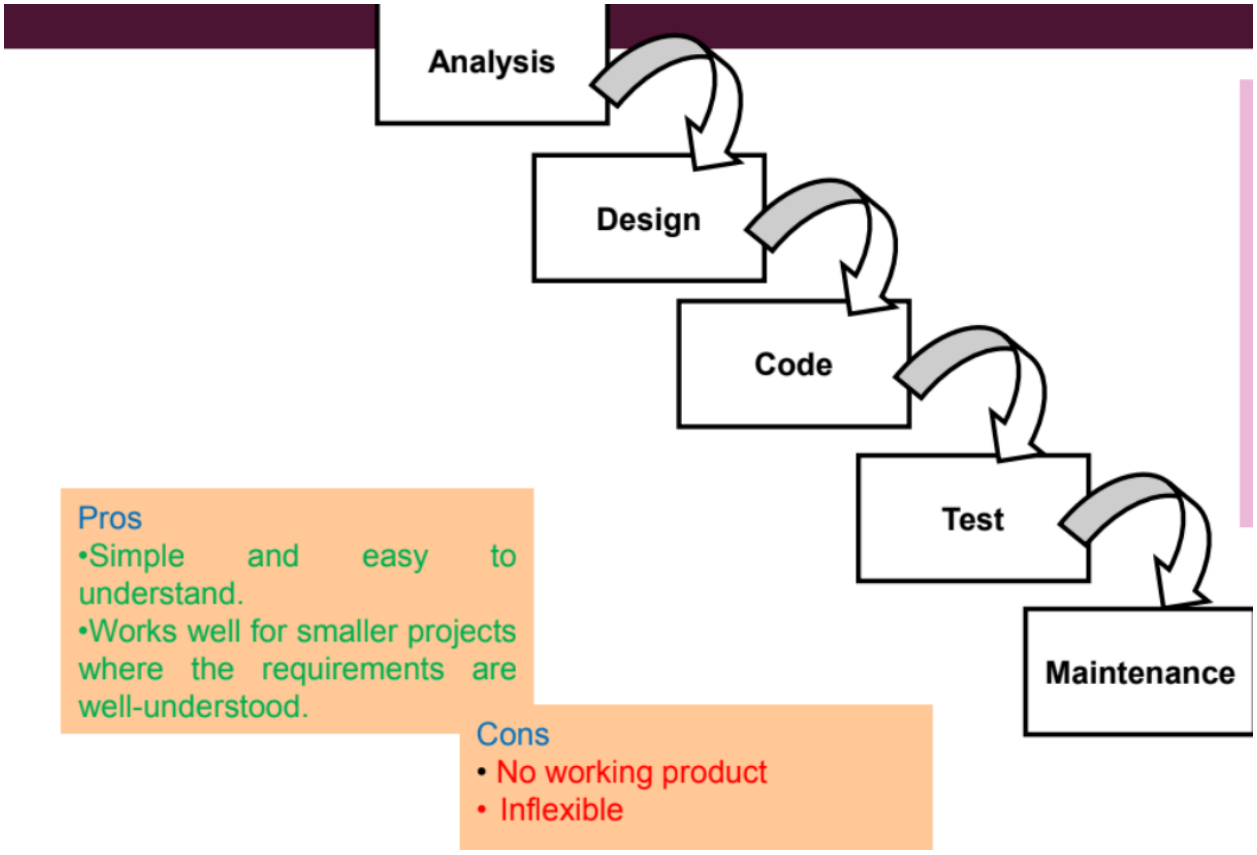
SDA MID I NOTES

SDLC MODELS

1. Waterfall Model

- A **linear and sequential** SDLC model where each phase (Requirements → Design → Implementation → Testing → Deployment → Maintenance) must be completed before the next one begins.
- Once a phase is finished, it is usually not revisited.
- **Key Idea:** Like water flowing down steps, no going back.
- **Best for:** Projects with fixed, well-defined requirements.



Pros
•Simple and easy to understand.
•Works well for smaller projects where the requirements are well-understood.

Cons
• No working product
• Inflexible

2. Incremental Model (SDLC)

Definition

The **Incremental Model** is a software development approach where the system is built and delivered in **increments (modules or parts)**. Each increment adds new features to the system until the full software is complete. Unlike Iterative (which focuses on improving the same version repeatedly), **Incremental focuses on delivering complete parts step by step**.

How It Works (Phases)

1. **Requirements Analysis** – Break the system into smaller functional parts.
2. **Design & Development of First Increment** – Build a fully working module (e.g., login system).
3. **Delivery of Increment** – Provide it to users for use/feedback.
4. **Next Increment** – Develop the next module (e.g., student registration, then fees, then exams).
5. **Repeat** – Continue until all increments are combined into the full system.

