Deliverable 1 Report
EECS 3311Z - Software Design

https://github.com/egeyesss/eecs3311-reservation

Student Name - Student Number - Github Name
Ege Yesilyurt - 219701739 - egeyesss
Adren Reuel Singh - 219191881- adrenreuel
Weiqin Situ - 219720432 - kevinsitu1706
Gurnoor Kahlon - 220596086 - gurnoor-kahlon
Marwan Al-Jaufy - 219253459 - maljaufy

Part I: Requirements Eliciting and Modeling **2**
Task 1: Use Case Diagram 3
Task 2: Activity Diagrams 4
Task 3: Sequence Diagrams 11
Part II: Design **17**
Task 1: Class Diagram 17
Part III: Report on AI-Assistant Adaptation **17**
Task 1: Explain the Role of the AI Assistant 17
Task 2: Arguments and Resolution 26
Task 3: Verifying AI Responses 27
Task 4: Application of AI-Suggested Content 27

Part I: Requirements Eliciting and Modeling

Actor 1: Registered User (Student/Faculty/Researcher combined)
Unique Responsibilities:

- Reserve Equipment - Search and book equipment with hourly fees and deposits (Req 3, 4)
- Modify/Cancel Reservation - Manage their own bookings (Req 8)
- Extend Reservation - Extend if equipment is available (Req 9)
- Make Payment - Pay for equipment usage (Req 10)

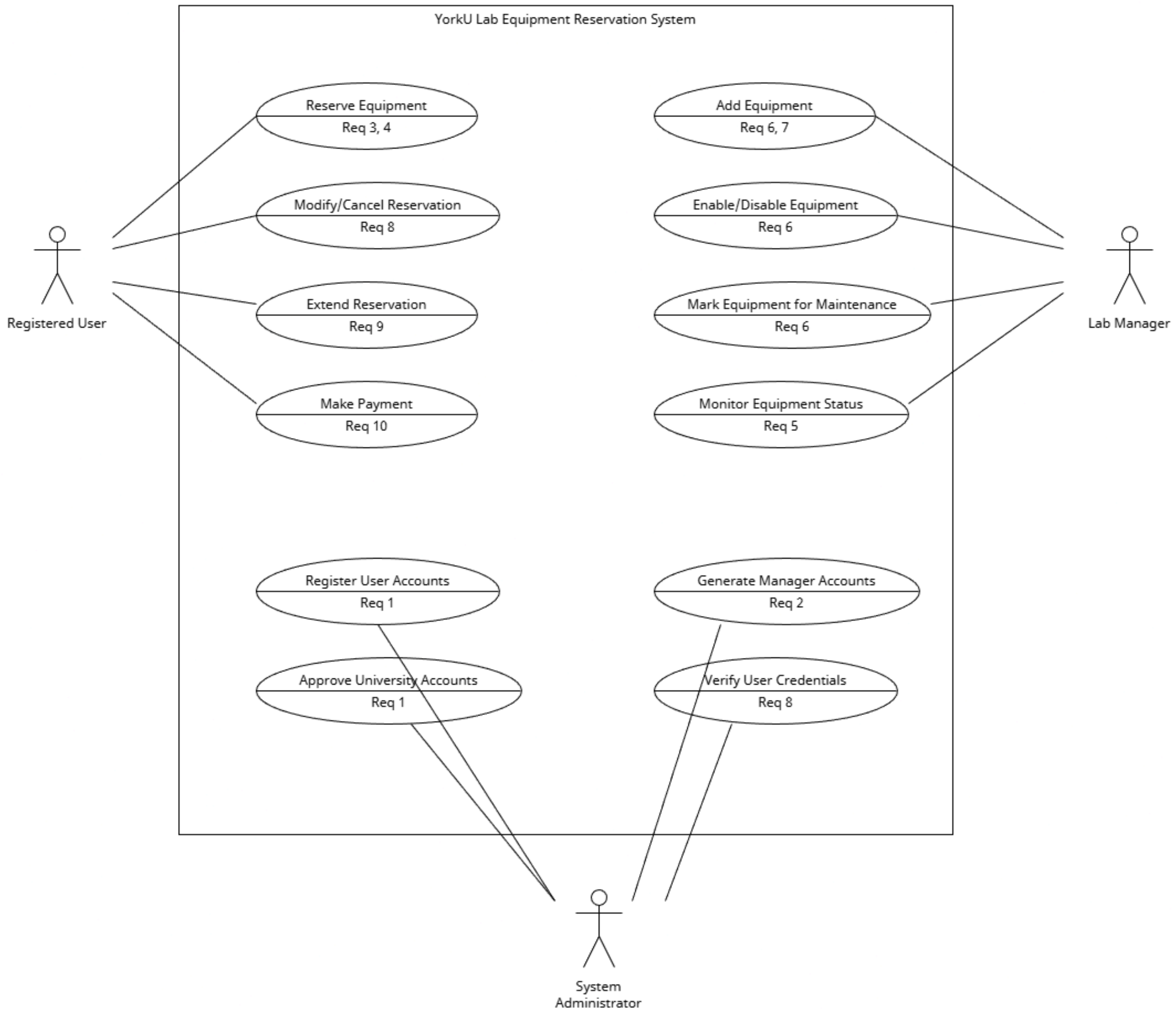
Actor 2: Lab Manager
Unique Responsibilities:

- Add Equipment - Add new equipment with unique ID, description, location (Req 6, 7)
- Enable/Disable Equipment - Control equipment availability (Req 6)
- Mark Equipment for Maintenance - Set equipment as unavailable (Req 6)
- Monitor Equipment Status - View sensor data and operation status (Req 5)

Actor 3: System Administrator (Head Lab Coordinator)
Unique Responsibilities:

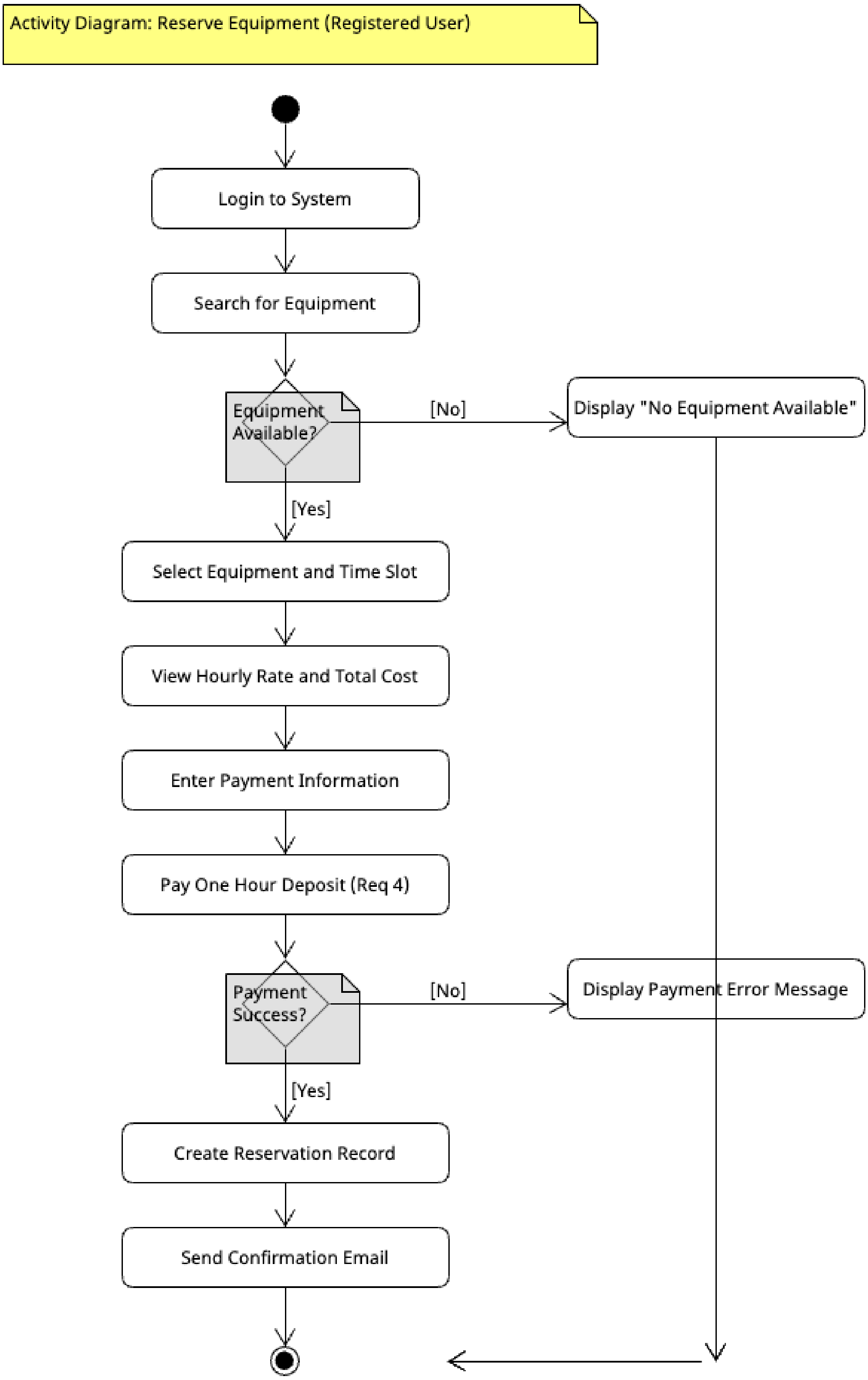
- Register User Accounts - Create accounts with email/password validation (Req 1)
- Approve University Accounts - Provide departmental approval for affiliated users (Req 1)
- Generate Manager Accounts - Auto-generate lab manager accounts (Req 2)
- Verify User Credentials - Validate student/staff IDs and certification numbers (Req 8)

