The company will be responsible for system security, so our software will not include support.

Departments [Use Cases]

HQ	Design	Manufacturing	Marketing
proposeConcept	designProduct	createProduct	advertiseProduct
transferConcept	approveDesign	transferProudct	advertiseEvent
	transferDesign		distributeCatalog

Inventory	Modeling	Treasury	HR
updateInventory addProducts deleteProducts	haveEvent	processPayroll	hireEmployee collateSalaries editEmployee removeEmployee handleCandidate handleEmployee
Placeorder	scheduleFittings	adjustSalary	
	trainModels		

Committee			
createCatalog			

Manufacturing Use Cases

Use Case #1: createProduct

Primary Actor: Head of Manufacturing

Supporting Actor: manufacturing managers and workers

Offstage Actor: Head of Design

Stakeholders: Head of Design and the Headquarters because the fashion brand depends on the product we produce.

Precondition: All the materials have been acquired and are stored in a warehouse so there is no need to order more raw materials. We are assuming that since we have all the raw materials that once we receive a design from the design team we can begin production. We are also assuming that we have all the proper machinery to create any product that the Design team sends.

Main Success Scenario:

1) The Head of Manufacturing receives the design for a product from the Head of the Design Team
2) The Head of Manufacturing will review the design and determine the raw materials needed for production.
3) The Head of Manufacturing will alert the Manager to collect the necessary raw materials from the warehouse in the manufacturing plant.
4) The head of manufacturing will tell the manager what machines are needed for creating the product.
5) The head of Manufacturing will verify the raw materials collected from the manager match the design before creating a product.
6) After verifying the correct raw materials, the manager will turn on the machines that are needed for producing the given product.
7) The workers will receive the raw materials from the manager to begin work.

8) The workers will start working on the given machines so the product can be produced.
9) The workers give the final product to the manager so he can then give it to the head of manufacturing for it to be quality tested.
10) If the TestProduct fails quality: redo steps 7-9.
11) If the TestProduct is approved then begin make X amount for the design team.

Extensions:

1A) The design lists the raw materials that they want to use for creating the product

6A) The machine does not turn on, so the manager must alert the head of manufacturing that the machine needs to be fixed.

8A) The manager will make sure the workers are working so the product can be created efficiently.

10A) quality can fail due to wrong size, holes, damaged materials, not up to standard.

Use Case #2: designDesign

Primary Actor: Head of Design Team

Supporting Actor: Head of Manufacturing & Design Team workers

Offstage Actor: Customers looking to purchase the product

Stakeholders: The Headquarters of the company has a vested interest in the design team and manufacturing communicating effectively to produce the best product possible for maximum revenue.

Precondition: We are assuming that the manufacturing plant has all the necessary rawMaterials to begin production when the design is sent to them.

Main Success Scenario:

1) The Head of Desing decides what work together type of fashion product they will create(shirt, pants, shoes, etc.)
2) Design Team workers will draw a sketch of the design and share it with the Head of Design Team.
3) The Head of Design Team will review all the sketches from the Design Team
4) Head of Design will select a design and share which one they will be developing with the Design Team.
5) The Head of Design and the Design Team will then work together to improve the design that was selected so it can be sent to production.
6) The Head of manufacturing and Design team will then create a final Design after working together, so it can then be sent to production.

Extensions:

3A) The head of manufacturing will either approve or disapprove of the design for production.

5A) Define colors associated with the design

5B) Define raw materials needed for the creation of the product

5C) Determine the sizes the product will be so manufacturing knows what to make

5D) Decide on a quantity to tell the manufacturing the correct amount to make

Inventory

Use Case: Place Order

Actor: Retailer

Precondition:

1. The Storage In-Charge (SIC) is available to place, modify and delete the order for retailers.

Main Success Scenario:

1. The Retailer arrives at the storage facility and meets the SIC.
2. The Retailer expresses interest in placing an order and provides their retailer id
3. SIC finds the information of retailer
4. System fetches and gives the retailer information like name and location
5. The Retailer requests the desired fashion items and specifies the quantity for each.
6. The SIC checks the real-time availability of each item
7. System confirms the quantity requested.
8. The SIC enters the finalized order details in the system, including, fashion item and quantity
9. System records the order details
10. The system generates a preliminary order summary.

11. The Retailer reviews the order summary and confirms the details with the SIC.

12. The SIC add the order

13. The system creates the order ID and updates the inventory to reflect reserved stock.

14. The system generates an order confirmation receipt, which the SIC prints to the Retailer.

15. The Retailer acknowledges receipt of the order confirmation

16. The SIC completes the order placement process.

Extensions:

1. If the Retailer id not exist

   a) SIC finds the retailer id

   b) System shows invalid

   c) SIC requests the details, like name, and location of retailer

   d) Retailer provides the details

   e) SIC enters the information

   f) System acknowledges the information

   g) SIC add the retailer

   h) System provides the retailer id

2. If the requested quantity is unavailable

   a) SIC add the item to order

   b) System shows unavailable

c) SIC informs the retailer that the requested quantity for the respective item is not available.

d) SIC finds the available quantity

e) System shows the available quantity for item

f) SIC informs the retailer about the available quantity

g) Retailer finalizes the adjustment to the order

h) SIC re-add the item to order with finalized information.

Use Case: Update Inventory

Actor: Warehouse Employee

Precondition:

None

Main Success Scenario:

1. The Warehouse Employee logs into the inventory management system
2. Employ enters the item id
3. System fetches the item details
4. Employee verifies the item description with the actual item.
5. Employee chooses to either add or remove of certain quantity of a fashion item from the storage.
6. System acknowledges the change and update the quantity of the item

7. The Warehouse Employee reviews the updated inventory records and confirms that changes are accurate.

Alternate Main Success Scenarios:

a) Add

1) Warehouse Employee add a certain quantity of an item

2) System acknowledges the information

b) Remove

1) Warehouse Employee remove a certain quantity of an item

2) System acknowledges the information

Extensions:

1. If the product id incorrect:

   a) The system prompts the employee to re-enter or correct the data.

   b) The employee reviews and corrects the details, and the system verifies the updates.

2. If the product id missing:

   a) The system prompts the employee to re-enter or correct the data.

   b) The employee requests the registered products in the inventory

   c) System shows the list of registered items

   d) Employee register the product with details like name, and description

e) System registers the product and creates the product id

3. Insufficient Stock for Removal:

a) Employee enters the quantity of an item to remove for unloading

b) System prompts the error

c) Employee reviews the information correctly

d) The Warehouse Employee adjusts the quantity for removal action.

e) System updates the product records

Modeling Department

Use Case #1:

haveEvent

Actor(s): [Head of Modeling], Team Managers, Marketing Department, Models

Preconditions:

1. The company can afford to book the venue
2. There are enough employees to cover the event
3. The modeling team knows schedules to accommodate to
4. There is a cleanup crew that cleans after the event

Main Success Scenario:

1. The Head of Modeling meets with their team to discuss the need for an event and where [Photoshoot -or- Fashion Show]
2. The Head delegates tasks to their team to book the event
    a. The Head directly communicates with the venue to book the time and with the marketing department to start the creation of necessary advertisements or possible collaborations. They will also plan the event's order.
    b. The Modeling manager delegates models to perform in the event
    c. The Clothing manager is responsible for fitting the models with the proper garments
    d. The Makeup manager is tasked with finding the perfect makeup and hair for each model and garment
3. Once the Head books a venue and lets the Marketing department know, they will tell the Modeling manager to pick 1 or more models to perform at the event (whichever is deemed necessary).
    a. In this exchange, the Head will give the modeling manager any celebrity profiles or collaborations the manager would need to know about.
    b. The modeling manager will send the selected models and celebrities to the clothing team to get fitted for the necessary garments
4. When the clothing team receives the models and details for the event, they will book multiple fittings for each model to get fitted for a garment.
5. The clothing manager will send the desired garment information for each model/celebrity to the makeup team.
    a. This team will plan out hair and makeup for each model/celebrity
6. On the day of the event, after all the fittings have been done, and planning has been finished, the entire modeling department will arrive at the venue, a couple

of hours (to days) before the event.

7. The models will go through this scenario:
    a. Arrive at the venue
    b. Go to the clothing team to get the assigned garment
    c. Transfer to hair and makeup, and get their planned designs.
    d. Go to the green room/waiting area to wait for the Head to assign them their task
8. Once all models/celebrities go through that process, the Head will delegate the tasks to each person that must perform and the order.
9. At the exact time of the event, the Head will motion to the first model/celebrity to begin their assigned task (walking the runway or getting their picture taken)
10. During the event, the Head will let each model/celebrity when their turn is
11. Once the event ends, each model will go to the clothing team to have their garments removed, then the makeup team to get their makeup removed.
12. Once the Head approves, everyone is prompted to leave for the day. The clean-up crew finishes it off.

Alternative Flows:

- A model/celebrity is unable to make the show
    - The head will contact each team to assign their garment and makeup to a new model
- Unforeseen weather events
    - The head will talk to marketing and HQ to determine if the weather is bad enough to cancel the event. If it is, each employee will be compensated for their time.
- A brand deal backs out
    - The head will communicate with the clothing and makeup teams to make sure the brand is not being shown on any model.

Use Case #2:

scheduleFitting

Actor(s): [Head of Modeling], Clothing Manager, Model Manager, Models, Manufacturing Department

Preconditions:

1. The design team has a design that they want to present at an event
2. There is a model to be fitted
3. The modeling team knows the model's schedule

4. There is a studio for the fitting to take place
5. There exists a storage unit for custom-fitted garments

Main Success Scenario:

1. The model manager decides on a model to take to an event.
2. The model manager gives the clothing manager and team the model's profile and schedule
3. The clothing manager sends the custom sizing and design for the model to the manufacturing department to be custom-made.
4. Once the garment is in the clothing manager's possession, the manager will review the model's schedule and propose times for the fitting to commence.
5. The model and clothing team will meet at a studio on the exact date and time of the fitting.
6. The model will put on the garment.
    a. The clothing team will check around for any issues with the garment, sizing, or looks wise.
    b. The clothing manager will take any necessary notes on the garment.
7. Once the first fitting is complete, the clothing manager will send the team on their way, and the garment back to manufacturing, along with the notes they took.
8. They will have 2 to 3 of these fittings before the garment is considered 'event-ready'
9. Once the garment has been fitted correctly, the clothing team will place it in storage, waiting for the next scheduled event.

Alternative Flows:

- A model/celebrity is unable to attend the fittings
    - The outfit will be made with only their measurements in mind, any quick adjustments happen at event time.
- The outfit is not 'event-ready' after 3 fittings
    - The clothing manager will ask the model to come to the event earlier than the rest to do last-minute alterations.

Treasury

Use Case: processPayroll
Level:
Primary Actor: Head of Treasury
Stakeholders & Interests:

- HoT: Wants fast and accurate processing of data, wants no payroll calculation errors, can result in financial losses, or penalties.
- HoHR: Wants to receive payroll data, will make salary changes or bonuses, address employee concerns with payroll.
- CEO/Headquarter: Wants payroll data to determine quarterly earnings.
- Government tax agencies: Wants accurate data of payroll taxes, make sure company adheres to labor laws and other standards.

Preconditions: employee data is complete and up-to-date

Main success scenario:

1. HoT provides employee data, e.g. hours worked, salary rate, benefits rate, overtime rate.
2. System records data and presents calculated gross pay based on employee data.
3. System calculates and presents new salary data after deductions from taxes, retirement contributions, insurance.
   HoT repeats 1-3 until all done.
4. System completes calculation & stores payroll data to repository.
5. System presents payslips and payroll data.
6. HoT sends payslips to required departments.
7. HoT sends payroll data to Headquarters.
8. HoT closes system.

Extensions:
1a. Employee data is invalid

1. System flags and indicates error on employee data.
2. HR updates or corrects employee data.
3. System receives new data and resumes.
4. HoT continues updating system with data.

2a. Employee work hours discrepancy

1. System flags and indicates discrepancy on employee data
2. Head of X Department reviews and approve discrepancy of unauthorized overtime or missing clock-ins.
   2a. Head of X Department disapproves and confront employee about discrepancy
      1. HoD approves discrepancy of data
         1a. HoD disapproves and forward data to HR
3. System accepts data and resumes
4. HR continues updating systems with data.

3a. Deductions exceed gross pay

1. System flags and indicate negative value for payroll
2. HoT performs deductions override, e.g. remove certain deductions based on company policy.
3. System recalculates employee salary based on overrides.
4. HoT continues updating system with data.

Use Case: adjustSalaries
Level:
Primary Actor: Head of Treasury
Stakeholders & Interests:
- HoT: Wants fast and accurate processing of data, wants no salary calculation errors, can result in incorrect payroll amount and penalties.
- HoHR: Wants to receive data to ensure it adheres to company policy, market standards, and employee performance.
- HoDs/Managers: Wants salary data that correctly equates to employee performance.
- Headquarters: Wants salary data to ensure employees are paid in correct amounts.
- Employee: Wants salaries that correctly recognizes their contributions and follows the market rate.

Preconditions: Employee data is complete & up-to-date.

Main success scenario:
1. HoT provides salary changes data, e.g. promotions, deductions, new market rate, or policy changes.
2. System records data and presents calculated gross salary based on existing employee data.
3. System completes calculation and presents data to HoT
4. HoT sends new salaries to HR and HoDs.
5. HR & HoDs approve new salaries.
6. HoF sends payroll data to Headquarters.
7. Headquarters approves new payroll.
8. HoT approves new payroll in system.
9. System stores new payroll data in repository.

Extensions:
2a. Insufficient budget detected
1. System indicates insufficient budget for new salary adjustment.
2. HoT notifies HR and Managers/HoDs about budget constraint with new salary adjustment.
    2a. HR & Managers/HoDs scale back salary increase
        1. HoT enters new salary adjustments into system.
            1a. Employees see new salary data and dispute with HR.
                1. HR adds delayed salary increase into system.
                2. System calculates current and future potential payroll, then updates

repository.

2. System recalculates current salaries and payroll, then updates repository.

2b. HR & Managers/HoDs delay pay increase to future

1. HoT enter new delayed salary adjustment to system
2. System adds future salary adjustment and updates repository.

9a. Employee rejects salary adjustments

1. Employee suggests readjustment of salary with HR/Manager/HoD.

1a. Readjustment is approved and request for readjustment is sent to HoT

1. HoT receives request and re-enter salary adjustment to system.
2. System recalculates salaries and payroll, then updates repository.

1b. Readjustment is rejected.

Human Resources Department

Use Case 1: manageEmployeeRecords

Actors:

- HR Assistant
- HR Manager
- Store Managers
- IT Specialist

Preconditions:

- Employee data from all stores (i.e., personal details, employment history, etc.) is collected and consistently formatted for entry into system.
- The mainframe system is set up with the custom software installed and configured.
- Security measures are in place.

Main Success Flow:

1. **Store Managers Collect Employee Information:**
    - Store Managers compile employee information using standardized forms.
    - Data includes new hires, terminations, promotions, and other relevant changes.
    - Information is sent to the headquarters.
2. **Store Managers Send Data to Headquarters:**
    - Store Managers format the data according to established standards and send it to the headquarters for further processing.
3. **HR Assistant Inputs Data into System:**
    - HR Assistant receives the data and enters it into the database.
4. **System Validates Data:**
    - The system runs validation checks to ensure data is consistent and adheres to established structure.
    - System raises errors and omissions.

5. **HR Assistant Reviews and Corrects Data:**

    - The HR Assistant reviews flagged errors, makes necessary corrections, and ensures data accuracy.

6. **System Updates Employee Records:**

    - After validation, the system updates the database with new or modified employee records

7. **HR Manager Generates Reports**

    - HR Manager uses the system to generate reports on employee records as necessary for decision making.

8. **HR Manager Reviews Records**

    - HR Manager reviews the reports generated.

Extensions:

- **4a. Data Format Issues Detected**:
    - If data provided by Store Managers is improperly formatted:
        - System Flags the Issue.
        - HR Assistant coordinates with the Store Managers to correct the formatting and re-submit.
        - The system resumes validation once data is corrected.
- **5a. Missing Employee Information**:
    - If the system detects missing or incomplete employee information:
        - System Flags Missing Data.
        - HR Assistant reaches out to the Store Manager to collect the missing information.
        - Once data is updated, the system proceeds with record maintenance.
- **7a. Inaccurate Reports Generated**:
    - If generated reports do not reflect expected results due to incorrect data entry:
        - HR Manager Flags Discrepancy.
        - HR Assistant reviews the records, identifies incorrect entries, and corrects them.
        - The system re-generates the report with updated information.

- **8a. Data Changes Not Approved**:
  - If data changes are not approved by the HR Manager:
    - The HR Assistant is notified to revert or correct the changes.
    - The system maintains an audit trail of modifications for transparency.

Postconditions:

- **Centralized Employee Database Updated**: A centralized, accurate, and up-to-date employee database is maintained.
- **Reports Generated**: Accurate employee reports are generated to support decision-making.
- **Improved Data Accuracy**: Data accuracy is ensured, reducing errors and improving decision-making quality.

Use Case: manageHiringProcess

Actors:

- HR Manager: Wants an efficient and fair hiring process to select best candidates.
- Hiring Manager: Needs to ensure that the chosen candidate have necessary skills and qualifications.
- IT Specialist: Ensures that the system used for managing candidate data and communications is functioning properly.
- Candidates: Want a transparent and timely process.

Preconditions:

- The job description, including skill, qualifications, and responsibilities, is already defined.
- Hiring budget has been established
- System for managing job applications and candidate records is operational.

Main Success Scenario:

- **HR Manager Posts Job Vacancy**:
  - The HR Manager creates and posts a job vacancy with the job description, qualifications, and other details.
- **Candidates Submit Applications**:
  - Candidates submit their applications.
- **System Records Applications**:
  - The system records all incoming applications and stores candidate information for future processing.

    - The system verifies that each application is complete and structured correctly.
- **HR Manager Screens Applications**:
    - The HR Manager screens applications for minimum qualifications and shortlists candidates based on criteria such as experience, education, and skills.
- **HR Manager Schedules Interviews**:
    - The HR Manager schedules interviews with shortlisted candidates.
    - Notifications are sent to candidates to confirm interview dates and times.
- **Hiring Manager Conducts Interviews**:
    - The Hiring Manager, along with other relevant interviewers, conducts the interviews and assesses candidates based on predefined criteria.
    - Interviewers record feedback for each candidate in the system.
- **System Generates Candidate Evaluation Report**:
    - The system compiles interview feedback and generates an evaluation report for each candidate.
    - HR Manager and Hiring Manager review the reports to determine the final candidate(s) for selection.
- **HR Manager Sends Offer to Selected Candidate**:
    - The HR Manager sends a job offer to the selected candidate(s), including details about salary, benefits, and start date.
    - Candidates are given a deadline to accept or decline the offer.
- **Candidate Accepts Job Offer**:
    - The selected candidate accepts the job offer.
    - The HR Manager records the candidate's acceptance in the system and finalizes hiring.
- **System Records Hiring Data**:
    - The system updates the employee records with the new hire's details, including personal information, position, salary, and start date.
    - The system notifies relevant departments (e.g., HR, Treasury) to prepare for the new hire.