Example (Scenario)

Suppose we are building an **Online Shopping System**:

- **Increment 1:** User login & registration.
- **Increment 2:** Product catalog & search.
- **Increment 3:** Shopping cart & checkout.
- **Increment 4:** Payment & order tracking.

Each increment is a **complete usable module** that integrates with the previous one.

Advantages

- Working software delivered early.
- Easier to test and debug smaller parts.
- Users can start using the system before it's fully complete.

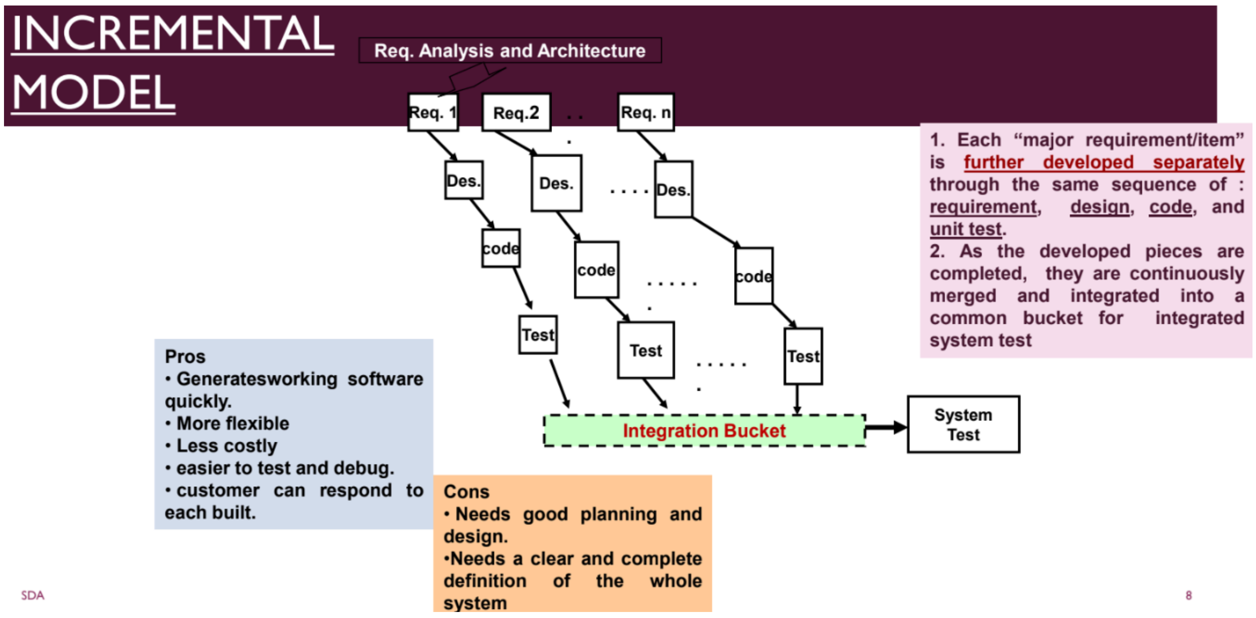
- Flexible to add new features in later increments.

Disadvantages

- Requires good planning and modular design.
- Integration of increments can be complex.
- If requirements change frequently, may cause rework.

Difference Between Iterative & Incremental

- **Iterative:** Improves the **same version** repeatedly until it's refined.
- **Incremental:** Builds the **system in parts (modules)** and delivers each part step by step.