Task 1: Use Case Diagram

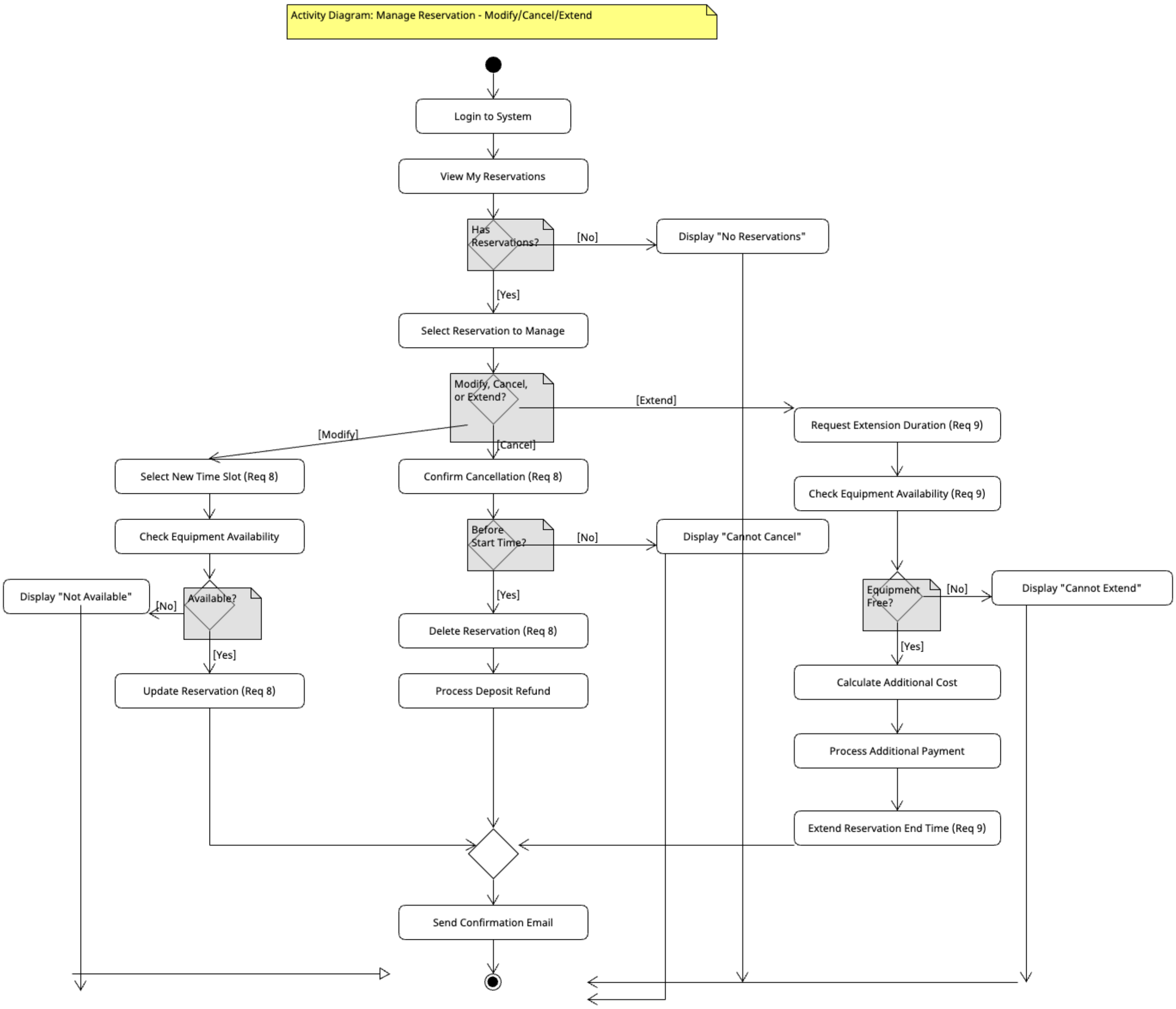


Task 2: Activity Diagrams

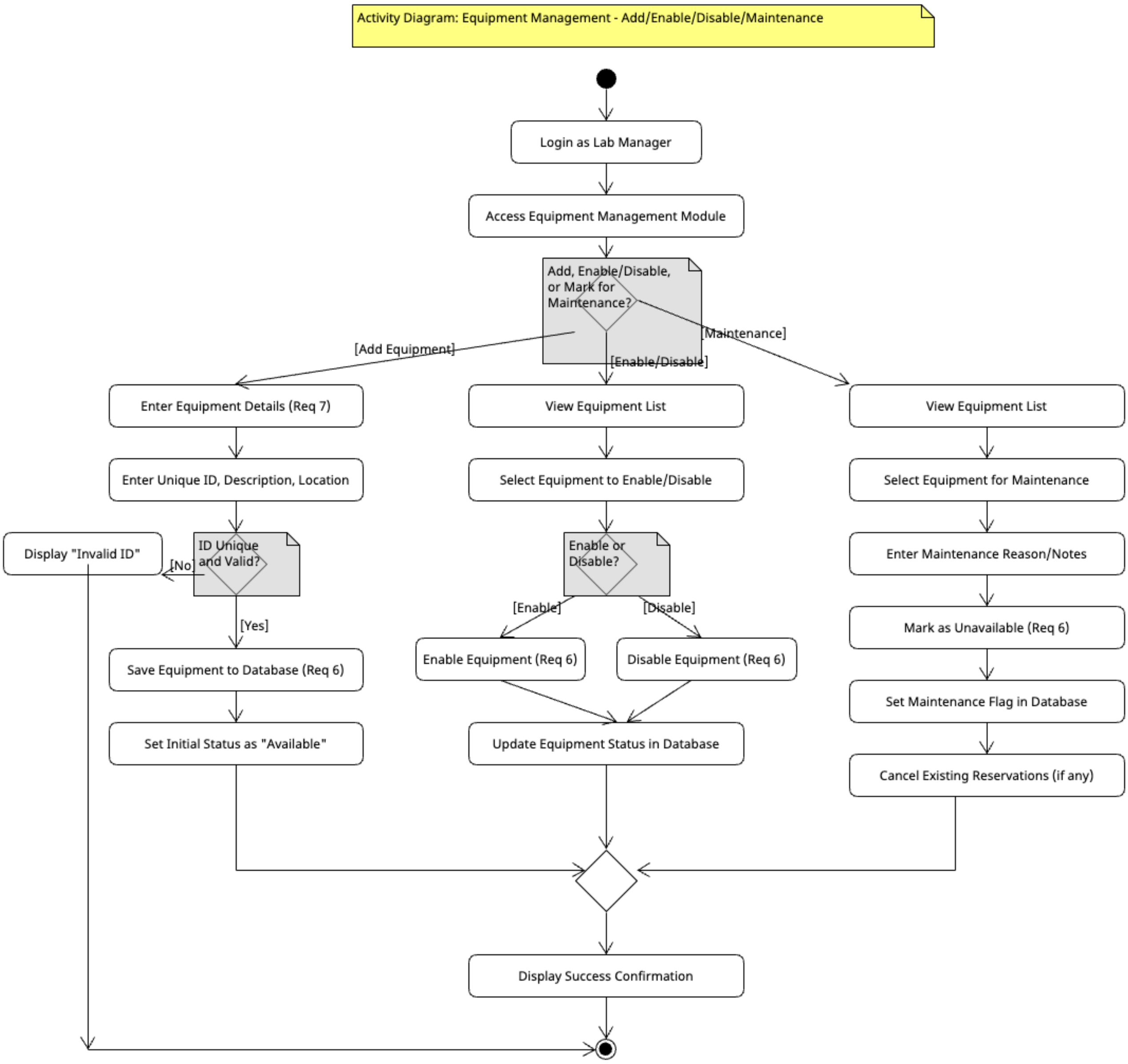
Actor 1: Registered User - Reserve Equipment and Payment


Activity Diagram: Reserve Equipment (Registered User)
Login to System
Search for Equipment
Equipment Available?
[No]
Display "No Equipment Available"
[Yes]
Select Equipment and Time Slot
View Hourly Rate and Total Cost
Enter Payment Information
Pay One Hour Deposit (Req 4)
Payment Success?
[No]
Display Payment Error Message
[Yes]
Create Reservation Record
Send Confirmation Email

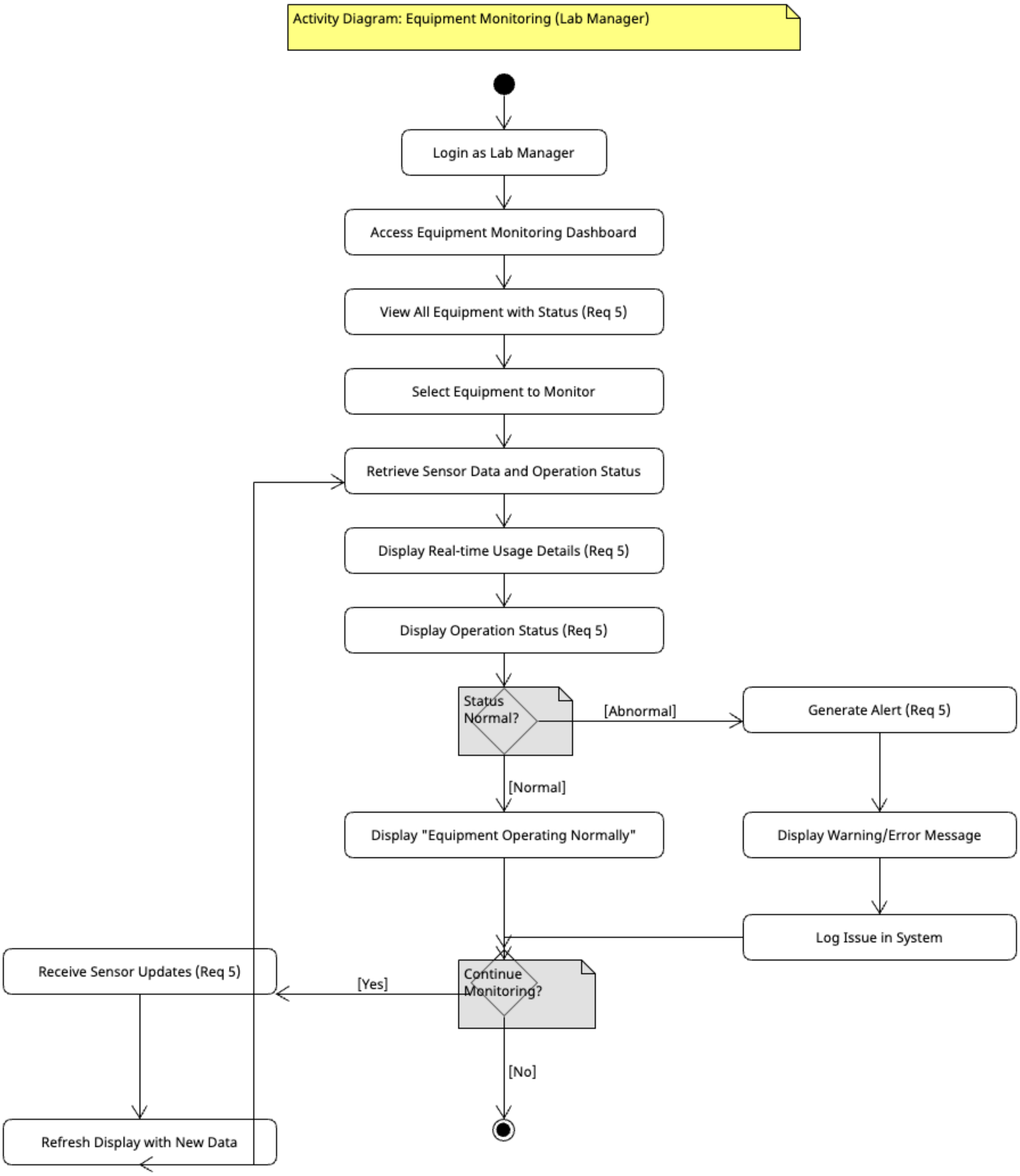
Actor 1: Registered User - Manage Reservation


Activity Diagram: Manage Reservation - Modify/Cancel/Extend
Login to System
View My Reservations
Has Reservations?
[No]
Display "No Reservations"
[Yes]
Select Reservation to Manage
Modify, Cancel, or Extend?
[Extend]
Request Extension Duration (Req 9)
[Modify]
[Cancel]
Select New Time Slot (Req 8)
Confirm Cancellation (Req 8)
Check Equipment Availability (Req 9)
Check Equipment Availability
Before Start Time?
[No]
Display "Cannot Cancel"
Display "Not Available"
[No]
Available?
[Yes]
Equipment Free?
[No]
Display "Cannot Extend"
[Yes]
Delete Reservation (Req 8)
[Yes]
Update Reservation (Req 8)
Process Deposit Refund
Calculate Additional Cost
Process Additional Payment
Extend Reservation End Time (Req 9)
Send Confirmation Email

