chapter 7: OCL + UML

🔍 Types of Data Models

In the world of data science and analytics, models are often categorized into three main types based on their purpose: Descriptive, Predictive, and Prescriptive. Each type serves a unique function in understanding data and supporting decision-making processes.

---

1. 📊 Descriptive Model

🔎 What It Does:

Descriptive models analyze historical data to understand patterns and relationships. They summarize what has happened in the past without predicting future outcomes.

🎯 Use Case:

- Identify customer segments based on behavior
- Understand trends in sales or user activity
- Generate reports and dashboards

🧠 Example:

A customer segmentation model groups users into categories such as

“frequent buyers,” “occasional buyers,” and “inactive users” based on purchase history.

✅ **Key Technique:**

- Clustering (e.g., K-Means)
- Association Rules (e.g., Market Basket Analysis)

---

2. 📈 Predictive Model

🔎 What It Does:

Predictive models use historical data to make informed predictions about future events. They estimate the likelihood of outcomes based on patterns.

🎯 Use Case:

- Predict which customers are likely to churn
- Forecast next month’s sales
- Estimate loan default risk

🧠 Example:

A churn prediction model analyzes past customer behavior to predict whether a user is likely to cancel their subscription within the next 30 days.

✅ **Key Techniques:**

- Regression
- Classification (e.g., Decision Trees, Random Forest, Logistic Regression)

---

3. 🧠 Prescriptive Model

🔎 What It Does:

Prescriptive models go beyond prediction and recommend actions to achieve desired outcomes. They consider constraints, goals, and possible decisions to suggest the best course of action.

🎯 Use Case:

- Optimize delivery routes for logistics
- Recommend personalized marketing strategies
- Decide the best inventory level based on demand forecasts

🧠 Example:

A route optimization model for a delivery company suggests the most efficient routes based on traffic, delivery deadlines, and fuel costs to minimize time and cost.

✅ **Key Techniques:**

- Optimization algorithms
- Simulation
- Reinforcement learning

🏗️ Model-Driven Engineering (MDE)

🔎 What is Model-Driven Engineering?

Model-Driven Engineering (MDE) is a software development approach that focuses on creating and transforming abstract models rather than writing low-level code directly. In MDE, models are first-class citizens—they represent the structure, behavior, and functionality of a system and can be automatically transformed into executable code or other models.

MDE aims to automate and simplify software development by raising the level of abstraction and enabling developers to work closer to the problem domain rather than the technology.

🎯 Why Use MDE?

- ✅ **Higher Abstraction: Focuses on business logic rather than**

implementation details.

- **✅ Automation: Enables automatic code generation, reducing repetitive coding tasks.**
- **✅ Consistency: Keeps documentation and implementation aligned through models.**
- **✅ Productivity: Speeds up development by using reusable model components.**
- **✅ Platform Independence: Models can be translated to different target platforms or technologies.**

🔧 Key Concepts in MDE

Concept	Description
Model	An abstract representation of a system or its part
Metamodel	A model that defines the language for creating other models
Model Transformation	The process of converting one model into another (e.g., model → code)
Code Generation	Automatically generating source code from models

🧠 Example:

Imagine you are building an online shopping system. Instead of

manually writing code for customers, products, and orders, you:

1. Create a model of the system using a modeling language like UML.
2. Use a tool to automatically generate database schemas, API code, and documentation from the model.
3. Apply transformations to produce different versions of the system (e.g., Java backend, .NET backend) using the same model.

This approach improves maintainability and reduces human error.

🏗️ Popular Tools and Frameworks

- Eclipse Modeling Framework (EMF)
- ATL (Atlas Transformation Language)
- Acceleo (for model-to-text transformations)
- Papyrus (UML modeling tool)
- JetBrains MPS

🗣️ Modeling Languages

🔍 What Are Modeling Languages?

A modeling language is a set of notations, symbols, and rules used to create abstract representations (models) of systems, processes, or data. These models help in understanding, designing, analyzing, and

communicating different aspects of software or system architectures.

Modeling languages are essential tools in Model-Driven Engineering (MDE) and system design, and they vary based on the domain they are intended to represent.

📚 Characteristics

- 🎯 **Domain-Specific: Each modeling language targets a particular domain (e.g., software, business, hardware).**
- 🧩 **Graphical or Textual: Models can be drawn visually (diagrams) or described in text.**
- 🔁 **Standardized Semantics: Modeling languages often follow standardized rules to ensure consistency.**

🧰 Common Modeling Languages

Here are some widely used modeling languages categorized by their application domain:

Modeling Language	Domain	Description
UML (Unified Modeling Language)	Object-Oriented Software	A general-purpose language for modeling

		object-oriented systems. Supports use case, class, sequence, activity, and state diagrams.
Simulink	Control Systems / Embedded Systems	A graphical language from MATLAB used for modeling, simulating, and analyzing dynamic systems like electrical or mechanical controllers.
Archimate	Enterprise Architecture	A high-level modeling language used for designing and visualizing business, application, and technology layers in enterprise architecture.
BPMN (Business Process Model and Notation)	Business Processes	Designed for modeling business workflows and processes. It provides easy-to-understand flowchart-style diagrams that

		bridge business and IT.
ER (Entity-Relationship Model)	Relational Databases	Used for modeling data structures, especially relational databases. It visually represents entities, their attributes, and relationships.

🎓 **Example Use Cases**

- **UML: Designing a student registration system using class and sequence diagrams.**
- **Simulink: Developing an automotive cruise control system with real-time simulation.**
- **Archimate: Mapping how a new business service fits into an enterprise IT infrastructure.**
- **BPMN: Modeling a loan approval workflow in a banking application.**
- **ER: Creating a schema for a library management system's database.**

Modeling languages help bridge the gap between abstract ideas and concrete implementations, allowing teams to design and reason about complex systems more effectively. Choosing the right modeling language depends on the system type, domain, and team expertise.

Use Case Diagram

Definition:

A use case diagram represents the functional behavior of a system from the user's point of view. It shows what the system does (not how) and the interactions between the system and its external actors (users or other systems).

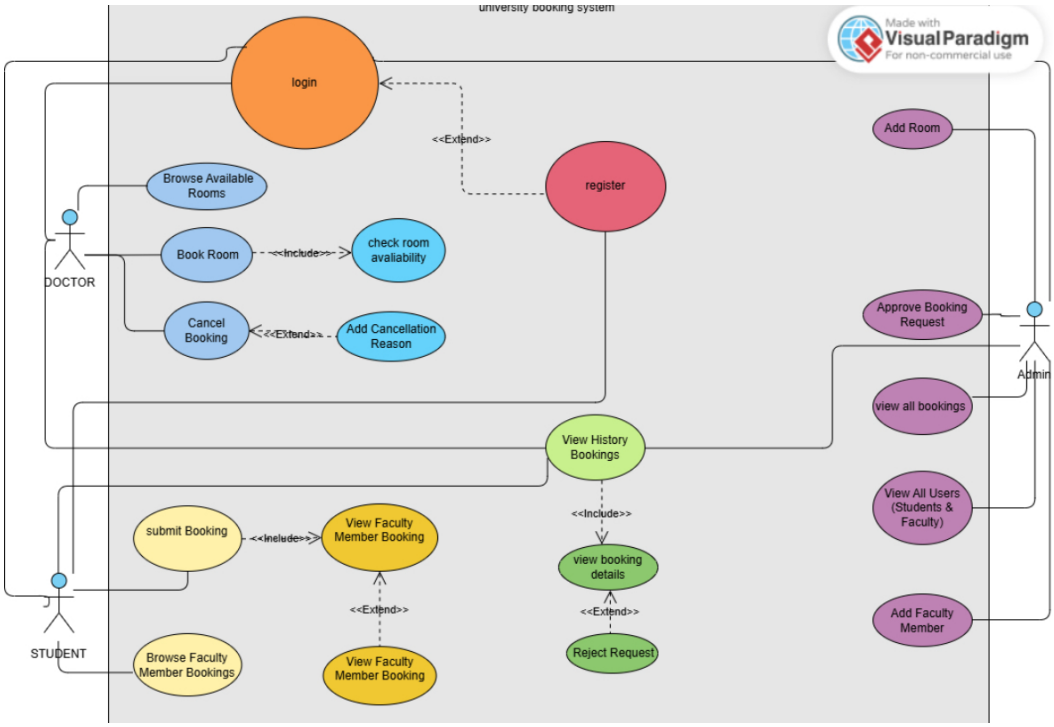
Elements:

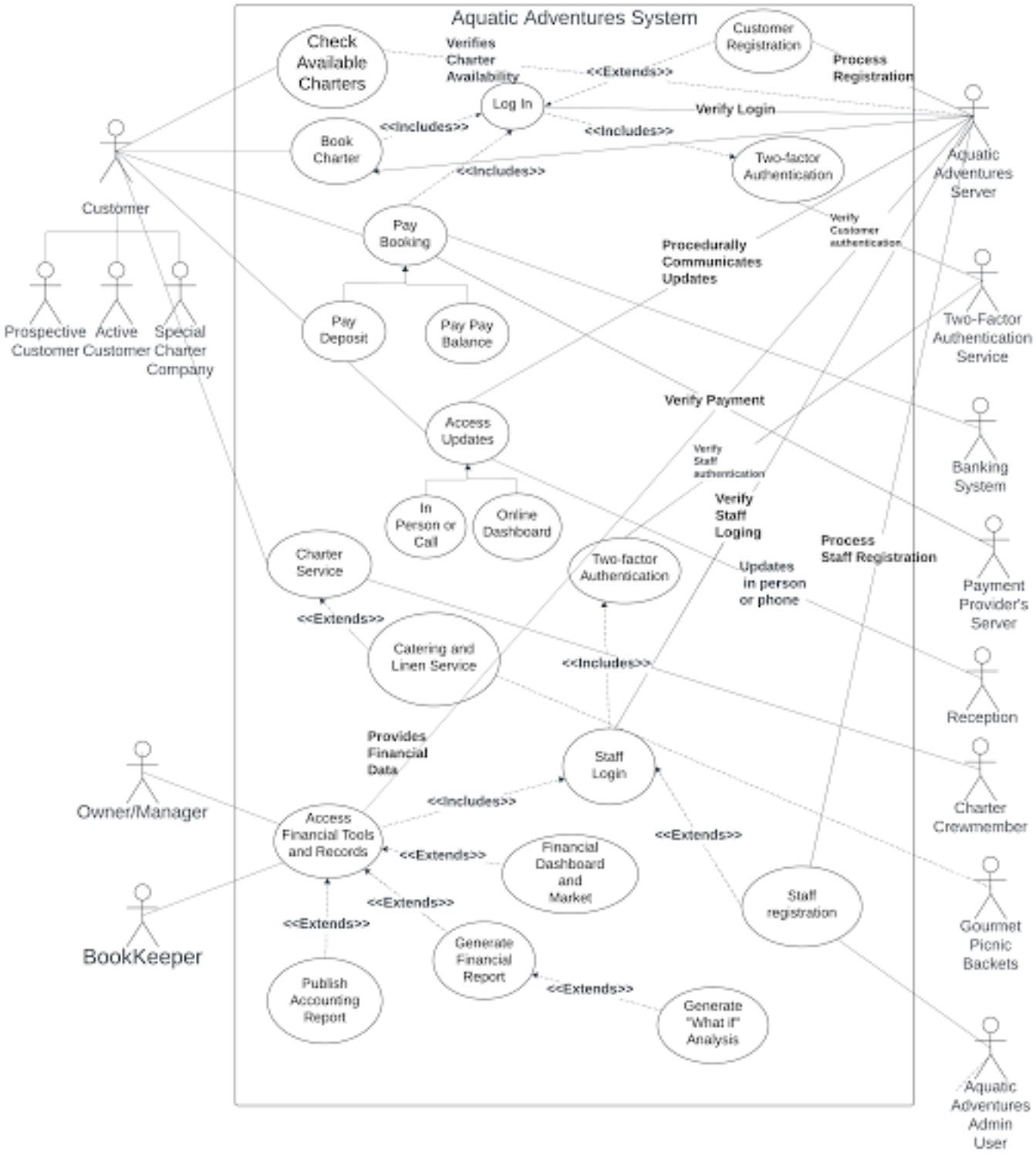
- Actor: Represents a user or external system that interacts with the system.
- Use Case: Describes a specific functionality or service provided by the system.
- System Boundary: Represents the scope of the system.
- Relationships: Includes associations, include, extend, and generalization.

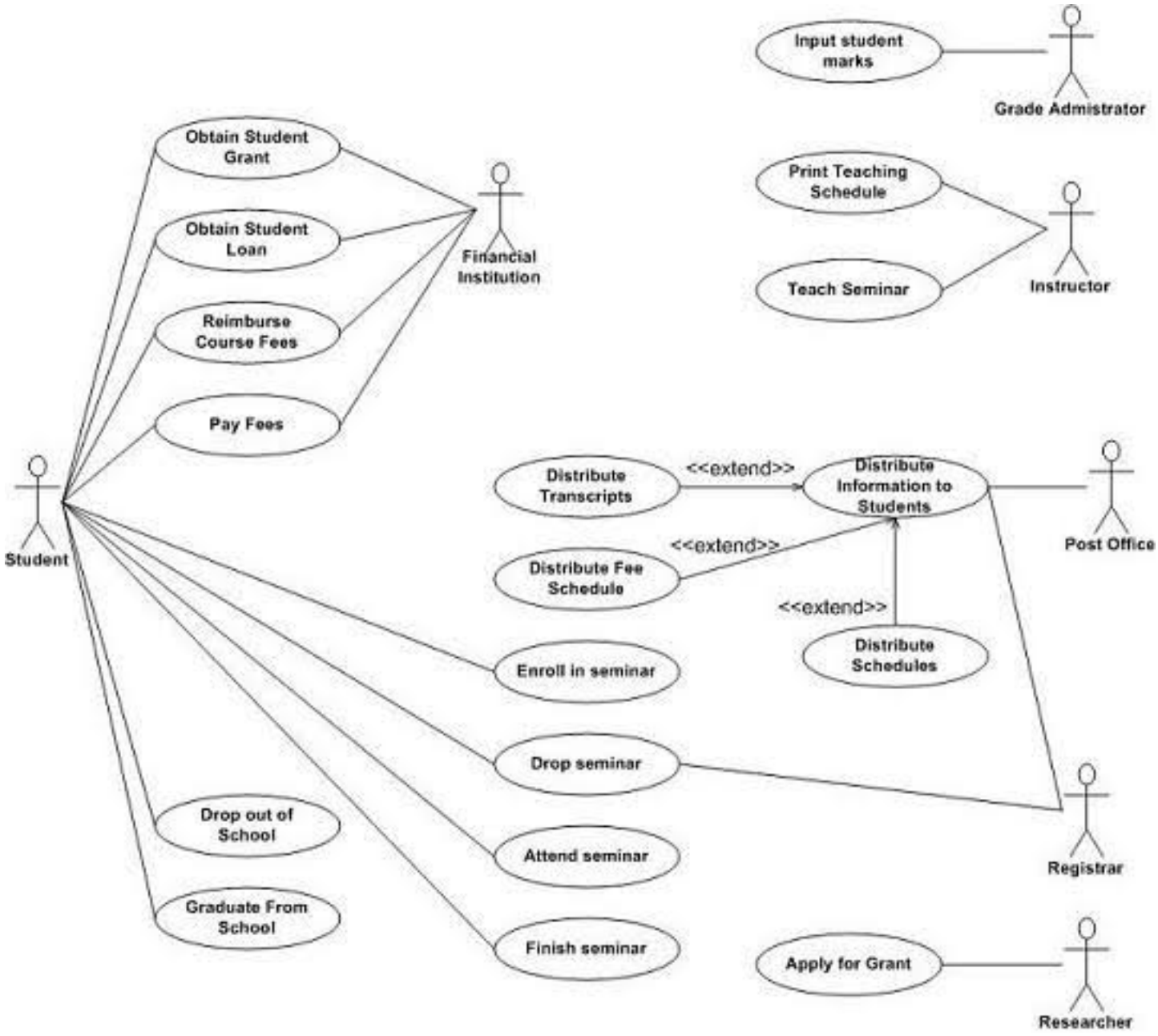
Example:

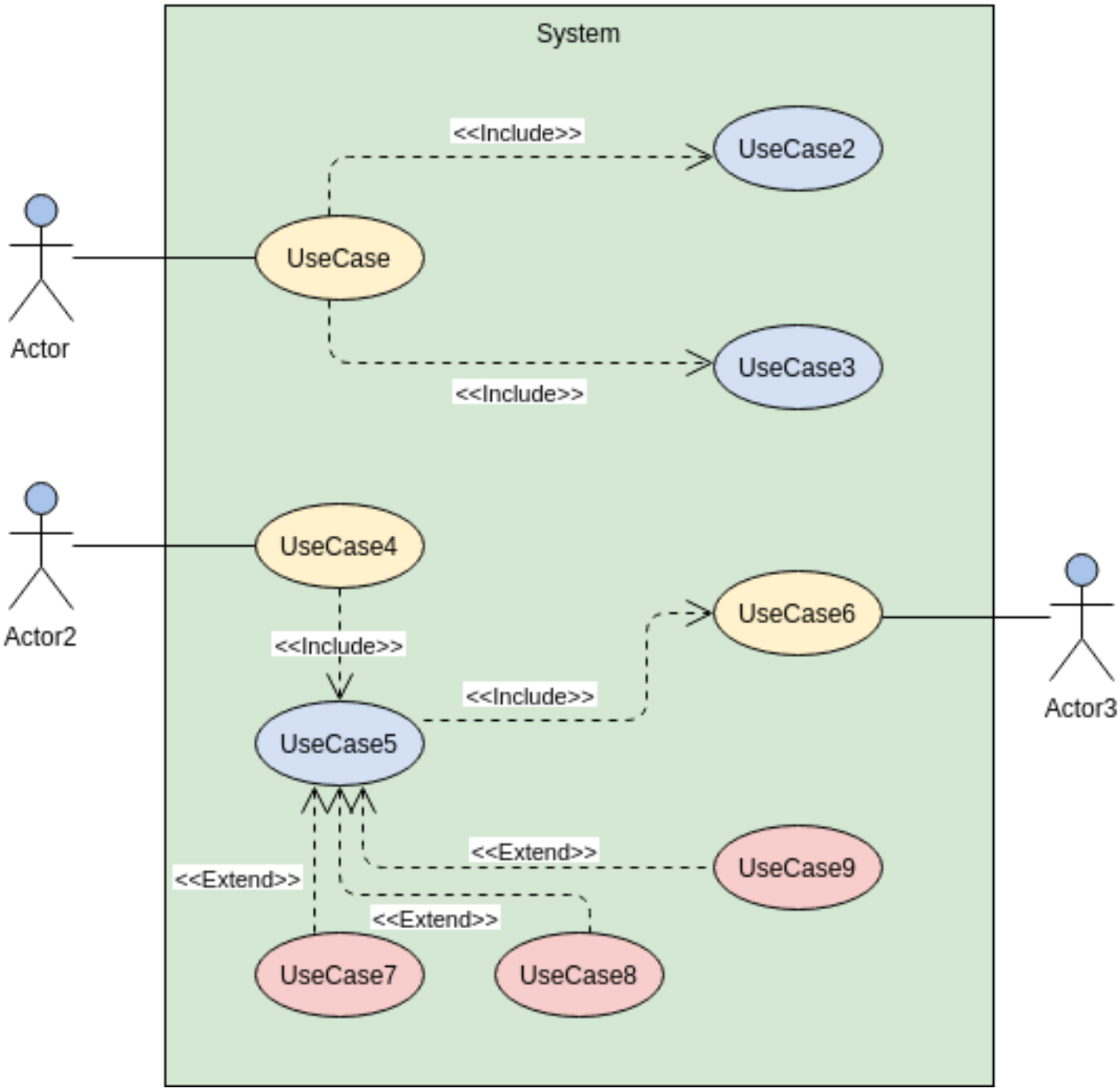
For an online bookstore:

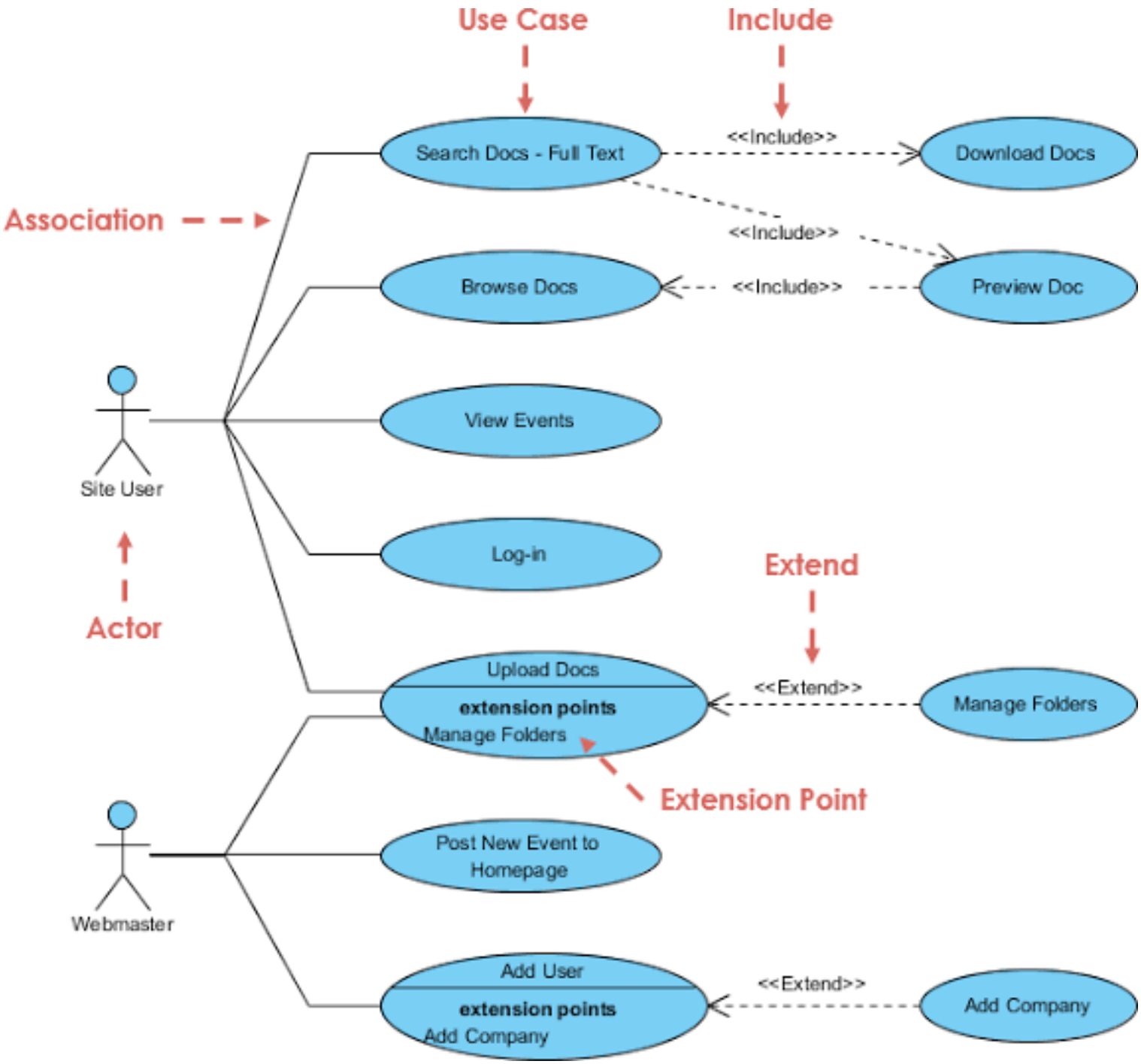
- Actors: Customer, Admin
- Use Cases: Browse Books, Place Order, Manage Inventory

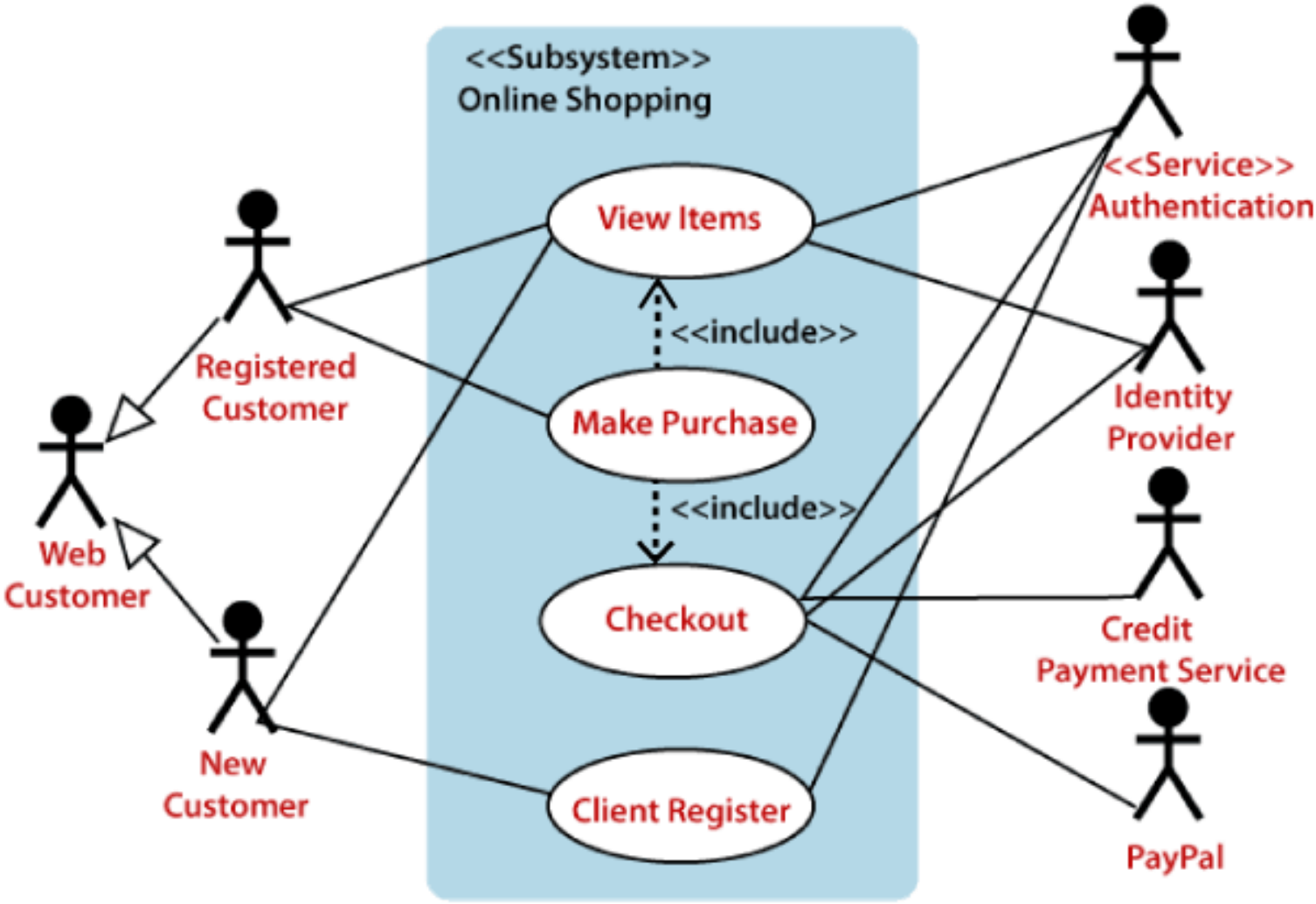

university booking system
Made with VisualParadigm For non-commercial use
login
<<Extend>>
register
Browse Available Rooms
Book Room
<<Include>>
check room avaliability
DOCTOR
Cancel Booking
<<Extend>>
Add Cancellation Reason
Add Room
Approve Booking Request
Admin
view all bookings
View History Bookings
<<Include>>
view booking details
<<Extend>>
Reject Request
View All Users (Students & Faculty)
Add Faculty Member
submit Booking
<<Include>>
View Faculty Member Booking
<<Extend>>
View Faculty Member Booking
STUDENT
Browse Faculty Member Bookings


Aquatic Adventures System
Check Available Charters
Verifies Charter Availability
Customer Registration
Process Registration
<<Extends>>
Log In
Verify Login
<<Includes>>
Book Charter
<<Includes>>
Two-factor Authentication
<<Includes>>
Aquatic Adventures Server
Customer
Pay Booking
Verify Customer authentication
Procedurally Communicates Updates
Prospective Customer
Active Customer
Special Charter Company
Pay Deposit
Pay Pay Balance
Two-Factor Authentication Service
Verify Payment
Access Updates
Verify Staff authentication
Banking System
In Person or Call
Online Dashboard
Verify Staff Loging
Process Staff Registration
Charter Service
Two-factor Authentication
Updates in person or phone
Payment Provider's Server
<<Extends>>
Catering and Linen Service
<<Includes>>
Reception
Provides Financial Data
Staff Login
Charter Crewmember
Owner/Manager
<<Includes>>
Access Financial Tools and Records
<<Extends>>
Financial Dashboard and Market
<<Extends>>
Staff registration
Gourmet Picnic Backets
BookKeeper
<<Extends>>
<<Extends>>
Generate Financial Report
Publish Accounting Report
<<Extends>>
Generate "What if" Analysis
Aquatic Adventures Admin User


Input student marks
Grade Administrator
Obtain Student Grant
Obtain Student Loan
Reimburse Course Fees
Pay Fees
Financial Institution
Print Teaching Schedule
Teach Seminar
Instructor
Student
Distribute Transcripts
<<extend>>
Distribute Information to Students
Post Office
Distribute Fee Schedule
<<extend>>
<<extend>>
Distribute Schedules
Enroll in seminar
Drop seminar
Drop out of School
Attend seminar
Graduate From School
Finish seminar
Registrar
Apply for Grant
Researcher


System
<<Include>>
UseCase2
Actor
UseCase
UseCase3
<<Include>>
Actor2
UseCase4
UseCase6
Actor3
<<Include>>
<<Include>>
UseCase5
<<Extend>>
UseCase9
<<Extend>>
<<Extend>>
UseCase7
UseCase8


Use Case
Include
Search Docs - Full Text
<<Include>>
Download Docs
Association
<<Include>>
Browse Docs
<<Include>>
Preview Doc
View Events
Site User
Log-in
Extend
Actor
Upload Docs
extension points
Manage Folders
<<Extend>>
Manage Folders
Extension Point
Post New Event to Homepage
Webmaster
Add User
extension points
Add Company
<<Extend>>
Add Company



Use Case Description

Definition:

A textual description of a use case providing detailed information about the flow of events, actors, preconditions, and postconditions.