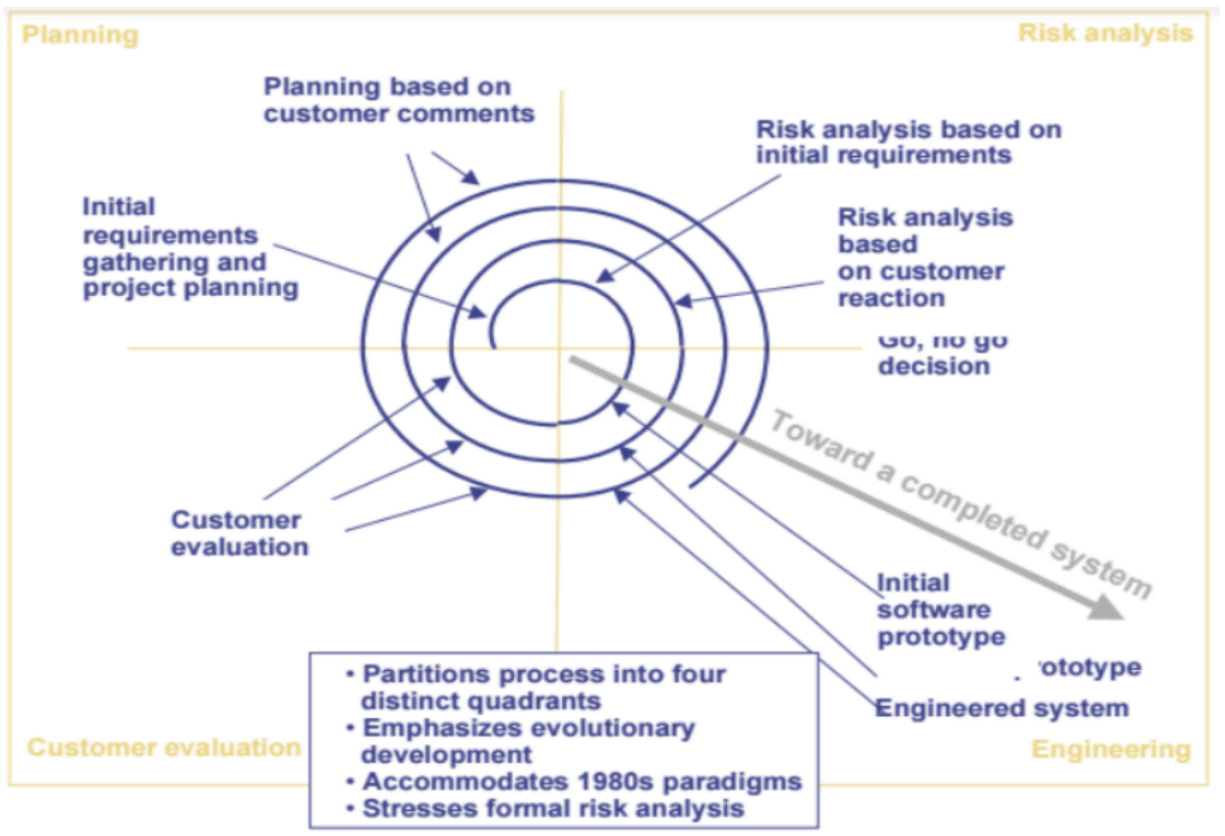
1. Each "major requirement/item" is further developed separately through the same sequence of : requirement, design, code, and unit test.
2. As the developed pieces are completed, they are continuously merged and integrated into a common bucket for integrated system test

Pros
• Generatesworking software quickly.
• More flexible
• Less costly
• easier to test and debug.
• customer can respond to each built.

Cons
• Needs good planning and design.
•Needs a clear and complete definition of the whole system

3. Iterative Model

- Development is done in **repeated cycles (iterations)**.
- Each iteration delivers a small working version of the software, which is improved in the next cycle until the final system is complete.
- **Key Idea:** Build → Review → Improve → Repeat.
- **Best for:** Large projects where requirements may evolve over time.


Planning
Risk analysis
Planning based on customer comments
Risk analysis based on initial requirements
Initial requirements gathering and project planning
Risk analysis based on customer reaction
Go, no go decision
Toward a completed system
Customer evaluation
Initial software prototype
ototype
Engineered system
Customer evaluation
Engineering
• Partitions process into four distinct quadrants
• Emphasizes evolutionary development
• Accommodates 1980s paradigms
• Stresses formal risk analysis