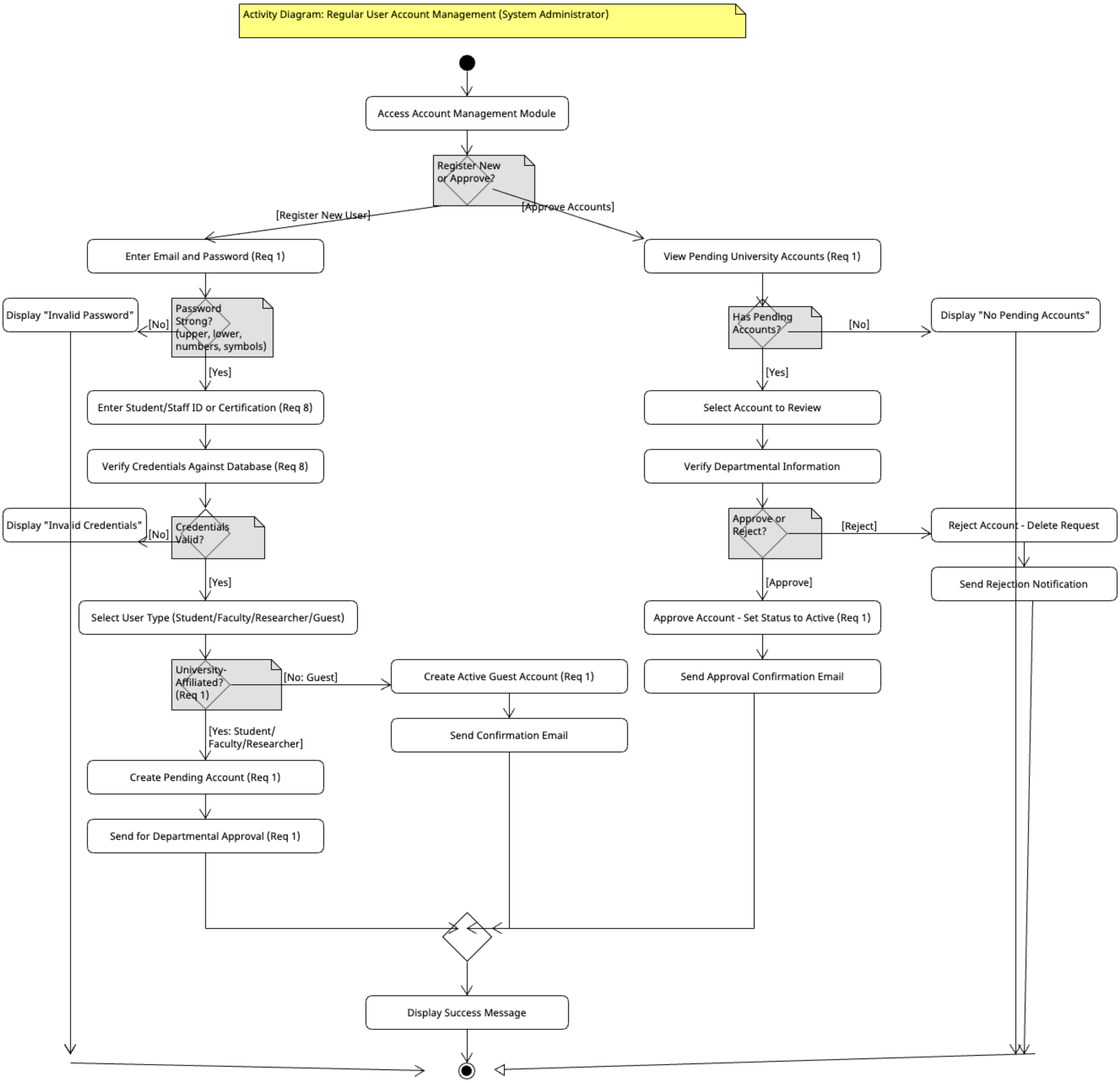
Actor 2: Lab Manager - Equipment Management


Activity Diagram: Equipment Management - Add/Enable/Disable/Maintenance
Login as Lab Manager
Access Equipment Management Module
Add, Enable/Disable, or Mark for Maintenance?
[Add Equipment]
[Enable/Disable]
[Maintenance]
Enter Equipment Details (Req 7)
Enter Unique ID, Description, Location
ID Unique and Valid?
[No]
Display "Invalid ID"
[Yes]
Save Equipment to Database (Req 6)
Set Initial Status as "Available"
View Equipment List
Select Equipment to Enable/Disable
Enable or Disable?
[Enable]
[Disable]
Enable Equipment (Req 6)
Disable Equipment (Req 6)
Update Equipment Status in Database
View Equipment List
Select Equipment for Maintenance
Enter Maintenance Reason/Notes
Mark as Unavailable (Req 6)
Set Maintenance Flag in Database
Cancel Existing Reservations (if any)
Display Success Confirmation

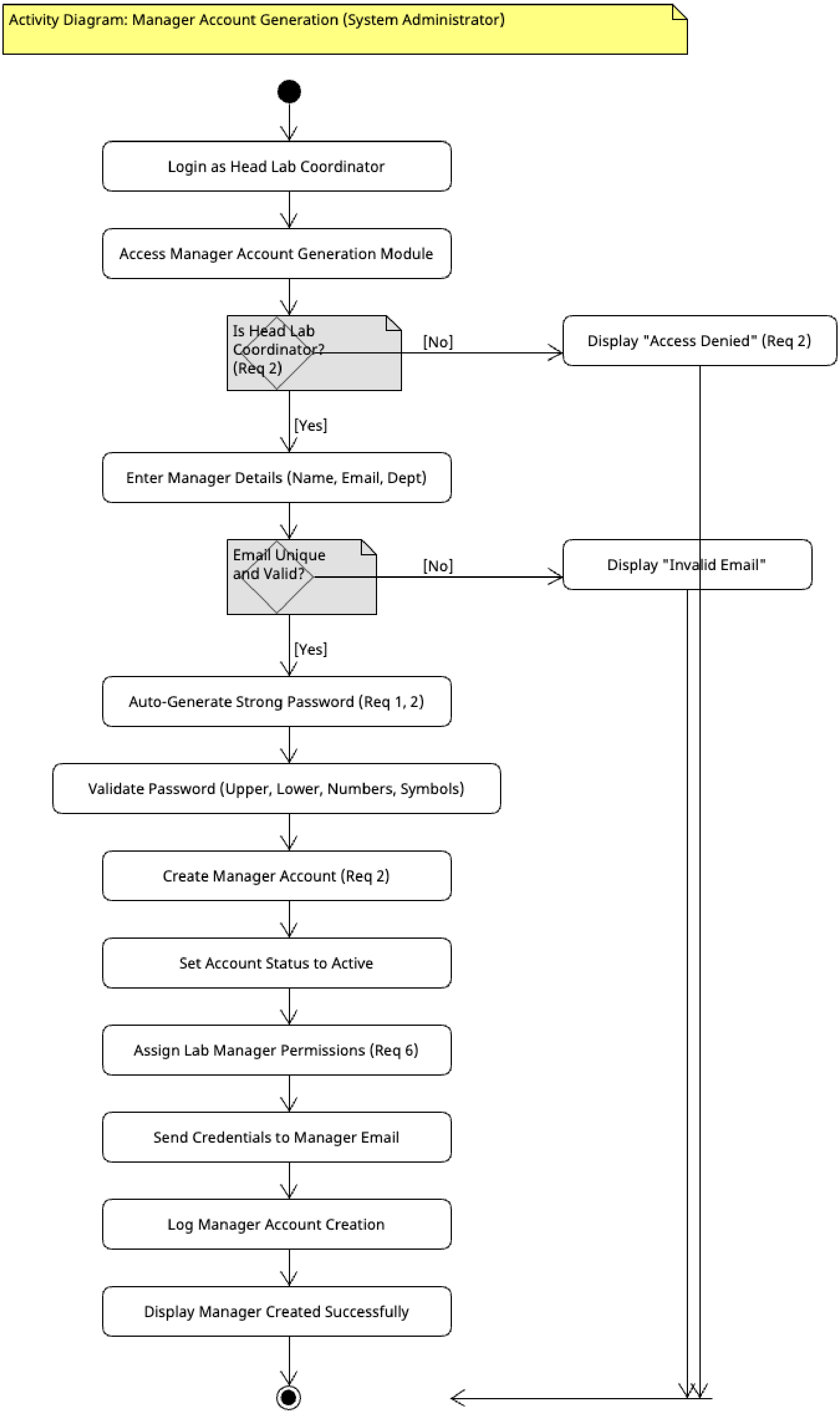
Actor 2: Lab Manager - Equipment Monitoring


Activity Diagram: Equipment Monitoring (Lab Manager)
Login as Lab Manager
Access Equipment Monitoring Dashboard
View All Equipment with Status (Req 5)
Select Equipment to Monitor
Retrieve Sensor Data and Operation Status
Display Real-time Usage Details (Req 5)
Display Operation Status (Req 5)
Status Normal?
[Abnormal]
Generate Alert (Req 5)
[Normal]
Display "Equipment Operating Normally"
Display Warning/Error Message
Log Issue in System
Continue Monitoring?
[Yes]
Receive Sensor Updates (Req 5)
Refresh Display with New Data
[No]

Actor 3: System Administrator - Account Management

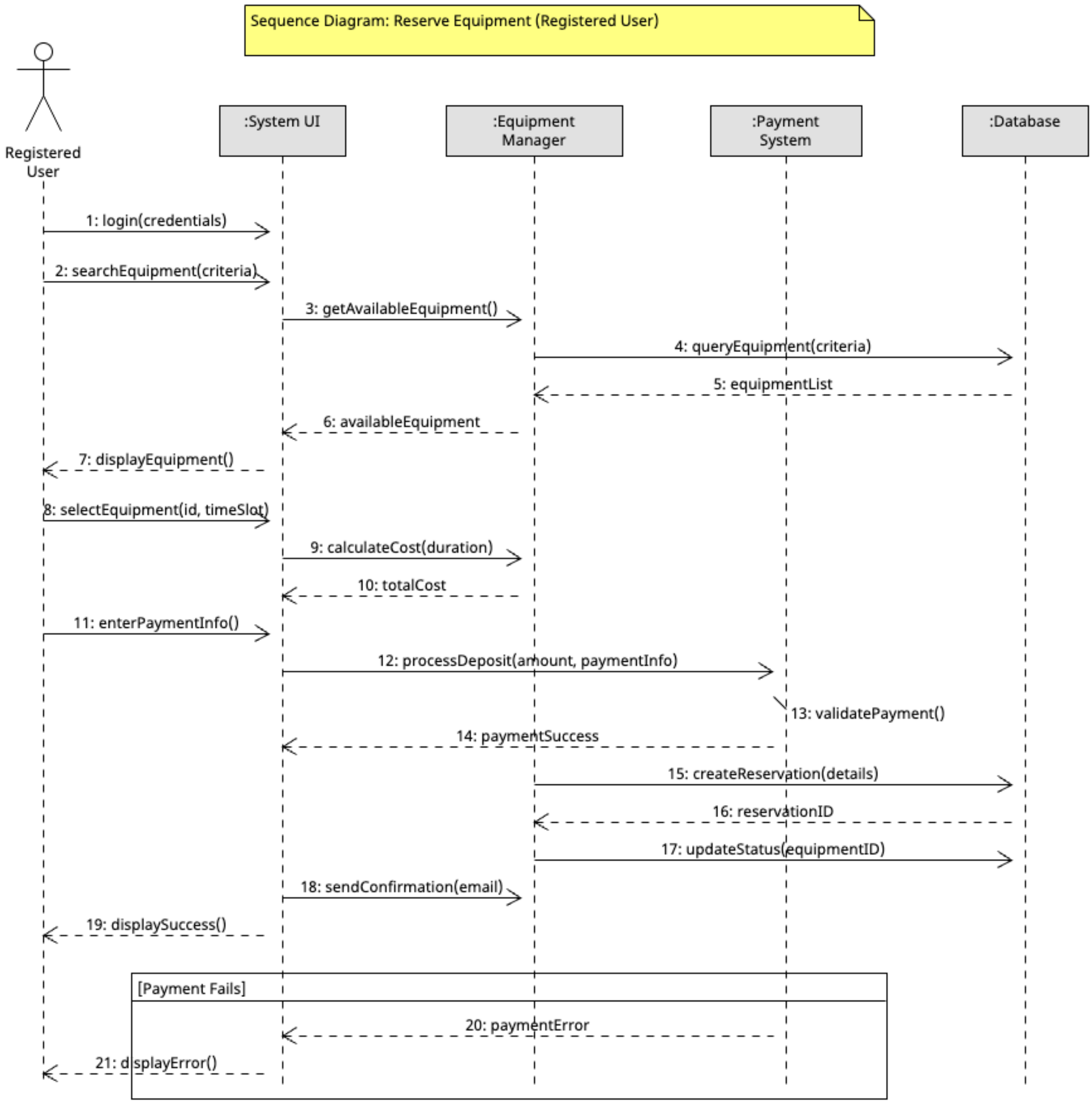

Activity Diagram: Regular User Account Management (System Administrator)
Access Account Management Module
Register New or Approve?
[Register New User]
[Approve Accounts]
Enter Email and Password (Req 1)
Password Strong? (upper, lower, numbers, symbols)
[No]
Display "Invalid Password"
[Yes]
Enter Student/Staff ID or Certification (Req 8)
Verify Credentials Against Database (Req 8)
Credentials Valid?
[No]
Display "Invalid Credentials"
[Yes]
Select User Type (Student/Faculty/Researcher/Guest)
University-Affiliated? (Req 1)
[No: Guest]
Create Active Guest Account (Req 1)
Send Confirmation Email
[Yes: Student/ Faculty/Researcher]
Create Pending Account (Req 1)
Send for Departmental Approval (Req 1)
View Pending University Accounts (Req 1)
Has Pending Accounts?
[No]
Display "No Pending Accounts"
[Yes]
Select Account to Review
Verify Departmental Information
Approve or Reject?
[Reject]
Reject Account - Delete Request
Send Rejection Notification
[Approve]
Approve Account - Set Status to Active (Req 1)
Send Approval Confirmation Email
Display Success Message