Typical Template:

- Use Case Name
- Actors
- Preconditions (what must be true before the use case starts)
- Postconditions (what is true after the use case completes)
- Main Flow (Basic Path)
- Alternative Flows
- Exceptions

Example:

Use Case Name: Place Order
Actors: Customer
Preconditions: Customer is logged in
Postconditions: Order is stored in the database
Main Flow:

1. Customer selects items and adds to cart
2. Customer proceeds to checkout
3. System calculates total and verifies inventory
4. Customer confirms order and payment
5. System creates order

Use Case Template

Name, description	Brief description of the process (what is happening) in the use case
Actors	List all actors (people, systems, etc.) associated with this requirement
Pre-condition	What must occur before the use case begins
Post-condition	What has occurred as a result of the use case

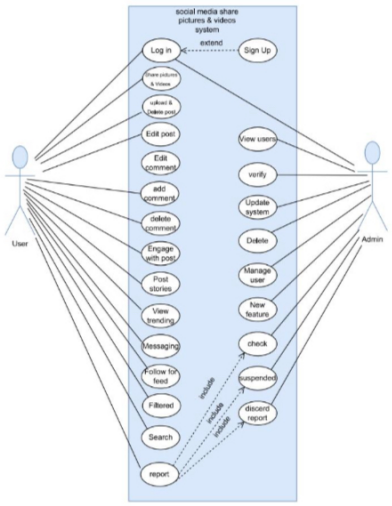
Main Success Path (primary flow)	Description of the sequence of activities in the most commonly completed path or flow. The main flow is the most routine path from the pre- to the post-conditions.
Actor Actions	**System Responses**
1. List the actor steps here	1.1 List the system steps here
2.	2.1

Alternate Path	A1	List the description here, and the less common sequences below that get to the post-conditions. Include any triggers, and where the use case resumes after this flow.
Actor Actions		**System Responses**
1. List the actor steps here		1.1 List the system steps here
2.		2.1

Exception Path	E1	List the description here, and the sequence of actions that prevents getting to the post-conditions. Include any triggers.
Actor Actions		**System Responses**
1. List the actor steps here		1.1 List the system steps here
2.		2.1

Scenarios (Insert additional rows for each scenario)

Scenario	Post-Conditions	Flow



This is the use case for the system

Some initial ideas for the system

Use Case:	**Social Media Share Picture and Video Functionality**
Primary Actor:	Social Media User
Goal:	To share pictures and videos on social media platform and manage their shared content.

The description for the user

Aspect	Description
Identifier	UC-001
Name	Log in
Initiator	User
Goal	To access the user's account by providing valid credentials
Precondition	User has registered for an account
Postcondition	User gains access to their account dashboard
Main Success Scenario	1. User navigates to the login page. 2. User enters their username/email and password. 3. . System verifies the credentials. 4. If credentials are valid, the system grants access to the user's account. 5. User gains access to their account dashboard.
Main Success Scenario	2. User enters their username/email and password.
Main Success Scenario	3. System verifies the credentials.
Main Success Scenario	4. If credentials are valid, the system grants access to the user's account.
Extensions	- If the credentials are invalid, the system displays an error message and prompts the user to retry. - If the user has forgotten their password, they can request a password reset link.

2. Use Case: Share Pictures & Videos

Aspect	Description
Identifier	UC-002
Name	Share Pictures & Videos
Initiator	User
Goal	To upload and share pictures or videos with followers or on profile
Precondition	User is logged into their account
Postcondition	Media (pictures or videos) is successfully uploaded and visible to the intended audience

---

Package Diagram

Definition:

A package diagram organizes elements of a system into related groups

to reduce complexity. It represents the high-level structure of a system.

Elements:

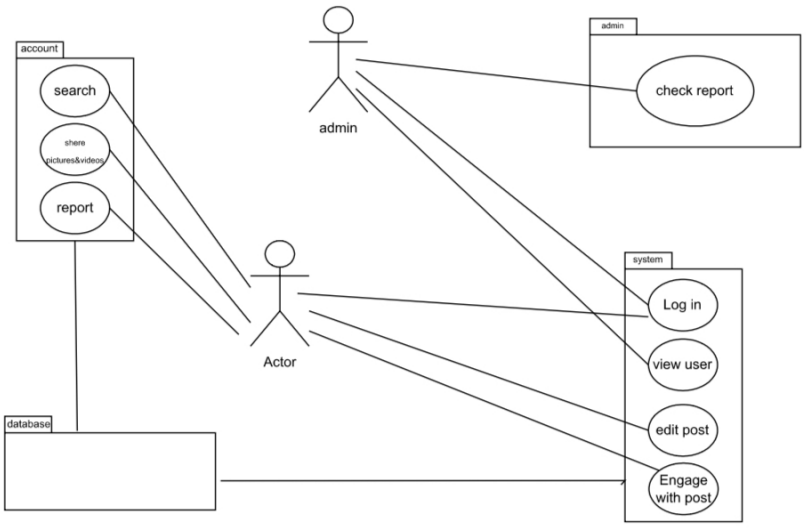
- Packages: Groups of related classes or components.
- Dependencies: Indicate relationships or usage between packages.

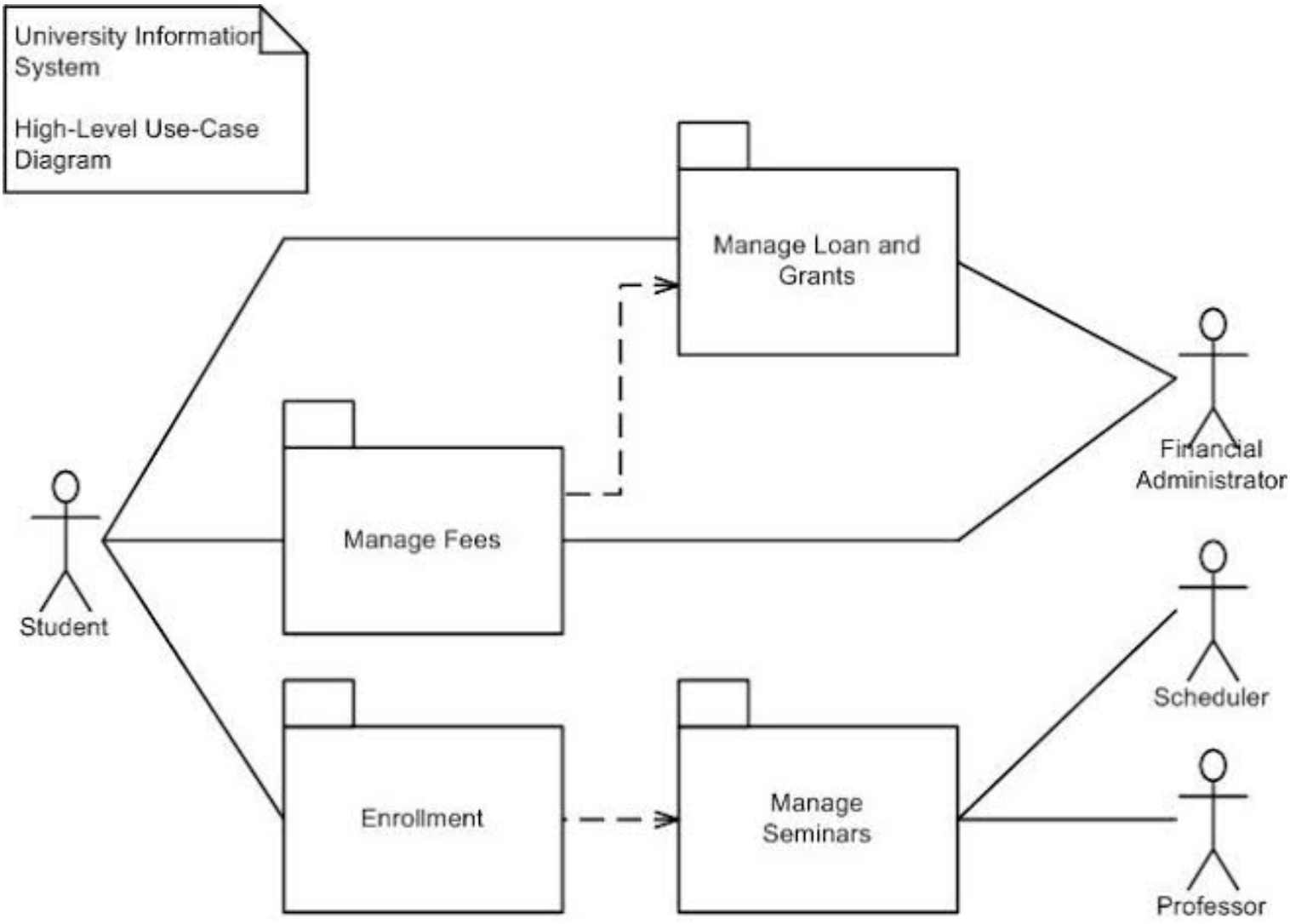
Example:

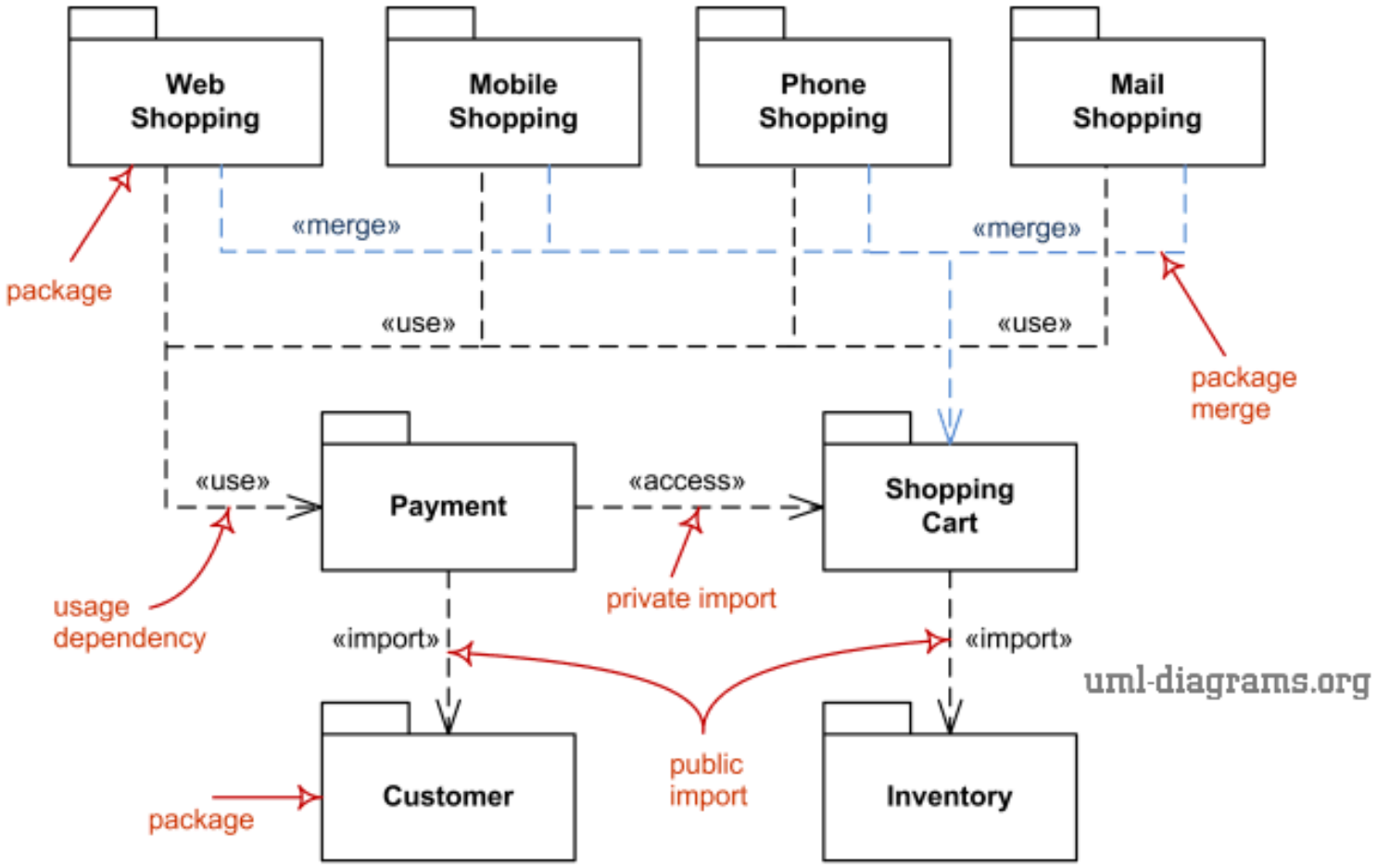
For an e-commerce platform:

- Packages: Authentication, ProductCatalog, OrderProcessing, PaymentGateway

---


account
search
shere
pictures&videos
report
admin
admin
check report
Actor
system
Log in
view user
edit post
Engage
with post
database


University Information System
High-Level Use-Case Diagram
Manage Loan and Grants
Manage Fees
Enrollment
Manage Seminars
Student
Financial Administrator
Scheduler
Professor



Activity Diagram

Definition:

An activity diagram models the flow of control or data in a system. It is useful for visualizing workflows and business processes.