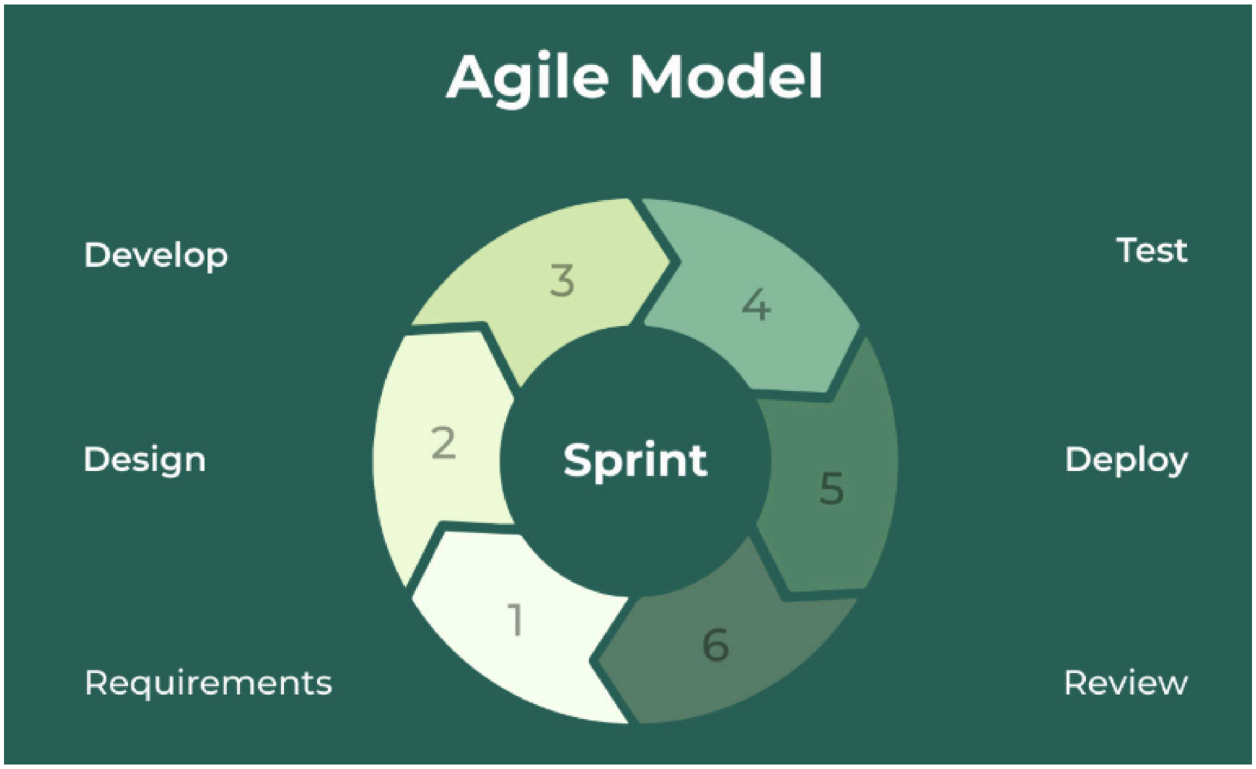
4. Prototype Model

- A **working prototype/mock-up** of the system is built quickly to show how the software will look and function.
- Users give feedback on the prototype, and based on that, the final system is developed.
- **Key Idea:** “Build a sa mple first, then refine into the real product.”
- **Best for:** Projects with unclear or incomplete requirements.

5. Agile Model

- A **flexible, iterative, and incremental** SDLC model.

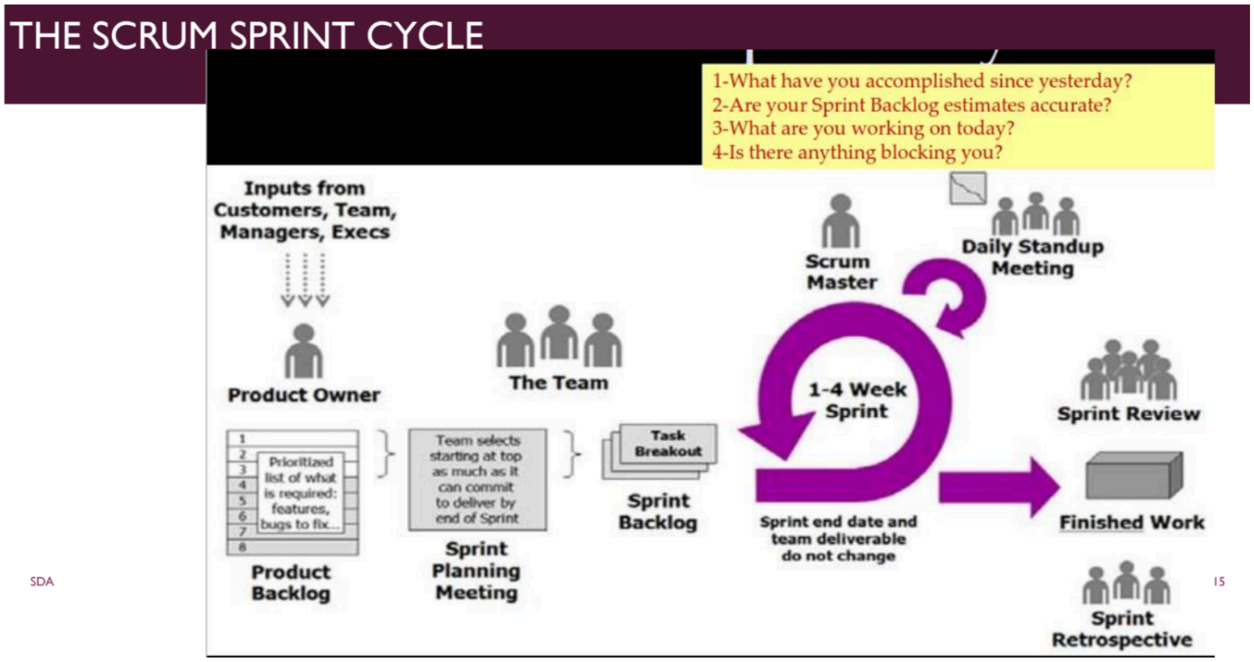
- Development is divided into **short cycles (sprints)**, each delivering a usable product feature.
- Involves **continuous user collaboration, adaptation to change, and frequent delivery** of working software.
- **Minimal documentation**: Focus on **working software** instead of lengthy documents.
- **Key Idea:** Customer collaboration + quick response to change.
- **Best for:** Projects with changing requirements and need for fast delivery.