Actor 3: System Administrator - Creating Manager Accounts

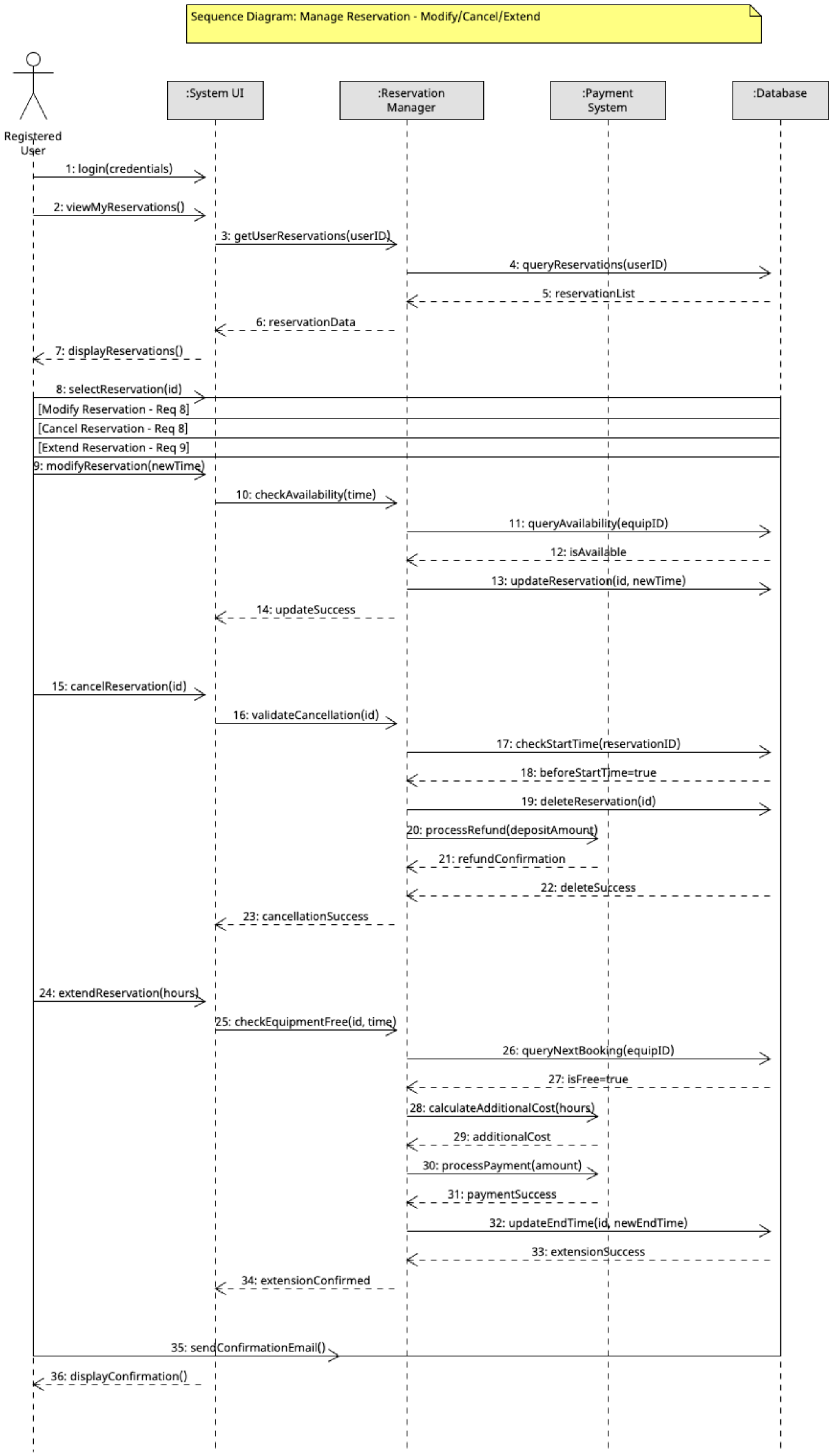

Activity Diagram: Manager Account Generation (System Administrator)
Login as Head Lab Coordinator
Access Manager Account Generation Module
Is Head Lab Coordinator? (Req 2)
[No]
Display "Access Denied" (Req 2)
[Yes]
Enter Manager Details (Name, Email, Dept)
Email Unique and Valid?
[No]
Display "Invalid Email"
[Yes]
Auto-Generate Strong Password (Req 1, 2)
Validate Password (Upper, Lower, Numbers, Symbols)
Create Manager Account (Req 2)
Set Account Status to Active
Assign Lab Manager Permissions (Req 6)
Send Credentials to Manager Email
Log Manager Account Creation
Display Manager Created Successfully

Task 3: Sequence Diagrams

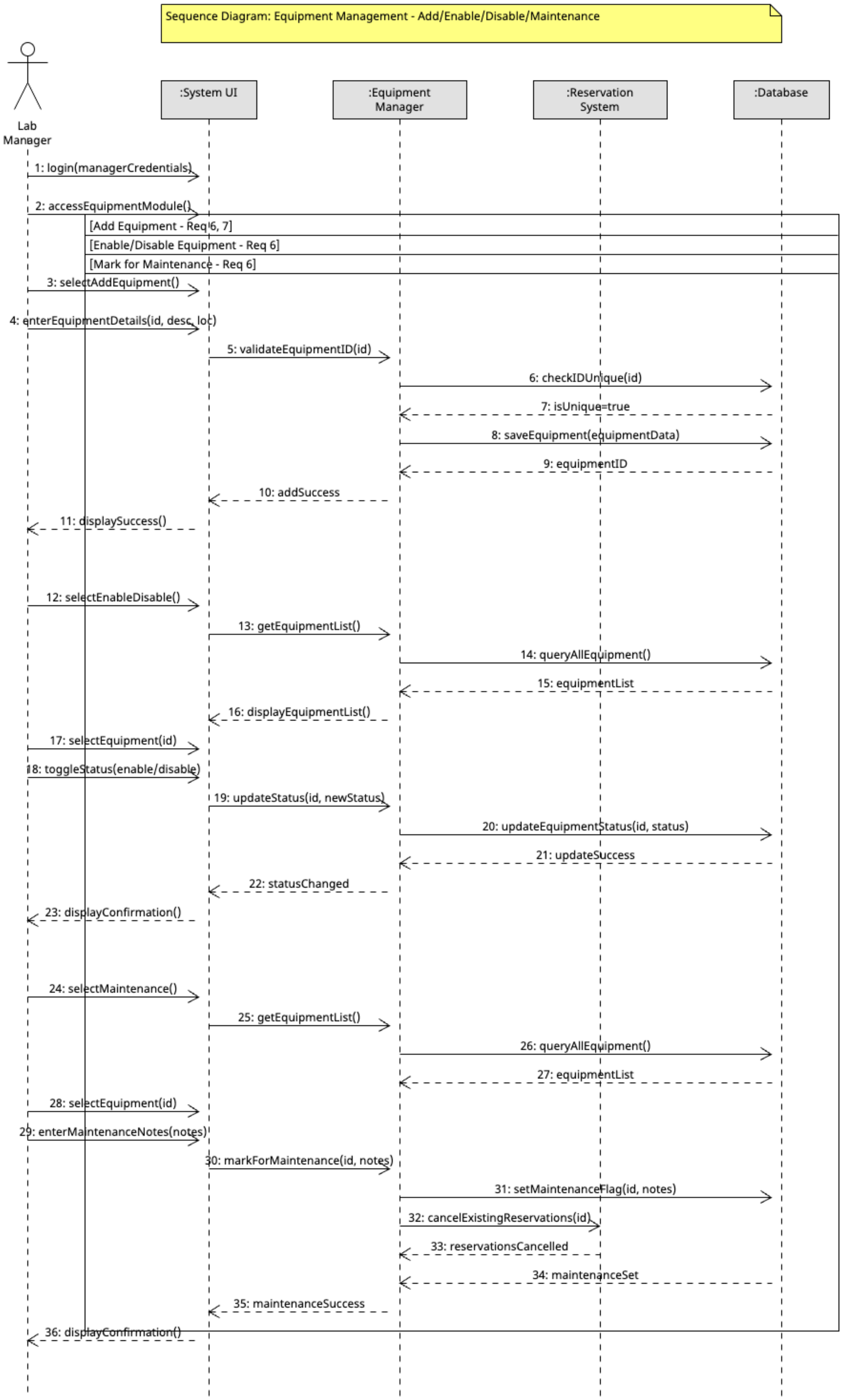
Actor 1: Registered User - Reserve Equipment and Payment


Sequence Diagram: Reserve Equipment (Registered User)
Registered User
:System UI
:Equipment Manager
:Payment System
:Database
1: login(credentials)
2: searchEquipment(criteria)
3: getAvailableEquipment()
4: queryEquipment(criteria)
5: equipmentList
6: availableEquipment
7: displayEquipment()
8: selectEquipment(id, timeSlot)
9: calculateCost(duration)
10: totalCost
11: enterPaymentInfo()
12: processDeposit(amount, paymentInfo)
13: validatePayment()
14: paymentSuccess
15: createReservation(details)
16: reservationID
17: updateStatus(equipmentID)
18: sendConfirmation(email)
19: displaySuccess()
[Payment Fails]
20: paymentError
21: displayError()