Extensions:

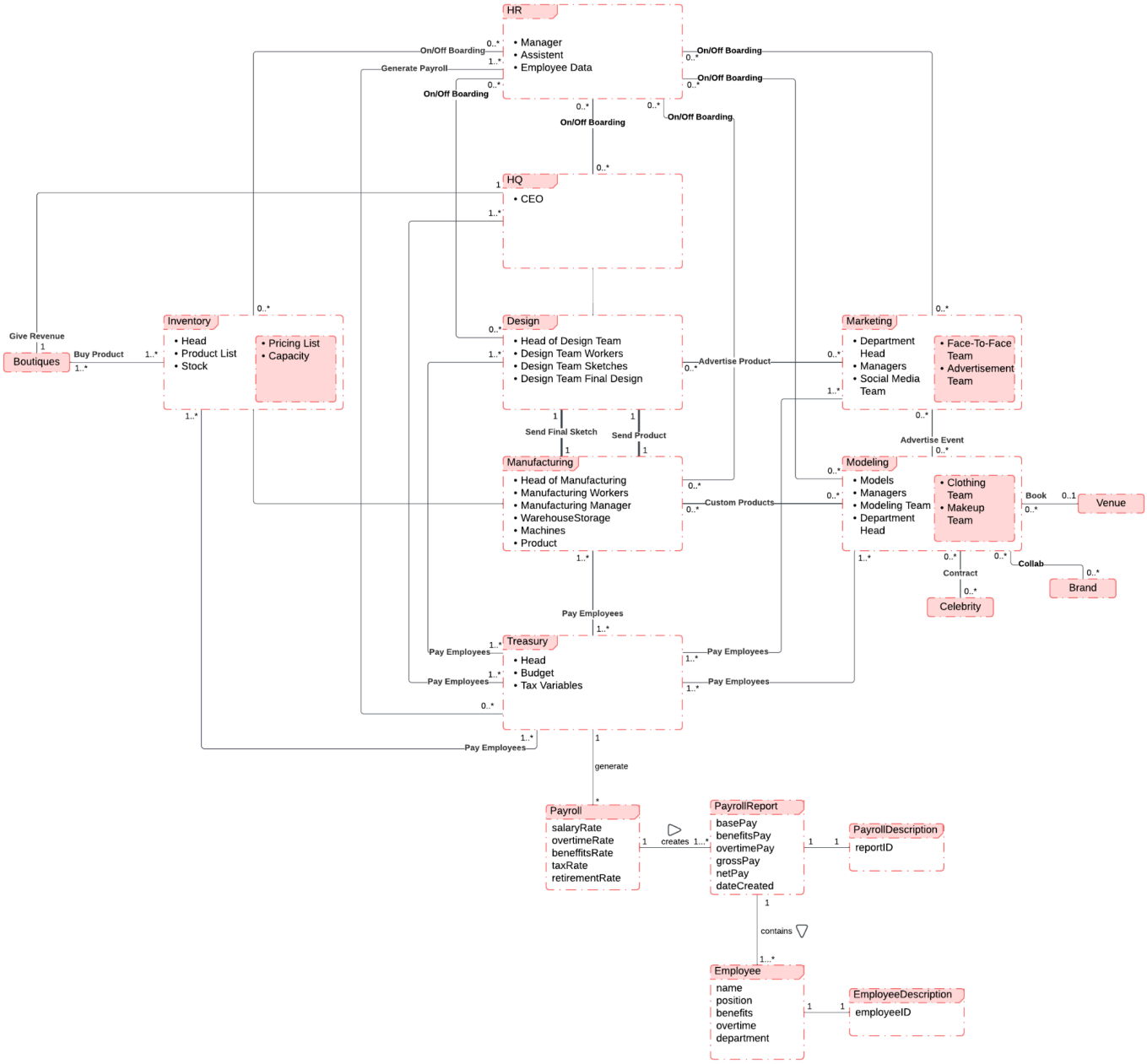
1. **3a. Incomplete Application**:
    - If a candidate submits an incomplete application:
        - System flags the issue and notifies the candidate to correct it.
        - Once the candidate provides the missing details, the system updates the application.
2. **5a. Candidate Unable to Attend Interview**:
    - If a candidate is unable to attend the scheduled interview:
        - System Allows Rescheduling.
        - The candidate can choose a new interview slot from available dates.
        - The HR Manager confirms the rescheduled interview.
3. **7a. No Suitable Candidates Found**:
    - If no suitable candidates are found after interviews:

        - HR Manager reviews the candidate pool again or decides to repost the job vacancy.
    - The hiring process resumes with new candidates.
4. **8a. Candidate Declines Job Offer**:
    - If the selected candidate declines the job offer:
        - HR Manager selects the next best candidate from the evaluation report.
        - The system sends a job offer to the next candidate.
        - The process repeats until a candidate accepts the offer.
5. **10a. Onboarding Requirements Not Met**:
    - If onboarding requirements (e.g., reference checks, documentation) are not completed:
        - System flags missing onboarding steps.
        - HR Manager contacts the candidate to provide the necessary documents.
        - The hiring process resumes once all onboarding requirements are met.

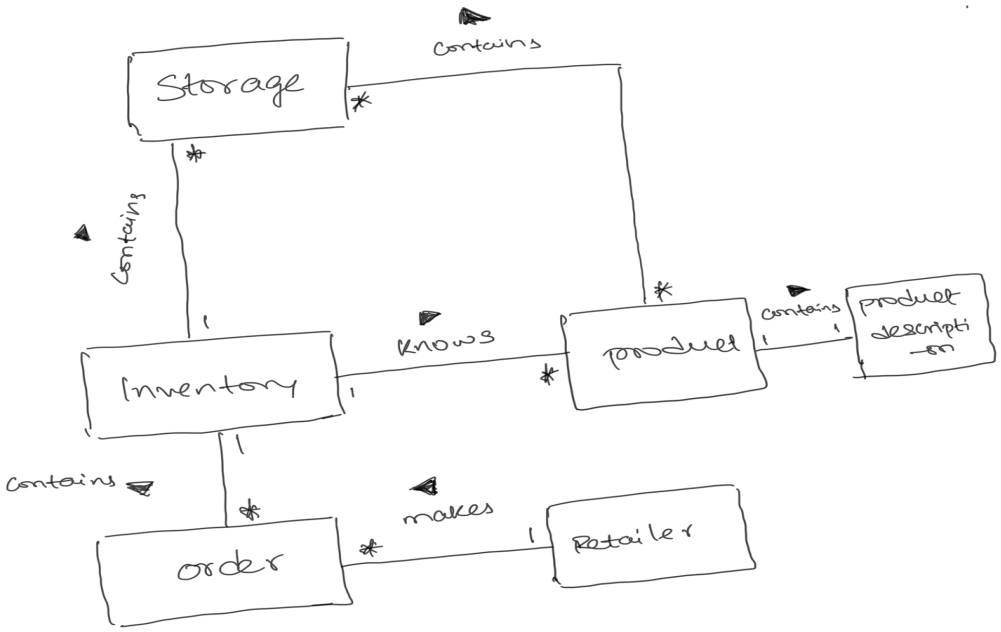
Postconditions:

- **Candidate Hired**: A suitable candidate is hired and employee records are updated accordingly.
- **Records Updated**: All relevant information, such as personal details, salary, and start date, is recorded in the employee management system.
- **Departments Notified**: Relevant departments are notified to prepare for the onboarding of the new employee.

Domain Model



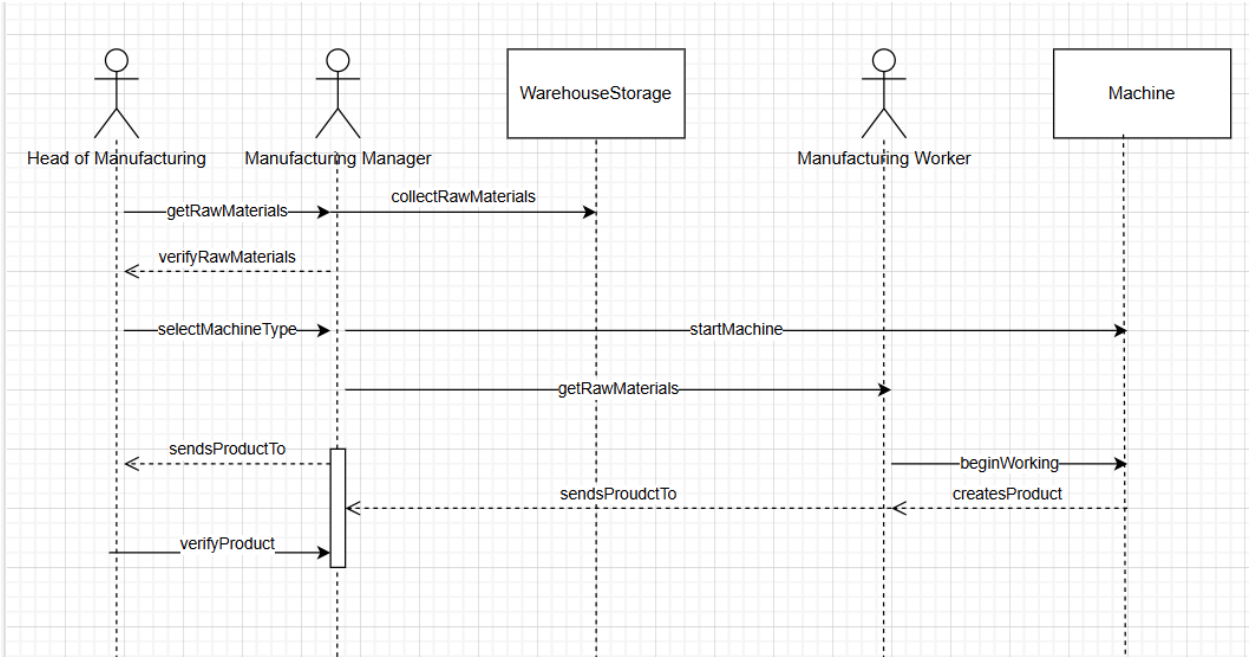
Inventory Extended Domain model



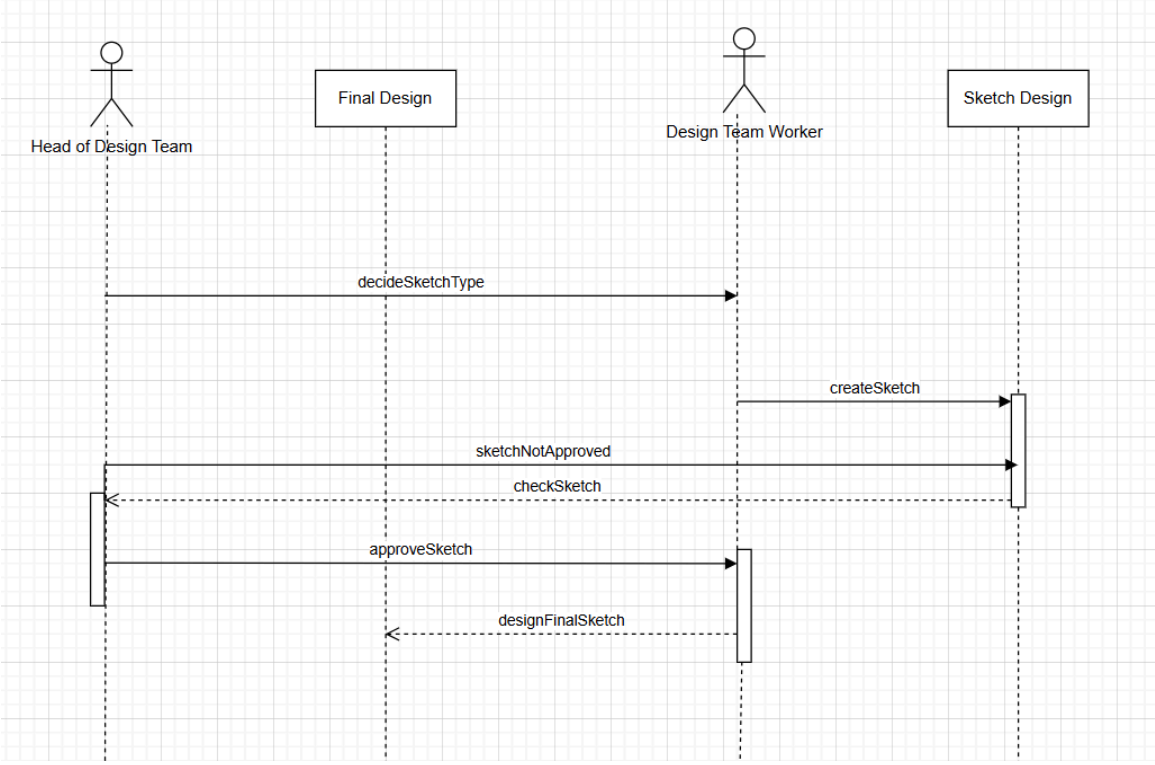
Interaction Diagrams

1. Manufacturing Department

CreateProduct



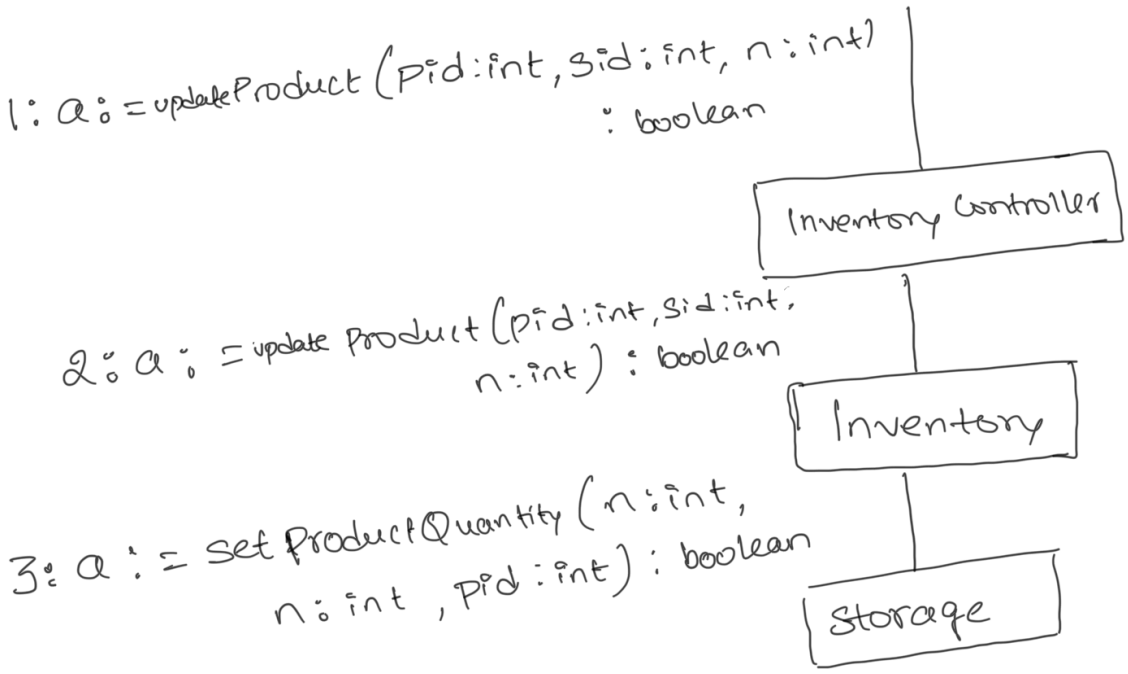
DesignProduct



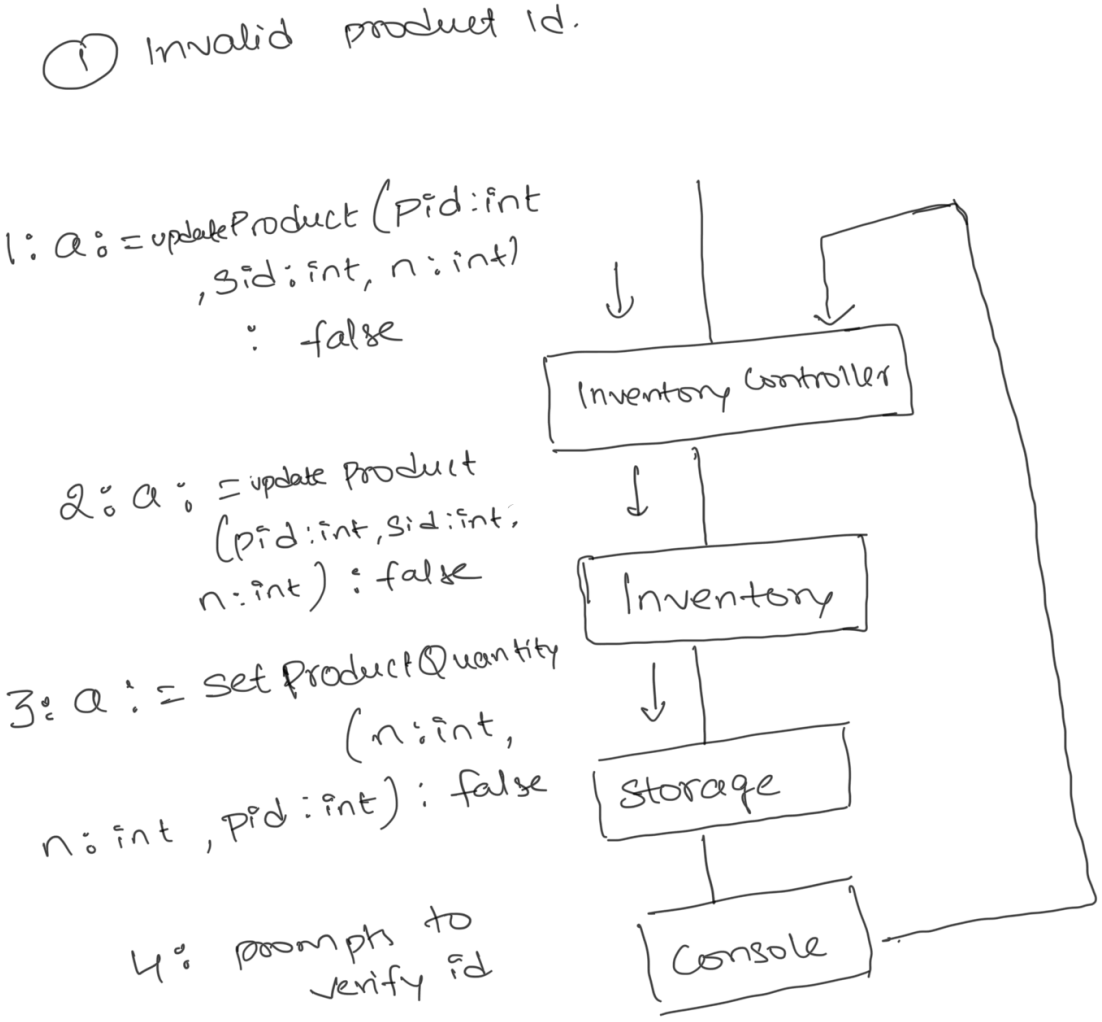
2. Inventory

Use Case: Update Product

Main success scenario



Extension: Invalid product Id



Extension: new product type/missing product id

② Missing/new product

1: a := updateProduct (pid: int
, sid: int, n: int)
: false

2: a := updateProduct
(pid: int, sid: int,
n: int) : false

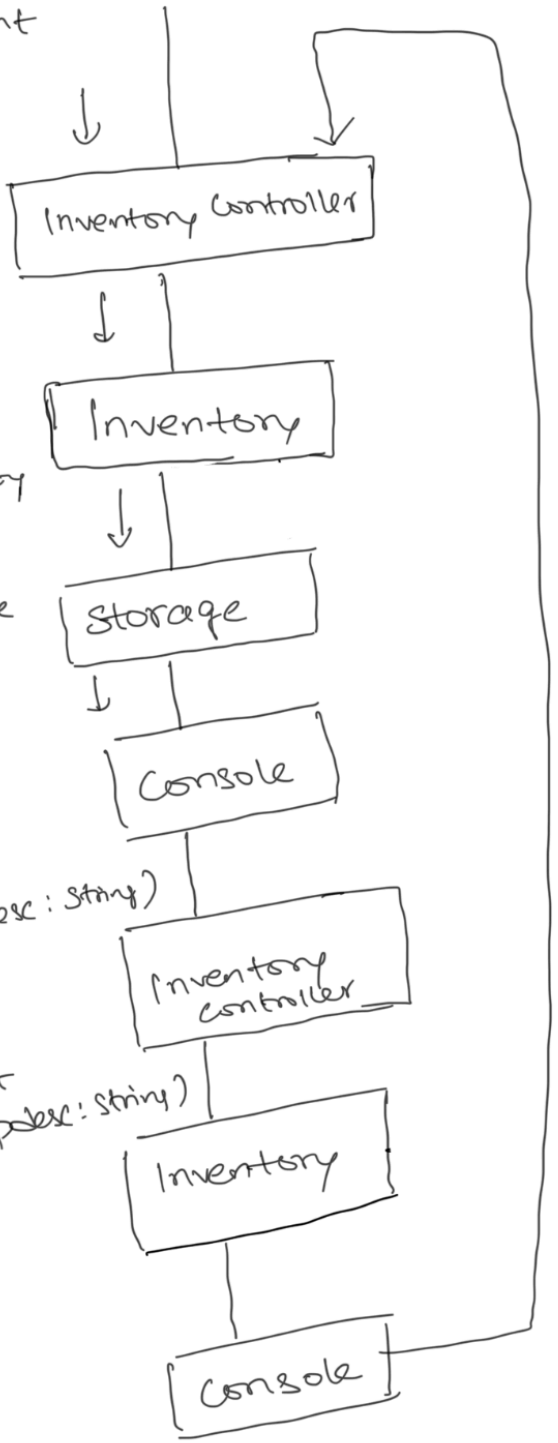
3: a := setProductQuantity
(n: int,
n: int, pid: int) : false

4: prompts to
verify id

5: b := registerProduct
(pname: string, pdesc: string)
: int

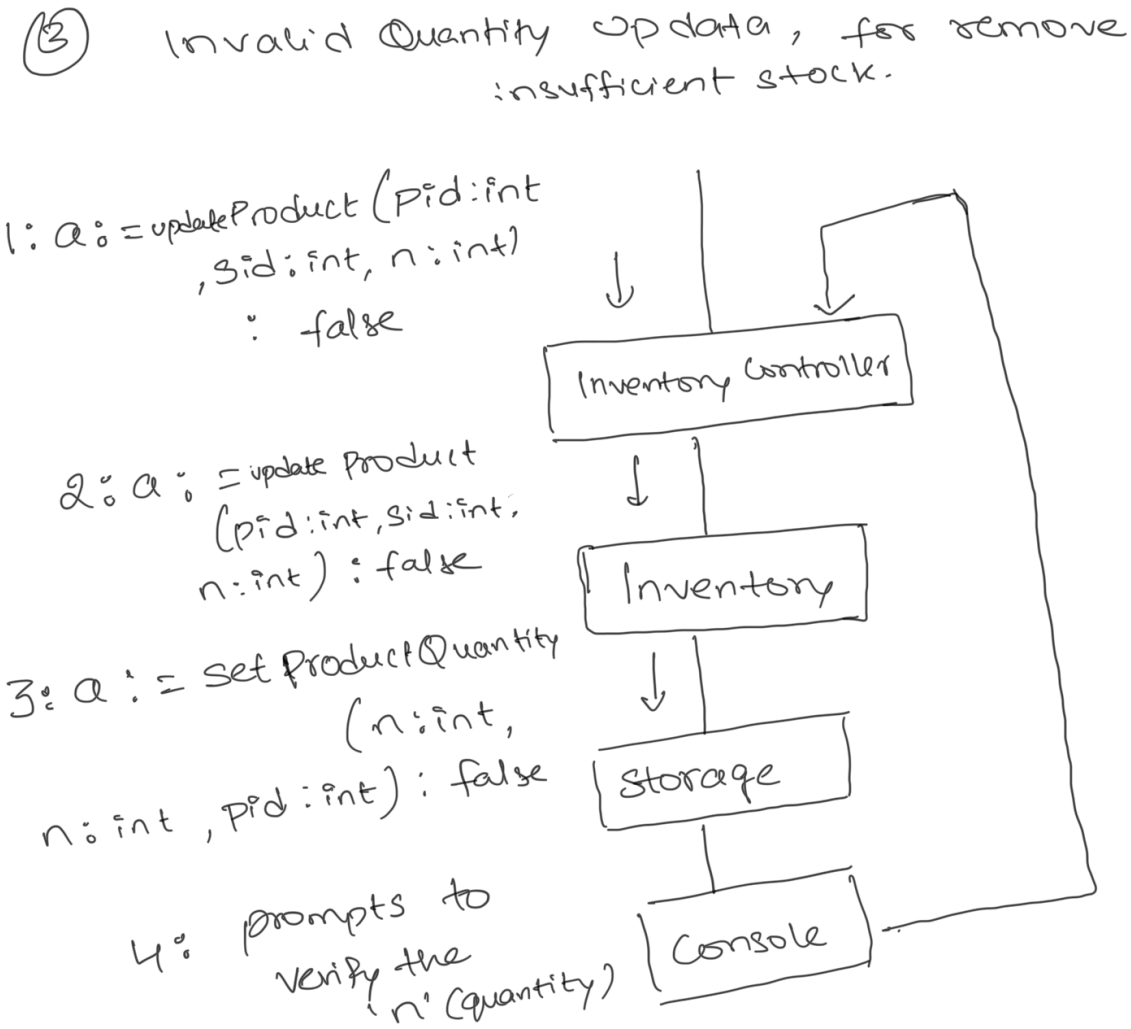
6: b.2 registerProduct
(pname: string, pdesc: string)
: int

7: prints b.