Plan-driven vs Agile Development

Aspect	Plan-Driven Development	Agile Development
Process	Separate, sequential stages (requirements → design → implementation → testing).	Activities (specification, design, coding, testing) are interleaved.

Aspect	Plan-Driven Development	Agile Development
Requirements	Defined in advance, relatively stable.	Emerge and evolve through user collaboration.
Iteration	Limited (within stages).	Continuous, with short development cycles.
Outputs	Planned documents & deliverables for each stage.	Working software, decided through negotiation with stakeholders.
Flexibility	Low (changes are costly).	High (changes integrated easily).
Best suited for	Large, safety-critical, regulated systems.	Dynamic business applications, startups, and projects needing fast delivery.


THE SCRUM SPRINT CYCLE

1-What have you accomplished since yesterday?
2-Are your Sprint Backlog estimates accurate?
3-What are you working on today?
4-Is there anything blocking you?

Inputs from Customers, Team, Managers, Execs
Product Owner
The Team
Scrum Master
Daily Standup Meeting
1-4 Week Sprint
Sprint Review
Prioritized list of what is required: features, bugs to fix...
Team selects starting at top as much as it can commit to deliver by end of Sprint
Task Breakout
Sprint Backlog
Sprint end date and team deliverable do not change
Finished Work
Product Backlog
Sprint Planning Meeting
Sprint Retrospective
SDA
15

JUDGE THE TYPE OF SDLC MODEL BY SCENARIOS AND JUSTIFY WHICH SHOULD BE USED.?

Scenario 1

A government organization needs a **payroll system** for its employees. All requirements are fixed, clear, and must follow legal rules. Once approved, the requirements cannot change.

Answer: Waterfall Model

Justification:

- Requirements are well-defined and stable.
- A strict sequence (requirements → design → coding → testing) ensures no missing legal rules.
- Waterfall is best for projects where **accuracy, compliance, and documentation** are more important than flexibility.

Scenario 2

A **Hospital Management System** is being developed. The hospital wants basic features (registration & appointments) first, and later additions like billing, pharmacy, and reports. They want the system delivered in **parts (modules)**.

Answer: Incremental Model

Justification:

- The system can be broken into independent modules (increments).
- Each increment (e.g., patient registration → billing → pharmacy) is delivered and used before the full system is complete.
- Users start benefitting early, and feedback can be incorporated into later increments.

Scenario 3

A startup is developing a new **mobile app for food delivery**. Requirements are uncertain, and users need to test a sample design before final development.

Answer: Prototype Model

Justification:

- Since requirements are unclear, a **prototype/mock-up** will help users visualize the app.
- Feedback from the prototype helps refine requirements.

- Reduces the risk of misunderstandings and builds confidence before actual coding.

Scenario 4

An **e-commerce platform** like Amazon is being built. The business environment changes frequently (new features, discounts, integrations). The company needs **fast delivery** and must adapt to customer demands quickly.

Answer: Agile Model

Justification:

- Agile delivers working features in **short sprints** (e.g., sprint 1 = login, sprint 2 = product catalog).
- Continuous user involvement ensures the platform matches market needs.
- Agile allows **quick response to changes**, which is critical for e-commerce competition.

Scenario 5

A **banking system** is being developed. The client wants a core system running quickly (basic transactions) but expects continuous improvements and refinements over time (loans, mobile banking, analytics).

Answer: Iterative Model

Justification:

- A basic version is developed first (core banking).
- Each iteration improves the system (add loans, then analytics, then mobile features).
- Suitable for projects where a **usable system is needed early** but must evolve with feedback.

Challenging SDLC Scenarios

Scenario 1: Defense System (Missile Control Software)

A government defense contractor is developing missile control software. Requirements are highly critical, must comply with strict safety standards, and no errors can be tolerated. Requirements are fixed and must be fully documented before implementation.

Answer: Waterfall Model

Justification:

- Safety-critical, compliance-heavy systems require complete documentation and rigorous verification.
- Requirements are stable and predefined (cannot risk frequent changes).
- Waterfall ensures a structured, controlled process with strong testing phases.

Scenario 2: AI-Based Healthcare Diagnostic Tool

A company is building a machine learning–based diagnostic tool for hospitals. Requirements are not fully clear since the system must evolve as doctors provide feedback and new medical research comes out. They need early versions to experiment with.

Answer: Iterative Model

Justification:

- Initial version can provide basic predictions.
- Each iteration can improve accuracy (better models, more datasets, more features).
- Iterative approach balances research-driven improvements with usable releases.