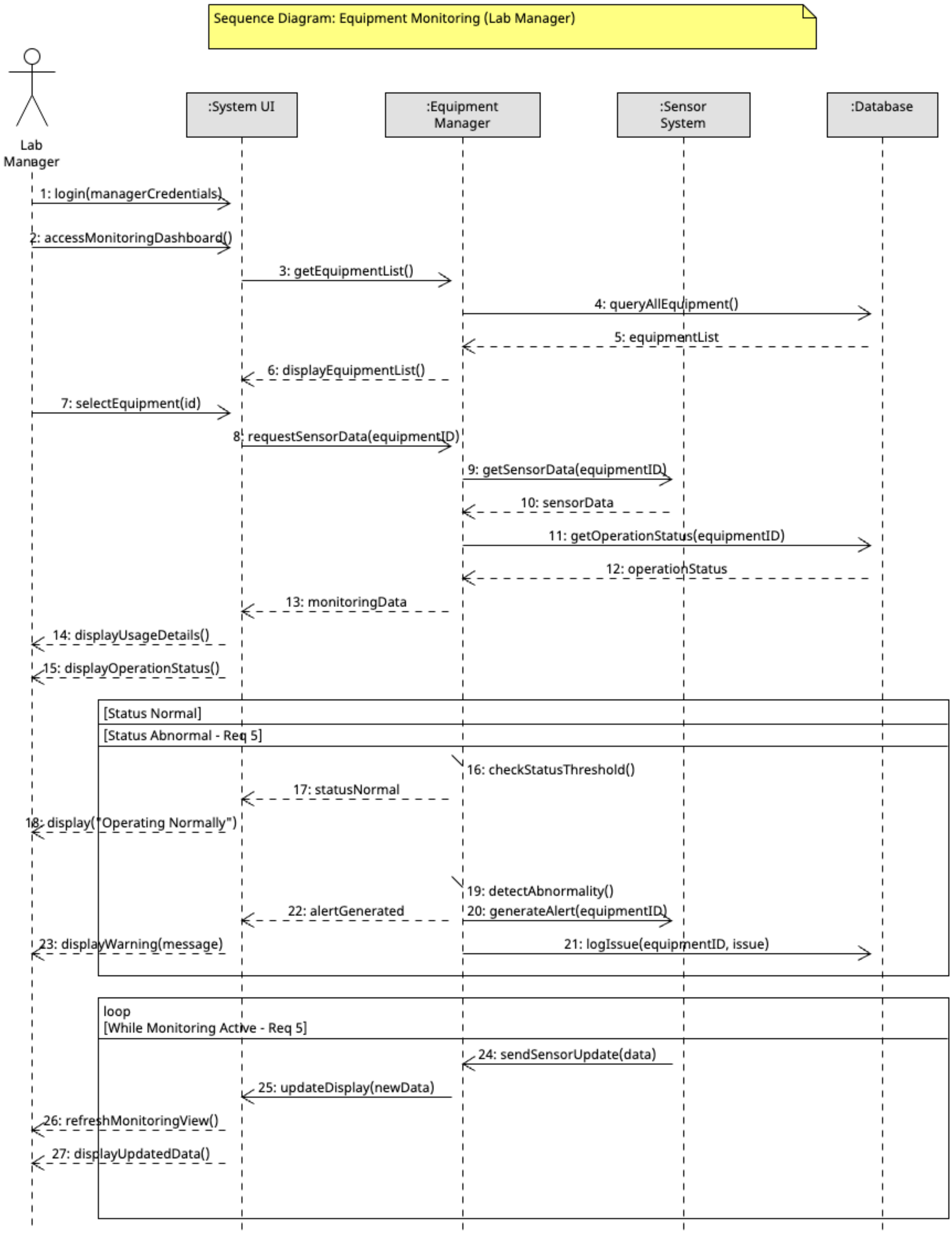
Actor 1: Registered User - Manage Reservation


Sequence Diagram: Manage Reservation - Modify/Cancel/Extend
Registered User
:System UI
:Reservation Manager
:Payment System
:Database
1: login(credentials)
2: viewMyReservations()
3: getUserReservations(userID)
4: queryReservations(userID)
5: reservationList
6: reservationData
7: displayReservations()
8: selectReservation(id)
[Modify Reservation - Req 8]
[Cancel Reservation - Req 8]
[Extend Reservation - Req 9]
9: modifyReservation(newTime)
10: checkAvailability(time)
11: queryAvailability(equipID)
12: isAvailable
13: updateReservation(id, newTime)
14: updateSuccess
15: cancelReservation(id)
16: validateCancellation(id)
17: checkStartTime(reservationID)
18: beforeStartTime=true
19: deleteReservation(id)
20: processRefund(depositAmount)
21: refundConfirmation
22: deleteSuccess
23: cancellationSuccess
24: extendReservation(hours)
25: checkEquipmentFree(id, time)
26: queryNextBooking(equipID)
27: isFree=true
28: calculateAdditionalCost(hours)
29: additionalCost
30: processPayment(amount)
31: paymentSuccess
32: updateEndTime(id, newEndTime)
33: extensionSuccess
34: extensionConfirmed
35: sendConfirmationEmail()
36: displayConfirmation()

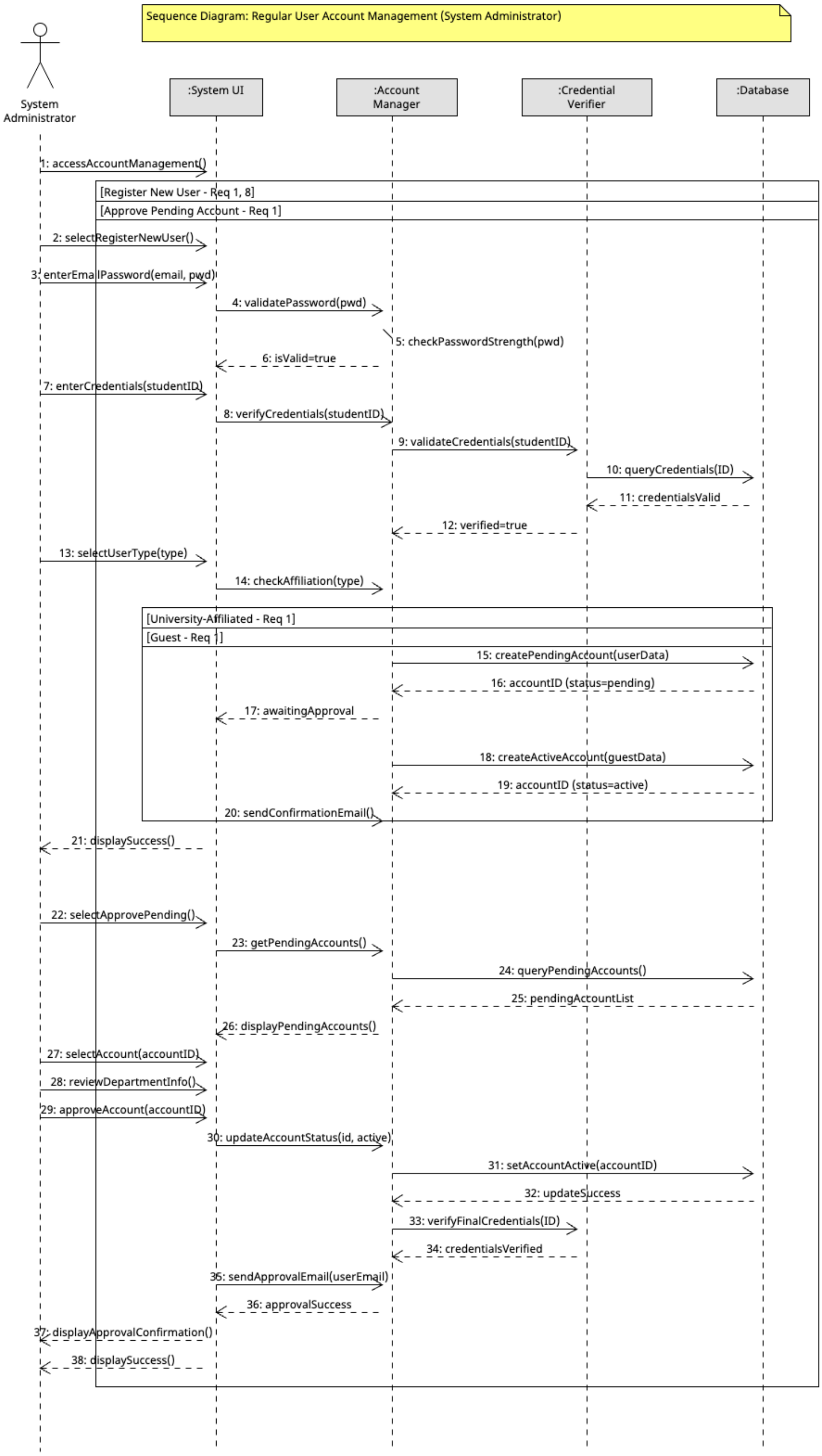
Actor 2: Lab Manager - Equipment Management


Sequence Diagram: Equipment Management - Add/Enable/Disable/Maintenance
Lab Manager
:System UI
:Equipment Manager
:Reservation System
:Database
1: login(managerCredentials)
2: accessEquipmentModule()
[Add Equipment - Req 6, 7]
[Enable/Disable Equipment - Req 6]
[Mark for Maintenance - Req 6]
3: selectAddEquipment()
4: enterEquipmentDetails(id, desc, loc)
5: validateEquipmentID(id)
6: checkIDUnique(id)
7: isUnique=true
8: saveEquipment(equipmentData)
9: equipmentID
10: addSuccess
11: displaySuccess()
12: selectEnableDisable()
13: getEquipmentList()
14: queryAllEquipment()
15: equipmentList
16: displayEquipmentList()
17: selectEquipment(id)
18: toggleStatus(enable/disable)
19: updateStatus(id, newStatus)
20: updateEquipmentStatus(id, status)
21: updateSuccess
22: statusChanged
23: displayConfirmation()
24: selectMaintenance()
25: getEquipmentList()
26: queryAllEquipment()
27: equipmentList
28: selectEquipment(id)
29: enterMaintenanceNotes(notes)
30: markForMaintenance(id, notes)
31: setMaintenanceFlag(id, notes)
32: cancelExistingReservations(id)
33: reservationsCancelled
34: maintenanceSet
35: maintenanceSuccess
36: displayConfirmation()

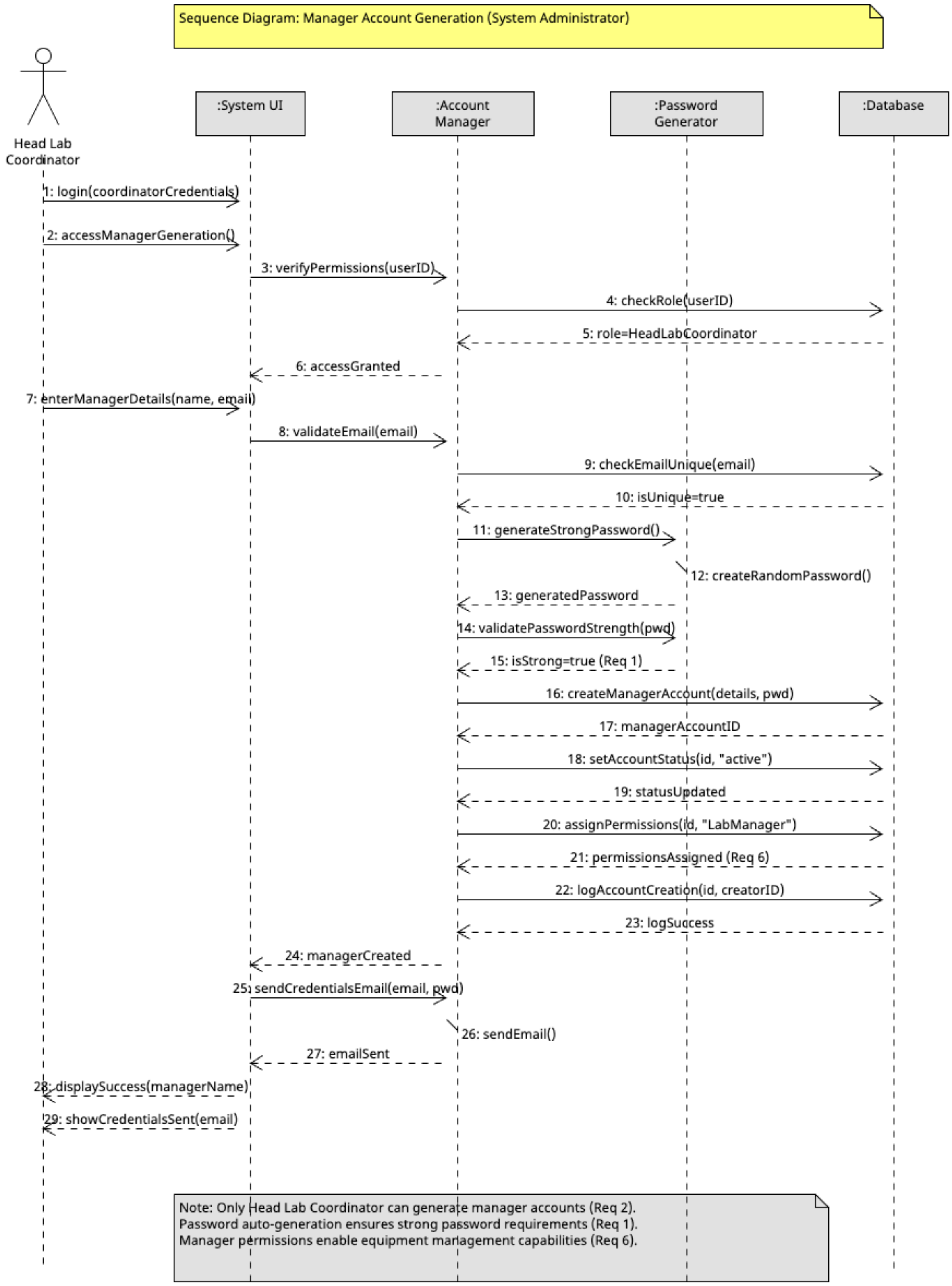
Actor 2: Lab Manager - Equipment Monitoring