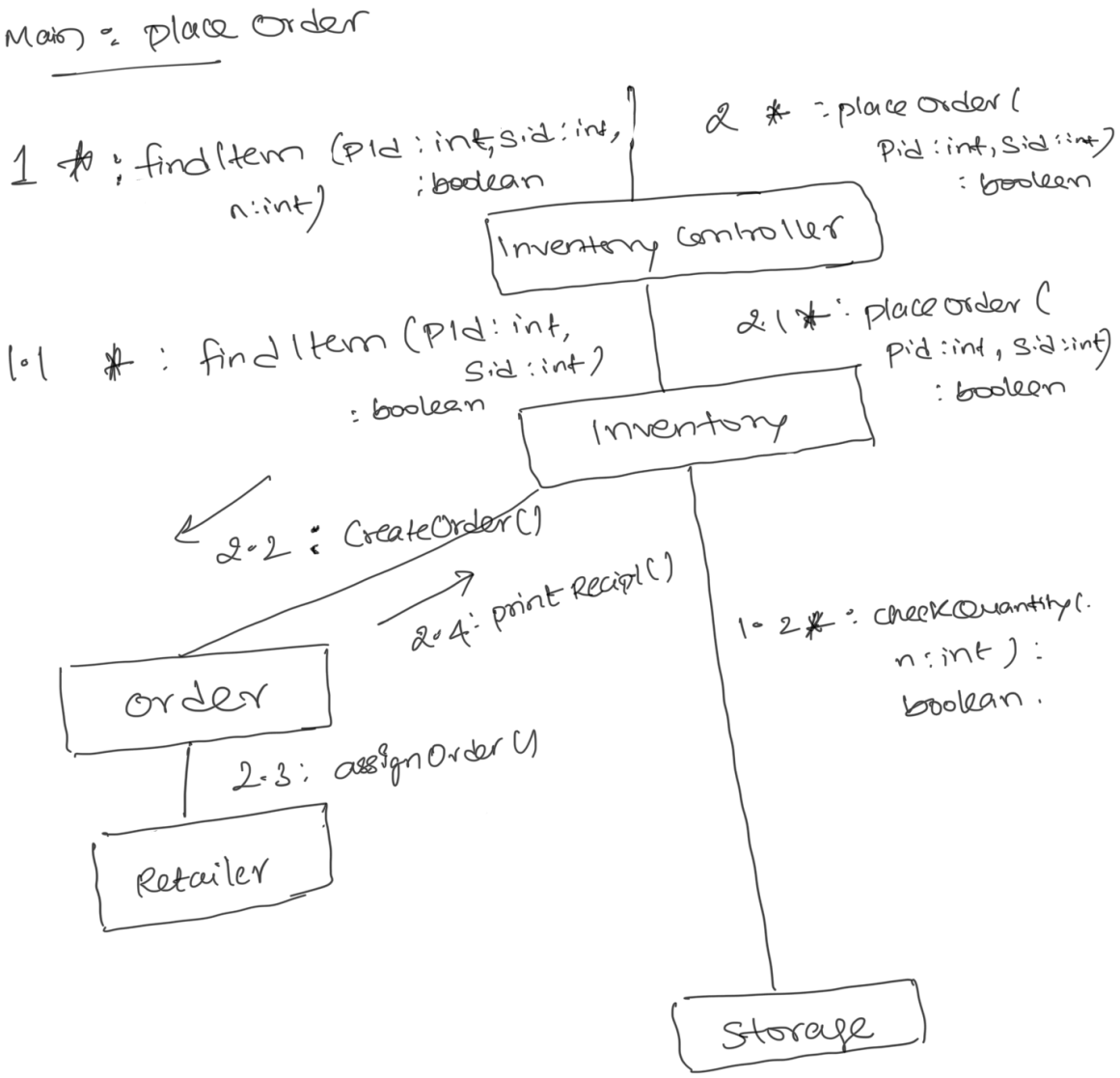


* perform the update again after
generating id
(or) registering product.

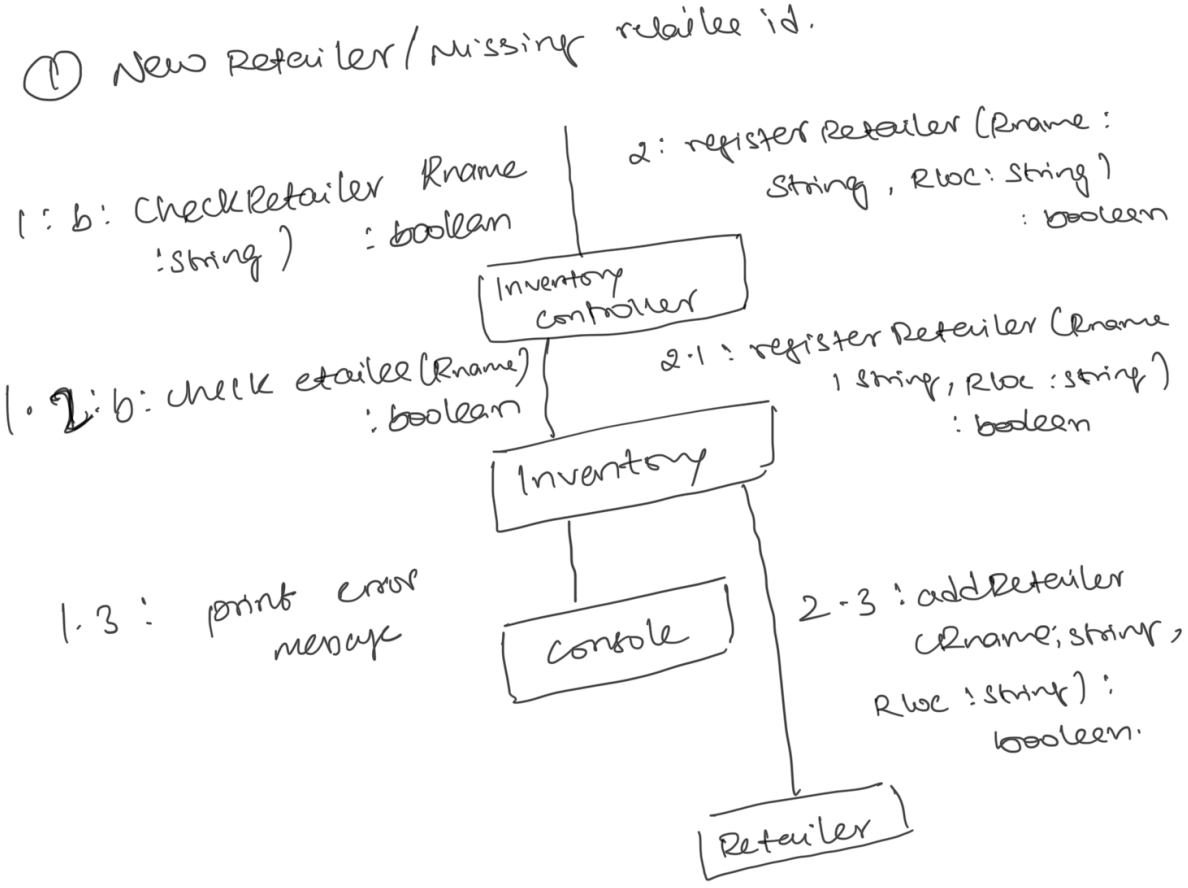
Extension: Exceeded quantity update for remove