Scenario 3: Online Learning Platform During a Pandemic

A university needs an online learning system immediately due to an emergency (pandemic). They need a working platform fast, even if it lacks advanced features, but it should be improved over time.

Answer: Incremental Model

Justification:

- Deliver core modules first (e.g., video lectures, chat).
- Later increments add features like exams, grading, AI tutors.
- Students and teachers can start using the system quickly while new features are rolled out step by step.

Scenario 4: Smart Home IoT Ecosystem

A company is creating a smart home ecosystem (lighting, appliances, security). Requirements are unclear because customers have different expectations. The company needs to experiment with prototypes to validate ideas before investing fully.

Answer: Prototype Model

Justification:

- Prototypes of smart devices and apps help users test features.
- Feedback shapes the final design (e.g., some customers may prioritize energy savings, others security).
- Reduces risk of developing the wrong system.

Scenario 5: Social Media Platform (Competing with TikTok)

A startup wants to create a new social media app. Requirements are constantly changing due to market competition. They must launch features quickly and respond to user feedback weekly.

Answer: Agile Model

Justification:

- Agile's sprint-based delivery allows new features (filters, stories, chat) to be launched quickly.
- Continuous feedback from early adopters drives product direction.
- Agile ensures adaptability to fast-changing user trends and competition.

Scenario 6: Satellite Software with Modular Subsystems

A space agency is building satellite software with separate modules for communication, navigation, and imaging. Each subsystem is independent but must be integrated later. Delivery time is strict, but requirements for each module are clear.

Answer: Incremental Model

Justification:

- Each module can be built and tested independently.
- Early delivery of communication module allows testing in real satellites before full system is ready.
- Integration happens after increments are delivered.

Multiplicity

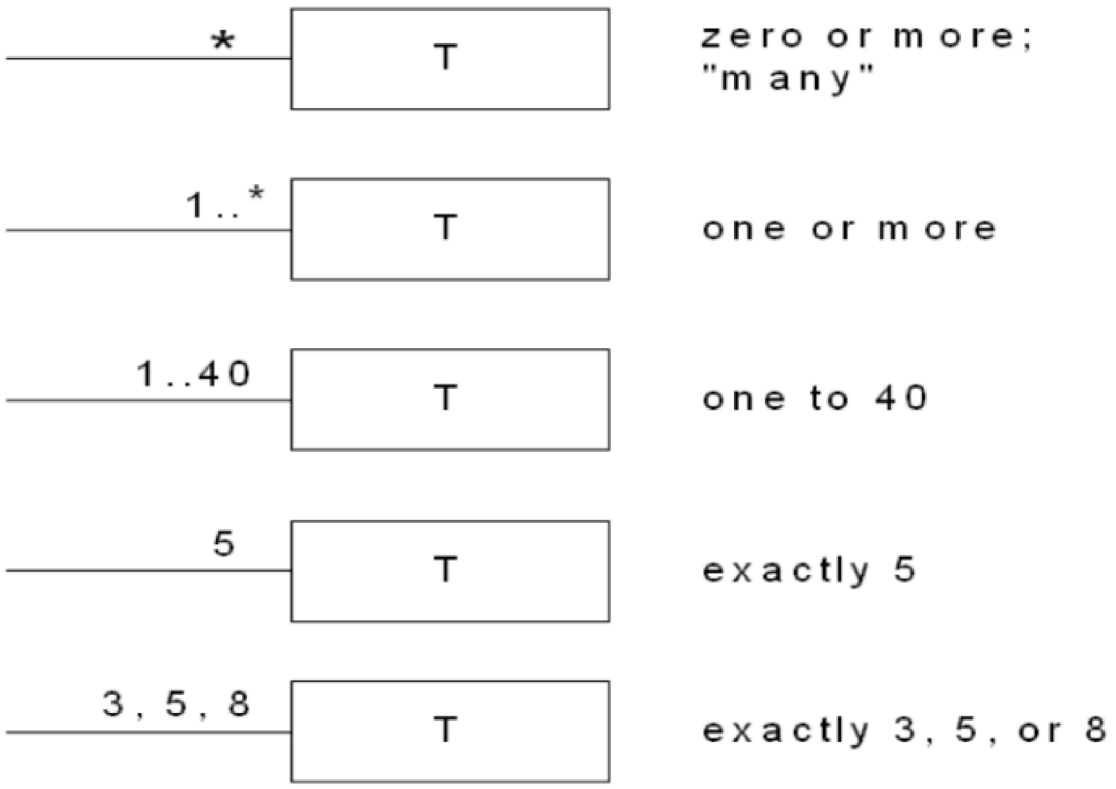
- **Definition:** Multiplicity specifies how many instances of one class (or entity) can be associated with a single instance of another class **at a given moment in time**.
- It describes the **cardinality** of a relationship in UML class diagrams or ER models.
- It is different from "over time," because multiplicity looks at associations **simultaneously** at a particular state of the system