Elements:

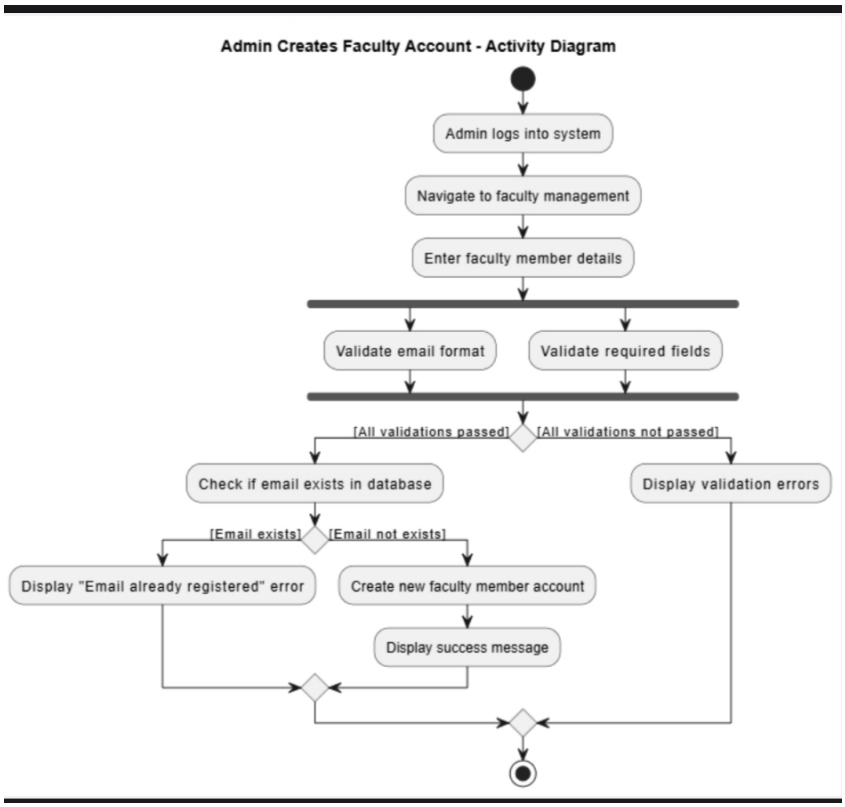
- Action/Activity States
- Transitions (Arrows)
- Start and End Nodes

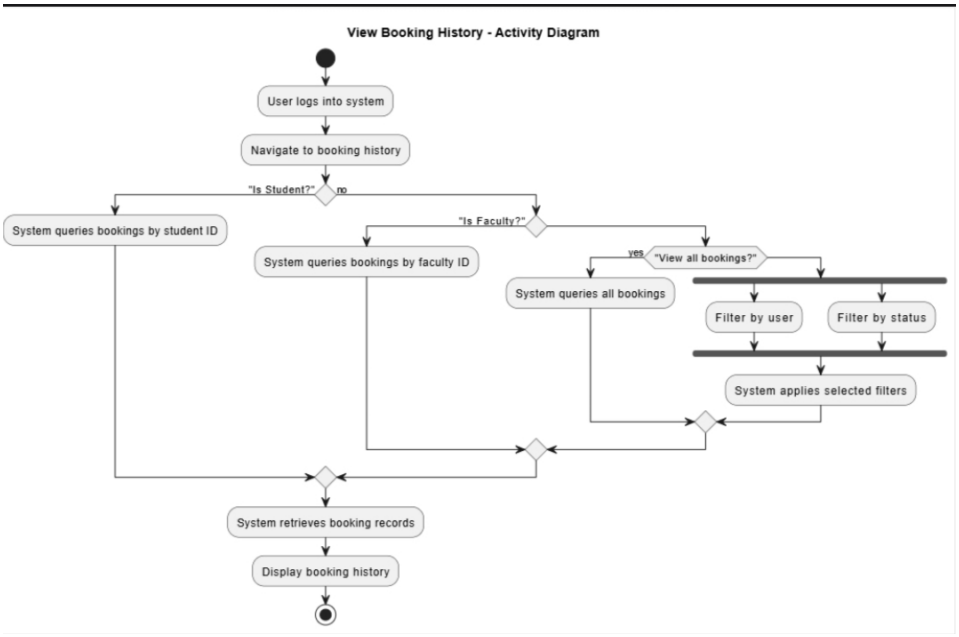
- Decision Nodes
- Parallel Bars (Fork/Join)

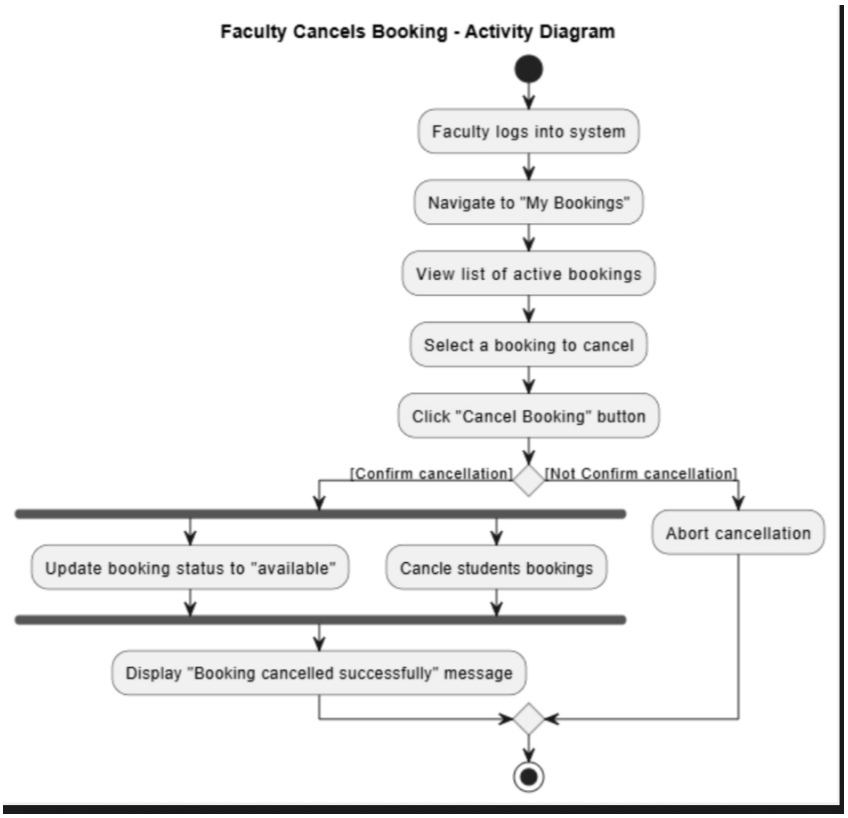
Example:

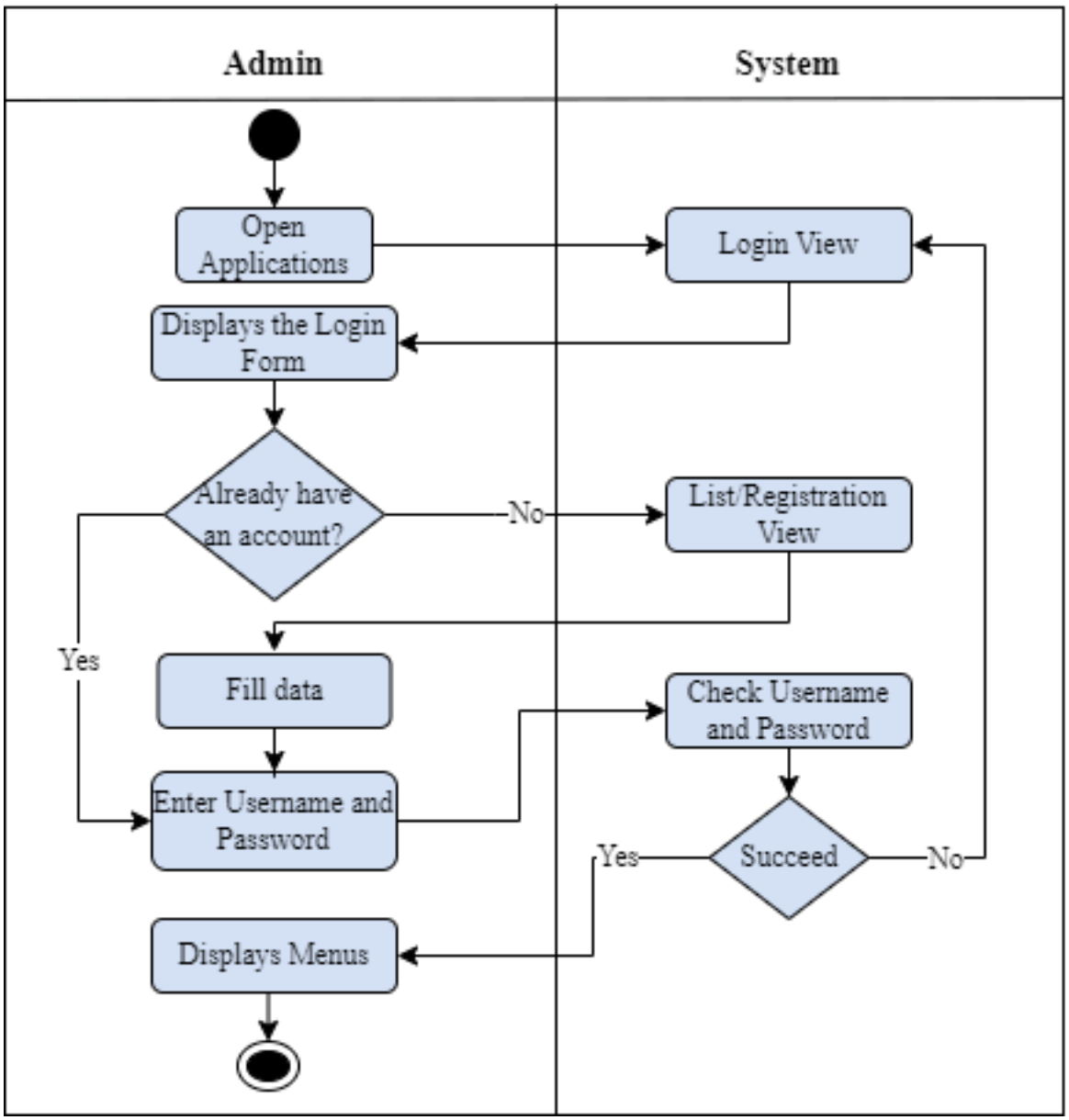
An order processing workflow:

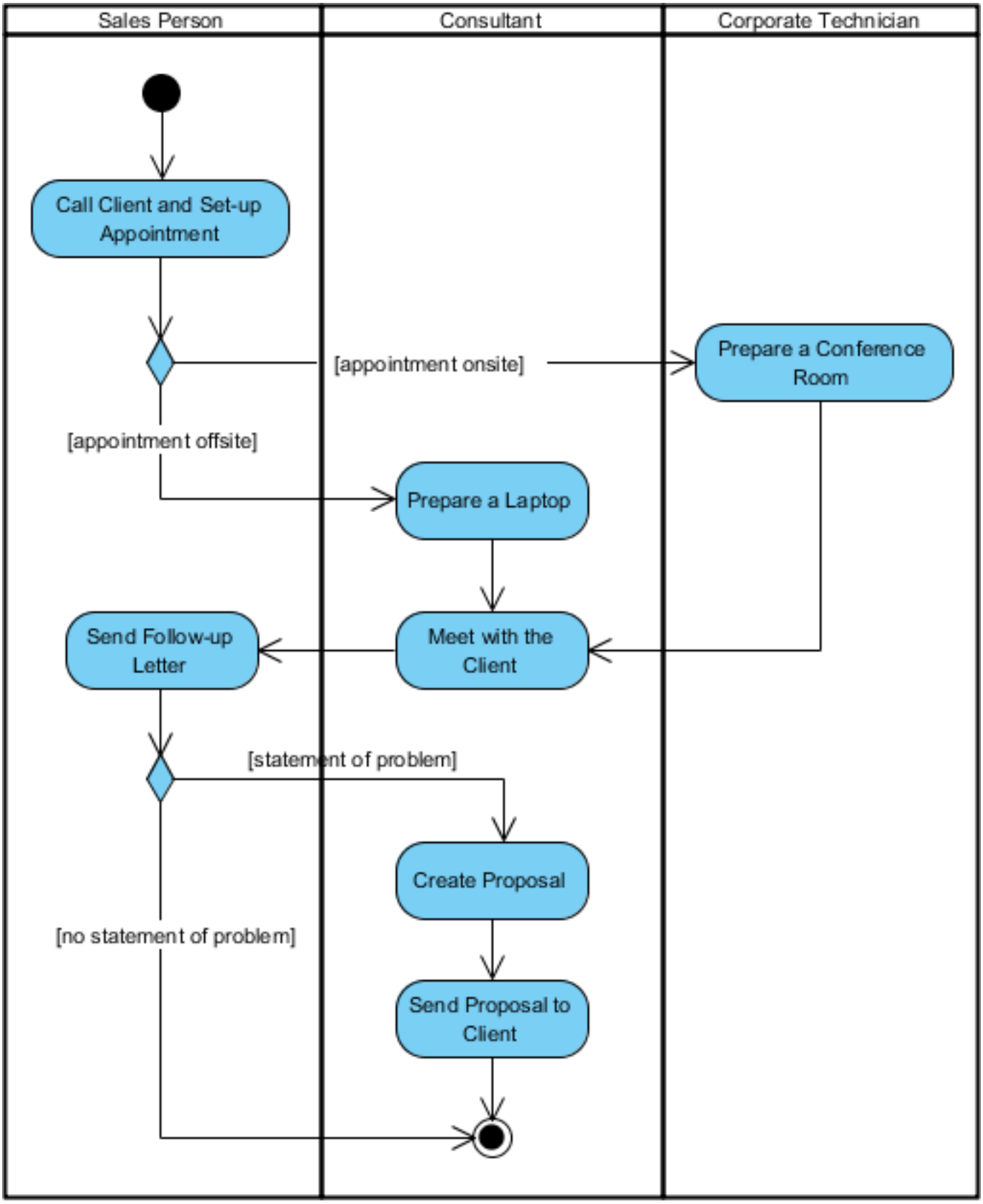
1. Start
2. Validate Order
3. Process Payment
4. Ship Order
5. Send Confirmation
6. End