Sequence Diagram: Equipment Monitoring (Lab Manager)
Lab Manager
:System UI
:Equipment Manager
:Sensor System
:Database
1: login(managerCredentials)
2: accessMonitoringDashboard()
3: getEquipmentList()
4: queryAllEquipment()
5: equipmentList
6: displayEquipmentList()
7: selectEquipment(id)
8: requestSensorData(equipmentID)
9: getSensorData(equipmentID)
10: sensorData
11: getOperationStatus(equipmentID)
12: operationStatus
13: monitoringData
14: displayUsageDetails()
15: displayOperationStatus()
[Status Normal]
[Status Abnormal - Req 5]
16: checkStatusThreshold()
17: statusNormal
18: display("Operating Normally")
19: detectAbnormality()
20: generateAlert(equipmentID)
22: alertGenerated
21: logIssue(equipmentID, issue)
23: displayWarning(message)
loop
[While Monitoring Active - Req 5]
24: sendSensorUpdate(data)
25: updateDisplay(newData)
26: refreshMonitoringView()
27: displayUpdatedData()

Actor 3: System Administrator - Account Management


Sequence Diagram: Regular User Account Management (System Administrator)
System Administrator
:System UI
:Account Manager
:Credential Verifier
:Database
1: accessAccountManagement()
[Register New User - Req 1, 8]
[Approve Pending Account - Req 1]
2: selectRegisterNewUser()
3: enterEmailPassword(email, pwd)
4: validatePassword(pwd)
5: checkPasswordStrength(pwd)
6: isValid=true
7: enterCredentials(studentID)
8: verifyCredentials(studentID)
9: validateCredentials(studentID)
10: queryCredentials(ID)
11: credentialsValid
12: verified=true
13: selectUserType(type)
14: checkAffiliation(type)
[University-Affiliated - Req 1]
[Guest - Req 1]
15: createPendingAccount(userData)
16: accountID (status=pending)
17: awaitingApproval
18: createActiveAccount(guestData)
19: accountID (status=active)
20: sendConfirmationEmail()
21: displaySuccess()
22: selectApprovePending()
23: getPendingAccounts()
24: queryPendingAccounts()
25: pendingAccountList
26: displayPendingAccounts()
27: selectAccount(accountID)
28: reviewDepartmentInfo()
29: approveAccount(accountID)
30: updateAccountStatus(id, active)
31: setAccountActive(accountID)
32: updateSuccess
33: verifyFinalCredentials(ID)
34: credentialsVerified
35: sendApprovalEmail(userEmail)
36: approvalSuccess
37: displayApprovalConfirmation()
38: displaySuccess()

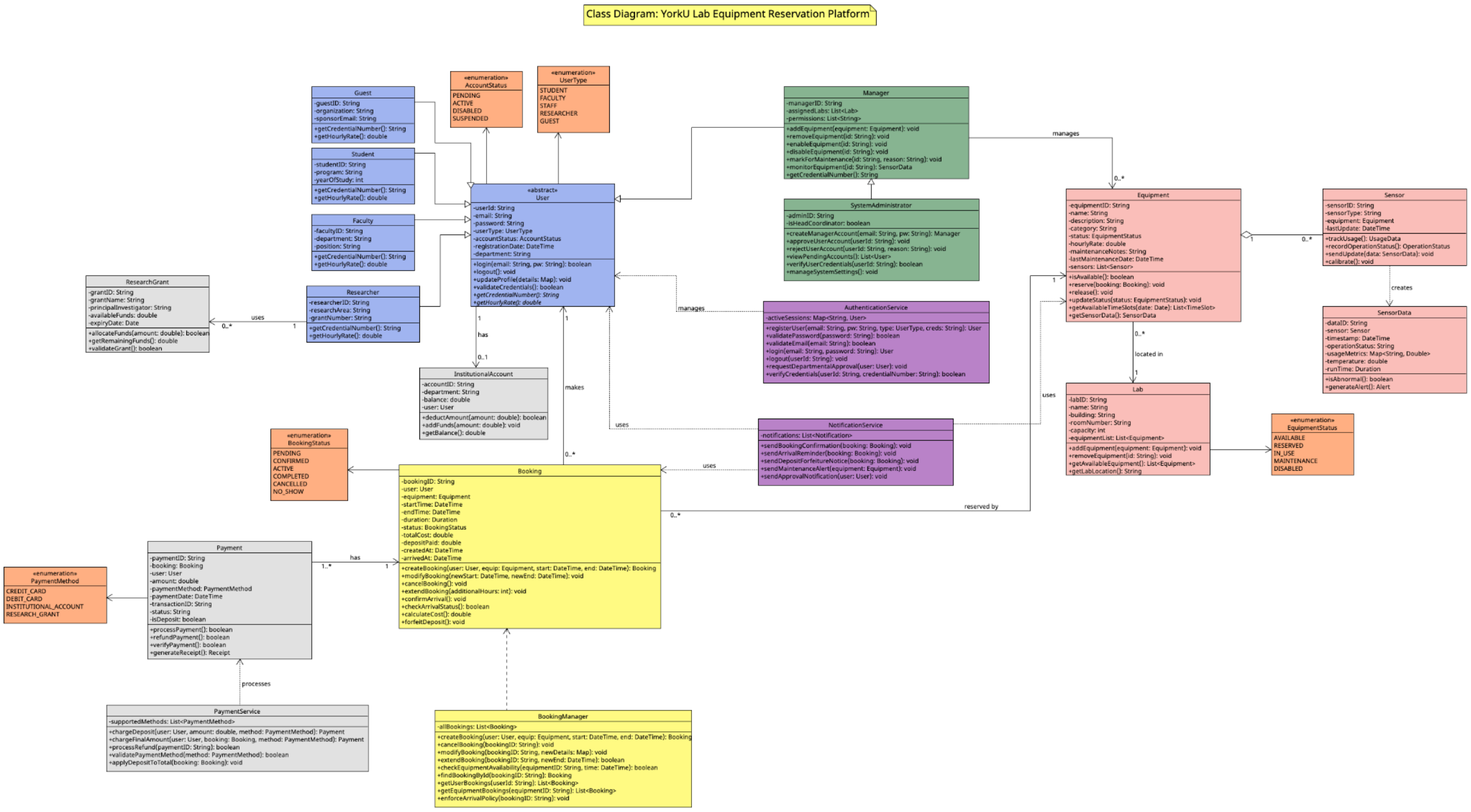
Actor 3: System Administrator - Creating Manager Accounts


Sequence Diagram: Manager Account Generation (System Administrator)
Head Lab Coordinator
:System UI
:Account Manager
:Password Generator
:Database
1: login(coordinatorCredentials)
2: accessManagerGeneration()
3: verifyPermissions(userID)
4: checkRole(userID)
5: role=HeadLabCoordinator
6: accessGranted
7: enterManagerDetails(name, email)
8: validateEmail(email)
9: checkEmailUnique(email)
10: isUnique=true
11: generateStrongPassword()
12: createRandomPassword()
13: generatedPassword
14: validatePasswordStrength(pwd)
15: isStrong=true (Req 1)
16: createManagerAccount(details, pwd)
17: managerAccountID
18: setAccountStatus(id, "active")
19: statusUpdated
20: assignPermissions(id, "LabManager")
21: permissionsAssigned (Req 6)
22: logAccountCreation(id, creatorID)
23: logSuccess
24: managerCreated
25: sendCredentialsEmail(email, pwd)
26: sendEmail()
27: emailSent
28: displaySuccess(managerName)
29: showCredentialsSent(email)

Note: Only Head Lab Coordinator can generate manager accounts (Req 2).
Password auto-generation ensures strong password requirements (Req 1).
Manager permissions enable equipment management capabilities (Req 6).