Examples

1. **One-to-One (1..1)**
    - A *Person* has exactly **one passport** at a given time.
    - Multiplicity: `Person 1..1 → Passport 1..1`.
2. **One-to-Many (1..*)**
    - A *Teacher* can teach **many students** at the same time.
    - Multiplicity: `Teacher→ Student 1..*`.
3. **Many-to-Many (..)**

- A *Student* can enroll in **many courses**, and a *Course* can have **many students**.
- Multiplicity: `Student * ↔ Course *`.

Key Idea: Multiplicity tells us **how many objects/records can be linked at once**, not how many over the lifetime of the system.



Aggregation (with Example and Code in C++)

Explanation:

Aggregation is a "has-a" relationship between two classes, where one class (whole) contains objects of another class (part). The contained objects can exist independently of the container.

Real-world Example:

A **Department** has multiple **Professors**, but Professors can exist without the Department (they can belong to another one).

C++ Code Example:

```
#include <iostream>
#include <vector>
using namespace std;

class Professor {
    string name;
public:
    Professor(string n) : name(n) {}
    void showInfo() {
        cout << "Professor: " << name << endl;
    }
};

class Department {
    string deptName;
    vector<Professor*> professors; // Aggregation (has-a relationship)
public:
    Department(string d) : deptName(d) {}

    void addProfessor(Professor* prof) {
        professors.push_back(prof);
    }

    void showDepartment() {
        cout << "Department: " << deptName << endl;
        for (auto p : professors) {
            p→showInfo();
        }
    }
};
```

```
int main() {
    Professor p1("Dr. Ali");
    Professor p2("Dr. Ayesha");

    Department cs("Computer Science");
    cs.addProfessor(&p1);
    cs.addProfessor(&p2);

    cs.showDepartment();

    // Professors can still exist without Department
    p1.showInfo();
    return 0;
}
```

Output:

```
Department: Computer Science
Professor: Dr. Ali
Professor: Dr. Ayesha
Professor: Dr. Ali
```

Generalization (with Example and Code in C++)

Explanation:

Generalization is an **inheritance relationship** (is-a). The subclass inherits properties and behaviors of the superclass.

Real-world Example:

A **Car** and a **Bike** are both **Vehicles**. They share common features like `speed` and `move()`, but also have their own unique behaviors.

C++ Code Example:

```
#include <iostream>
using namespace std;
```

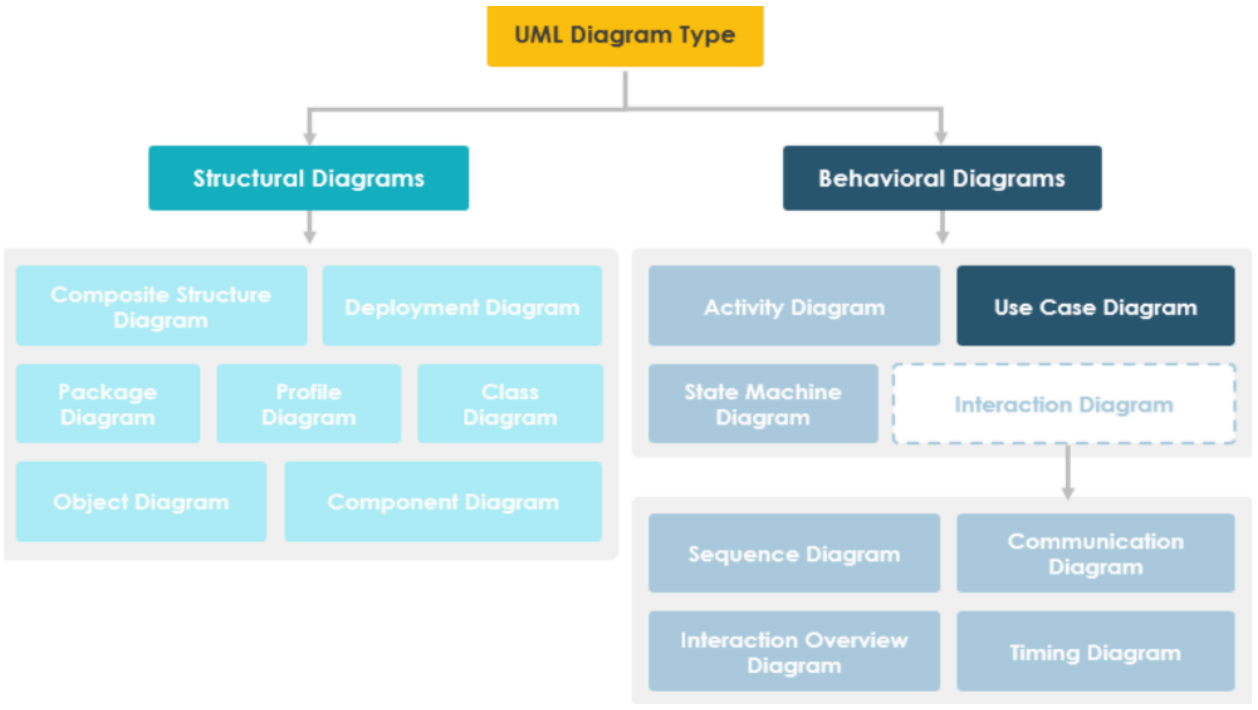
```
class Vehicle { // Superclass
protected:
    int speed;
public:
    Vehicle(int s) : speed(s) {}
    virtual void move() {
        cout << "Vehicle is moving at speed " << speed << " km/h" << endl;
    }
};

class Car : public Vehicle { // Subclass
public:
    Car(int s) : Vehicle(s) {}
    void move() override {
        cout << "Car drives smoothly at " << speed << " km/h" << endl;
    }
};

class Bike : public Vehicle { // Subclass
public:
    Bike(int s) : Vehicle(s) {}
    void move() override {
        cout << "Bike rides at " << speed << " km/h" << endl;
    }
};

int main() {
    Vehicle* v1 = new Car(100);
    Vehicle* v2 = new Bike(60);

    v1→move();
    v2→move();

    delete v1;
    delete v2;
    return 0;
}
```