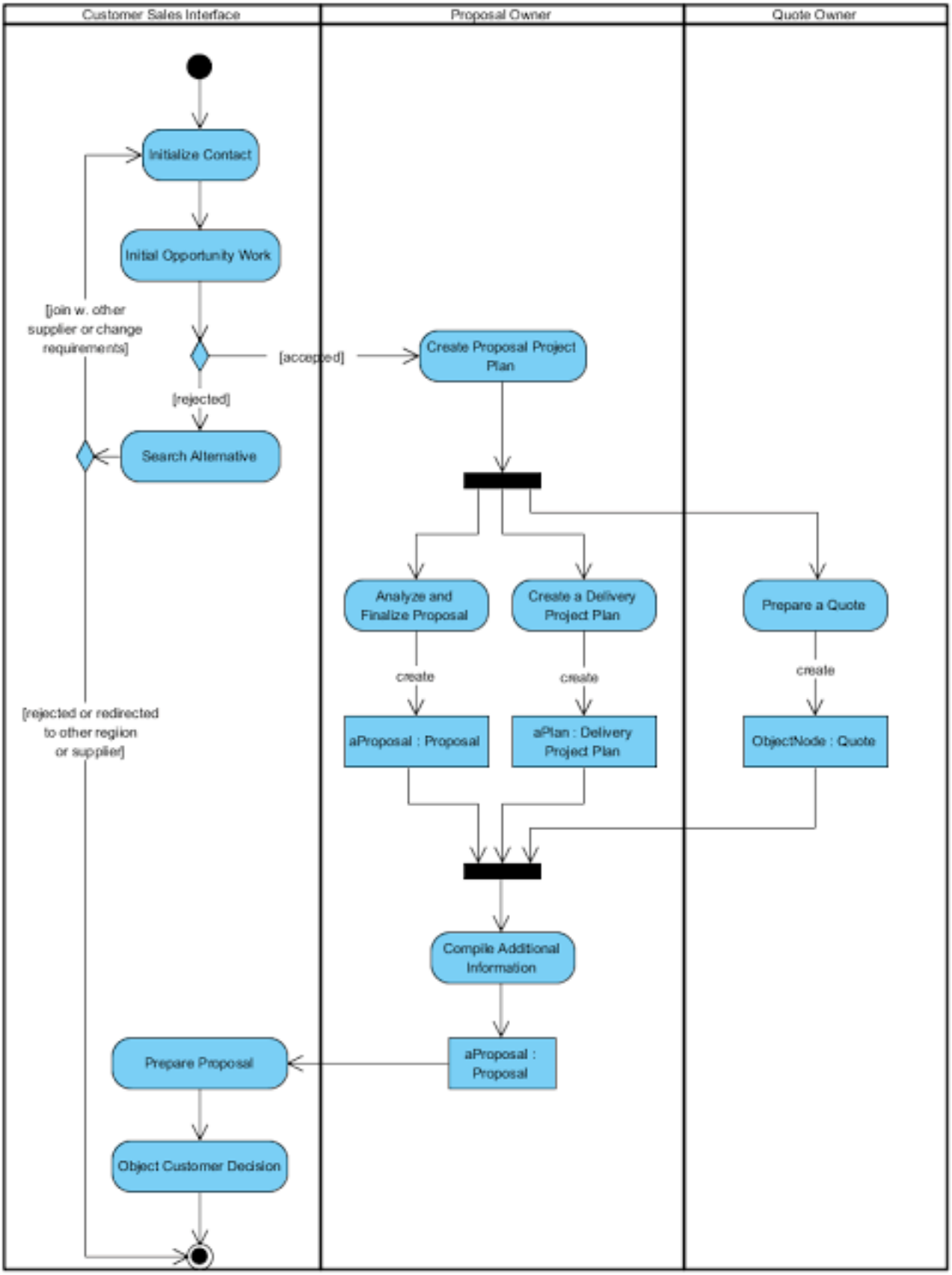



View Booking History - Activity Diagram
User logs into system
Navigate to booking history
"Is Student?"
no
System queries bookings by student ID
"Is Faculty?"
System queries bookings by faculty ID
yes
"View all bookings?"
System queries all bookings
Filter by user
Filter by status
System applies selected filters
System retrieves booking records
Display booking history


Faculty Cancels Booking - Activity Diagram
Faculty logs into system
Navigate to "My Bookings"
View list of active bookings
Select a booking to cancel
Click "Cancel Booking" button
[Confirm cancellation]
[Not Confirm cancellation]
Update booking status to "available"
Cancle students bookings
Abort cancellation
Display "Booking cancelled successfully" message


Admin
System
Open Applications
Login View
Displays the Login Form
Already have an account?
No
List/Registration View
Yes
Fill data
Check Username and Password
Enter Username and Password
Succeed
Yes
No
Displays Menus


Sales Person
Consultant
Corporate Technician
Call Client and Set-up Appointment
[appointment onsite]
Prepare a Conference Room
[appointment offsite]
Prepare a Laptop
Send Follow-up Letter
Meet with the Client
[statement of problem]
Create Proposal
[no statement of problem]
Send Proposal to Client


Customer Sales Interface
Proposal Owner
Quote Owner
Initialize Contact
Initial Opportunity Work
[join w. other supplier or change requirements]
[accepted]
Create Proposal Project Plan
[rejected]
Search Alternative
Analyze and Finalize Proposal
Create a Delivery Project Plan
Prepare a Quote
create
create
create
aProposal : Proposal
aPlan : Delivery Project Plan
ObjectNode : Quote
[rejected or redirected to other region or supplier]
Compile Additional Information
Prepare Proposal
aProposal : Proposal
Object Customer Decision

Sequence Diagram

Definition:

A sequence diagram models the interaction between objects in a time sequence. It shows how processes operate with one another and in what order.

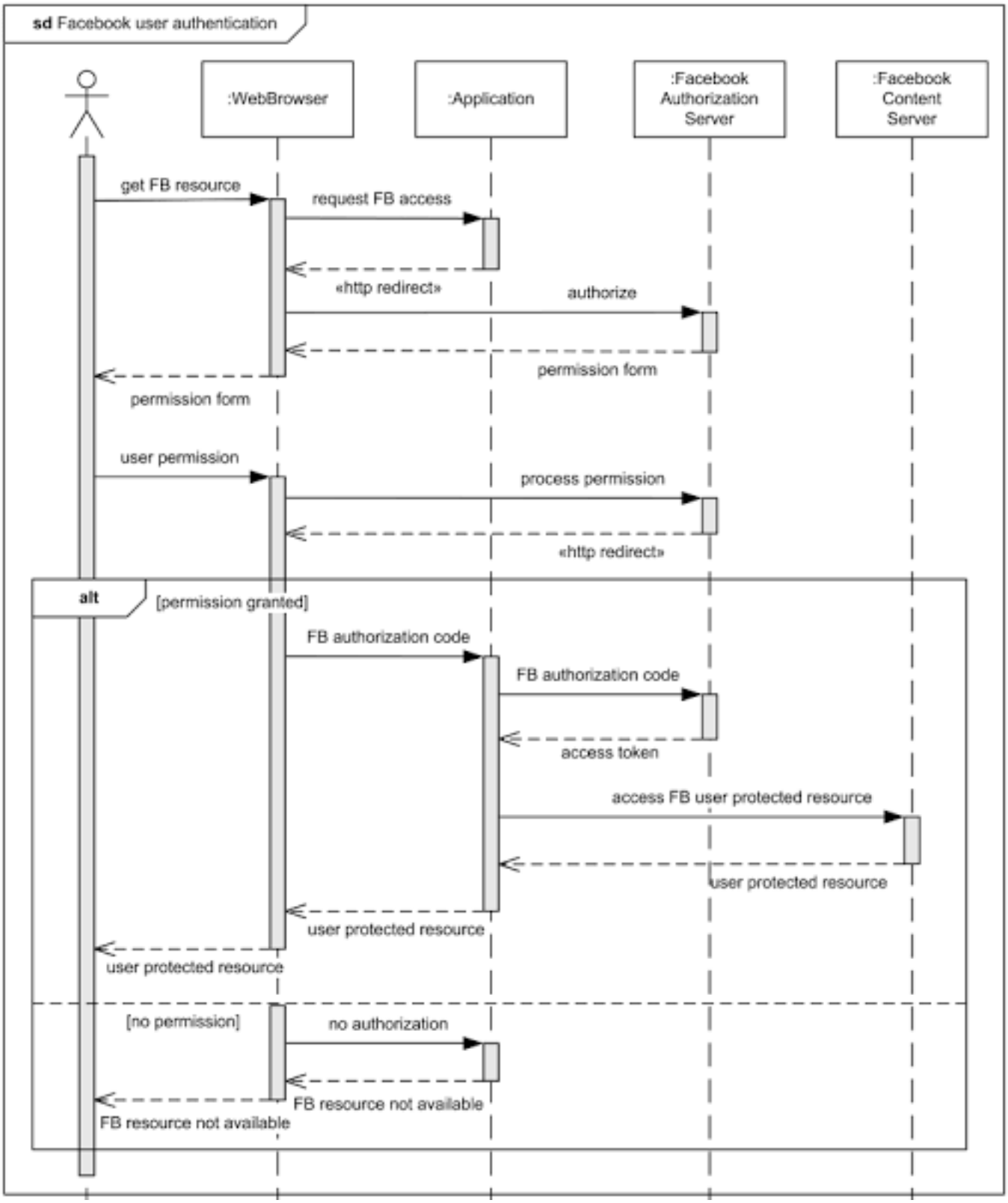
Elements:

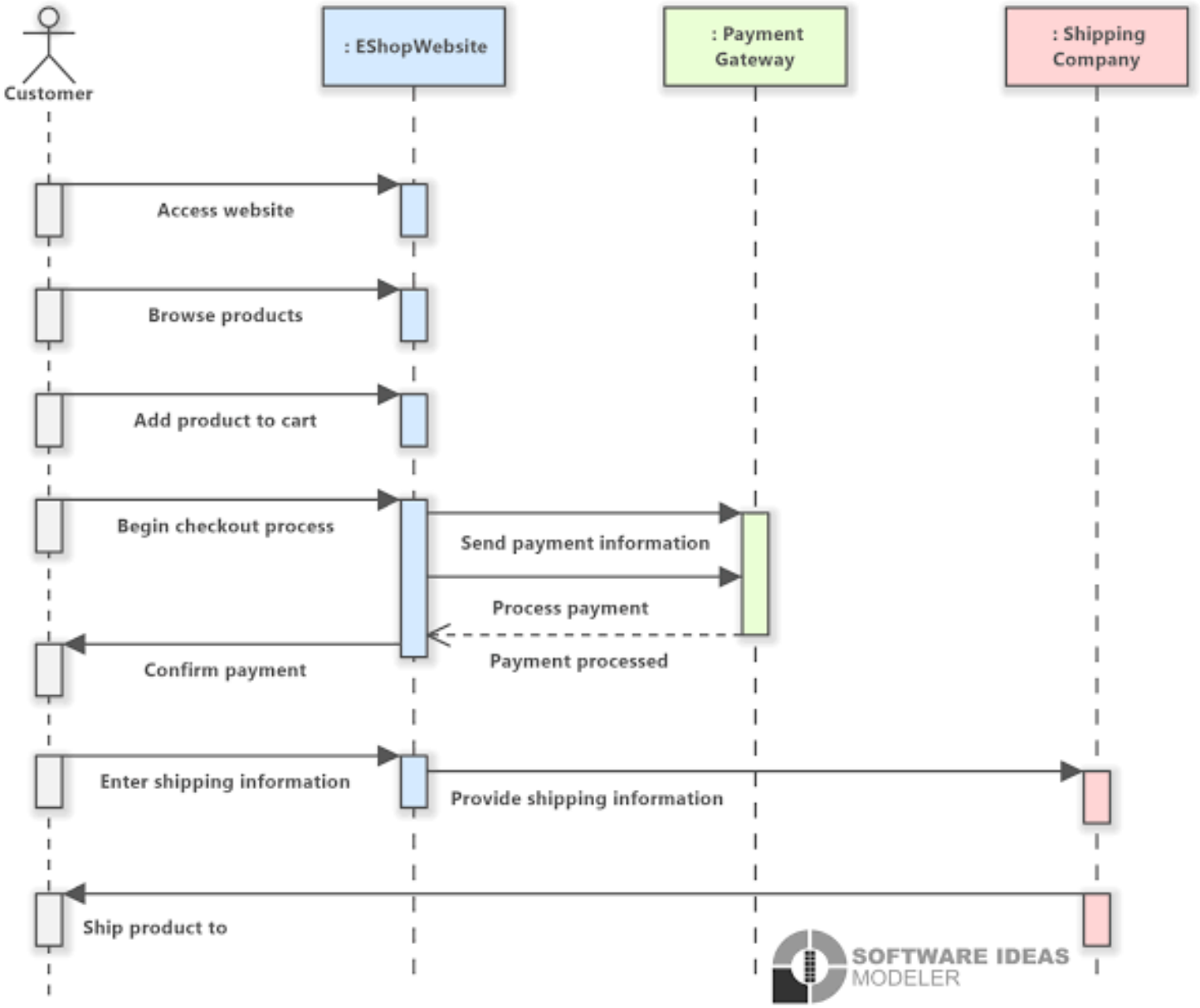
- Actors and Objects (Lifelines)
- Messages (arrows with method calls)
- Activation Bars (indicate control)
- Return Messages

Example:

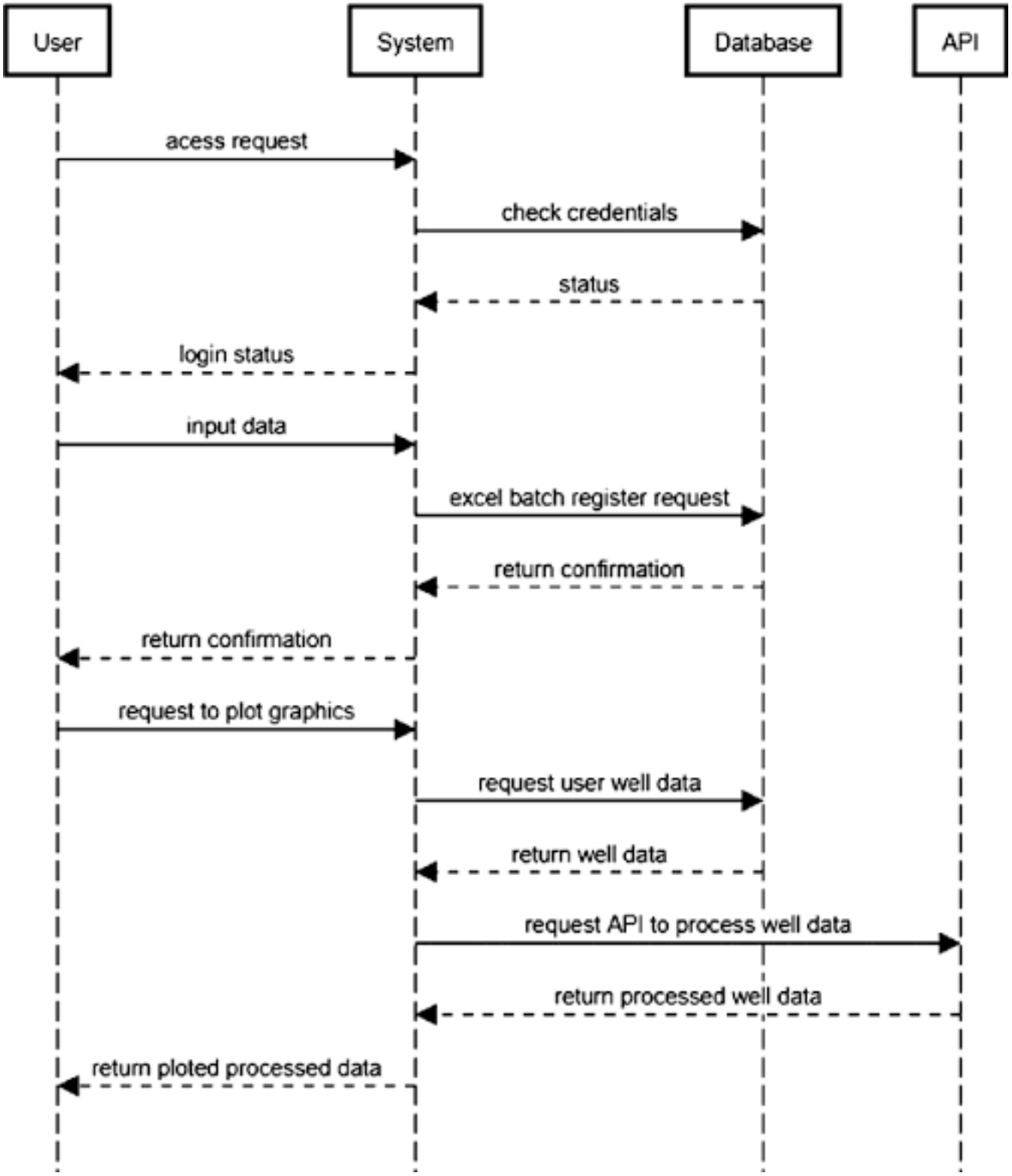
Ordering a book:

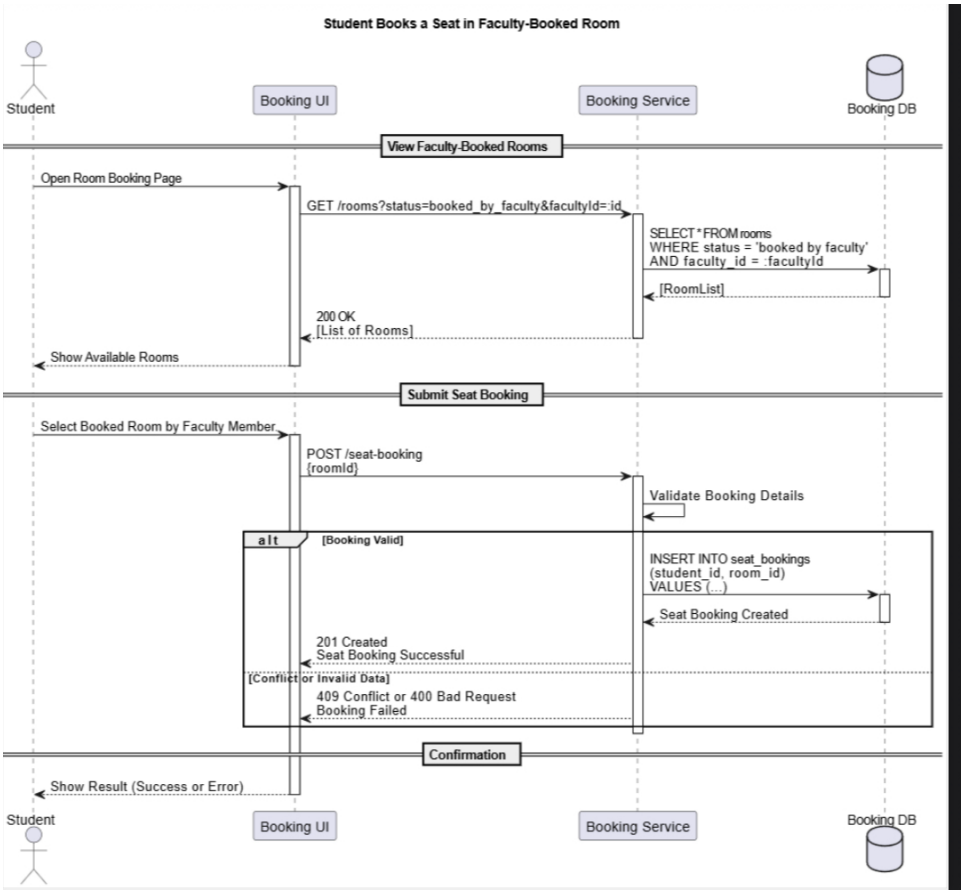
- Customer → OrderService: placeOrder()
- OrderService → Inventory: checkStock()
- OrderService → Payment: processPayment()
- OrderService → Customer: sendConfirmation()

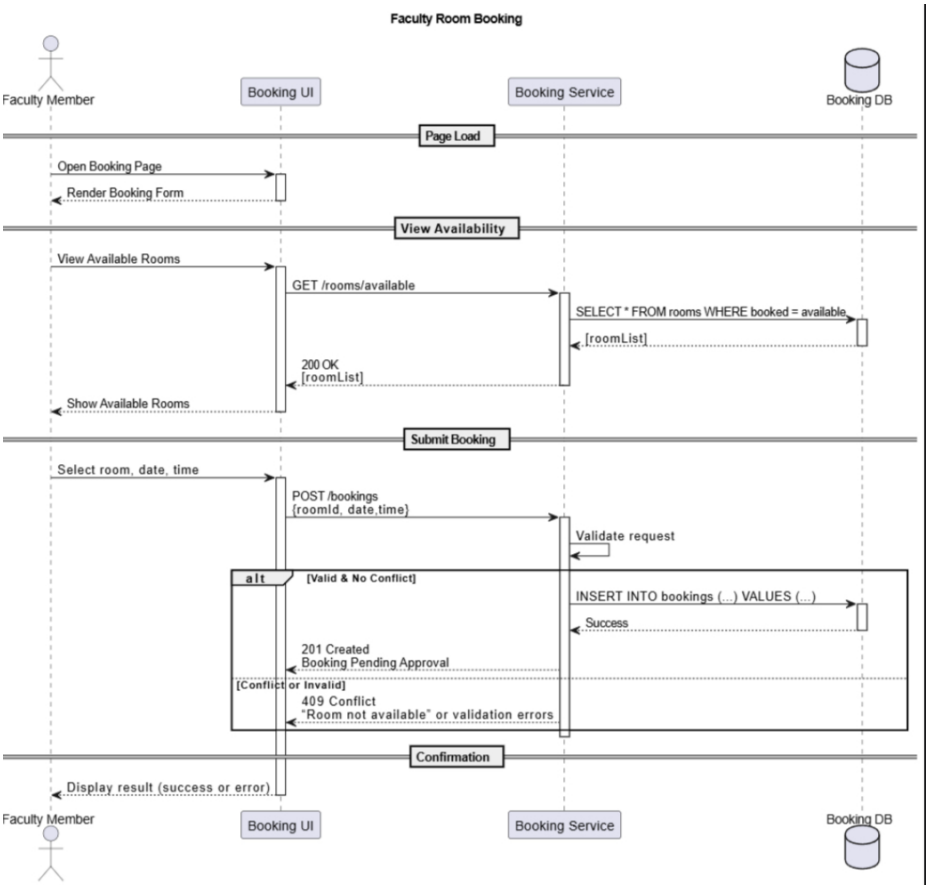

sd Facebook user authentication
:WebBrowser
:Application
:Facebook Authorization Server
:Facebook Content Server
get FB resource
request FB access
«http redirect»
authorize
permission form
permission form
user permission
process permission
«http redirect»
alt
[permission granted]
FB authorization code
FB authorization code
access token
access FB user protected resource
user protected resource
user protected resource
user protected resource
[no permission]
no authorization
FB resource not available
FB resource not available

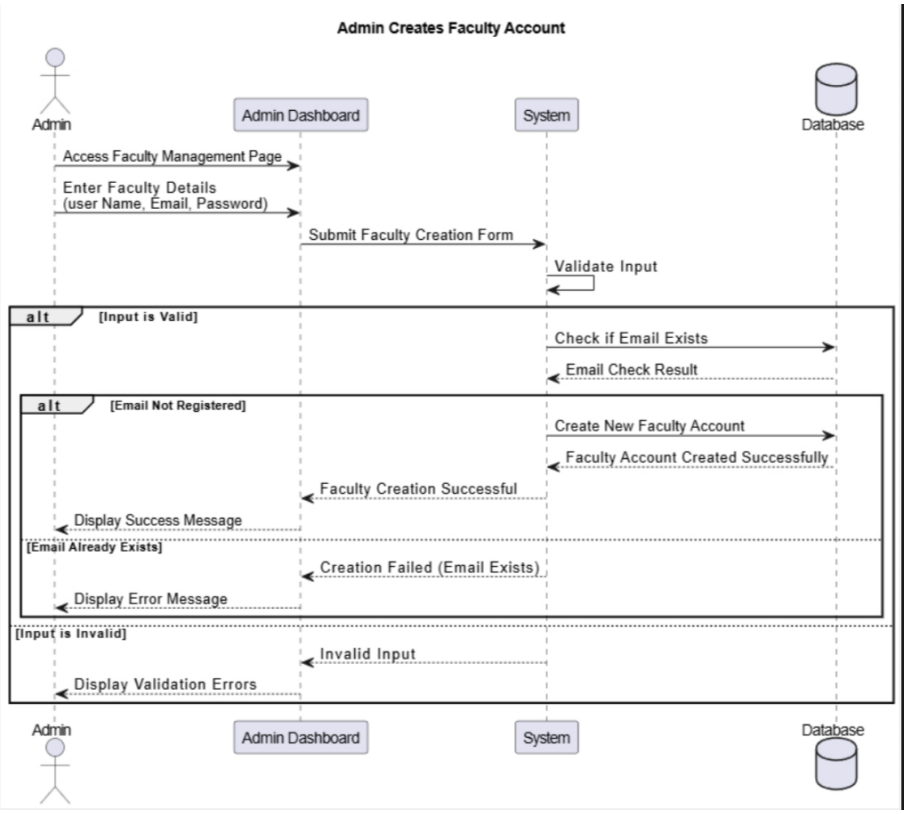

Customer
: EShopWebsite
: Payment Gateway
: Shipping Company
Access website
Browse products
Add product to cart
Begin checkout process
Send payment information
Process payment
Payment processed
Confirm payment
Enter shipping information
Provide shipping information
Ship product to
SOFTWARE IDEAS MODELER

Sequence Diagram User-Platform




Student Books a Seat in Faculty-Booked Room
Student
Booking UI
Booking Service
Booking DB
View Faculty-Booked Rooms
Open Room Booking Page
GET /rooms?status=booked_by_faculty&facultyId=:id
SELECT * FROM rooms
WHERE status = 'booked by faculty'
AND faculty_id = :facultyId
[RoomList]
200 OK
[List of Rooms]
Show Available Rooms
Submit Seat Booking
Select Booked Room by Faculty Member
POST /seat-booking
{roomId}
Validate Booking Details
alt
[Booking Valid]
INSERT INTO seat_bookings
(student_id, room_id)
VALUES (...)
Seat Booking Created
201 Created
Seat Booking Successful
[Conflict or Invalid Data]
409 Conflict or 400 Bad Request
Booking Failed
Confirmation
Show Result (Success or Error)
Student
Booking UI
Booking Service
Booking DB


Faculty Room Booking
Faculty Member
Booking UI
Booking Service
Booking DB
Page Load
Open Booking Page
Render Booking Form
View Availability
View Available Rooms
GET /rooms/available
SELECT * FROM rooms WHERE booked = available
[roomList]
200 OK
[roomList]
Show Available Rooms
Submit Booking
Select room, date, time
POST /bookings
{roomId, date,time}
Validate request
alt
[Valid & No Conflict]
INSERT INTO bookings (...) VALUES (...)
Success
201 Created
Booking Pending Approval
[Conflict or Invalid]
409 Conflict
"Room not available" or validation errors
Confirmation
Display result (success or error)
Faculty Member
Booking UI
Booking Service
Booking DB



Class Diagram

Definition:

A class diagram models the static structure of a system. It shows the system's classes, their attributes, methods, and relationships.

Elements:

- Class: Contains name, attributes, and operations
- Associations: Relationships between classes
- Generalization: Inheritance relationship
- Multiplicity: Specifies how many instances can be associated

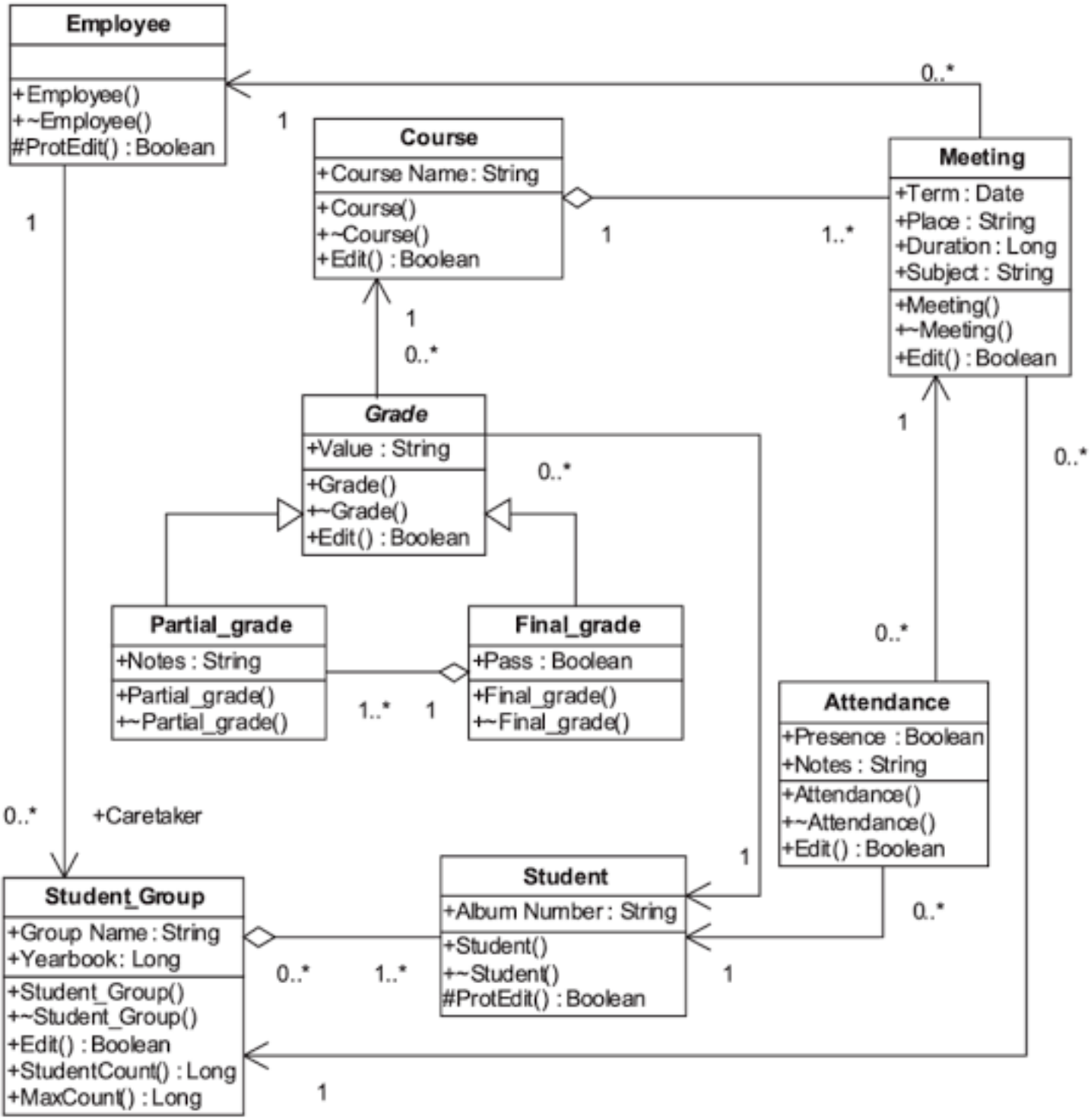
Example:

Classes:

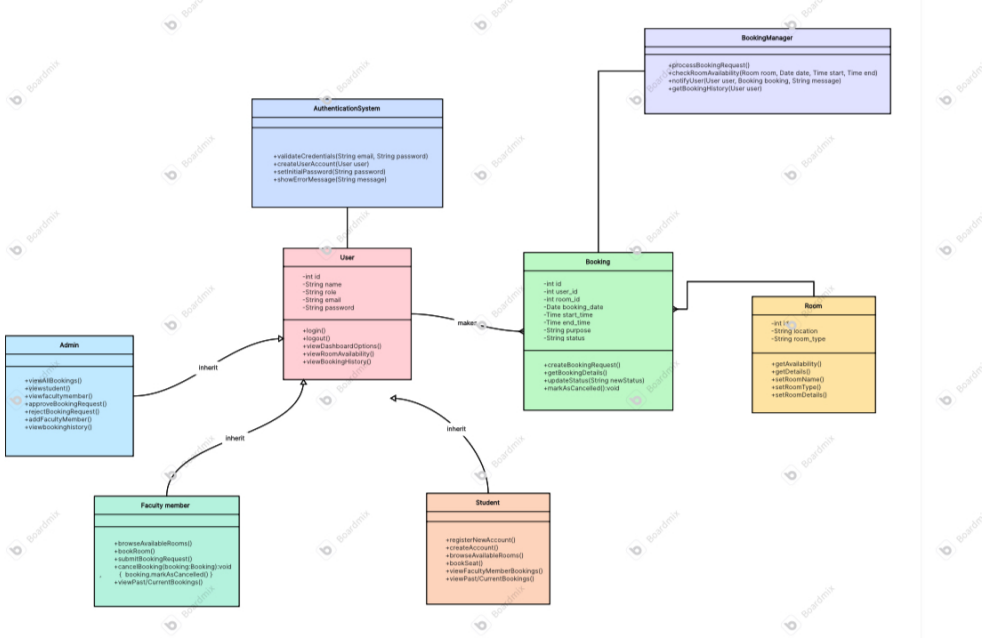
- Customer: name, email, login()
- Order: orderId, date, calculateTotal()
- Product: productId, price

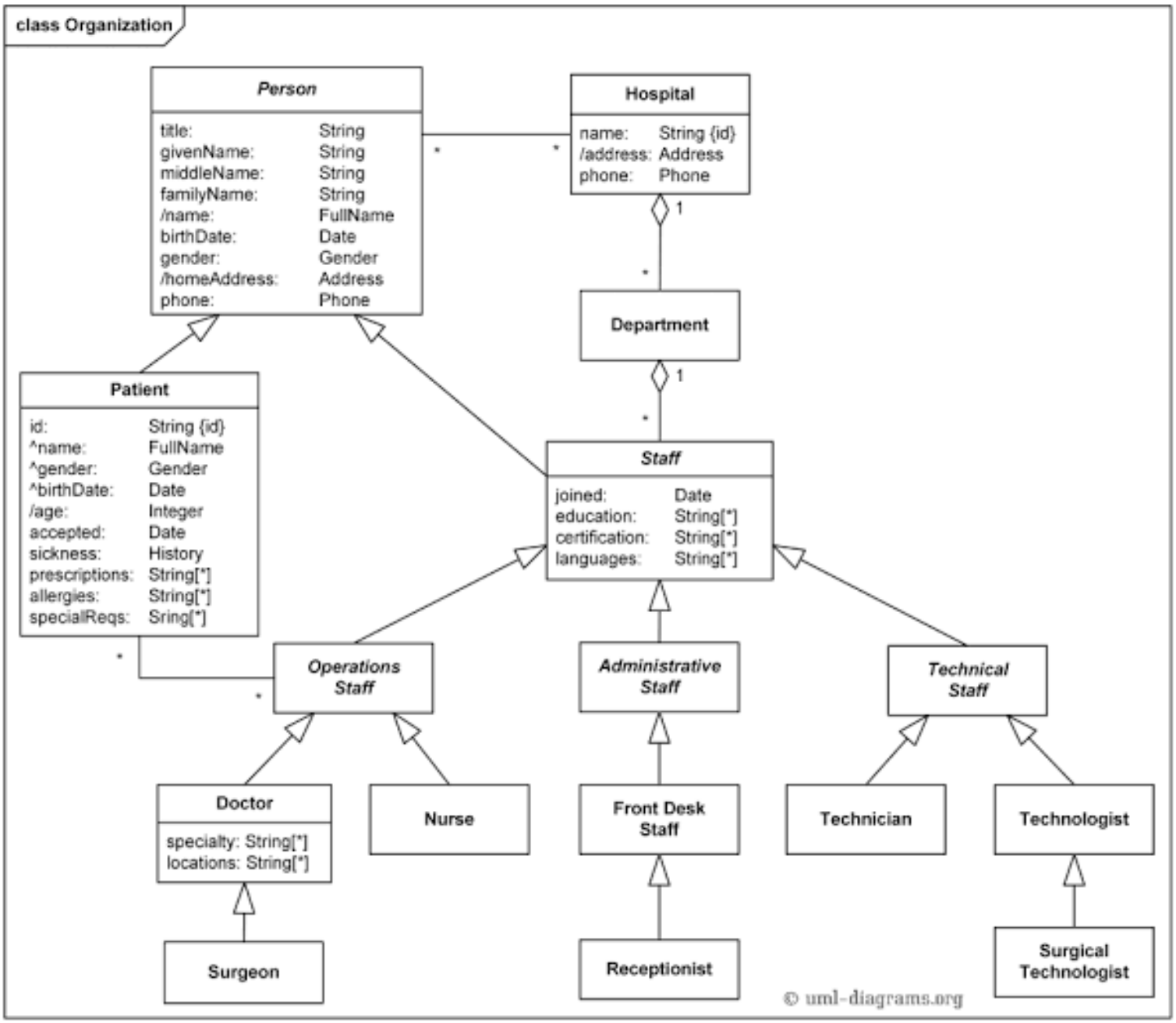
Relationships:

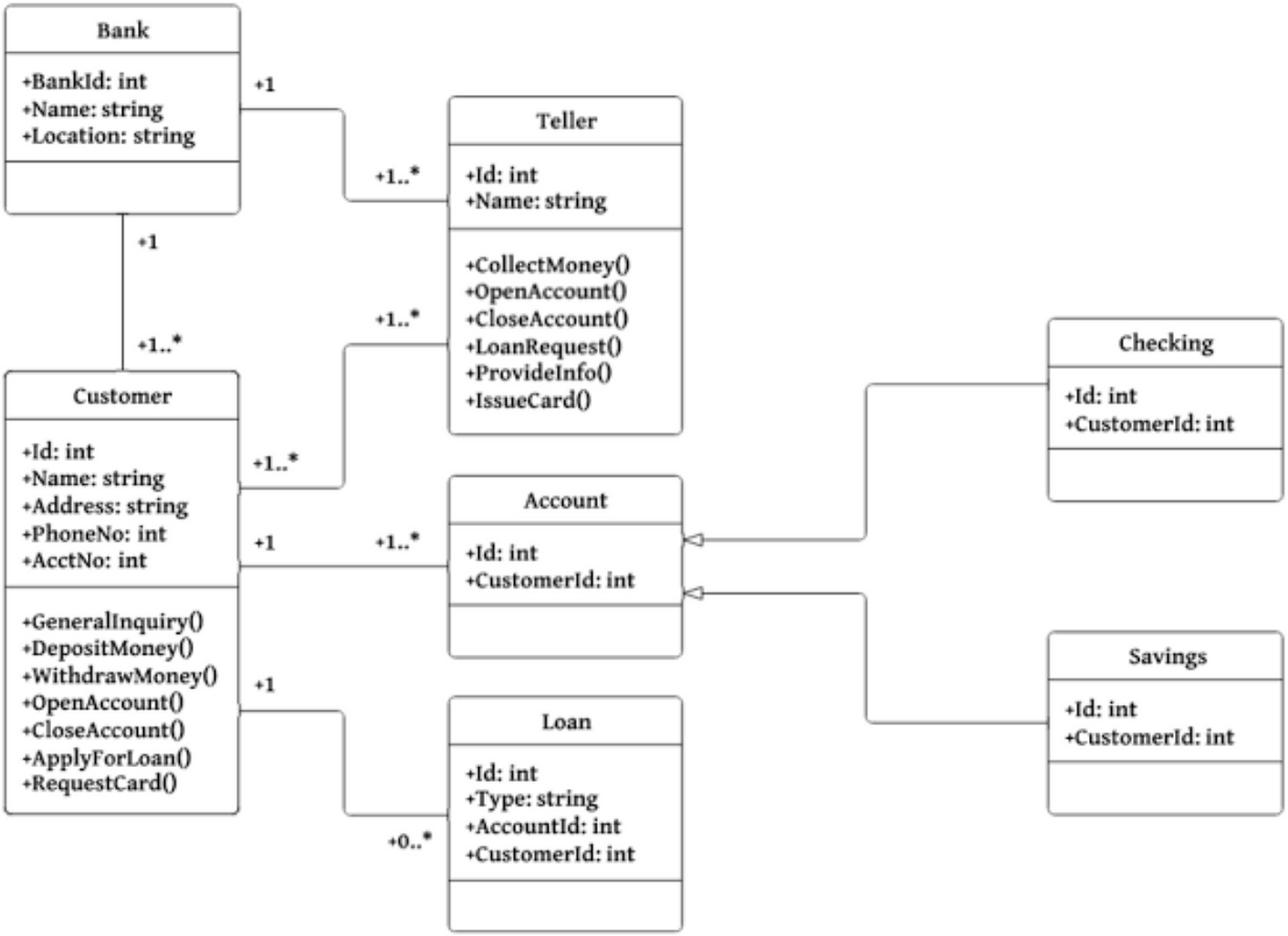
- Customer — Order (One-to-Many)
- Order — Product (Many-to-Many)


Employee
+Employee()
+~Employee()
#ProtEdit() : Boolean
Course
+Course Name : String
+Course()
+~Course()
+Edit() : Boolean
Meeting
+Term : Date
+Place : String
+Duration : Long
+Subject : String
+Meeting()
+~Meeting()
+Edit() : Boolean
Grade
+Value : String
+Grade()
+~Grade()
+Edit() : Boolean
Partial_grade
+Notes : String
+Partial_grade()
+~Partial_grade()
Final_grade
+Pass : Boolean
+Final_grade()
+~Final_grade()
Attendance
+Presence : Boolean
+Notes : String
+Attendance()
+~Attendance()
+Edit() : Boolean
Student_Group
+Group Name : String
+Yearbook: Long
+Student_Group()
+~Student_Group()
+Edit() : Boolean
+StudentCount() : Long
+MaxCount() : Long
Student
+Album Number : String
+Student()
+~Student()
#ProtEdit() : Boolean
0..*
1
1
1..*
1
0..*
1
0..*
0..*
1
0..*
1..*
1
0..*
+Caretaker
1
0..*
1..*
1
1
0..*

university booking system




class Organization
Person
title: String
givenName: String
middleName: String
familyName: String
/name: FullName
birthDate: Date
gender: Gender
/homeAddress: Address
phone: Phone
Hospital
name: String {id}
/address: Address
phone: Phone
Department
Patient
id: String {id}
^name: FullName
^gender: Gender
^birthDate: Date
/age: Integer
accepted: Date
sickness: History
prescriptions: String[*]
allergies: String[*]
specialReqs: Sring[*]
Staff
joined: Date
education: String[*]
certification: String[*]
languages: String[*]
Operations Staff
Administrative Staff
Technical Staff
Doctor
specialty: String[*]
locations: String[*]
Nurse
Front Desk Staff
Technician
Technologist
Surgeon
Receptionist
Surgical Technologist
1
*
© uml-diagrams.org



System Sequence Diagram (SSD)

Definition:

A System Sequence Diagram (SSD) is a type of UML interaction diagram that focuses on the interaction between external actors and the system during a particular use case. It shows the sequence of messages exchanged between an actor (such as a user) and the system (represented as a black box) to carry out a scenario of a use

case.

Purpose:

The main purpose of an SSD is to:

- Understand the behavior of the system in response to external events.
- Capture the flow of messages at the system boundary (not inside internal objects).
- Help in identifying system operations that will later become methods in the system design.

Key Elements:

1. Actors: Represented by stick figures on the left. They initiate events by sending messages to the system.
2. System Box: A rectangle that represents the system as a single object.
3. Messages: Arrows from the actor to the system indicating system operations triggered by user actions.
4. Return Messages: Optional arrows showing data returned from the system to the actor.
5. Sequence: Messages are arranged vertically to indicate the order of execution.

Example Scenario: Place an Order

Use Case: Customer places an order on an e-commerce website.

System Sequence Diagram Flow:

Actor: Customer
System: Online Shopping System

Sequence:
1. Customer → System: selectProduct(productId)
2. Customer → System: addToCart(productId, quantity)
3. Customer → System: viewCart()
4. Customer → System: checkout()
5. Customer → System: enterPaymentDetails(cardInfo)
6. Customer → System: confirmOrder()
7. System → Customer: displayOrderConfirmation()

Benefits of SSD:

- Clarifies interactions between the user and system in early design.
- Helps identify necessary system operations.
- Supports creation of more detailed design diagrams such as

class and sequence diagrams.

- Useful in requirements analysis to validate scenarios with stakeholders.

———

When to Use SSD:

- During use case analysis to describe system behavior per use case scenario.
- When defining the system boundary and interaction between external entities and the system.
- As a communication tool between analysts, designers, and stakeholders.

———

Notes:

- SSD does not show internal class or object interactions—only between actors and the system.
- It usually focuses on one use case scenario at a time.
- The system is considered a black box, and internal details are hidden.

UML Diagrams Are NOT Enough!

Why UML Diagrams Are Insufficient Alone

UML (Unified Modeling Language) provides a powerful set of visual modeling diagrams for representing object-oriented systems. These diagrams—such as class diagrams, use case diagrams, sequence diagrams, and activity diagrams—are excellent for communicating structure and behavior.

However, UML diagrams alone are not enough to fully and precisely specify the complete semantics and constraints of a system. Visual notations often leave room for ambiguity and incompleteness. For example:

- A class diagram may define associations between classes but cannot enforce constraints like "a person must have exactly one passport."
- A state diagram may describe transitions but not capture guard conditions in a formal, verifiable way.

To address this, we need a formal language that can complement UML and allow us to define precise rules and constraints.