Part II: Design

Task 1: Class Diagram


Class Diagram: YorkU Lab Equipment Reservation Platform
Guest
-guestID: String
-organization: String
-sponsorEmail: String
+getCredentialNumber(): String
+getHourlyRate(): double
«enumeration»
AccountStatus
PENDING
ACTIVE
DISABLED
SUSPENDED
«enumeration»
UserType
STUDENT
FACULTY
STAFF
RESEARCHER
GUEST
Student
-studentID: String
-program: String
-yearOfStudy: int
+getCredentialNumber(): String
+getHourlyRate(): double
Faculty
-facultyID: String
-department: String
-position: String
+getCredentialNumber(): String
+getHourlyRate(): double
«abstract»
User
-userId: String
-email: String
-password: String
-userType: UserType
-accountStatus: AccountStatus
-registrationDate: DateTime
-department: String
+login(email: String, pw: String): boolean
+logout(): void
+updateProfile(details: Map): void
+validateCredentials(): boolean
+getCredentialNumber(): String
+getHourlyRate(): double
ResearchGrant
-grantID: String
-grantName: String
-principalInvestigator: String
-availableFunds: double
-expiryDate: Date
+allocateFunds(amount: double): boolean
+getRemainingFunds(): double
+validateGrant(): boolean
uses
0..*
1
Researcher
-researcherID: String
-researchArea: String
-grantNumber: String
+getCredentialNumber(): String
+getHourlyRate(): double
1
has
0..1
InstitutionalAccount
-accountID: String
-department: String
-balance: double
-user: User
+deductAmount(amount: double): boolean
+addFunds(amount: double): void
+getBalance(): double
1
makes
0..*
uses
Manager
-managerID: String
-assignedLabs: List<Lab>
-permissions: List<String>
+addEquipment(equipment: Equipment): void
+removeEquipment(id: String): void
+enableEquipment(id: String): void
+disableEquipment(id: String): void
+markForMaintenance(id: String, reason: String): void
+monitorEquipment(id: String): SensorData
+getCredentialNumber(): String
manages
0..*
SystemAdministrator
-adminID: String
-isHeadCoordinator: boolean
+createManagerAccount(email: String, pw: String): Manager
+approveUserAccount(userId: String): void
+rejectUserAccount(userId: String, reason: String): void
+viewPendingAccounts(): List<User>
+verifyUserCredentials(userId: String): boolean
+manageSystemSettings(): void
manages
AuthenticationService
-activeSessions: Map<String, User>
+registerUser(email: String, pw: String, type: UserType, creds: String): User
+validatePassword(password: String): boolean
+validateEmail(email: String): boolean
+login(email: String, password: String): User
+logout(userId: String): void
+requestDepartmentalApproval(user: User): void
+verifyCredentials(userId: String, credentialNumber: String): boolean
Equipment
-equipmentID: String
-name: String
-description: String
-category: String
-status: EquipmentStatus
-hourlyRate: double
-maintenanceNotes: String
-lastMaintenanceDate: DateTime
-sensors: List<Sensor>
+isAvailable(): boolean
+reserve(booking: Booking): void
+release(): void
+updateStatus(status: EquipmentStatus): void
+getAvailableTimeSlots(date: Date): List<TimeSlot>
+getSensorData(): SensorData
1
0..*
Sensor
-sensorID: String
-sensorType: String
-equipment: Equipment
-lastUpdate: DateTime
+trackUsage(): UsageData
+recordOperationStatus(): OperationStatus
+sendUpdate(data: SensorData): void
+calibrate(): void
creates
SensorData
-dataID: String
-sensor: Sensor
-timestamp: DateTime
-operationStatus: String
-usageMetrics: Map<String, Double>
-temperature: double
-runTime: Duration
+isAbnormal(): boolean
+generateAlert(): Alert
1
0..*
located in
1
uses
Lab
-labID: String
-name: String
-building: String
-roomNumber: String
-capacity: int
-equipmentList: List<Equipment>
+addEquipment(equipment: Equipment): void
+removeEquipment(id: String): void
+getAvailableEquipment(): List<Equipment>
+getLabLocation(): String
«enumeration»
EquipmentStatus
AVAILABLE
RESERVED
IN_USE
MAINTENANCE
DISABLED
NotificationService
-notifications: List<Notification>
+sendBookingConfirmation(booking: Booking): void
+sendArrivalReminder(booking: Booking): void
+sendDepositForfeitureNotice(booking: Booking): void
+sendMaintenanceAlert(equipment: Equipment): void
+sendApprovalNotification(user: User): void
uses
uses
reserved by
0..*
«enumeration»
BookingStatus
PENDING
CONFIRMED
ACTIVE
COMPLETED
CANCELLED
NO_SHOW
Booking
-bookingID: String
-user: User
-equipment: Equipment
-startTime: DateTime
-endTime: DateTime
-duration: Duration
-status: BookingStatus
-totalCost: double
-depositPaid: double
-createdAt: DateTime
-arrivedAt: DateTime
+createBooking(user: User, equip: Equipment, start: DateTime, end: DateTime): Booking
+modifyBooking(newStart: DateTime, newEnd: DateTime): void
+cancelBooking(): void
+extendBooking(additionalHours: int): void
+confirmArrival(): void
+checkArrivalStatus(): boolean
+calculateCost(): double
+forfeitDeposit(): void
Payment
-paymentID: String
-booking: Booking
-user: User
-amount: double
-paymentMethod: PaymentMethod
-paymentDate: DateTime
-transactionID: String
-status: String
-isDeposit: boolean
+processPayment(): boolean
+refundPayment(): boolean
+verifyPayment(): boolean
+generateReceipt(): Receipt
has
1..*
1
«enumeration»
PaymentMethod
CREDIT_CARD
DEBIT_CARD
INSTITUTIONAL_ACCOUNT
RESEARCH_GRANT
processes
PaymentService
-supportedMethods: List<PaymentMethod>
+chargeDeposit(user: User, amount: double, method: PaymentMethod): Payment
+chargeFinalAmount(user: User, booking: Booking, method: PaymentMethod): Payment
+processRefund(paymentID: String): boolean
+validatePaymentMethod(method: PaymentMethod): boolean
+applyDepositToTotal(booking: Booking): void
BookingManager
-allBookings: List<Booking>
+createBooking(user: User, equip: Equipment, start: DateTime, end: DateTime): Booking
+cancelBooking(bookingID: String): void
+modifyBooking(bookingID: String, newDetails: Map): void
+extendBooking(bookingID: String, newEnd: DateTime): boolean
+checkEquipmentAvailability(equipmentID: String, time: DateTime): boolean
+findBookingById(bookingID: String): Booking
+getUserBookings(userId: String): List<Booking>
+getEquipmentBookings(equipmentID: String): List<Booking>
+enforceArrivalPolicy(bookingID: String): void

Note: Class Diagram is hard to read as it is here so I will be uploading a separate PDF and a PNG of this class diagram for easier view.

Part III: Report on AI-Assistant Adaptation

Task 1: Explain the Role of the AI Assistant

Throughout this deliverable, We primarily used Claude Sonnet 4.5 through Perplexity AI as the AI assistant to help clarify design decisions and validate our approach to the requirements. We want to emphasize that we developed a good portion of the diagrams independently after studying the requirements, and only consulted the AI for specific guidance on structuring decisions and UML diagram verification as we weren't too familiar with UMLet and the whole procedure prior to this project.

Part I - Task 1: Use Case Diagram

AI Assistant Used: Perplexity AI (Claude Sonnet 4.5)

Prompts We Used:

"We have identified three main actors for the Lab Equipment Reservation system: Regular User, Lab Manager, and System Administrator. Can you help us understand how to properly divide the remaining use cases (requirements) between Actor 3 (System Administrator) and Actor 2 (Lab Manager)?

AI Output:

The AI suggested the following division of requirements:

```
Actor 2: Lab Manager
Add Equipment - Add new equipment with unique ID, description, location
(Req 6, 7)
Enable/Disable Equipment - Control equipment availability (Req 6)
Mark Equipment for Maintenance - Set equipment as unavailable (Req 6)
Monitor Equipment Status - View sensor data and operation status (Req 5)

Actor 3: System Administrator (Head Lab Coordinator)
Register User Accounts - Create accounts with email/password validation
(Req 1)
Approve University Accounts - Provide departmental approval for
affiliated users (Req 1)
Generate Manager Accounts - Auto-generate lab manager accounts (Req 2)
Verify User Credentials - Validate student/staff IDs and certification
numbers (Req 8)
```

“Dividing the remaining requirements as such allows us to show that the System Administrator is above the Lab Manager as a Head Lab Coordinator and it handles tasks that are above the level of a Lab Manager. Also, it should be noted that the System Administrator, having the most power, is naturally also able to do things that lab managers can do (monitoring equipment status, marking equipment for maintenance, etc.)”

Our Understanding:

It was initially easy to come up with Actors and the names for these actors and it was also fairly straightforward to come up with the use cases for Actor 1: Registered User. We had some problems dividing the remaining requirements between the other 2 Actors since we weren’t sure how to divide some things like registering user accounts and we also weren't sure if monitoring equipment status should be given to both actors as a use case but AI has assured us that since System Administrator is a level above Lab Manager, it will also be able to do things that Lab Manager does without having to have it explicitly stated.

Part I - Task 2 & 3: Activity and Sequence Diagrams

AI Assistant Used: Perplexity AI (Claude Sonnet 4.5)

Prompts We Used:

[1] *"For Equipment Management (Add/Enable/Disable/Maintenance), what's the best way to structure the activity diagram to show all three operations without making it too complex/hard to read?"*

[2] *"I need help generating UMLet XML code for our activity diagram structure. Can you provide the code syntax?"*

AI Output:

[1] "*Great question! For the Equipment Management activity diagram, you want to balance completeness with clarity. Here's the best approach:*
Use a single activity diagram with three parallel branches stemming from one decision node. Here's the flow:

```
Start → Login as Lab Manager → Access Equipment Management Module
Decision Node: "Choose Action?" with three branches:
Branch 1: Add Equipment
Enter equipment details (ID, description, location - Req 7)
Validate unique ID
If valid: Save to database with status "Available"
If invalid: Display error and loop back
Branch 2: Enable/Disable
View equipment list
Select equipment
Toggle status
Update database (Req 6)
Branch 3: Mark for Maintenance
View equipment list
Select equipment
Enter maintenance notes
Set status to "Unavailable" (Req 6)
Cancel existing reservations if any
All branches merge → Display success message → End
```

This structure keeps related operations in one diagram while clearly separating the flows. The decision node at the top makes it easy to see the three options, and each branch shows its complete workflow without cluttering the other branches.

For the sequence diagram, I'd recommend focusing on just one branch (probably Add Equipment since it has the most validation steps) to keep it readable."

[2] *"Absolutely! Here's the UMLet XML structure for your Equipment Management activity diagram. I'll show you the key components:*

Decision Node Syntax:

```
<element>
  <id>UMLSpecialState</id>
  <coordinates>
    <x>500</x>
    <y>300</y>
    <w>60</w>
    <h>60</h>
  </coordinates>
  <panel_attributes>type=decision</panel_attributes>
  <additional_attributes/>
</element>
```

Activity Box Syntax:

```
<element>
  <id>UMLState</id>
  <coordinates>
    <x>400</x>
    <y>400</y>
    <w>250</w>
    <h>40</h>
  </coordinates>
  <panel_attributes>Enter Equipment Details (Req 7)</panel_attributes>
  <additional_attributes/>
</element>
```

Flow Arrow with Labels:

```
<element>
  <id>Relation</id>
  <coordinates>
    <x>520</x>
    <y>350</y>
    <w>100</w>
    <h>70</h>
  </coordinates>
  <panel_attributes>lt=&lt;-
[Add Equipment]</panel_attributes>
  <additional_attributes>10.0;50.0;10.0;10.0</additional_attributes>
</element>
```

For your three branches, position them at different x-coordinates:
Add Equipment: x=200-400
Enable/Disable: x=450-650
Maintenance: x=700-900
This creates clean parallel paths that converge at the bottom.

Key tips:
Use lt=&lt;- for arrows pointing down/left
Use [condition] for decision branch labels
Adjust coordinates to avoid overlapping boxes
Set bg=color in panel_attributes for colored boxes

Would you like me to provide the complete XML for one of the branches?” **No**

Our Understanding:
[1] *We already had an idea of the basic flow for Equipment Management in our head but weren't sure whether to create three separate activity diagrams for more readability or how to combine them into one in a way that doesn’t make it too hard to read. The AI's suggestion to use a single diagram with parallel branches made sense because all operations (Add/Enable/Disable/Maintenance) are performed by the same actor (Lab Manager) and accessed from the same module. But then, We still modified the AI's approach in several ways: We added more detailed validation steps for the Add Equipment branch (checking not just ID uniqueness but also format validation), included explicit error handling paths that loop back to allow corrections, and added a decision node in the Maintenance branch to check if there are existing reservations that need to be cancelled. The AI's structure provided a good framework, but we reorganized it with more realistic error scenarios and edge cases that a real system would need to handle based on our understanding of Requirements 6 and 7.*

[2] *The AI's response really helped us understand the UMLet XML syntax, which we were unfamiliar with initially. I know we didn’t need to code in XML syntax and we could just create elements and move them around manually and that’s what we have done but having the AI give us importable code templates to start with helped us with our workflow. The code examples created some templates that showed us the basic structure of decision nodes, activity boxes, and flow arrows. We used it as a template to understand the diagram rules and then built our diagrams on top of those basic building blocks it has provided. Of course, we had to adjust all the coordinates manually to fit our specific layout, added custom colors to different sections, and expanded the flow with additional elements the AI didn't include, such as merge nodes and error activities.*

Part II: Class Diagram Prompts

AI Assistant Used: Perplexity AI (Claude Sonnet 4.5)

Prompts I Used:
[1] *"What tool should I use to create a professional class diagram? I have seen some other friends use different tools as they found them to be easier to create class diagrams with. Should I continue with UMLet or are there any other better websites/techs to use?”*

[2] *"Can you help me structure the class hierarchy? I'm thinking User should be abstract with Student, Faculty, Staff, Researcher, Guest kind of subclasses, and Manager should probably extend User as well right?"*

[3] *"The class diagram looks messy with too many overlapping lines as you can see from the attached screenshot of our class diagram so far. How can I reorganize it to be cleaner like other professional examples I have attached in this prompt?" (attached some class diagram examples I have found on the web and our current diagram)*