Use case: Place Order
Main success scenario

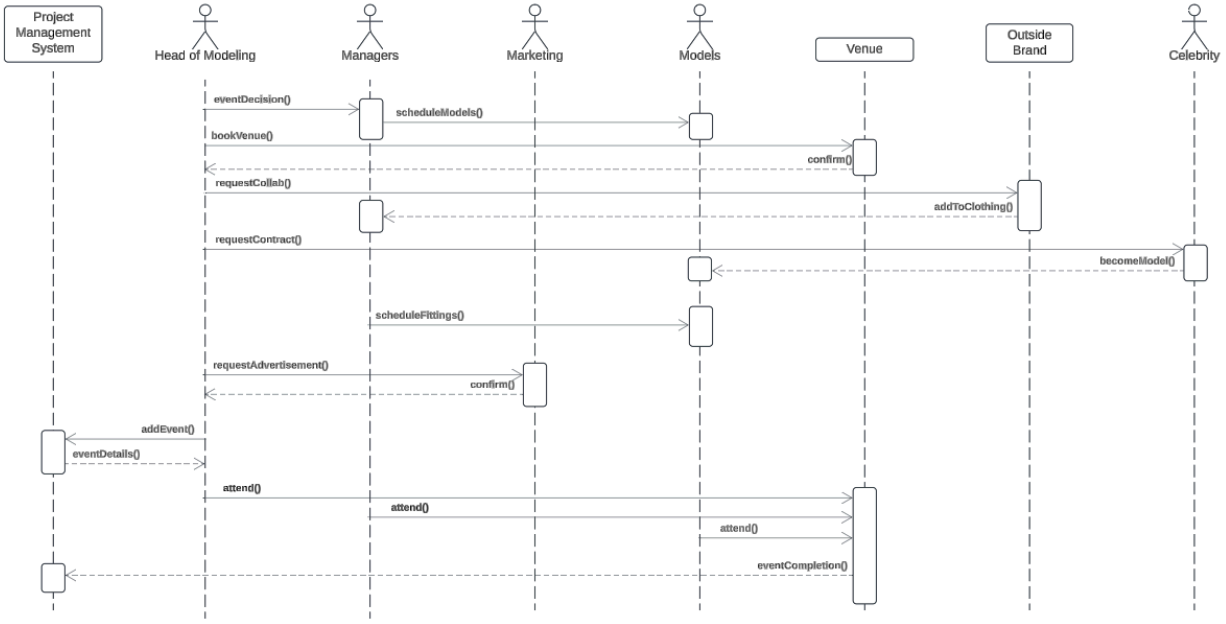


Extension: new retailer / missing retailer id

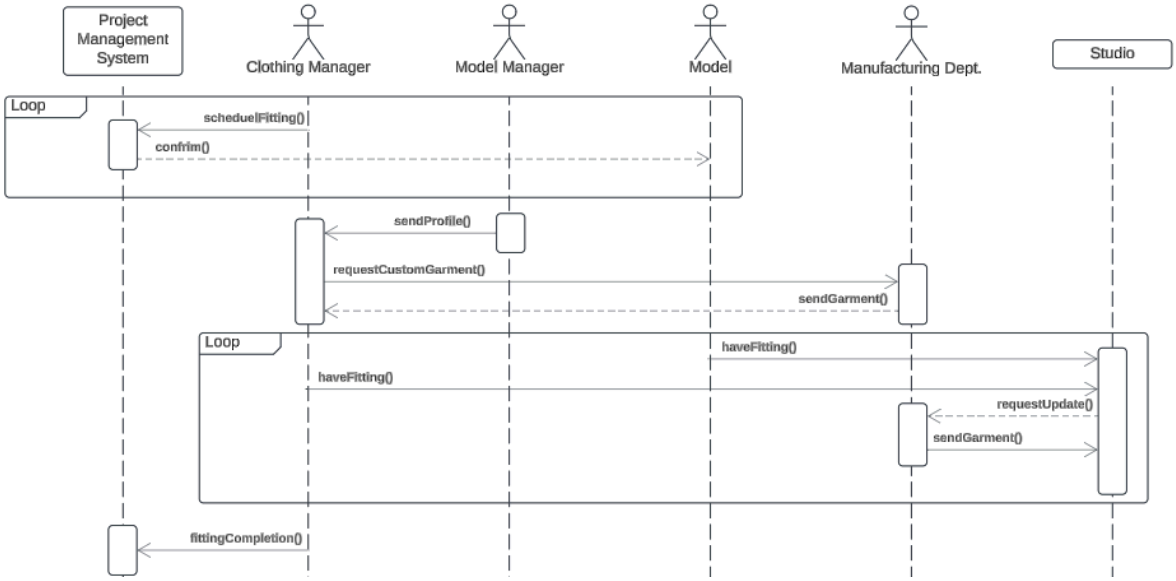


3. Modeling Department

a. haveEvent

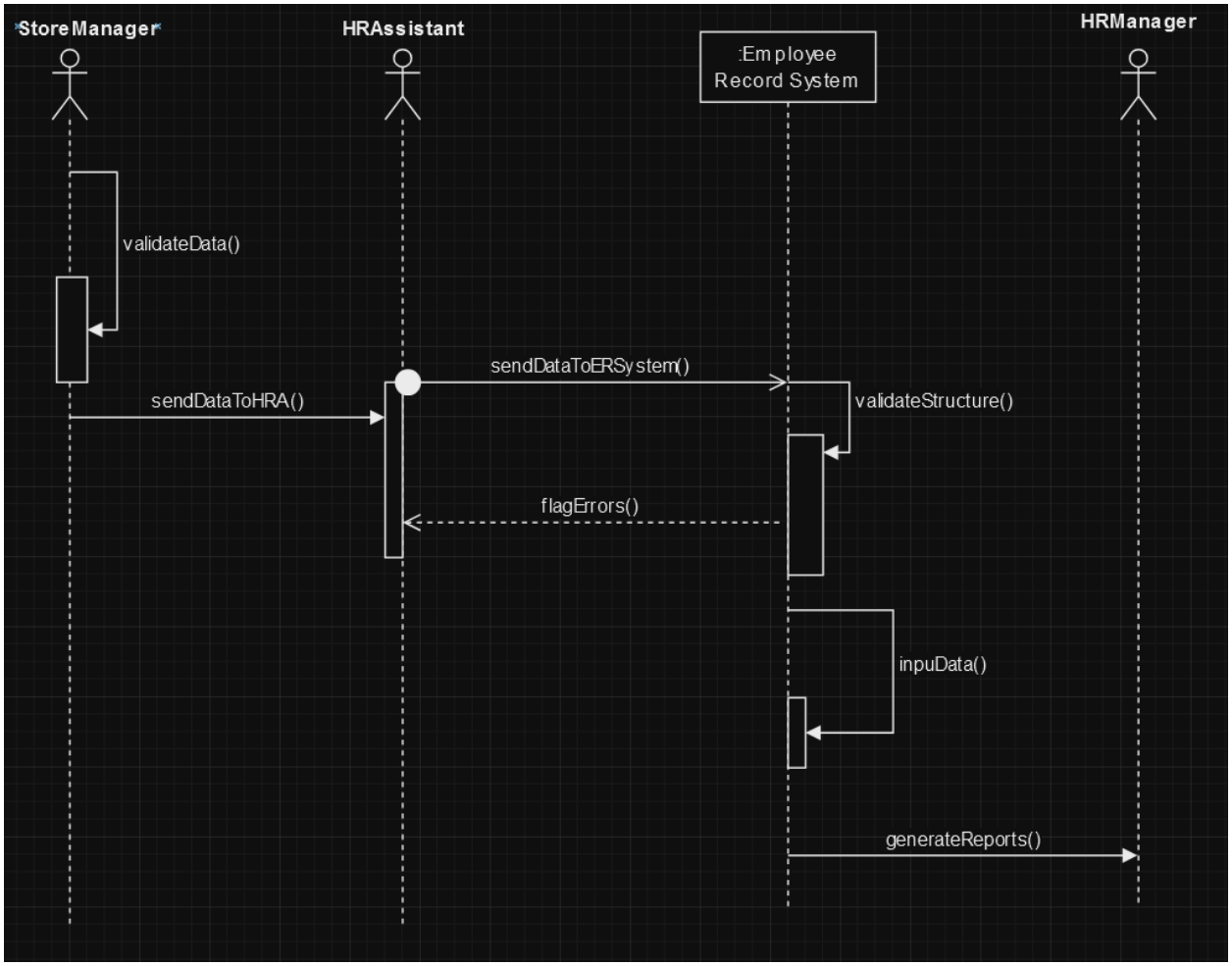


b. scheduleFitting

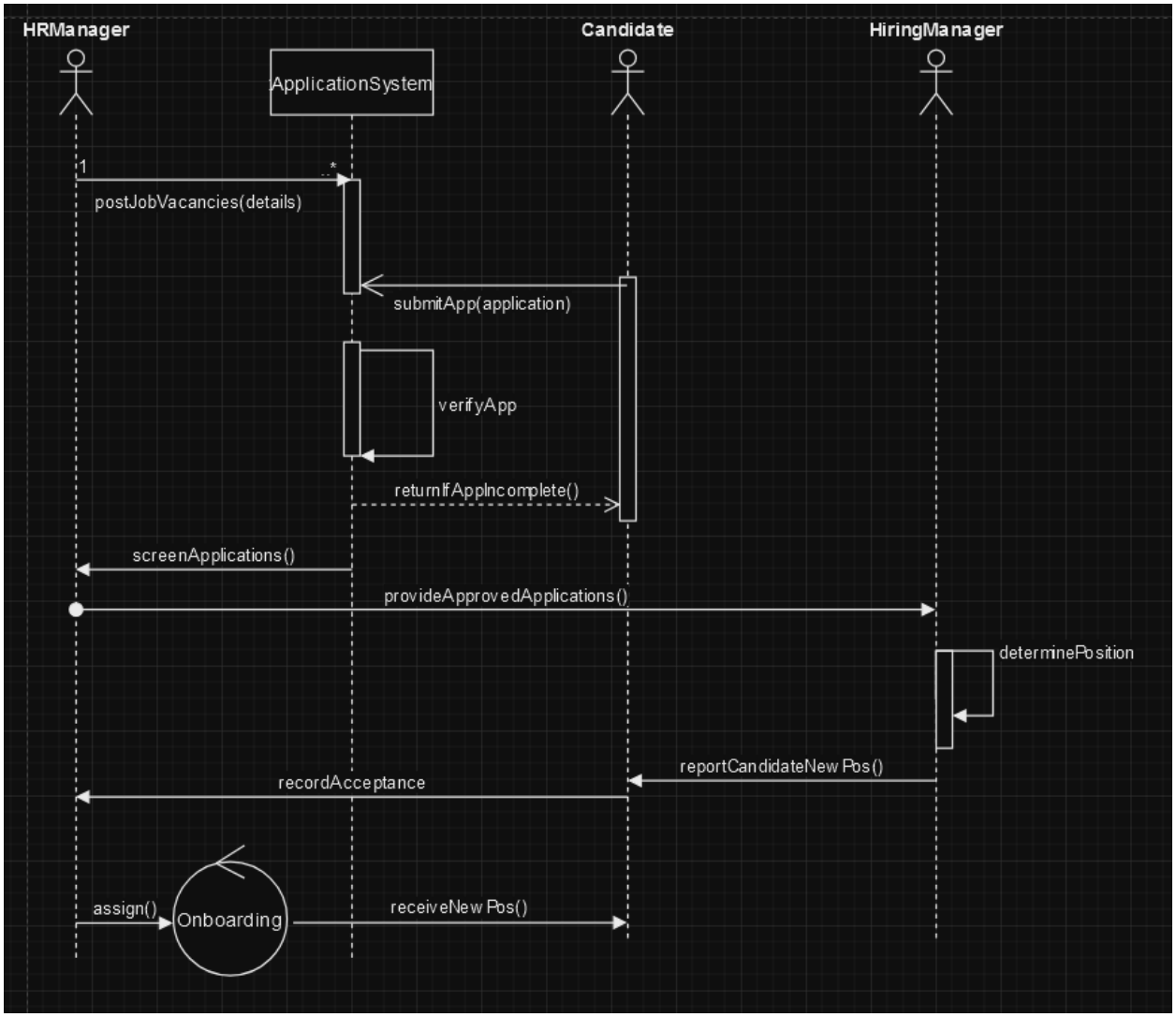


4. HR

a. Use Case: manageEmployeeRecords

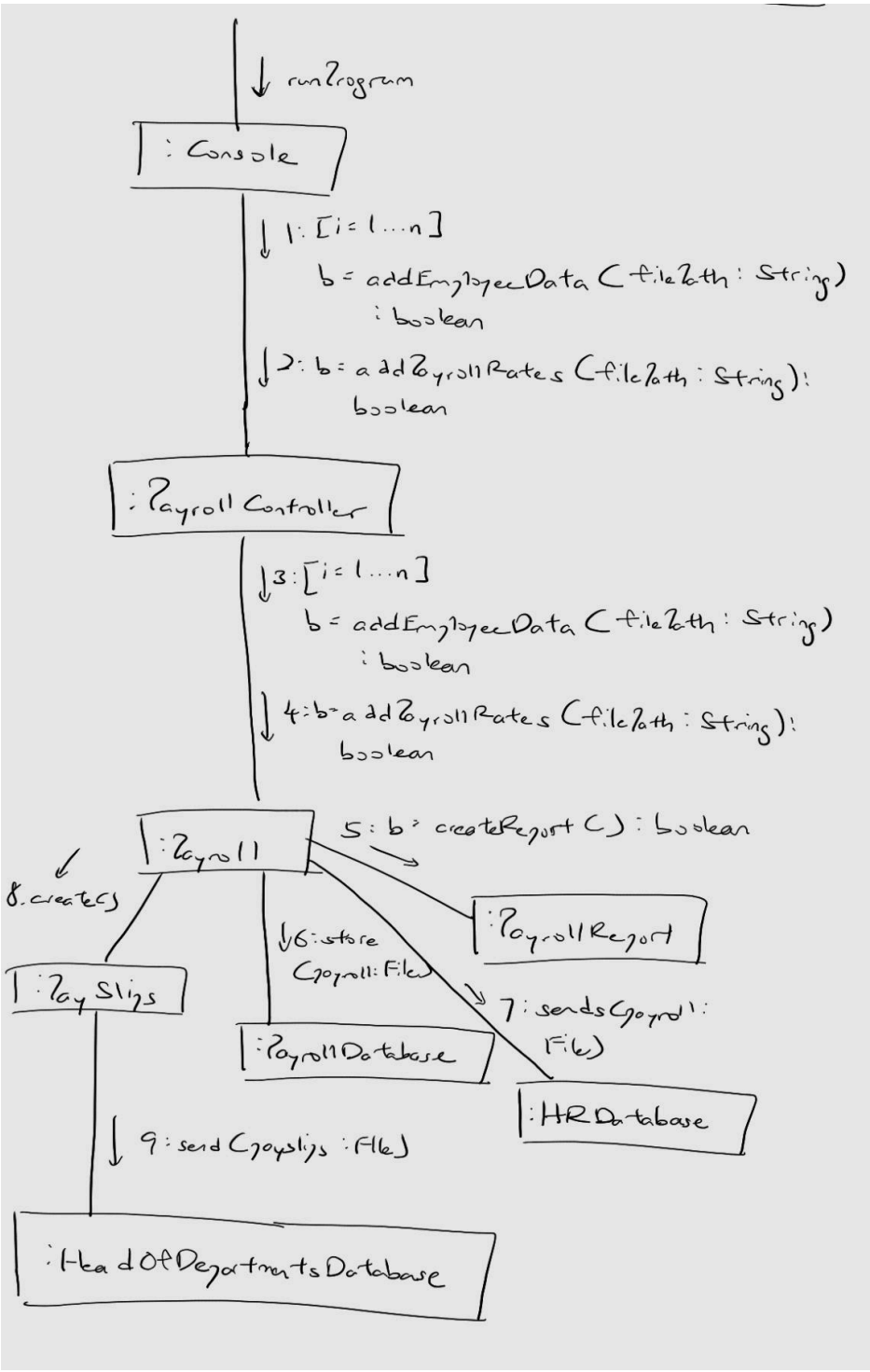


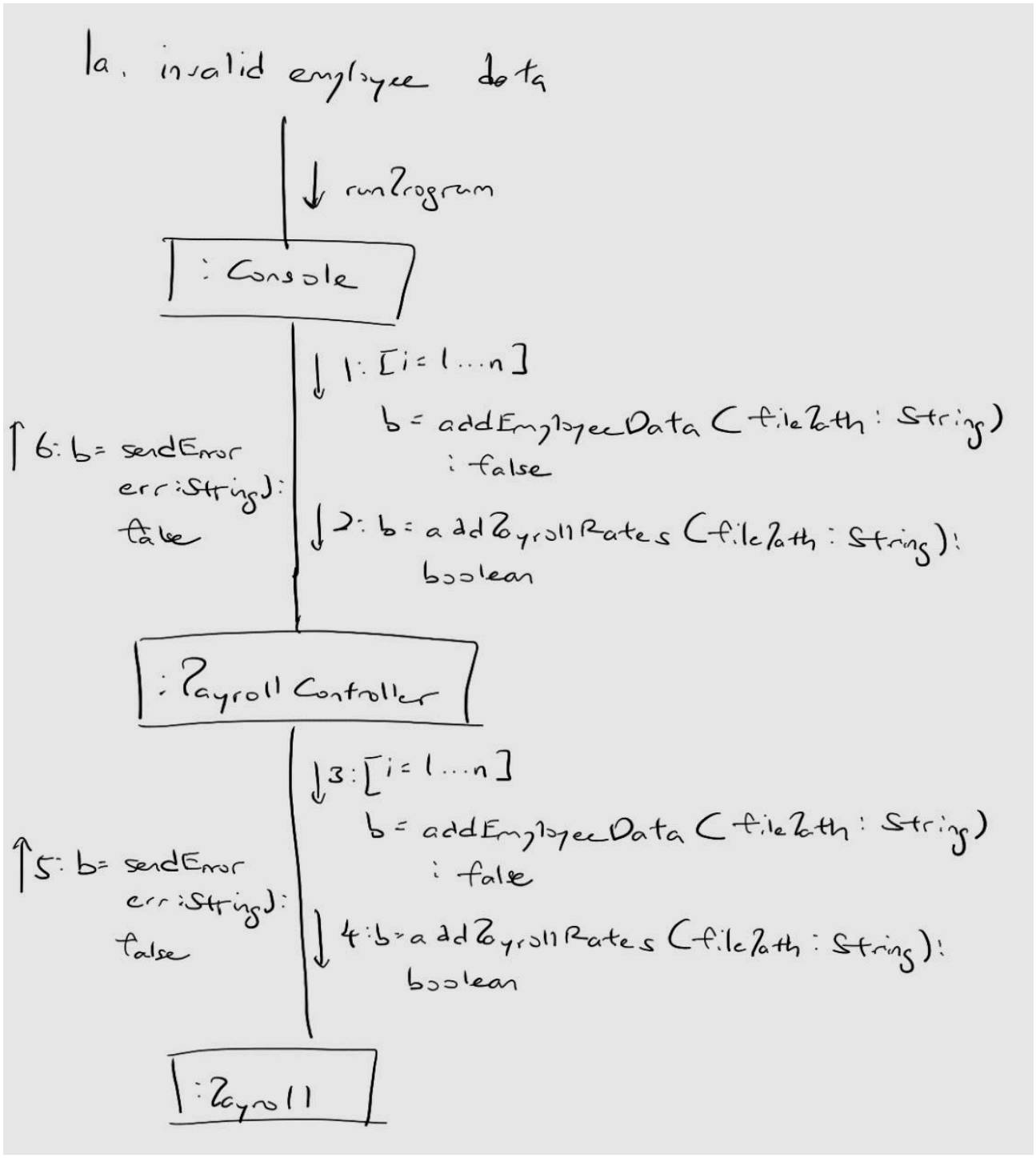
b. Use Case: manageHiringProcess

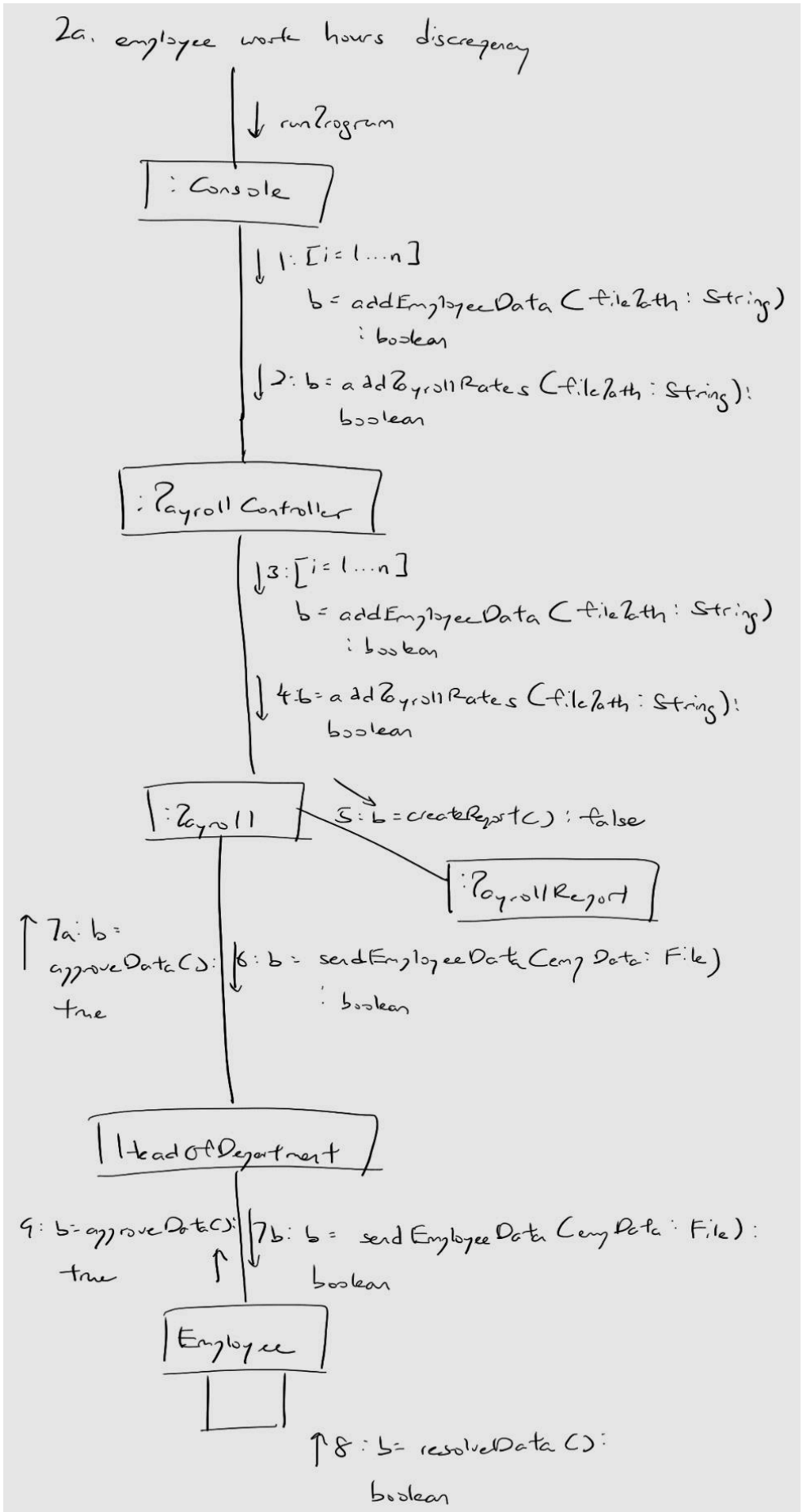

HRManager
ApplicationSystem
Candidate
HiringManager
1
*
postJobVacancies(details)
submitApp(application)
verifyApp
returnIfAppIncomplete()
screenApplications()
provideApprovedApplications()
determinePosition
reportCandidateNewPos()
recordAcceptance
assign()
Onboarding
receiveNewPos()