The Need for Specification Languages

- Specifications are formal definitions of behavior and rules that cannot be fully visualized.
- We need something that can augment UML without replacing it

—a non-graphical “add-on” that provides precision.

- The ideal specification language must be:
- Formal and unambiguous
- Object-oriented friendly
- Able to express preconditions, postconditions, invariants, constraints, etc.

———

Why Not First-Order Logic?

First-order logic (FOL) is a powerful formalism used in mathematics and AI. However, FOL is:

- Not object-oriented
- Lacks support for modeling classes, inheritance, and object relationships directly
- Not easily integrated with UML’s object-oriented models

Thus, while useful in some domains, FOL does not align well with object-oriented system design.

———

Object Constraint Language (OCL)

OCL was introduced as a textual complement to UML. It is:

- A declarative language for describing rules applied to UML models.
- Used to specify:
- Class invariants
- Preconditions and postconditions on operations
- Derivation rules for attributes
- Constraints on associations and multiplicities

Example:

```
context Person
inv: self.age > 0

context Order::submit()
pre: self.items->size() > 0
post: self.status = 'submitted'
```

This OCL snippet:

- Ensures a person’s age is positive
- Requires that an order has at least one item before submission
- Ensures the order’s status changes after submission

———

Are There Other Options?

Yes, other formal specification languages exist, such as:

- Z notation
- Alloy
- VDM (Vienna Development Method)

However, these are:

- Not tightly integrated with UML
- Often more mathematical and harder for software engineers to adopt in practice

---

Why Use OCL?

- Standardized: OCL is part of the UML standard (by OMG).
- Designed for OO: Fits seamlessly with classes, objects, and UML semantics.
- Tool Support: Many UML tools support OCL evaluation and validation.
- Readable: Easier to understand for software engineers compared to mathematical notations.

Conclusion

While UML diagrams are powerful for visualization and communication, they lack the expressiveness needed to define all the necessary system

rules and behavior. To build robust, correct, and verifiable systems, we need a formal specification language. OCL (Object Constraint Language) provides that missing piece—bringing precision and formalism to UML modeling without requiring a completely new modeling approach.

Constraints (Invariants), Contexts, and self in OCL

What is a Constraint (Invariant)?

In Object Constraint Language (OCL), a constraint—particularly an invariant—is a boolean expression that must always evaluate to either true or false.

An invariant defines a rule that must always hold true for every instance of a class during its lifetime, ensuring that the object remains in a valid state.

Context of a Constraint

Every OCL constraint is written within a specific context—this means the constraint is associated with a particular element in the UML

model, such as:

- A class
- An association class
- An interface

This tells us where the constraint logically “belongs” and what data it can access.

The context is specified as:
context <ClassName>

Using self in OCL

Inside an OCL expression, the keyword self is used to refer to the current instance of the context class.

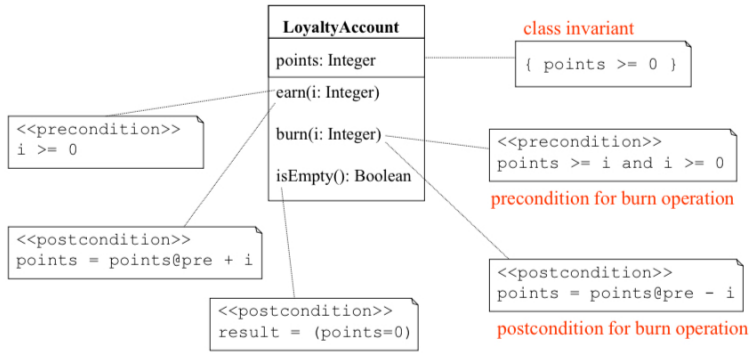
For example, if your context is the Person class, then:
self.age means the age property of that specific Person instance.

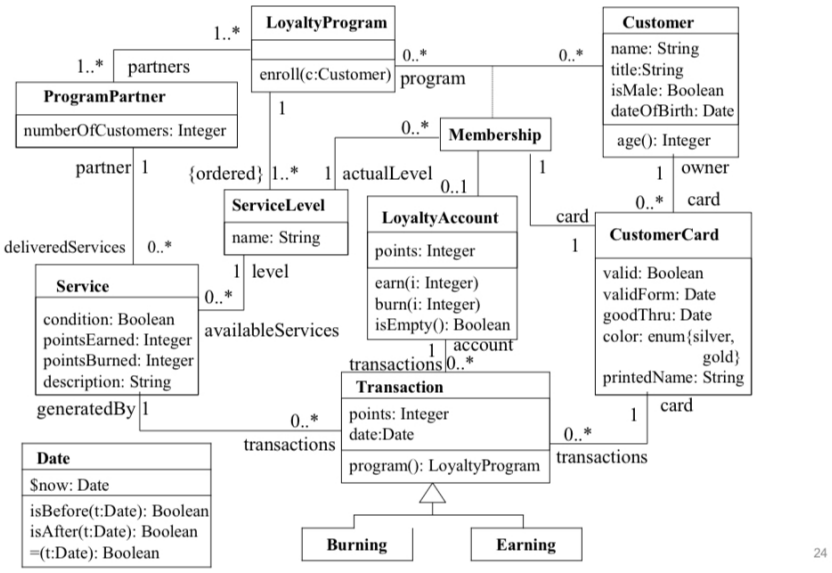
Defining Invariants

You can optionally name an invariant to make it more readable and traceable:

context Person
inv PositiveAge: self.age > 0

This defines an invariant named PositiveAge for the Person class, stating that a person's age must always be positive.





OCL Invariants on Attributes and Associations

🔷 1. Invariants on Attributes

Invariants can be defined directly on attributes of a class to enforce business rules.

✅ Example:

context Customer

invariant agerestriction: age >= 18

This invariant ensures that every Customer must be at least 18 years old.

✅ Another Example:

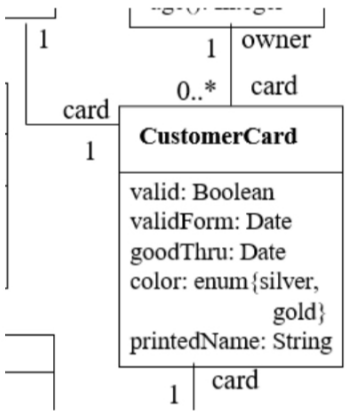
context CustomerCard

invariant correctDates: validFrom.isBefore(goodThru)

- validFrom and goodThru are attributes of type Date.
- isBefore(Date): Boolean is a standard OCL operation for Date.

Meaning: The card's validFrom date must be before its goodThru (expiry) date.

🔷 2. Invariants Using Navigation Over Association Ends



In OCL, we can navigate associations using the role name to reach related objects.

✅ Example: context CustomerCard

invariant: owner.age >= 18

- owner is an association from CustomerCard to Customer.
- owner.age accesses the age of the associated Customer.

This invariant ensures that the card owner is at least 18 years old.

🔷 3. Invariants with String Operations (Using Navigation)

You can use navigation and string operations to build expressions.

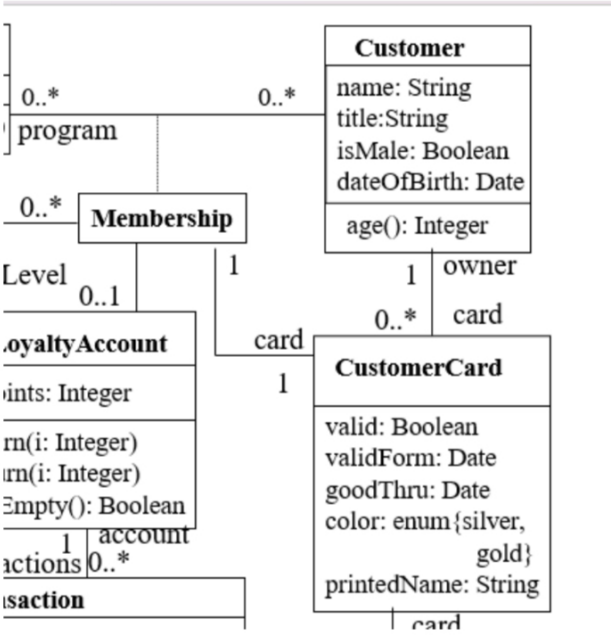
✅ **Example:context CustomerCard**

```
invariant printedName:
  printedName = owner.title.concat(' ').concat(owner.name)
```

- owner refers to a Customer.
  - owner.title and owner.name are strings.
  - concat(String): String is a built-in OCL string operation.

This invariant ensures the printedName on the card is the title and name of the owner, separated by a space.

🔷 **4. Invariants on Associations with Multiplicity "Many"**



When an association has a multiplicity greater than 1 (*), it results in a collection (Set or Bag) in OCL.

OCL provides collection operations such as ->includes, ->size, ->forAll, and more.

✅ **Example: context CustomerCard**

```
invariant correctCard:
  owner.Membership->includes(membership)
```

Explanation:

- CustomerCard has an association to Membership.
- owner refers to a Customer who can have multiple Memberships.
- The invariant checks that the membership linked to the CustomerCard is among the memberships of its owner.

Operations:

- ->includes(element) checks if the collection contains a specific element.

OCL Type Hierarchy

OCL (Object Constraint Language) defines a strong, well-structured type hierarchy that supports both basic types (like numbers and

strings) and object-oriented types (like classes and collections). This hierarchy ensures type safety and enables type-specific operations in OCL expressions.

🔷 1. Root Type: OclAny

At the top of the OCL type hierarchy is OclAny.

- All other types inherit from OclAny.
- It defines the most general operations, such as:
- =, <> (equality and inequality)
- oclIsTypeOf(), oclIsKindOf()
- oclIsUndefined()

🔷 2. Primitive Types

Primitive data types represent basic values. These types are directly derived from OclAny.

- Integer → Whole numbers (e.g., 1, 42)
- Real → Floating-point numbers (e.g., 3.14)
- Boolean → true or false
- String → Text data (e.g., “hello”)

Each primitive type supports standard operations:

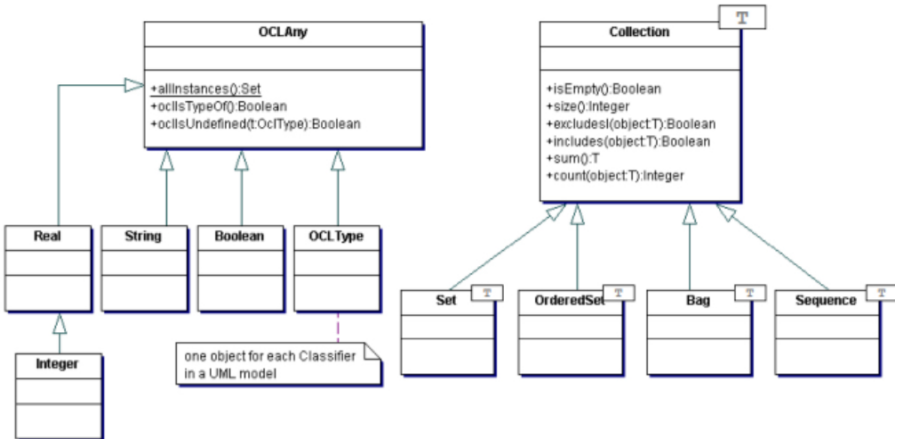
Type	Example Operation	Description
Integer	x + 2, x < y	Arithmetic and comparison
Real	x / 3.0, x > y	Real number operations
Boolean	a and b, not a	Logical operations
String	s.concat(" world")	String concatenation

🔷 3. Collection Types

OCL provides types for working with multiple elements.

All collection types are subtypes of Collection(T) where T is the type of the elements.

Type	Description
Set(T)	Unordered, no duplicates
Bag(T)	Unordered, allows duplicates
Sequence(T)	Ordered, allows duplicates
OrderedSet(T)	Ordered, no duplicates



OCL Collection Operations :

1. size

Returns the number of elements in a collection.

Example: context Order

invariant: items->size() > 0

2. isEmpty

Checks if the collection is empty.

Example:context ShoppingCart

invariant: products->isEmpty()

3. notEmpty

Checks if the collection is not empty.

Example: context ShoppingCart

invariant: products->notEmpty()

4. sum()

Returns the sum of numeric values in a collection.

Example: context Invoice

invariant: lineItems.price->sum() <= totalAmount

5. count(object)

Counts how many times a specific object appears in the collection.

Example:context Library

invariant: books->count(Book.allInstances()->any(b | b.title = 'OCL Guide')) <= 5

6. excludes(object)

Checks if an object is not in the collection.

Example: context Customer

invariant: orders->excludes(Order.allInstances()->any(o | o.status = 'Cancelled'))

7. includes(object)

Checks if an object is in the collection.

Example: context Customer

invariant: orders->includes(self.recentOrder)

8. includesAll(collection)

Checks if all elements of another collection are in the current

collection.
Example: context Warehouse
invariant: inventory->includesAll(requiredProducts)

Filtering and Transformation Operations

9. select(e:T | boolean expression)
Filters elements that satisfy a condition.
Example: context Order
invariant: items->select(i | i.quantity > 0)->notEmpty()

10. reject(e:T | boolean expression)
Filters elements that do not satisfy a condition.
Example: context Student
invariant: courses->reject(c | c.grade < 50)->size() >= 3

11. collect(e:T | value expression)
Transforms each element in the collection into another value.
Example:context Invoice
invariant: lineItems->collect(i | i.price * i.quantity)->sum() = total

12. forAll(e:T | boolean expression)
Checks if all elements satisfy the condition.

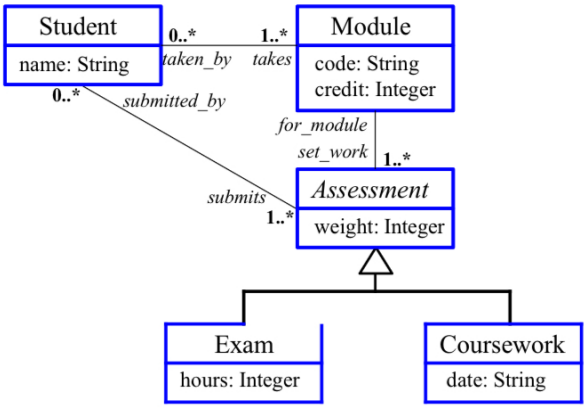
Example:context Course
invariant: students->forAll(s | s.age >= 18)

13. exists(e:T | boolean expression)
Checks if at least one element satisfies the condition.
Example: context Order
invariant: items->exists(i | i.discount > 0)

14. iterate(e:T1; r:T2 = initial | expression)
Iterates over the collection and accumulates a result.
Example: context Invoice
invariant: lineItems->iterate(i:Item; total:Real = 0 | total + (i.price * i.quantity)) = totalAmount

---

As :



a) Modules can be taken only if they have more than seven students registered.

b) The assessments for a module must total 100%.

c) Students must register for 120 credits each year.

d) Students must take at least 90 credits of Computer Science (CS) modules each year.

e) All modules must have at least one assessment worth over 50%.

f) Students can only have assessments for modules they are actively taking.