AI Output:
[1] *Since you've already been using UMLet for your Part I diagrams (activity and sequence diagrams), I strongly recommend continuing with UMLet for your class diagram to maintain consistency throughout your deliverable.*

Reasons to stick with UMLet:
Consistent visual style - Your entire submission will have a unified look
You're already familiar with the drawing diagrams using UMLet from Part I
Academic appropriateness - UMLet is designed for academic UML diagrams
No learning curve - You won't waste time learning a new tool

Alternative if you want a more polished look:
If you find UMLet's class diagram output too basic, draw.io (diagrams.net) is an excellent free alternative that:
Has better auto-layout features
Provides cleaner, more professional-looking outputs
Still supports all UML notation standards
Can export to PNG/PDF for your submission

However, this would create a visual inconsistency with your Part I diagrams.
Our recommendation: *Use UMLet. The slightly less polished look is acceptable for academic submissions, and consistency across your deliverable is more important than having the "prettiest" diagram. Professors care more about correct UML notation and complete requirements coverage than visual aesthetics.*

[2] *Excellent thinking! Your proposed hierarchy is sound. Let me validate and expand on your design:*
User Hierarchy - Correct Approach:

```
<<abstract>> User
    ↑
    ├── Student
    ├── Faculty
    ├── Staff
    ├── Researcher
    ├── Guest
    └── Manager
            ↑
            └── SystemAdministrator (Head Lab Coordinator)
```

Why this works:

1. ***User as Abstract*** *- Perfect choice because:*
    - *There should never be a "generic" user object in the system*
    - *All users must have a specific type (Req 1 mentions students, faculty, researchers, guests)*
    - *Shared attributes (email, password, userId) belong at this level*

2. ***Student/Faculty/Staff/Researcher/Guest as subclasses** - Correct because:*
    - *Each has different getHourlyRate() implementations (Req 3: students $10, faculty $15, researchers $20, guests $30)*
    - *Each might have unique attributes (studentID, facultyID, researcherID, etc.)*
    - *Polymorphism: You can write code that works with "User" but behaves differently for each type*

3. ***Manager extends User** - Good design because:*
    - *Managers ARE users who also happen to have management privileges*
    - *They still need to login, have accounts, etc.*
    - *This follows the "is-a" relationship rule in OOP*

4. ***SystemAdministrator extends Manager** - Excellent because:*
    - *Req 2 specifies the "head lab coordinator" has special privileges (create manager accounts)*
    - *They can do everything a Manager can PLUS system administration*
    - *This models the organizational hierarchy naturally*

Additional recommendations:
Add a `userType: UserType` attribute to the User class (use an enumeration)
Consider making `getHourlyRate()` an abstract method in User, forcing all subclasses to implement it
Manager should have additional methods like `addEquipment()`, `disableEquipment()` (Req 6)
SystemAdministrator should have methods like `createManagerAccount()` (Req 2), `approveUserAccount()` (Req 1)
This hierarchy is clean, follows OOP principles, and maps directly to your requirements. Well done!

***[3]** Class diagram readability is crucial for professional software design documentation. Looking at complex systems like your Lab Equipment Reservation Platform, here's how to create a clean, organized layout:*

Recommended Layout Structure:

1. ***Top Section - Enumerations and Core Entities***

Place your enumerations (UserType, AccountStatus, EquipmentStatus) on top of the parent , stacked vertically or horizontally.
Color them distinctly (yellow/orange works well) to differentiate them from domain classes
Position your core User class and its hierarchy in the top-center area
This establishes the foundation since users are central to your system

2. ***User Hierarchy Organization***

Place the abstract User class at the top
Arrange subclasses (Student, Faculty, Researcher, Guest) in a row or grid below User
Keep inheritance arrows short and vertical - avoid long diagonal lines
Position Manager and SystemAdministrator to the right of the user hierarchy since they're specialized user types with administrative functions

3. ***Right Side - Equipment Domain***

Group all equipment-related classes on the right side of the diagram:
Equipment class in the upper-right
Lab class near Equipment (they're closely associated by Req 7)
Sensor and SensorData classes adjacent to Equipment
This creates a clear "equipment zone" that's visually separated from user management

4. ***Center - Booking System***

Position Booking and BookingManager classes in the center of the diagram
This placement is strategic because:
Booking connects to User (left) via "makes" association
Booking connects to Equipment (right) via "reserved by" association
Center placement minimizes line crossing distance

5. ***Bottom Section - Services and Supporting Classes***

Payment system (Payment, PaymentService, PaymentMethod enum) on the bottom-left
Supporting payment classes (InstitutionalAccount) below Payment
Service classes (AuthenticationService, NotificationService) on the bottom-right
BookingStatus enum near Booking class for proximity

Line Management Strategies:
Use dependency arrows (dashed lines) sparingly - They should connect service classes to the entities they use
Inheritance arrows should flow top-to-bottom - This is the natural reading direction
Association arrows should be the shortest possible paths - Adjust class positions to minimize distance
When lines must cross, ensure they cross at right angles rather than parallel running
Group related associations together - If multiple classes point to the same target, route them from the same side

Color Coding for Visual Clarity:
Enumerations: Yellow/Orange
User classes: Blue/Cyan
Manager classes: Green
Equipment classes: Pink/Salmon
Booking classes: Yellow
Payment classes: Gray/Light gray
Service classes: Purple/Violet

Spacing Guidelines:
Maintain at least 50-100 pixels between unrelated classes
Keep related classes closer (20-50 pixels)
Leave enough space for association labels and multiplicities
Don't crowd the diagram - white space improves readability

Final Tips:
Test your layout by printing it: Can you read all text at normal size?
Follow the "left-to-right, top-to-bottom" information flow when possible
Services and utilities go at the edges/bottom since they support the core domain
Keep your most important classes (User, Equipment, Booking) in prominent central positions

This organization creates clear visual zones that help readers quickly understand which classes collaborate, following the principle that "spatial proximity indicates conceptual relationship.

Our Understanding:
[1] *We were debating whether to switch to draw.io or some other tool (Lucidchart) since some of our friends that have done this course said they found other websites easier to work with. The AI made a good point about maintaining visual consistency across all deliverable parts since we have already used UMLet for activity and sequence diagrams and we also strongly obsess over formatting and visuals of our work. Therefore, we decided to stick with UMLet. Learning a new tool mid-project would have also caused us to waste time since we can just do the same thing on UMLetino as well. But I have noted the AI's recommendation about Lucidchart since it is more commonly used in the industry and I will be having that in the back of my head for when I have to create a diagram again.*

[2] *We have already sketched the User hierarchy structure before consulting the AI, but weren't confident about whether the Manager should extend User or be a separate entity. The AI's validation confirmed our initial design was sound and helped us understand the polymorphism benefits (different getHourlyRate() implementations for each user type). However, we disagreed with the AI's suggestion to make getHourlyRate() an abstract method. We implemented it as a concrete method in the User class with a switch statement on userType instead, which seemed to be cleaner than forcing every subclass to override it. I also added several attributes and methods the AI didn't mention, like certificationNumber (as vaguely mentioned in Req 8,) and booking-related methods.*

[3] *Our initial layout was completely disorganized with lines crossing everywhere, making it unreadable. The AI's zone-based layout strategy (enums closeby, users center-top, equipment right, services bottom) gave me a clear reorganization plan. We followed most of the positioning advice, particularly placing Booking in the center since it connects both User and Equipment, but made adjustments based on actual line routing. For example, We moved the BookingStatus enum closer to the Booking class rather than keeping it in the top-left with other enums, which significantly reduced arrow length. We also applied the color-coding scheme the AI suggested (blue for users, pink for equipment, green for managers, etc.), which made the diagram much more readable at a glance.*

Task 2: Arguments and Resolution

Throughout this deliverable 1, We encountered several instances where our understanding differed from the AI's suggestions. Below are the key disagreements and their resolutions:

Disagreement 1: *getHourlyRate() Implementation*
AI's Suggestion: *Make getHourlyRate() an abstract method forcing all User subclasses to implement their own version.*
Our Challenge: *This would create unnecessary code duplication, each subclass would just return a constant (10.0, 15.0, 20.0, 30.0).*
Resolution: *We decided to implement it as a concrete method in the User class with a switch statement on userType. Since Req 3 defines rates as fixed values, centralizing this logic in one place is cleaner and more maintainable than five separate implementations.*

Disagreement 2: *Diagram Flow Logic and Error Handling*
AI's Suggestion: *For activity diagrams, the AI provided simplified flows with basic "loop back on error" patterns. For sequence diagrams, it didn't emphasize validation ordering.*
Our Challenge: *The AI's flows lacked realistic error handling details and proper validation sequencing for security-critical operations.*
Resolution: *We enhanced both diagram types:*

- *Activity diagrams: Added explicit error paths with format validation before uniqueness checks, separate error messages, and a decision node in the Maintenance branch to check for existing reservations before marking equipment unavailable*
- *Sequence diagrams: Reordered the User Registration flow to validate credentials before creating accounts (validate → verify → then create), following the "fail-fast" security principle*

This made the diagrams implementation-ready rather than just conceptual.

Disagreement 3: *InstitutionalAccount and ResearchGrant as Separate Classes*
AI's Suggestion: *Create full classes with attributes like balance, availableFunds, expiryDate for institutional accounts and research grants.*
Our Initial Doubt*: This seemed over-engineered since Req 10 just says "supports" these payment methods—why not store them as simple strings?*
Resolution: *After researching university financial systems, we decided to accept the AI's approach with some simplifications. Universities have strict compliance requirements and research grants need detailed spending reports. These reasons can justify separate classes, but I removed some excessive validation methods the AI suggested and focused on core functionality: balance checking and fund allocation.*

Task 3: Verifying AI Responses

We decided to use a three-step verification process:
(1) Requirements cross-reference against Req 1-10,
(2) UML 2.5 standards validation using course materials and uml-diagrams.org,
(3) Logical consistency checks for real-world feasibility.

Example: *Equipment-Lab Relationship Verification*
AI Suggestion*: Simple association arrow between Equipment and Lab.*
Verification Steps:
Requirement: *Req 7 states equipment has a "lab location"—implies Equipment must belong to a Lab*
UML Standards: *Reviewed aggregation (hollow diamond) vs. composition (filled diamond) vs. association rules*
Logic: *Equipment can be transferred between labs but shouldn't exist unassigned*
Outcome: *Modified to aggregation with Lab (1) ◇→ (0..*) Equipment. This correctly shows Equipment belongs to exactly one Lab but can be transferred, following UML 2.5 aggregation semantics.*

Example: *Composition for Equipment-Sensor*
AI Suggestion: *Use composition (◆) for Equipment-Sensor relationship.*
Verification: *Req 5 states sensors track equipment usage. UML composition means child cannot exist without parent. Logically, sensors are physically attached to equipment.*
Outcome: *Accepted. Composition correctly models that sensors have no independent existence outside their equipment.*

Task 4: Application of AI-Suggested Content

Original Prompt: *"The class diagram looks messy with too many overlapping lines. How can I reorganize it to be cleaner like professional examples?"*

AI Response Applied:
"Place enumerations close-by stacked vertically or horizontally. Position User hierarchy in top-center. Position Booking and BookingManager in center—this placement is strategic because Booking connects to User (left) and Equipment (right), minimizing line crossing. Group all equipment-related classes on the right side. Services and payment classes on the bottom."

Where Applied: *This layout structure appears directly in our final class diagram, some enums are top-left-ish while others are closer to the parent, User hierarchy top-center, Booking central, Equipment right-side, services bottom.*

Rationale: *The AI's zone-based approach follows the "spatial proximity indicates conceptual relationship" principle and immediately solved our arrow-crossing problem. We learnt that it's a proven UML organization pattern and it made our diagram professional and readable. We only modified it slightly by moving some enum closer to their own respective class to shorten arrows, but kept the overall zone strategy intact.*