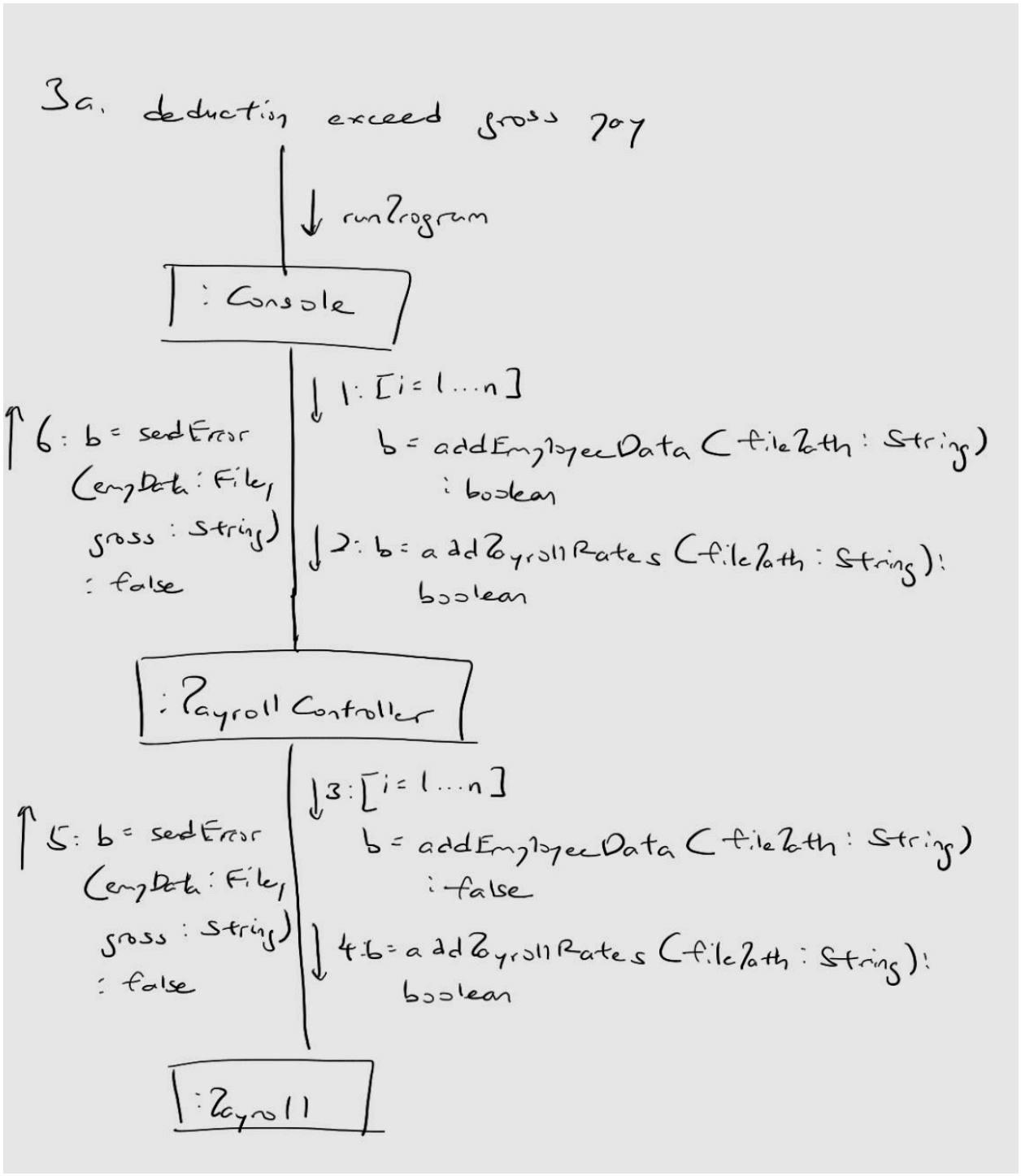
5. Treasury

Use Case: processPayroll

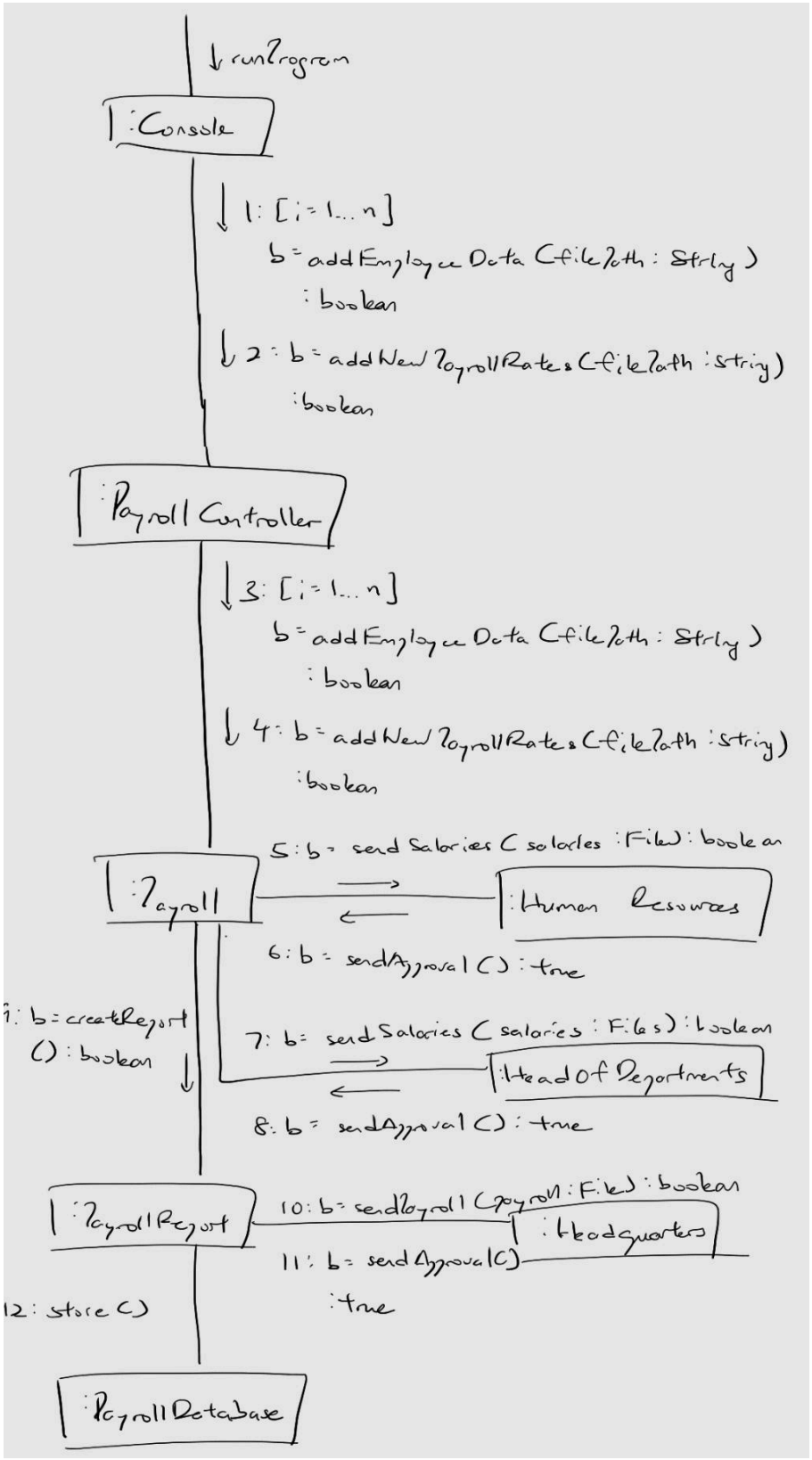


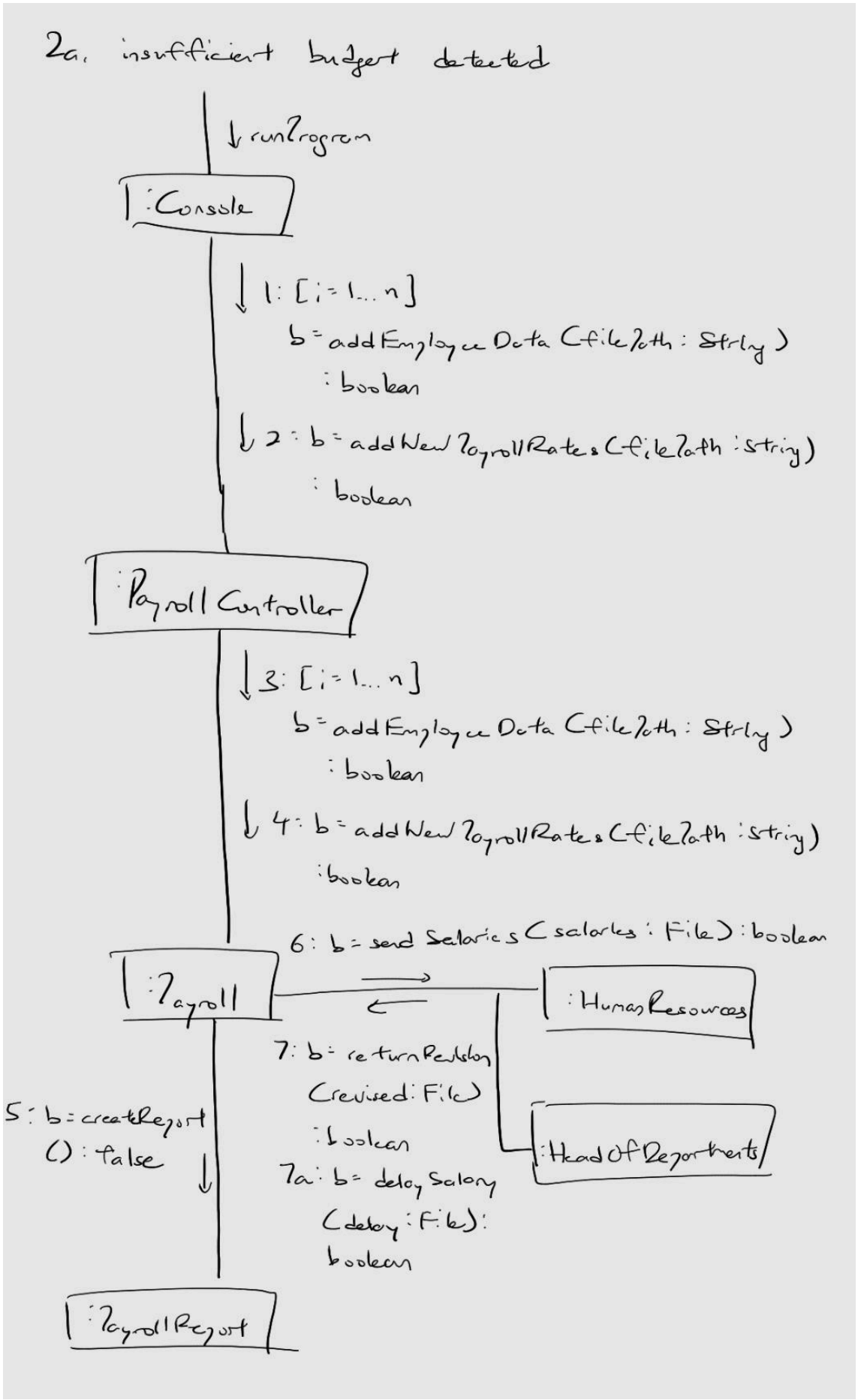

1a. invalid employee data
: runProgram
: Console
1: [i = 1...n]
b = addEmployeeData (filePath : String) : false
2: b = addPayrollRates (filePath : String) : boolean
6: b = sendError (err : String) : false
: PayrollController
3: [i = 1...n]
b = addEmployeeData (filePath : String) : false
4: b = addPayrollRates (filePath : String) : boolean
5: b = sendError (err : String) : false
: Payroll

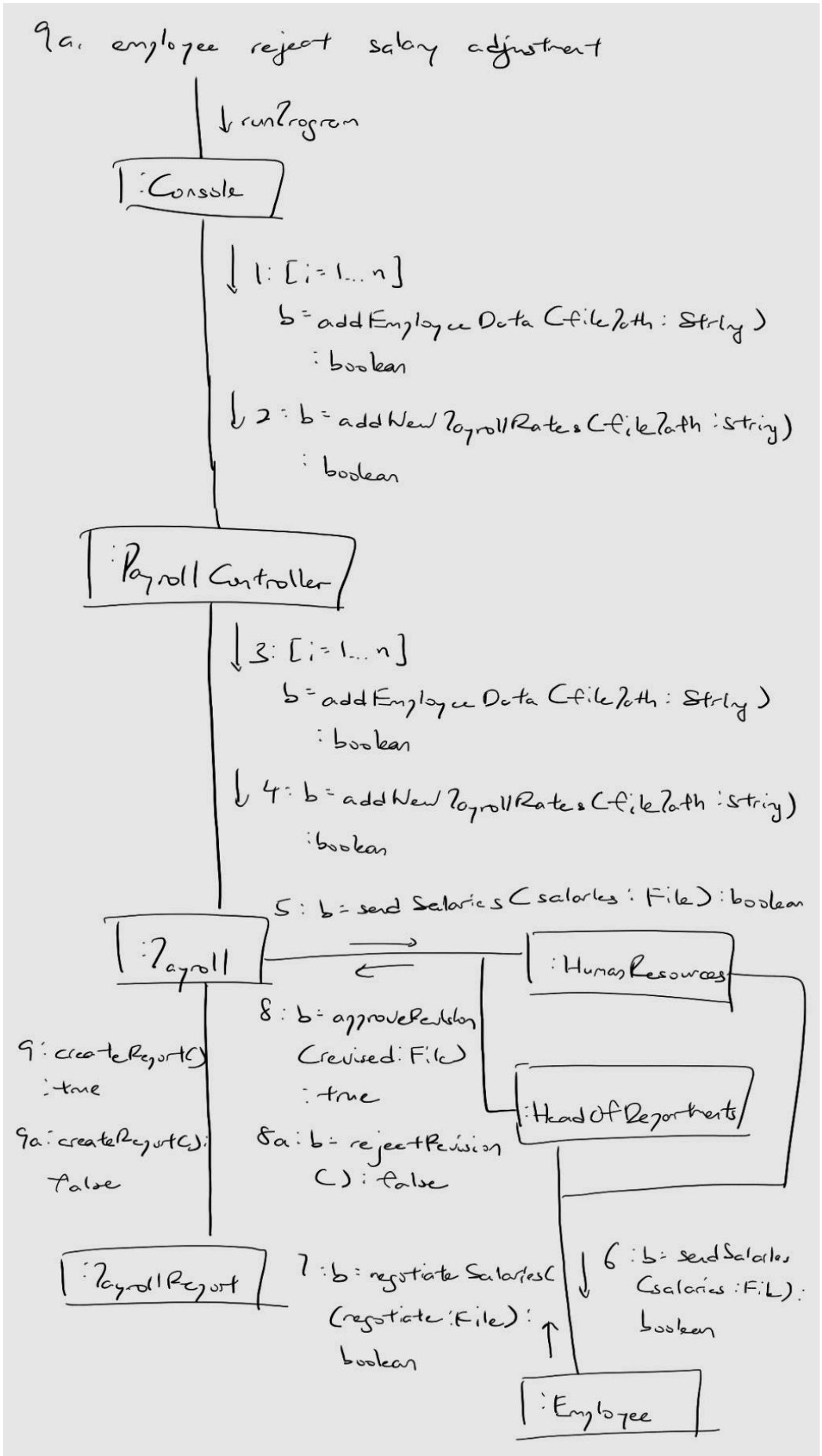

2a. employee work hours discrepancy
runProgram
: Console
1: [i = 1...n]
b = addEmployeeData (filePath : String) : boolean
2: b = addPayrollRates (filePath : String) : boolean
: PayrollController
3: [i = 1...n]
b = addEmployeeData (filePath : String) : boolean
4: b = addPayrollRates (filePath : String) : boolean
: Payroll
5: b = createReport() : false
: PayrollReport
7a: b = approveData(): true
6: b = sendEmployeeData(empData: File) : boolean
HeadOfDepartment
9: b = approveData(): true
7b: b = sendEmployeeData(empData: File) : boolean
Employee
8: b = resolveData() : boolean


3a. deduction exceed gross pay
↓ runProgram
: Console
↓ 1: [i = 1...n]
b = addEmployeeData (filePath : String) : boolean
↓ 2: b = addPayrollRates (filePath : String) : boolean
↑ 6: b = sendError (empData : File, gross : String) : false
: PayrollController
↓ 3: [i = 1...n]
b = addEmployeeData (filePath : String) : false
↓ 4: b = addPayrollRates (filePath : String) : boolean
↑ 5: b = sendError (empData : File, gross : String) : false
: Payroll

Use Case: adjustSalaries

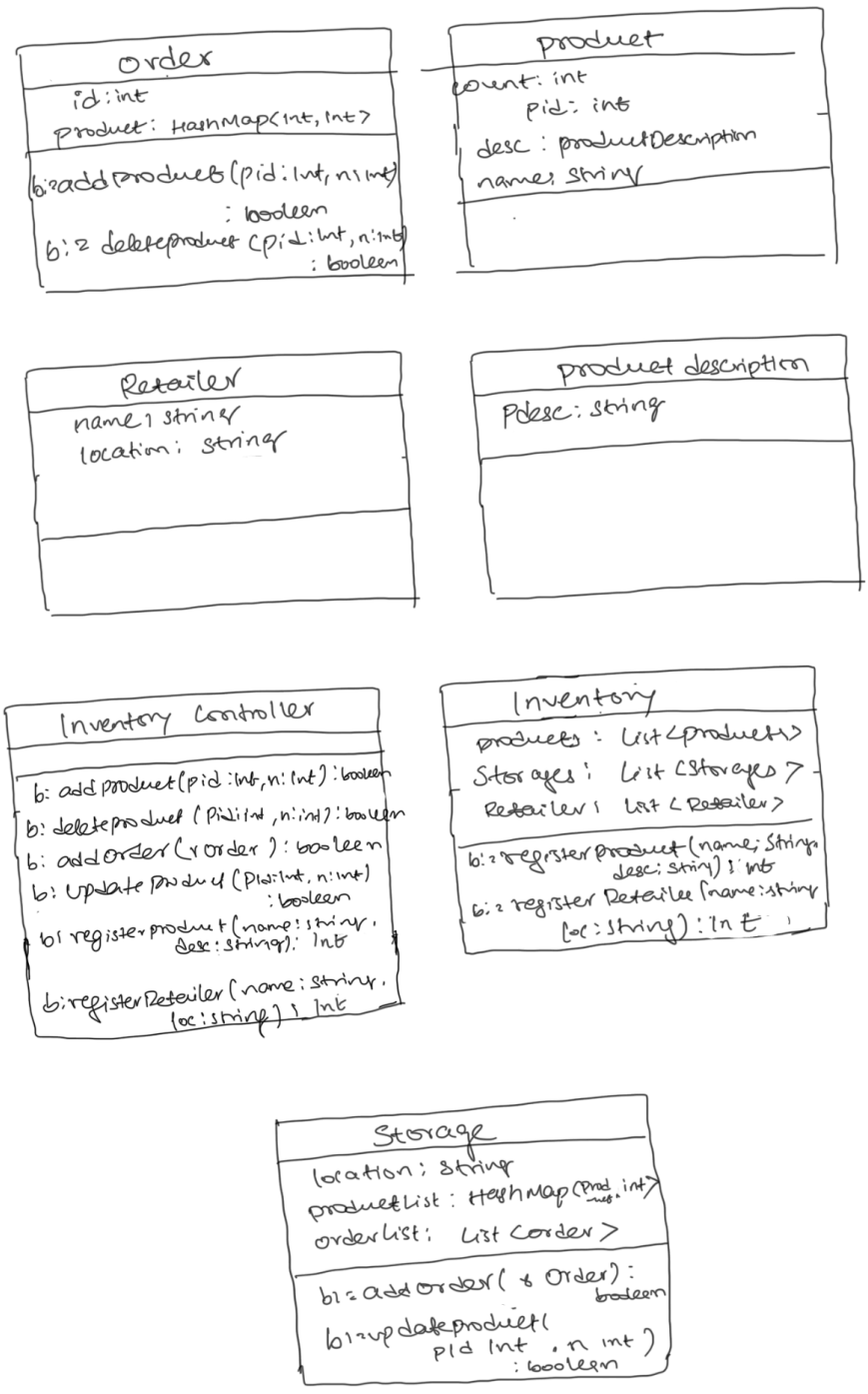



2a. insufficient budget detected
runProgram
:Console
1: [i = 1...n]
b = addEmployeeData(filePath: String)
: boolean
2: b = addNewPayrollRates(filePath: String)
: boolean
:PayrollController
3: [i = 1...n]
b = addEmployeeData(filePath: String)
: boolean
4: b = addNewPayrollRates(filePath: String)
: boolean
6: b = sendSalaries(salaries: File): boolean
:Payroll
:HumanResources
7: b = returnRevision
(revised: File)
: boolean
7a: b = delaySalary
(delay: File):
boolean
:HeadOfDepartments
5: b = createReport
(): false
:PayrollReport

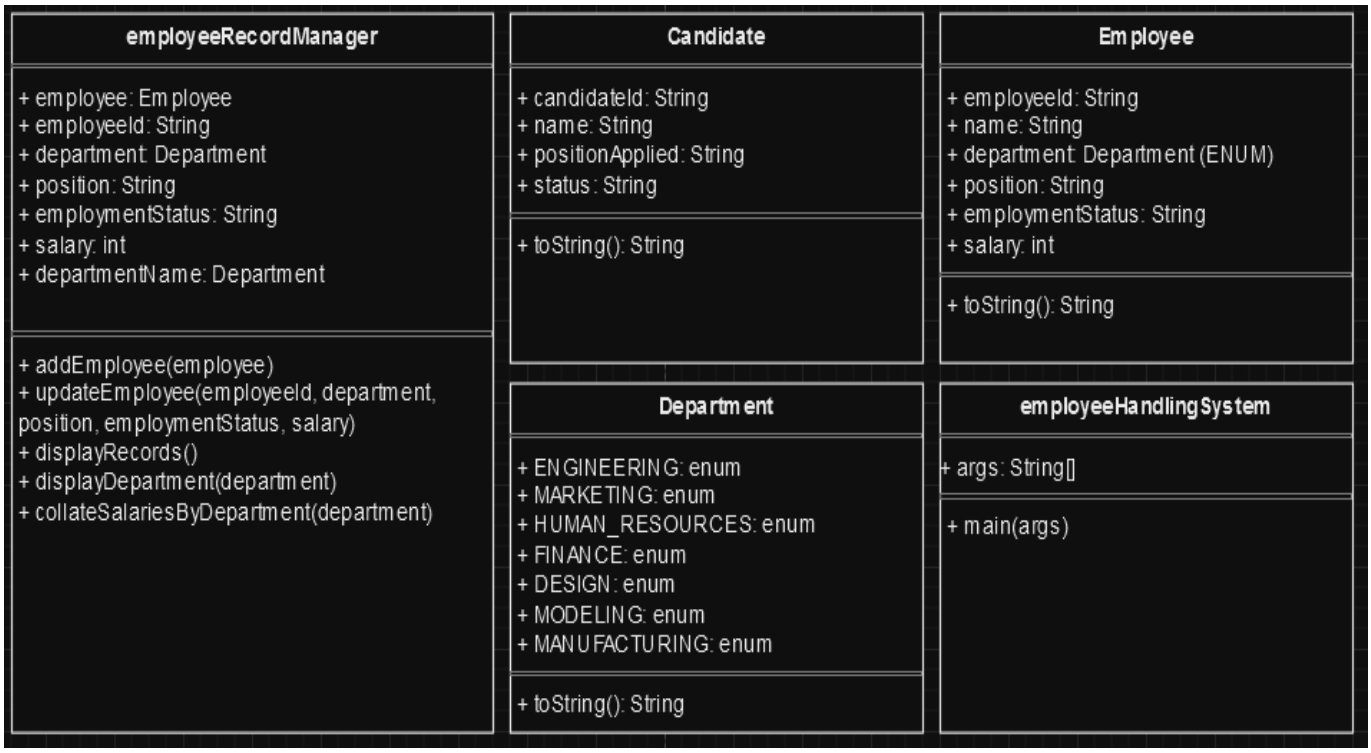

9a. employee reject salary adjustment
runProgram
:Console
1: [i=1...n]
b = addEmployeeData(filePath: String)
: boolean
2: b = addNewPayrollRates(filePath: String)
: boolean
:PayrollController
3: [i=1...n]
b = addEmployeeData(filePath: String)
: boolean
4: b = addNewPayrollRates(filePath: String)
: boolean
5: b = sendSalaries(salaries: File): boolean
:Payroll
:HumanResources
8: b = approveRevision
(revised: File)
: true
8a: b = rejectRevision
(): false
:HeadOfReportents
9: createReport()
: true
9a: createReport():
false
:PayrollReport
7: b = negotiateSalaries(
negotiate: File):
boolean
6: b = sendSalaries
(salaries: File):
boolean
:Employee

Class Diagrams

Inventory:



HR:


employeeRecordManager
+ employee: Employee
+ employeeId: String
+ department: Department
+ position: String
+ employmentStatus: String
+ salary: int
+ departmentName: Department
+ addEmployee(employee)
+ updateEmployee(employeeId, department, position, employmentStatus, salary)
+ displayRecords()
+ displayDepartment(department)
+ collateSalariesByDepartment(department)
Candidate
+ candidateId: String
+ name: String
+ positionApplied: String
+ status: String
+ toString(): String
Department
+ ENGINEERING: enum
+ MARKETING: enum
+ HUMAN_RESOURCES: enum
+ FINANCE: enum
+ DESIGN: enum
+ MODELING: enum
+ MANUFACTURING: enum
+ toString(): String
Employee
+ employeeId: String
+ name: String
+ department: Department (ENUM)
+ position: String
+ employmentStatus: String
+ salary: int
+ toString(): String
employeeHandlingSystem
+ args: String[]
+ main(args)

Design

DesignSketch
sketchID : String sketchDescrioption: String
getSketchID() : string getSketchDescription() : string

DesignTeamWorkers
Workers : String Sketches: List<DesignSketch>
getSketches() : List<DesingSketch> getWorker(): String

DesignType
typeName: ENUM Description: ENUM

FinalDesign
designName: String Colors: List<String> rawMaterials: List<String> Sizes: List<String> Quantity: Integer
Void setColor(List<String> colors): List<String> Void setRawMaterials(List<String> rawMaterials); List<String> Void setSizes(List<String> sizes); List<String> Void setquantity(int quantities); int List<String> getColors(); List<String> List<String> getRawMaterials(); List<String> List<String> getSizes(); List<String>

HeadOfDesignTeam
Sketches: List<DesignSketches> FinalDesign: FinalDesign
approveFinalDesign() : FInalDesign isApproved: boolean

Manufacturing

HeadOfManufacturing
Manager: ManufacturingManager Machine: Machines Warehouse: WarehouseStorage Design: Final Design
reviewDesign(design): void getRawMaterials(List<String> materials, List<Integer> quantity) :String reviewRawMaterials(List<String> materials, List<Integer> quantity) : String selectMachineType(): Machines approveProduct():boolean

Machines
PowerOn() :boolean PowerOff() :boolean createsProduct() : FinalProduct

ManufacturingWorkers
receivesRawMaterials(List<String> materials, List<Integer> quantity) :String beginWorkingOnMachines() :boolean

createProduct() :FinalProduct sendProductToManage() : FinalProduct

ManufacturingManager
Warehouse: WarehouseStorage headofMan: HeadOfManufacturing Machine: MachineOperations
getRawMaterials(String materials, int quantity) :String decrementRawMaterialCount(String material, int quantity):int startMachine(): boolean stopMachine(): boolean startProduction(): boolean stop Production(): boolean

WarehouseStorage
receiveRawMaterials(String material, int quantity) :String decementMaterialCount(String material, int quantity): Int

Treasury:

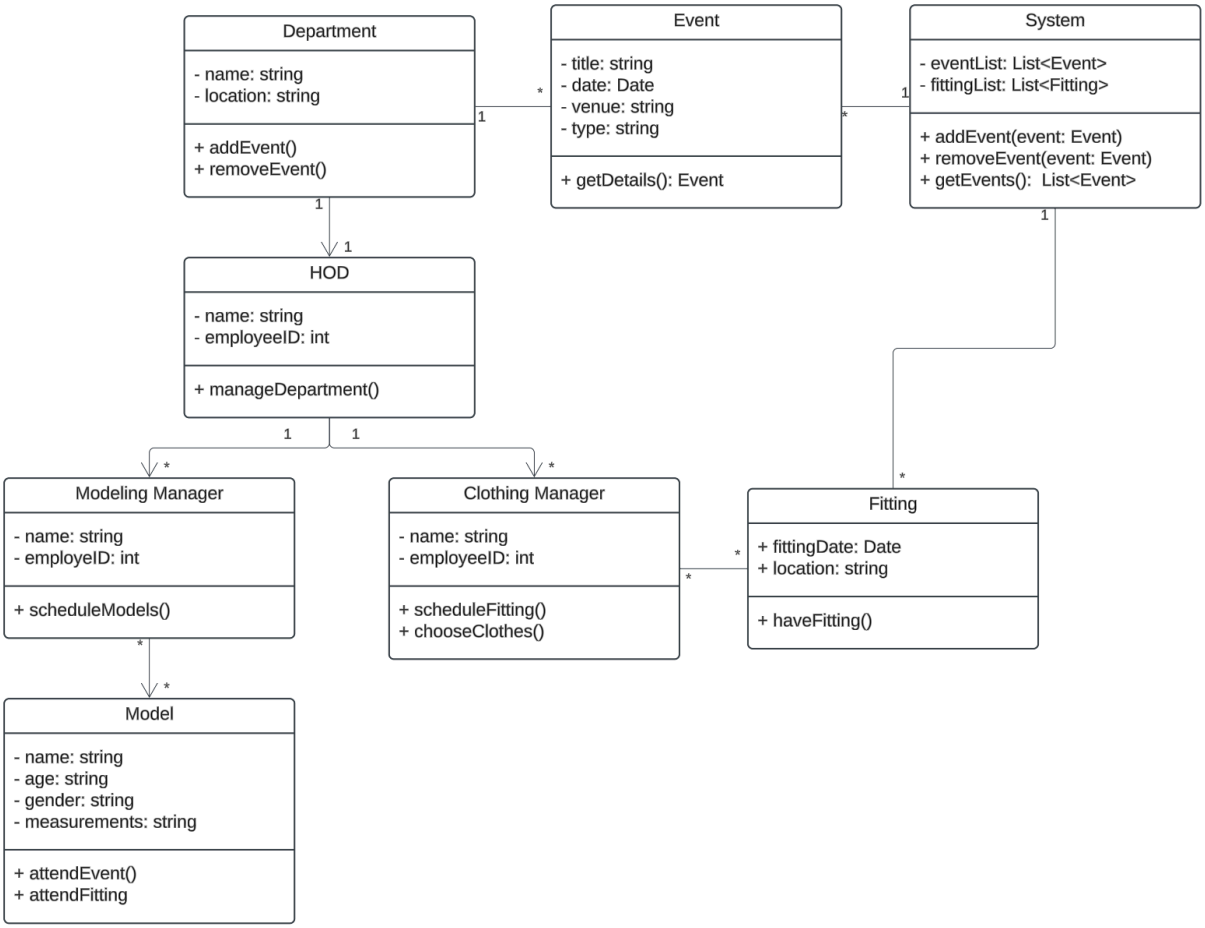
PayrollController
[i = 1 … n] b:= addEmployeeData(filePath:String):boolean b:= addPayrollRates(filePath:String):boolean b:=processPayroll():boolean

Payroll
[i = 1 … n] b:= addEmployeeData(filePath:String):boolean b:= addPayrollRates(filePath:String):boolean b:=processPayroll():boolean

PayrollReport
basePay:Double benefitsPay:Double overtimePay:Double grossPay:Double netPay:Double
totalBase:=getTotalBasePay():double totalBenefits:=getTotalBenefitsPay():double totalOvertime:=getTotalOvertimePay():double totalGross:=getTotalGrossPay():double totalNet:=getTotalNetPay()

Employee
employeeID:String name:String department:String position:String benefits:Double overtime:Double salary:Double
salary:=getSalary():double

Modeling


Department
- name: string
- location: string
+ addEvent()
+ removeEvent()
Event
- title: string
- date: Date
- venue: string
- type: string
+ getDetails(): Event
System
- eventList: List<Event>
- fittingList: List<Fitting>
+ addEvent(event: Event)
+ removeEvent(event: Event)
+ getEvents(): List<Event>
HOD
- name: string
- employeeID: int
+ manageDepartment()
Modeling Manager
- name: string
- employeeID: int
+ scheduleModels()
Clothing Manager
- name: string
- employeeID: int
+ scheduleFitting()
+ chooseClothes()
Fitting
+ fittingDate: Date
+ location: string
+ haveFitting()
Model
- name: string
- age: string
- gender: string
- measurements: string
+ attendEvent()
+ attendFitting

Interface List

Treasury:

```
public interface EmployeeInterface {
        Employee getEmployeeDetails();
        double getSalary();
        void setSalary(double newSalary);
}
public interface PayrollControllerInterface {
        boolean addEmployeeData(String filePath);
        boolean addPayrollRates(String filePath);
        boolean processPayroll(String filepath);
}
public interface PayrollInterface {
        boolean addEmployeeData(String filePath);
        boolean addPayrollRates(String filePath);
        boolean processPayroll(String fileDirectory);
        void createPayroll(String fileDirectory);
        void createPayslip(String fileDirectory);
        PayrollApproval retrieveApproval(String reportID);
}
public interface PayrollReportInterface {
        double getTotalBasePay();
        double getTotalBenefitsPay();
        double getTotalOvertimePay();
        double getTotalGrossPay();
        double getTotalNetPay();
}
```

Inventory:

```
public interface Storage {
    void addOrder(Order order);
    void updateProductCount(String productId, int quantityChange);
}
public interface Retailer {
     String getName();
      String getLocation();
}
public interface InventoryManagement {
    void registerProduct(Product product);
    void registerRetailer(Retailer retailer);
```

```
    void confirmOrder(Order order);
}
public interface ProductDescription {
        String getDescription();
}
public interface InventoryController {
        void registerProduct(int id, String name, double price);
        void registerRetailer(String name, String location);
        void addOrder(int id);
        void updateProductQuantity(int productId, int quantity, String location);
}
```

HR:

(none)

Modeling:

```
public interface Event {
        Event addEvent(Employee[] models, Boolean type, String celebrity, Boolean collab)
        void endEvent();
}

public interface Fitting {
        Fitting addFitting(Employee model, Product garment, LocalDateTime date);
        void endFitting();
}

public interface HOD {
        void haveEvent(Boolean type);
        Boolean requestAdvertisement(Event event);
        Boolean requestContract(String celebrity);
        Boolean requestCollab(String brand);
}

public interface Manager{
        Fitting requestFitting(Employee Model);
        Boolean scheduleModel(Employee Model);
}
```

Manufacturing

```
Public interface MachineOperations{
        Boolean isRunning();
        Void startMachine();
        Void stopMachine();
        void startProduction();
        Void stopProduction();
}
Public interface RawMaterialHandler{
        Void getRawMaterials(String materials, int quantity);
        Void decrementRawMaterialCount(String material, int quantity);
}
```

Design

```
Public interface DesignSpecifications{
        Void setColor(List<String> colors);
        Void setRawMaterials(List<String> rawMaterials);
        Void setSizes(List<String> sizes);
        Void setquantity(int quantities);
        List<String> getColors();
        List<String> getRawMaterials();
        List<String> getSizes();
        Int getQuantities();
}
Public interface FinalDesignApproval{
        Void approveFinalDesign(FinalDesign design);
        boolean  isApproved(FinalDesign design);

}
```

Implementation

Treasury

```
/**
 * @author Kenny
 */

package treasuryDepartment;

import java.util.Scanner;

/**
 * Stores an instance of Payroll that is used to
 * call the Payroll object methods.
 */
public class PayrollController implements PayrollControllerInterface{

	private PayrollController controller;

	private Payroll payroll = null;

	private PayrollController (PayrollController controller) {
		this.controller = controller;
	}

	public PayrollController() {};

	/**
	 * Run the payroll program
	 */
	public void run() {

		System.out.println("Welcome to the Payroll System");


		Scanner scan = new Scanner(System.in);
		boolean exit = false;

		while (!exit) {

			System.out.println("1. Load Department Employee Data List");
			System.out.println("2. Load Payroll Rates");
			System.out.println("3. Process Payroll & Payslips");
			System.out.println("4. Exit program");

			int choice = scan.nextInt();
			scan.nextLine();

			switch (choice) {

				case 1:
					System.out.println("Enter path of Department Employee Data File:");
					String filePath = scan.nextLine();
					if (addEmployeeData(filePath)) {;
						System.out.println("Department Employee Data added successfully\n");
					}
					else {
						System.out.println("Invalid Employee Data file\n");
					}
					break;
				case 2:
					System.out.println("Enter path of Payroll Rate File:");
					String filePath2 = scan.nextLine();
					if (addPayrollRates(filePath2)) {
						System.out.println("Payroll Rate added successfully\n");
					}
					else {
						System.out.println("Invalid Payroll Rate file\n");
```

```
                                                }
                                                break;
                                        case 3:
                                                System.out.println("Enter path to store Payroll Report & Payslips: (e.g.
User\\Documents\\(files stored in here))");
                                                String filePath3 = scan.nextLine();
                                                if (processPayroll(filePath3)) {
                                                        System.out.println("Payroll successfully processed and saved\n");
                                                }
                                                else {
                                                        System.out.println("No Employee Data or Payroll Data found\n");
                                                }
                                                break;
                                        case 4:
                                                System.out.println("Closing program...");
                                                exit = true;
                                                break;
                                        default:
                                                System.out.println("Incorrect choice. Try again.");
                                }
                        }
                        scan.close();
                }

                @Override
                public boolean addEmployeeData(String filePath) {

                        return this.getPayrollInstance().addEmployeeData(filePath);
                }

                @Override
                public boolean addPayrollRates(String filePath) {

                        return this.getPayrollInstance().addPayrollRates(filePath);
                }

                @Override
                public boolean processPayroll(String filePath) {

                        return this.getPayrollInstance().processPayroll(filePath);
                }

                /**
                 * To define one instance of the class Payroll to
                 * call Payroll methods
                 * @return
                 */
                private Payroll getPayrollInstance() {

                        if (payroll == null) {
                                payroll = new Payroll();
                        }
                        return payroll;
                }
}


/**
 * @author Kenny
 */

package treasuryDepartment;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import tEditor.PoorTextEditor;

/*
 * Information Expert that knows Employee and Rates
 */
public class Payroll implements PayrollInterface{
```

```
	// department name to filepath repository
	private ArrayList<Employee> employeeRepo = new ArrayList<Employee>();
	private String payrollData = null;
	private Map<String, PayrollReport> payrollReportRepo = new HashMap<>();

	PoorTextEditor editor = new PoorTextEditor();

	@Override
	public boolean addEmployeeData(String employeeData) {

		editor.processTextFile(employeeData);

		if (editor.getRepository().isEmpty()) {
			return false;
		}

		String[] temp = editor.getArrayNames();
		for (String s : temp) {

			// storing employees to repo
			Employee employee = new Employee(s, editor.retrieveValue(s, "name"),
editor.retrieveValue(s,
"position"),
editor.retrieveValue(s,
"department"),
editor.getDoubleValue(s, "work hours"),
editor.getDoubleValue(s, "benefits"),
editor.getDoubleValue(s, "overtime"),
editor.getDoubleValue(s, "salary"));
			employeeRepo.add(employee);
		}
		return true;
	}

	@Override
	public boolean addPayrollRates(String payrollData) {

		editor.processTextFile(payrollData);

		if (editor.getRepository().isEmpty()) {
			System.out.println("Incorrect path to Department Employee Data");
			return false;
		}

		this.payrollData = payrollData;
		return true;
	}

	@Override
	public boolean processPayroll(String fileDirectory) {

		if (employeeRepo.isEmpty() || payrollData == null) {
			return false;
		}

		File directory = new File(fileDirectory);
		if (!directory.exists()) {

			System.out.println("Directory does not exist");
			return false;
		}

		calculatePayroll();

		createPayroll(fileDirectory);
		createPayslip(fileDirectory);
		return true;
	}

	@Override
	public PayrollApproval retrieveApproval(String reportID) {
		// TODO Auto-generated method stub
		return null;
	}
```

```
	private void calculatePayroll() {

		editor.processTextFile(payrollData);
		double salaryRate = editor.getDoubleValue("payrollRate", "salaryRate");
		double benefitsRate = editor.getDoubleValue("payrollRate", "benefitsRate");
		double overtimeRate = editor.getDoubleValue("payrollRate", "overtimeRate");
		double taxRate = editor.getDoubleValue("payrollRate", "taxRate");
		double retirementRate = editor.getDoubleValue("payrollRate", "retirementRate");

		for (int i = 0; i < employeeRepo.size(); i++) {

			double basePay = Double.parseDouble(String.format("%.2f", salaryRate *
getPositionBasePay(employeeRepo.get(i).getPosition())));
			double benefitsPay = Double.parseDouble(String.format("%.2f", benefitsRate * employeeRepo.get(i).getBenefits()));
			double overtimePay = Double.parseDouble(String.format("%.2f", overtimeRate *
employeeRepo.get(i).getOvertime()));
			double grossPay = Double.parseDouble(String.format("%.2f", basePay + benefitsPay + overtimePay));
			double taxDeduction = Double.parseDouble(String.format("%.2f", grossPay * (taxRate / 100.0)));
			double retirementDeduction = Double.parseDouble(String.format("%.2f", grossPay * (retirementRate / 100.0)));
			double netPay = Double.parseDouble(String.format("%.2f", grossPay - taxDeduction - retirementDeduction));

			PayrollReport subreport = new PayrollReport(basePay, benefitsPay, overtimePay, grossPay, netPay);

			if (payrollReportRepo.containsKey(employeeRepo.get(i).getDepartment())) {

				PayrollReport report = payrollReportRepo.get(employeeRepo.get(i).getDepartment());
				PayrollReport newReport = new PayrollReport(basePay + report.getTotalBasePay(),

		benefitsPay + report.getTotalBenefitsPay(),

		overtimePay + report.getTotalOvertimePay(),

		grossPay + report.getTotalGrossPay(),

		netPay + report.getTotalNetPay());
				payrollReportRepo.put(employeeRepo.get(i).getDepartment(), newReport);
			}
			else {
				payrollReportRepo.put(employeeRepo.get(i).getDepartment(), subreport);
			}
			employeeRepo.get(i).setSalary(netPay);
		}
	}

	private double getPositionBasePay(String position) {

		double baseSalary = -1;

		switch (position) {

		case "employee":
			baseSalary =  3000.00;
			break;
		case "manager":
			baseSalary =  6000.00;
			break;
		default:
			System.out.println("No such job position");
		}
		return baseSalary;
	}

	@Override
	public void createPayroll(String fileDirectory) {

		BufferedWriter writer = null;

		try {

			LocalDateTime timeNow = LocalDateTime.now();
			DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MMddyyyyHHmmss");
			String formattedDateTime = timeNow.format(formatter);

			writer = new BufferedWriter(new FileWriter(fileDirectory + "Payroll" + formattedDateTime + ".txt"));

			double salaries = 0;

			for (Map.Entry<String, PayrollReport> entry : payrollReportRepo.entrySet()) {

				String departmentName = entry.getKey();
```

```
                                        PayrollReport report = entry.getValue();

                                        writer.write(departmentName + "\n");
                                        writer.write("Total Base Pay: " + report.getTotalBasePay() + "$\n");
                                        writer.write("Total Benefits Pay: " + report.getTotalBenefitsPay() + "$\n");
                                        writer.write("Total Overtime Pay: " + report.getTotalOvertimePay() + "$\n");
                                        writer.write("Total Gross Pay: " + report.getTotalGrossPay() + "$\n");
                                        writer.write("Total Net Pay: " + report.getTotalNetPay() + "$\n");
                                        writer.write("\n");

                                        salaries += report.getTotalNetPay();
                                }

                                writer.write("================================================================================\n");
                                writer.write("Total Salaries Paid: " + salaries + "$\n");
                        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (writer != null) {
                    writer.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

                }

                @Override
                public void createPayslip(String fileDirectory) {

                        BufferedWriter writer = null;

                        for (int i = 0; i < employeeRepo.size(); i++) {

                                try {

                                        Employee employee = employeeRepo.get(i);
                                        writer = new BufferedWriter(new FileWriter(fileDirectory + "PaySlip_" + employee.getEmployeeID() +
".txt"));

                                        writer.write("Employee ID: " + employee.getEmployeeID() + "\n");
                                        writer.write("Name: " + employee.getName() + "\n");
                                        writer.write("\n");
                                        writer.write("Salary: " + employee.getSalary() + "$");


                                } catch (IOException e) {
                            e.printStackTrace();
                        } finally {
                            try {
                                if (writer != null) {
                                    writer.close();
                                }
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                        }
                                }
                }
}


/**
 * @author Kenny
 */

package treasuryDepartment;

public class PayrollReport implements PayrollReportInterface{

                private double basePay, benefitsPay, overtimePay, grossPay, netPay;

                public PayrollReport (double basePay, double benefitsPay, double overtimePay,
                                                                        double grossPay, double netPay) {

                        this.basePay = basePay;
                        this.benefitsPay = benefitsPay;
                        this.overtimePay = overtimePay;
```

```
                this.grossPay = grossPay;
                this.netPay = netPay;
        }

        @Override
        public double getTotalBasePay() {
                return basePay;
        }

        @Override
        public double getTotalBenefitsPay() {
                return benefitsPay;
        }

        @Override
        public double getTotalOvertimePay() {
                return overtimePay;
        }

        @Override
        public double getTotalGrossPay() {
                return grossPay;
        }

        @Override
        public double getTotalNetPay() {
                return netPay;
        }
}


/**
 * @author Kenny
 */

package treasuryDepartment;

public class Employee implements EmployeeInterface{

        private String employeeID, name, position, department;
        private double workHours, benefits, overtime, salary;

        public Employee() {};

        public Employee(String employeeID, String name, String position, String department,
                                                        double workHours, double benefits, double overtime, double salary) {

                this.employeeID = employeeID;
                this.name = name;
                this.position = position;
                this.department = department;
                this.workHours = workHours;
                this.benefits = benefits;
                this.overtime = overtime;
                this.salary = salary;
        }

        @Override
        public Employee getEmployeeDetails() {

                return this;
        }

        public String getEmployeeID() {

                return employeeID;
        }

        public String getName() {

                return name;
        }

        public String getPosition() {

                return position;
        }

        public String getDepartment() {
```

```
		return department;
	}

	public double getWorkHours() {

		return workHours;
	}

	public double getBenefits() {

		return benefits;
	}

	public double getOvertime() {

		return overtime;
	}

	@Override
	public double getSalary() {

		return salary;
	}

	@Override
	public void setSalary(double newSalary) {
		this.salary = newSalary;
	}

}


/**
 * @author Kenny
 */
package tEditor;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.LinkedHashMap;
import java.util.Map;

/**
 * A very limited text parser, editor, and printer
 *
 * Necessary functionalities are all called from
 * public methods
 */
public class PoorTextEditor {

	private Map<String, Object> repository = new LinkedHashMap<>();

	/**
	 * Takes a text file in the format of:
	 *
	 * e.g. FruitsInventory.txt
	 *
	 * item1:
	 * name/apple
	 * price/4.12
	 * quantity/500
	 *
	 * item2:
	 * name/banana
	 * price/1.32
	 * quantity/2000
	 *
	 * A ":" indicates the start of an array item
	 * A "/" indicates the separation of a key value pair
	 *
	 * When an array item (item1) has been successfully added to a HashMap
	 * and its contents into a nested HashMap, it stops further updating
	 * to the array item when encountering a new array item (item2) as
	 * indicated by ":"
	 *
	 * Repository is updated with information gathered from text file
	 *
```

```
	 * @param filePath path of file, relative or absolute
	 */
	public void processTextFile(String filePath) {

		repository = new LinkedHashMap<>();
		parseTextToRepo(filePath);
	}

	/**
* Writes the data from the repository (HashMap) to a text file.
* The output format will look like:
*
* item1:
* name/apple
* price/4.12
* quantity/500
*
* item2:
* name/banana
* price/1.32
* quantity/2000
*
* @param filePath desired path of the file to write to
*/
	public void writeToTextFile(String filePath) {

		writeDataToFile(filePath);
	}

	/**
* Writes the data from the repository (HashMap) to a text file.
* The output format will look like:
*
* item1:
* name/apple
* price/4.12
* quantity/500
*
* item2:
* name/banana
* price/1.32
* quantity/2000
*
* @param hashmap repository
* @param filePath desired path of the file to write to
*/
	public void writeToTextFile(Map<String, Object> givenRepo, String filePath) {

		writeDataToFile(givenRepo, filePath);
	}

	/**
	 * @return the HashMap repository
	 */
	public Map<String, Object> getRepository() {

		return repository;
	}

	/**
	 * To retrieve a specific value for a specific key under a specific array
	 * item
	 * @param arrayName specific name of array item
	 * @param keyName specific name of key name
	 * @return value associated with key and array item
	 */
	public String retrieveValue(String arrayName, String keyName) {

		return retrieveValueFromKey(arrayName, keyName);
	}

	public double getDoubleValue(String arrayName, String keyName) {

		return retrieveDoubleFromKey(arrayName, keyName);
	}

	/**
	 * To set a specific value for a specific key under a specific array
	 * item
	 * @param arrayName specific name of array item
```

```
 * @param keyName specific name of key name
 * @param value replacement value
 */
public void setValue(String arrayName, String keyName, String value) {

		setValueFromKey(arrayName, keyName, value);
}

/**
 * Retrieves a HashMap of Key Value data under a specified array
 * item name
 * @param arrayItemName specific array item name
 * @return HashMap of Key Value data
 */
public Map<String, Object> getArrayItems(String arrayName){

		return getArrayItemList(arrayName);
}

/**
 * Creates a String array of all array item names
 * @return String array of all array item names
 */
public String[] getArrayNames() {

		return getArrayNamesList();
}

/**
 * Remove an array item with the given array name
 * @param arrayName
 */
public void removeArrayItem(String arrayName) {

		removeArrayItemByKey(arrayName);
}

/**
 * Retrieves a HashMap of Key Value data under a specified array
 * item name
 * @param arrayItemName specific array item name
 * @return HashMap of Key Value data
 */
private Map<String, Object> getArrayItemList(String arrayItemName) {

		if (checkRepositoryData()) {

			return null;
		}

		Map<String, Object> arrayItem;

		if (repository.containsKey(arrayItemName)) {

			arrayItem = (Map<String, Object>) repository.get(arrayItemName);
		}
		else {

			System.out.println("ArrayItem: " + arrayItemName + " not found");
			return null;
		}
		return arrayItem;

}

/**
 * To retrieve a specific value for a specific key under a specific array
 * item
 * @param arrayName specific name of array item
 * @param keyName specific name of key name
 * @return value associated with key and array item
 */
private String retrieveValueFromKey(String arrayName, String keyName) {

		if (checkRepositoryData()) {

			return null;
		}

		String value = null;
```

```
                if (repository.containsKey(arrayName)) {

                        Map<String, Object> arrayItem = (Map<String, Object>) repository.get(arrayName);

                        if (arrayItem.containsKey(keyName)) {

                                value = (String) arrayItem.get(keyName);

                                if (value == null) {

                                        System.out.println("Value not found for key: " + keyName + " under arrayItem: " +
arrayName);
                                }
                        }
                        else {
                                System.out.println("Key: " + keyName + " not found under arrayItem: " + arrayName);
                        }
                }
                else {
                        System.out.println("ArrayItem: " + arrayName + " not found");
                }
                return value;
        }

        /**
         * To retrieve a specific double value for a specific key under a specific
         * array item
         * @param arrayName specific name of array item
         * @param keyName specific name of key name
         * @return value associated with key and array item
         */
        private double retrieveDoubleFromKey(String arrayName, String keyName) {

                if (checkRepositoryData()) {

                        return -1;
                }

                String value = null;

                if (repository.containsKey(arrayName)) {

                        Map<String, Object> arrayItem = (Map<String, Object>) repository.get(arrayName);

                        if (arrayItem.containsKey(keyName)) {

                                value = (String) arrayItem.get(keyName);

                                if (value == null) {

                                        System.out.println("Value not found for key: " + keyName + " under arrayItem: " +
arrayName);
                                }
                        }
                        else {
                                System.out.println("Key: " + keyName + " not found under arrayItem: " + arrayName);
                        }
                }
                else {
                        System.out.println("ArrayItem: " + arrayName + " not found");
                }
                return Double.parseDouble(value);
        }

        /**
         * To set a specific value for a specific key under a specific array
         * item
         * @param arrayName specific name of array item
         * @param keyName specific name of key name
         * @param value replacement value
         */
        private void setValueFromKey(String arrayName, String keyName, String value) {

                if (!checkRepositoryData()) {

                        if (repository.containsKey(arrayName)) {

                                Map<String, Object> arrayItem = (Map<String, Object>) repository.get(arrayName);
```

```
                                        if (arrayItem.containsKey(keyName)) {

                                                arrayItem.replace(keyName, value);
                                        }
                                        else {
                                                System.out.println("Key: " + keyName + " not found under arrayItem: " + arrayName);
                                        }
                                }
                                else {
                                        System.out.println("ArrayItem: " + arrayName + " not found");
                                }
                        }
}

/**
 * Takes a text file in the format of:
 *
 * e.g. FruitsInventory.txt
 *
 * item1:
 * name/apple
 * price/4.12
 * quantity/500
 *
 * item2:
 * name/banana
 * price/1.32
 * quantity/2000
 *
 * A ":" indicates the start of an array item
 * A "/" indicates the separation of a key value pair
 *
 * When an array item (item1) has been successfully added to a HashMap
 * and its contents into a nested HashMap, it stops further updating
 * to the array item when encountering a new array item (item2) as
 * indicated by ":"
 *
 * Repository is updated with information gathered from text file
 *
 * @param filePath path of file, relative or absolute
 */
private void parseTextToRepo(String filePath){

        // to store array items in HashMap
        Map<String, Object> arrayItemList = new LinkedHashMap<>();
        Map<String, Object> currentItem = null;
        String currentArrayItem = null;
        // more efficient compared to Scanner
        BufferedReader reader = null;

        try {

                reader = new BufferedReader(new FileReader(filePath));
                String currentLine;

                while ((currentLine = reader.readLine()) != null) {

                        // remove trailing whitespace
                        currentLine = currentLine.trim();

                        // check for new array item or reached the end of an array item
                        if (currentLine.endsWith(":")){

                                // if array item already exists
                                if ( currentArrayItem != null && currentItem != null) {

                                        arrayItemList.put(currentArrayItem, currentItem);
                                }
                                // getting array item name without ":" or whitespace
                                currentArrayItem = currentLine.substring(0, currentLine.length() - 1).trim();
                                // create new array item to store items
                                currentItem = new LinkedHashMap<>();
                        }
                        // otherwise process key and value pairs
                        else if (currentItem != null && currentLine.contains("/")) {

                                // split current line into key and value pair at "/"
                                String[] keyValue = currentLine.split("/", 2);

                                if (keyValue.length == 2) {
```

```
                                                String key = keyValue[0].trim();
                                                String value = keyValue[1].trim();
                                                currentItem.put(key, value);
                                        }
                                }
                        }
                        // last processed array item is added to list
                        if (currentArrayItem != null && currentItem != null) {
                                arrayItemList.put(currentArrayItem, currentItem);
                        }

                } catch (IOException e) {
                        e.printStackTrace();
                } finally {
        try {
            if (reader != null) {
                reader.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
                repository = arrayItemList;
            }

            /**
 * Writes the data from the repository (HashMap) to a text file.
 * The output format will look like:
 *
 * item1:
 * name/apple
 * price/4.12
 * quantity/500
 *
 * item2:
 * name/banana
 * price/1.32
 * quantity/2000
 *
 * @param filePath path of the file to write to
 */
private void writeDataToFile(String filePath) {

            if (!checkRepositoryData()) {

                BufferedWriter writer = null;

        try {

            writer = new BufferedWriter(new FileWriter(filePath));

            // Iterate over the repository map
            for (Map.Entry<String, Object> entry : repository.entrySet()) {

                String arrayName = entry.getKey();
                Map<String, String> arrayItem = (Map<String, String>) entry.getValue();

                // writing the array name followed by ":"
                writer.write(arrayName + ":\n");

                // write all key-value pairs under array item
                for (Map.Entry<String, String> keyValueEntry : arrayItem.entrySet()) {
                    writer.write(keyValueEntry.getKey() + "/" + keyValueEntry.getValue() + "\n");
                }

                // adding blank line to separate array items
                writer.write("\n");
            }

            // debug
            System.out.println("Data written to " + filePath + " successfully.");

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (writer != null) {
                    writer.close();
                }
```

```
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

            }
}

/**
 * Writes the data from the repository (HashMap) to a text file.
 * The output format will look like:
 *
 * item1:
 * name/apple
 * price/4.12
 * quantity/500
 *
 * item2:
 * name/banana
 * price/1.32
 * quantity/2000
 *
 * @param givenRepo hashmap repository
 * @param filePath path of the file to write to
 */
private void writeDataToFile(Map<String, Object> givenRepo, String filePath) {

            if (!checkRepositoryData()) {

                        BufferedWriter writer = null;

        try {

            writer = new BufferedWriter(new FileWriter(filePath));

            // Iterate over the repository map
            for (Map.Entry<String, Object> entry : givenRepo.entrySet()) {

                String arrayName = entry.getKey();
                Map<String, String> arrayItem = (Map<String, String>) entry.getValue();

                // writing the array name followed by ":"
                writer.write(arrayName + ":\n");

                // write all key-value pairs under array item
                for (Map.Entry<String, String> keyValueEntry : arrayItem.entrySet()) {
                    writer.write(keyValueEntry.getKey() + "/" + keyValueEntry.getValue() + "\n");
                }

                // adding blank line to separate array items
                writer.write("\n");
            }

            // debug
            System.out.println("Data written to " + filePath + " successfully.");

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (writer != null) {
                    writer.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

            }
}

/**
 * Checks if repository is empty
 * @return true if empty, otherwise false
 */
private boolean checkRepositoryData() {

            if (repository.isEmpty()) {
                        System.out.println("Repository not initialized with data");
                                return true;
```

```
                }
            return false;
    }

    /**
             * Creates a String array of all array item names
             * @return String array of all array item names
             */
    private String[] getArrayNamesList() {

            String[] arrayNamesList = new String[repository.size()];

            int index = 0;
            for (Map.Entry<String, Object> entry : repository.entrySet()) {

                    if (!entry.getKey().isEmpty()) {
                            arrayNamesList[index] = entry.getKey();
                            index++;
                    }
            }
            return arrayNamesList;
    }

    /**
             * Remove an array item with the given array name
             * @param arrayName
             */
    private void removeArrayItemByKey(String arrayName) {

            if (!checkRepositoryData()) {

                    if (repository.containsKey(arrayName)) {

                            repository.remove(arrayName);
                            System.out.println("ArrayItem: " + arrayName + " has been successfully removed");
                    }
                    else {
                            System.out.println("ArrayItem: " + arrayName + " not found");
                    }
            }
    }
}
```

Manufacturing

```
package Manufacturing;

import Design.FinalDesignApproval;
import Design.DesignSketch;
import Design.DesignSpecifications;
import Design.FinalDesign;

import java.util.List;

/*
@author Doyle Chism
 */
public class HeadOfManufacturing implements FinalDesignApproval {


    //variables
    private ManufacturingManager manager;
```

```
private Machines machines;
private WarehouseStorage warehouse;
private DesignSpecifications design;



//constructor
public HeadOfManufacturing(ManufacturingManager manager, WarehouseStorage warehouse, Machines machines) {
    this.manager = manager;
    this.warehouse = warehouse;
    this.machines = machines;

}

//Head of manufacturing tasks that are needed to complete

//head of manufacturing receives design from the design team
private void receiveDesign(FinalDesign design) {
    //assuming there is a design team we are receiving the correct information
   if(isApprovedDesign(true)){
       this.design = design;
       System.out.println("Design Approved");
       reviewDesign();
   }
   else{
       System.out.println("Design Not Approved");
   }
}


@Override
public void approveFinalDesign(FinalDesign finalDesign) {

    if(finalDesign != null){
        finalDesign.isApproved();
        System.out.println("Final Design Approved");
    }
}

@Override
public void isApprovedDesign(FinalDesign finalDesign) {

    return finalDesing.isApproved();
}

@Override
public void setApprovedDesign(FinalDesign finalDesign) {

    if(finalDesign != null && finalDesign.isApproved){
        this.design = finalDesign;
    }
}

//head of manufacturing reviews design from the design team
public void reviewDesign() {
    System.out.println("Reviewing design details for: " + design.getDesignName());

    List<String> requiredMaterials = design.getRawMaterials();
    List<Integer> materialCount = design.getRawMaterialCount();

    System.out.println("Below is a list of the required materials: ");
    for (int i = 0; i < requiredMaterials.size(); i++) {
        System.out.println("- " + requiredMaterials.get(i) + ":" + materialCount.get(i));
    }
    manager.collectRawMaterials(requiredMaterials, materialCount);
    selectMachines();
    ;
}
```

```
    //head of manufacturing verifies raw materials from the manager
    public void reviewRawMaterials(List<String> materials, List<Integer> quantity) {

        System.out.println("Verifying the raw materials collected by the Manager");
        boolean materialsMatchDesign = true;

        for (int i = 0; i < materials.size(); i++) {
            if (!materials.get(i).equals(design.getRawMaterials().get(i))) {
                System.out.println("Incorrect quantity of raw materials: " + materials.get(i));
                materialsMatchDesign = false;
            } else {
                System.out.println("Correct quantity of raw materials: " + materials.get(i));
            }
        }

        if (materialsMatchDesign) {
            System.out.println("All materials are approved by the head of manufacturer");
            manager.startManufacturing();
        } else {
            System.out.println("Some materials are not approved by the head of manufacturer");
        }

    }

    public void verifyRawMaterials() {
        System.out.println("Verifying the raw materials collected by the Manager");
        manager.startManufacturing();
    }

    //head of manufacturing decides from the design what machines to use
    public void selectMachines() {
        System.out.println("Selecting the required machine for production");
        List<String> selectedMachines = machines.selectMachine(design.getRequiredMachineType());

        System.out.println("Machine selected for production:");
        for (String machine : selectedMachines) {
            System.out.println("- " + machine);
        }
        machines.startProduction();

    }



}

package Manufacturing;

public class Machines implements MachineOperations {


    private boolean isRunning;
    private boolean isProducing;

    public Machines() {
        isRunning = false;
        isProducing = false;
    }
    //isRunning
    @Override
    public boolean isRunning() {

        return isRunning;
    }

    //checkStatus
    @Override
    public void startMachine() {
```

```
        if(!isRunning) {
            isRunning = true;
            System.out.println("Machine is running.");
        }else{
            System.out.println("Machine is already running.");
        }
    }

    @Override
    public void stopMachine() {
        if(isRunning) {
            isRunning = false;
            System.out.println("Machine is stopped.");
        }
        else{
            System.out.println("Machine is not running.");
        }
    }

    @Override
    public void startProduction() {

        if(isRunning && !isProducing) {
            isProducing = true;
            System.out.println("Machine is producing.");
        }
        else if(!isRunning){
            System.out.println("Machine is not producing.");
        }
        else{
            System.out.println("Machine is already producing.");
        }
    }

    @Override
    public void stopProduction() {

        if(isProducing){
            isProducing = false;
            System.out.println("Machine is stopped.");
        }
        else{
            System.out.println("Machine is not producing.");
        }
    }

}
package Manufacturing;

import java.util.List;

/*
@author
 */
public class ManufacturingManager implements MachineOperations, RawMaterialHandler{

    private WarehouseStorage warehouse;
    private HeadOfManufacturing headOfManufacturing;
    private List<ManufacturingWorkers> workers;
    private MachineOperations machines;

    public ManufacturingManager(WarehouseStorage warehouse, HeadOfManufacturing headOfManufacturing,
                                List<ManufacturingWorkers> workers, MachineOperations machines) {
        this.warehouse = warehouse;
        this.headOfManufacturing = headOfManufacturing;
        this.workers = workers;
    }
    //collect the raw materials for the Head of Manufacturing
    @Override
    public void getRawMaterials(String material, int quantity) {
```

```
    System.out.println("material: " + material + "quantity: " + quantity);
    warehouse.retrieveRawMaterials(material, quantity);
}

//decrement rawMaterialCount from the warehouseStorage
@Override
public void decrementRawMaterialCount(String material, int quantity) {
    System.out.println("material: " + material + "new quantity: " + quantity);
    warehouse.decrementMaterialCount(material, quantity);
}
//verify machines to use from the head of manufacturing

@Override
public boolean isRunning() {
    return machines.isRunning();
}

//start the machines that the head of manufacturing told to start
@Override
public void startMachine() {

    if(machines.isRunning()){
        System.out.println("Machine is already running");
    }
    else{
        machines.startMachine();
    }
}

@Override
public void stopMachine() {

    if(machines.isRunning()){
       System.out.println("Machine is already running");
       machines.stopMachine();
    }
    else{
      System.out.println("Machine is already stopped");
    }
}
@Override
public void startProduction(){

    if(machines.isRunning()){
        System.out.println("machine is already running");
    }else{
        machines.startProduction();
    }
}

@Override
public void stopProduction() {

    if(machines.isRunning()){
        System.out.println("machine is already running");
        machines.stopMachine();
    }
    else{
        System.out.println("machine is already stopped");
    }
}


//supervise the workers to make sure they are working
/*
After the product is completed
 */
public void beginProduction(int quantity) {
    System.out.println("begin mass production of "+ quantity + "units");
```

```
        if(!machines.isRunning()){
            startMachine();
        }
        for(int i=0; i < quantity; i++){
            System.out.println("production of " + i + "units");
            machines.startProduction();
        }
        machines.stopProduction();
        machines.stopMachine();
    }


}
package Manufacturing;

import java.util.HashMap;
import java.util.Map;

public class WarehouseStorage {


    //hold the materials for production
    private Map<String, Integer> materialStock;

    public WarehouseStorage() {
        materialStock = new HashMap<>();
    }
    //manager will retrieve(materials) from the warehouse
    public void retrieveRawMaterials(String material, int quantity){
        if(materialStock.containsKey(material)){
            materialStock.put(material, quantity);
            decrementMaterialCount(material, quantity);
        }
        else{
            System.out.println("Material " + material + " does not exist");
        }
    }

    public void decrementMaterialCount(String material, int quantity){
        if(materialStock.containsKey(material)){
            int currentQuantity = materialStock.get(material);
            int newQuantity = currentQuantity - quantity;
            materialStock.put(material, newQuantity);
            System.out.println("Material " + material + " has been decreased to " + newQuantity);
        }
        else{
            System.out.println("Material " + material + " does not exist");
        }
    }

    //update stock amount



}
```

Design

```
package Design;

import java.util.List;

public class FinalDesign implements DesignSpecifications {

    private final String designName;
    private List<String> colors;
    private List<String> rawMaterials;
```

```
    private List<String> sizes;
    private int quantity;
    private boolean approved;


    public FinalDesign(String designName) {
        this.designName = designName;
        this.approved = false;
    }


    @Override
    public void setColor(List<String> colors) {
        this.colors = colors;
        System.out.println("Color for the design is: " + colors);
    }

    @Override
    public void setRawMaterials(List<String> rawMaterials) {
        this.rawMaterials = rawMaterials;
        System.out.println("Raw materials for the design is: " + rawMaterials);
    }

    @Override
    public void setSizes(List<String> sizes) {
        this.sizes = sizes;
        System.out.println("Sizes for the design is: " + sizes);
    }

    @Override
    public void setQuantities(int quantities) {
        this.quantity = quantities;
        System.out.println("Quantities for the design is: " + quantities);
    }

    @Override
    public List<String> getColors() {
        return colors;
    }

    @Override
    public List<String> getRawMaterials() {
        return rawMaterials;
    }

    @Override
    public List<String> getSizes() {
        return sizes;
    }

    @Override
    public int getQuantities() {
        return quantity;
    }

    public void displayAllDesignSpecifications() {
        System.out.println("All Final Design Specifications:");
        System.out.println("Design Name: " + designName);
        System.out.println("Colors: " + colors);
        System.out.println("Raw Materials: " + rawMaterials);
        System.out.println("Sizes: " + sizes);
        System.out.println("Quantities: " + quantity);
    }
}
package Design;

public enum DesignType{;

    private final String typeName;
    private final String description;
```

```
    DesignType(String typeName, String description) {
        this.typeName = typeName;
        this.description = description;
    }
    public String getTypeName() {
        return typeName;
    }
    public String getDescription() {
        return description;
    }


}
package Design;

public class DesignSketch {

  private final String sketchId;
  private final String sketchDescription;

  public DesignSketch(String sketchId, String sketchDescription) {
      this.sketchId = sketchId;
      this.sketchDescription = sketchDescription;
  }
  public String getSketchId() {
      return sketchId;
  }
  public String getSketchDescription() {
      return sketchDescription;
  }
}
package Design;

import java.util.List;

public class DesignTeamWorkers {

    private String worker;
    private List<DesignSketch> sketches;

    public DesignTeamWorkers(String worker) {
        this.worker = worker;

    }


    public DesignSketch createSketch(String sketchID, String sketchDescription){

        DesignSketch sketch = new DesignSketch(sketchID, sketchDescription);
        sketches.add(sketch);
        System.out.println(sketch);
        return sketch;
    }
    public List<DesignSketch> getSketches() {
          return sketches;
    }
    public String getWorker() {
        return worker;
    }



}
package Design;

import java.util.List;

public class HeadOfDesignTeam implements FinalDesignApproval {
```

```
    private List<DesignSketch> sketches;
    private FinalDesign finalDesign;

    public HeadOfDesignTeam(List<DesignSketch> sketches, FinalDesign finalDesign) {
        this.sketches = sketches;
        this.finalDesign = finalDesign;
    }

    @Override
    public void approveFinalDesign(FinalDesign finalDesign) {

        if(finalDesign != null) {
           System.out.println(sketches);
         finalDesign.setApproved(true);
         this.finalDesign = finalDesign;
    }
        else{
            System.out.println("No final Design approved");
        }



    @Override
    public boolean isApproved(FinalDesign finalDesign) {
            if(finalDesign != null && finalDesign.setApproved(true)) {
                finalDesign.displayAllDesignSpecifications();
                return true;
            }
            else{
            System.out.println("No final Design approved");
            return false;
            }
        }

    }



    public void setDesignSpecifications(DesignSpecifications design) {

        design.setColor(design.getColors());
        design.setRawMaterials(design.getRawMaterials());
        design.setSizes(design.getSizes());
        design.setQuantities(design.getQuantities());

        System.out.println("Head of Design has set all the specifications" + design);
    }


}
```

HR:

```
/**
 * @author Sam Gumm
 */
public class employeeHandlingSystem {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // Add a new employee
        Employee newEmployee1 = new Employee("E123", "Alice Johnson", Department.ENGINEERING, "Software Engineer", "Active", 80000);
        Employee newEmployee2 = new Employee("E124", "John Smith", Department.MARKETING, "Marketing Specialist", "Active", 90000);
        Employee newEmployee3 = new Employee("E125", "Emma Brown", Department.HUMAN_RESOURCES, "HR Coordinator", "Active", 100000);
        Employee newEmployee4 = new Employee("E126", "Michael Davis", Department.FINANCE, "Financial Analyst", "Active", 110000);
        Employee newEmployee5 = new Employee("E127", "Sophia Wilson", Department.ENGINEERING, "DevOps Engineer", "Active", 70000);

        System.out.println("Employee '" + newEmployee1.name + "' in department: " + newEmployee1.department);

        // Add a new candidate
        Candidate newCandidate = new Candidate("C456", "Bob Smith", "Data Analyst");
        System.out.println("Candidate '" + newCandidate.candidateId + "' with name: " + newCandidate.name);

        // Employee management
        employeeRecordManager handler = new employeeRecordManager();
        handler.addEmployee(newEmployee1);
        handler.addEmployee(newEmployee2);
        handler.addEmployee(newEmployee3);
        handler.addEmployee(newEmployee4);
        handler.addEmployee(newEmployee5);
        handler.displayRecords();
        // Switching employee to new department
        handler.updateEmployee("E123", Department.DESIGN, "Software Engineer", "Active", 100000);
```

```
        handler.displayRecords();
        // print out just a department -> DESIGN
        handler.displayDepartment(Department.DESIGN);
        System.out.println(handler.collateSalariesByDepartment(Department.DESIGN));
    }
}

/**
 * @author Sam Gumm
 */
public enum Department {
    ENGINEERING,
    MARKETING,
    HUMAN_RESOURCES,
    FINANCE,
    DESIGN,
    MODELING,
    MANUFACTURING;

    @Override
    public String toString() {
        switch (this) {
            case ENGINEERING: return "Engineering";
            case MARKETING: return "Marketing";
            case HUMAN_RESOURCES: return "Human Resources";
            case FINANCE: return "Finance";
            case DESIGN: return "Design";
            case MODELING: return "Modeling";
            case MANUFACTURING: return "Manufacturing";
            default: throw new IllegalArgumentException();
        }
    }
}


import java.util.ArrayList;
import java.util.List;

/**
 * @author Sam Gumm
 */
public class employeeRecordManager {
    private List<Employee> employeeList = new ArrayList<>();


    /**
     * @param employee
     */
    // Add a new employee
    public void addEmployee(Employee employee) {
        employeeList.add(employee);
    }
```

```
    /**
     * @param employeeId
     * @param department
     * @param position
     * @param employmentStatus
     * @param salary
     */
    // Update employee record
    public void updateEmployee(String employeeId, Department department, String position, String employmentStatus, int salary) {
        for (Employee emp : employeeList) {
            if (emp.employeeId.equals(employeeId)) {
                emp.department = department;
                emp.position = position;
                emp.employmentStatus = employmentStatus;
                return;
            }
        }
        System.out.println("Employee not found: " + employeeId);
    }

    // Display employee records
    public void displayRecords() {
        for (Employee emp : employeeList) {
            System.out.println(emp);
        }
    }


    /**
     * @param departmentName
     */
    public void displayDepartment(Department departmentName) {
        for (Employee emp : employeeList) {
            if(emp.department == departmentName) {
                System.out.println(emp);
            }
        }
    }

    public int collateSalariesByDepartment(Department department) {
        int totalSalary = 0;
        for (Employee emp : employeeList) {
            if(emp.department == department) {
                totalSalary += emp.salary;
            }
        }
        return totalSalary;
    }
}


/**
 * @author Sam Gumm
 */
```

```
class Employee {
    String employeeId;
    String name;
    Department department;
    String position;
    String employmentStatus;
    int salary;

    Employee(String employeeId, String name, Department department, String position, String employmentStatus, int salary) {
        this.employeeId = employeeId;
        this.name = name;
        this.department = department;
        this.position = position;
        this.employmentStatus = employmentStatus;
        this.salary = salary;
    }


    /**
     * @return String
     */
    @Override
    public String toString() {
        return String.format("ID: %s, Name: %s, Department: %s, Position: %s, Status: %s, Salary: %d",
                employeeId, name, department, position, employmentStatus, salary);
    }
}

/**
 * @author Sam Gumm
 */
class Candidate {
    String candidateId;
    String name;
    String positionApplied;
    String status;

    Candidate(String candidateId, String name, String positionApplied) {
        this.candidateId = candidateId;
        this.name = name;
        this.positionApplied = positionApplied;
        this.status = "Applied";
    }


    /**
     * @return String
     */
    @Override
    public String toString() {
        return String.format("ID: %s, Name: %s, Position Applied: %s, Status: %s",
                candidateId, name, positionApplied, status);
    }
}
```

Inventory

```
/**
 * @ Mani Raj
 */
public class Product {
                private int id;
                  private String name;
                  private double price;
                  private ProductDescription productDescription;

                  public Product(int id, String name, double price) {
                     this.id = id;
                     this.name = name;
                     this.price = price;

                  }

                  public void setProductDescription( ProductDescription productDescription)
                  {
                            this.productDescription = productDescription;
                  }

                  public int getId() {
                     return id;
                  }

                  public String getName() {
                     return name;
                  }

                  public double getPrice() {
                     return price;
                  }

                  public ProductDescription getProductDescription() {
                     return productDescription;
                  }
}


/**
 * @ Mani Raj
 */
public class Order {
            private int id;
            private Map<Integer, Integer> productList; // productId mapped to quantity

            private String path="";

            public Order(int id) {
                        this.id = id;
                        this.productList = new HashMap<>();
            }

            public void addProduct(int productId, int quantity) {
                        productList.put(productId, quantity);
                        // Update the text file
                        PoorTextEditor textEditor = new PoorTextEditor();
                        textEditor.setValue("Order" + id, "product" + productId, String.valueOf(quantity));
                        textEditor.writeToTextFile(path);
            }

            public void deleteProduct(int productId) {
                        productList.remove(productId);
```

```
                        // Update the text file
                        PoorTextEditor textEditor = new PoorTextEditor();
                        textEditor.setValue("Order" + id, "product" + productId, null);
                        textEditor.writeToTextFile(path);
                }

                public Map<Integer, Integer> getProductList() {
                        return productList;
                }

                public void setPath(String filePath)
                {
                        path =filePath;
                }
                public int getId() {
                        return id;
                }
}


/**
 * @ Mani Raj
 */
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import txedt.PoorTextEditor;

public class BasicInventManage implements InventoryManagement {
                private List<Product> products;
                private List<Retailer> retailers;
                private List<Storage> storages;

                private String path = "";

                public BasicInventManage() {

                        this.products = new ArrayList<>();
                        this.retailers = new ArrayList<>();
                        this.storages = new ArrayList<>();

                }

                @Override
                public void registerProduct(Product product) {
                        products.add(product);
                        // Update the text file
                        PoorTextEditor textEditor = new PoorTextEditor();
                        textEditor.setValue("Product" + product.getId(), "name", product.getName());
                        textEditor.setValue("Product" + product.getId(), "price", String.valueOf(product.getPrice()));
                        textEditor.writeToTextFile(path );
                }

                @Override
                public void registerRetailer(Retailer retailer) {
                        retailers.add(retailer);
                        // Update the text file
                        PoorTextEditor textEditor = new PoorTextEditor();
                        textEditor.setValue("Retailer" + retailer.getName(), "location", retailer.getLocation());
                        textEditor.writeToTextFile(path );
                }

                @Override
                public void confirmOrder(Order order) {
                        double total = 0;
                        for (Map.Entry<Integer, Integer> entry : order.getProductList().entrySet()) {
                                Product product = getProductById(entry.getKey());
                                if (product != null) {
                                        total += product.getPrice() * entry.getValue();
```

```
                        }
                    }
                    System.out.println("Order ID: " + order.getId() + " Total: $" + total);
                    // Update the text file
                    PoorTextEditor textEditor = new PoorTextEditor();
                    textEditor.setValue("Order" + order.getId(), "total", String.valueOf(total));
                    textEditor.writeToTextFile(path );
            }

            private Product getProductById(int productId) {
                    for (Product product : products) {
                            if (product.getId() == productId) {
                                    return product;
                            }
                    }
                    return null;
            }
            public void setPath(String filePath)
            {
                    path =filePath;
            }
}

/**
 * @ Mani Raj
 */
public class BasicInventoryController implements InventoryController{

   private BasicInventManage inventory = null;
   private BasicStorage store=null;

   public BasicInventoryController() {}

   /**
    * Runs the inventory management system.
    */
   public void run() {
      System.out.println("Welcome to the Inventory Management System");

      Scanner scan = new Scanner(System.in);
      boolean exit = false;

      while (!exit) {
         System.out.println("1. Register Product");
         System.out.println("2. Register Retailer");
         System.out.println("3. Add Product Quantity");
         System.out.println("4. Remove Product Quantity");
         System.out.println("5. Exit Program");

         int choice = scan.nextInt();
         scan.nextLine(); // Consume newline

         switch (choice) {
            case 1:
               System.out.println("Enter Product ID:");
               int productId = scan.nextInt();
               scan.nextLine(); // Consume newline
               System.out.println("Enter Product Name:");
               String productName = scan.nextLine();
               System.out.println("Enter Product Price:");
               double productPrice = scan.nextDouble();
               registerProduct(productId, productName, productPrice);
               break;
            case 2:
               System.out.println("Enter Retailer ID:");
               int retailerId = scan.nextInt();
               scan.nextLine(); // Consume newline
               System.out.println("Enter Retailer Name:");
               String retailerName = scan.nextLine();
               System.out.println("Enter Retailer Location:");
```

```
                String retailerLocation = scan.nextLine();
                registerRetailer(retailerName, retailerLocation);
                break;
            case 3:
                System.out.println("Enter Product ID to add quantity:");
                int addProductId = scan.nextInt();
                System.out.println("Enter Quantity to Add:");
                int addQuantity = scan.nextInt();
                scan.nextLine(); // Consume newline
                System.out.println("Enter Storage Location:");
                String addLocation = scan.nextLine();
                updateProductQuantity(addProductId, addQuantity, addLocation);
                break;
            case 4:
                System.out.println("Enter Product ID to remove quantity:");
                int removeProductId = scan.nextInt();
                System.out.println("Enter Quantity to Remove:");
                int removeQuantity = scan.nextInt();
                scan.nextLine(); // Consume newline
                System.out.println("Enter Storage Location:");
                String removeLocation = scan.nextLine();
                updateProductQuantity(removeProductId, removeQuantity, removeLocation);
                break;
            case 5:
                System.out.println("Exiting program...");
                exit = true;
                break;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    }
    scan.close();
}
t
    public void registerProduct(int id, String name, double price) {
        getInventoryInstance().registerProduct(new Product(id, name, price));
        System.out.println("Product registered successfully: " + name);
    }

    public void registerRetailer(String name, String location) {
        getInventoryInstance().registerRetailer(new BasicRetailer(name, location));
        System.out.println("Retailer registered successfully: " + name);
    }

    public void addOrder(int id) {
        boolean success = getStorageInsatnce().addOrder(new Order(id));
        if (success) {
            System.out.println("order added");
        } else {
            System.out.println("Failed to add product quantity. Check if the product or location exists.");
        }
    }

    public void updateProductQuantity(int productId, int quantity, String location) {
        boolean success = getStorageInsatnce().updateProductCount(productId, quantity);
        if (success) {
            System.out.println("updated " + quantity + " units of product ID " + productId + " from storage at " + location);
        } else {
            System.out.println("Failed to remove product quantity. Check if the product or location exists and has sufficient quantity.");
        }
    }

    // Singleton method to get the Inventory instance
    private InventoryManagement getInventoryInstance() {
        if (inventory == null) {
            inventory = new BasicInventManage();
        }
        return inventory;
    }
     // Singleton method to get the Storage  instance
```

```
    private Storage getStorageInsatnce()
    {
                if (store == null) {
            store= new BasicStorage();
        }
        return store;
    }
}

/**
 * @ Mani Raj
 */
public class BasicProductDescription implements ProductDescription{

                private String description;

                public BasicProductDescription(String description) {
                    this.description = description;
                }

                public String getDescription() {
                    return description;
                }
            }

/**
 * @ Mani Raj
 */
public class BasicRetailer implements Retailer{
            private String name;
            private String location;

            public BasicRetailer(String name, String location) {
                        this.name = name;
                        this.location = location;
            }
            public String getName() {
                        return name;
            }

            public String getLocation() {
                        return location;
            }
}

/**
 * @ Mani Raj
 */
public class BasicStorage implements Storage {
            private String location;
            private Map<Integer, Integer> availableProducts; // Product ID mapped to Quantity
            private List<Order> orderList;

            public BasicStorage() {

            }

            public BasicStorage(String location) {
                        this.location = location;
                        this.availableProducts = new HashMap<>();
                        this.orderList = new ArrayList<>();
            }

            @Override
            public Boolean addOrder(Order order) {
                        orderList.add(order);
                        return true;
            }

            @Override
```

```
	public Boolean updateProductCount(int productId, int quantityChange) {
		availableProducts.put(productId, availableProducts.getOrDefault(productId, 0) + quantityChange);
		return true;
	}
	public String getLocation() {
		return location;
	}

	public Map<Integer, Integer> getAvailableProducts() {
		return availableProducts;
	}
}
```

Modeling

```
package Modeling;

import HR.src.Employee;

import java.time.LocalDateTime;
import java.util.Scanner;

public class ModelingDepartment {
    HOD hod = new HOD();
    Manager manager = new Manager();

    public void runTasks() {
        System.out.println("Please choose an action you want to take: \n 1: haveEvent \n 2: scheduleFitting");
        Scanner s = new Scanner(System.in);
        int c = s.nextInt();

        switch (c){
            case 1:
                Boolean type = false;
                Boolean collab = false;
                System.out.println("What type of event?\n1:Photoshoot\n2:Fashion Show");
                int x = s.nextInt();
                switch (x) {
                    case 1:
                        type = true;
                }
                System.out.println("Type a name of a Celebrity (enter for none)");
                String celebrity = s.next();
                System.out.println("Will you collab with a brand? (Y/N)");
                String ch = s.next();
                switch (ch) {
                    case "Y":
                        collab = true;
                }

                hod.haveEvent(type, celebrity, collab);
            case 2:
                System.out.println("What model? (put a name)");
                String model = s.next();
                System.out.println("What month (1-12)");
                int month = s.nextInt();
                System.out.println("What day (1-31)");
                int day = s.nextInt();
                System.out.println("What hour? (1-24)");
                int hour = s.nextInt();
                LocalDateTime date = LocalDateTime.of(2024, month, day, hour, 0);

                Employee e = new Employee("1", model, "Modeling", "Model", "Employed");

                manager.requestFitting(e);
```

```
        }
    }
}


package Modeling;

import HR.src.Employee;

public interface IHOD {
    void haveEvent(Boolean type, String celebrity, Boolean collab);

    Boolean requestAdvertisement(Event event);

    Boolean requestContract(String celebrity);

    Boolean requestCollab(String brand);
}

class HOD implements IHOD {

    Manager manager = new Manager();

    @Override
    public void haveEvent(Boolean type, String celebrity, Boolean collab) {
        Employee[] models = manager.getModels();

        Event event = new Event();
        event.addEvent(1,models, type, celebrity, collab);
        requestAdvertisement(event);
    }

    @Override
    public Boolean requestAdvertisement(Event event) {
        return false;
    }

    @Override
    public Boolean requestContract(String celebrity) {
        return null;
    }

    @Override
    public Boolean requestCollab(String brand) {
        return null;
    }
}

package Modeling;

import HR.src.Employee;

import java.time.LocalDateTime;

public interface IFitting {
    Fitting addFitting(int id, Employee model, String garment, LocalDateTime date);

    void endFitting();
}

class Fitting implements IFitting {
    int id;
    Employee model;
    String garment;
    LocalDateTime date;
    Boolean completionStatus;

    @Override
    public Fitting addFitting(int id, Employee model, String garment, LocalDateTime date) {
        this.id = id;
```

```
        this.model = model;
        this.garment = garment;
        this.date = date;
        this.completionStatus = false;
        return null;
    }

    @Override
    public void endFitting() {
        this.completionStatus = true;
    }
}

package Modeling;

import HR.src.Employee;

public interface IEvent {
    Event addEvent(int id, Employee[] models, Boolean type, String celebrity, Boolean collab);
    void endEvent();
}

class Event implements IEvent {
    int id;
    Employee[] models;
    Boolean type; //true for photoshoot, false for fashion show
    String celebrity;
    Boolean collab;
    Boolean completionStatus;


    @Override
    public Event addEvent(int id, Employee[] models, Boolean type, String celebrity, Boolean collab) {
        this.id = id;
        this.models = models;
        this.type = type;
        this.celebrity = celebrity;
        this.collab = collab;
        this.completionStatus = false;

        return this;
    }

    @Override
    public void endEvent() {
        this.completionStatus = true;
    }
}

package Modeling;

import HR.src.Employee;

import java.time.LocalDateTime;
import java.util.Scanner;

public class ModelingDepartment {
    HOD hod = new HOD();
    Manager manager = new Manager();

    public void runTasks() {
        System.out.println("Please choose an action you want to take: \n 1: haveEvent \n 2: scheduleFitting");
        Scanner s = new Scanner(System.in);
        int c = s.nextInt();

        switch (c){
            case 1:
                Boolean type = false;
                Boolean collab = false;
                System.out.println("What type of event?\n1:Photoshoot\n2:Fashion Show");
```

```
            int x = s.nextInt();
            switch (x) {
               case 1:
                  type = true;
            }
            System.out.println("Type a name of a Celebrity (enter for none)");
            String celebrity = s.next();
            System.out.println("Will you collab with a brand? (Y/N)");
            String ch = s.next();
            switch (ch) {
               case "Y":
                  collab = true;
            }

            hod.haveEvent(type, celebrity, collab);
         case 2:
            System.out.println("What model? (put a name)");
            String model = s.next();
            System.out.println("What month (1-12)");
            int month = s.nextInt();
            System.out.println("What day (1-31)");
            int day = s.nextInt();
            System.out.println("What hour? (1-24)");
            int hour = s.nextInt();
            LocalDateTime date = LocalDateTime.of(2024, month, day, hour, 0);

            Employee e = new Employee("1", model, "Modeling", "Model", "Employed");

            manager.requestFitting(e, date);
      }
   }
}
```

Marketing

Main:

```
import Modeling.ModelingDepartment;

import java.util.Scanner;

public class StartingApplication {
   static ModelingDepartment modelingDepartment = new ModelingDepartment();
   PayrollController payrollController = new PayrollController();

   public static void main(String[] args) {
      Scanner s = new Scanner(System.in);

      System.out.println("--WELCOME TO FASHION EMPIRE--\n You are logged in as: IT SPECIALIST\n\nWhich Department would you like to go to?"
+
            "\n1: HR Department" +
            "\n2: Treasury" +
            "\n3: Manufacturing" +
            "\n4: Modeling" +
            "\n5: Inventory");
      int choice = s.nextInt();

      switch (choice){
         case 1:
            break;
```

```
            case 2:
                payrollController.run();
                break;
            case 3:
                break;
            case 4:
                modelingDepartment.runTasks();
                break;
            case 5:
                break;
        }

    }
}
```