Output:

```
Car drives smoothly at 100 km/h
Bike rides at 60 km/h
```

1. UML Diagrams & its Categories

- **Definition:** UML (Unified Modeling Language) diagrams are standardized visual representations of a system's structure and behavior. They help developers, stakeholders, and designers understand and communicate system requirements.
- **Categories:**
    - **Structural Diagrams** (focus on static aspects, e.g., Class, Object, Component, Deployment).
    - **Behavioral Diagrams** (focus on dynamic aspects, e.g., Use Case, Sequence, Activity, State Machine).
- **Example:** A **Class Diagram** showing `Student`, `Course`, and `Enrollment` helps illustrate the relationships in a university system.



Association

Description:

A general "has-a" relationship between two classes. Both can exist independently. A `Teacher` teaches a `Student`.

```
class Student {
public:
    std::string name;
};

class Teacher {
public:
    void teach(Student& s) {
        std::cout << "Teaching " << s.name << std::endl;
    }
};
```

Inheritance

Description:

A "is-a" relationship. One class (child) inherits from another (parent).
A `Dog` is an `Animal`.

```
class Animal {
public:
    void eat() {
        std::cout << "Eating..." << std::endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        std::cout << "Barking..." << std::endl;
    }
};
```

Realization / Implementation

Description:

A class implements an interface (pure abstract class in C++).
A `Printer` implements `Printable`.

```
class Printable {
public:
    virtual void print() = 0; // Pure virtual
};

class Printer : public Printable {
public:
    void print() override {
        std::cout << "Printing..." << std::endl;
    }
};
```

Dependency

Description:

A "uses-a" relationship. One class depends on another temporarily.
A `ReportGenerator` uses a `Database` to fetch data.

```
class Database {
public:
    std::string fetchData() {
        return "Data from DB";
    }
};

class ReportGenerator {
public:
    void generate(Database& db) {
        std::cout << "Generating report: " << db.fetchData() << std::endl;
    }
};
```

Aggregation

Description:

A "has-a" relationship where the contained object can exist independently.
A `Team` has `Players`, but players can exist without a team.

```
class Player {
public:
    std::string name;
};

class Team {
```

```
public:
    std::vector<Player*> players;

    void addPlayer(Player* p) {
        players.push_back(p);
    }
};
```

Composition

Description:

A "has-a" relationship where the contained object's lifetime is bound to the container.
A `House` has `Rooms`, and rooms don't exist without the house.

```
class Room {
public:
    Room() {
        std::cout << "Room created" << std::endl;
    }
};

class House {
private:
    Room room1;
    Room room2;

public:
    House() : room1(), room2() {
        std::cout << "House created with rooms" << std::endl;
    }
};
```

Summary Table

Concept	Relationship	Lifetime Dependency	Example Class A → B
Association	Has-a	Independent	Teacher → Student
Inheritance	Is-a	N/A	Dog → Animal
Realization	Implements	N/A	Printer → Printable
Dependency	Uses-a	Temporary	ReportGenerator → Database
Aggregation	Has-a	Independent	Team → Player
Composition	Has-a	Strong (bound)	House → Room

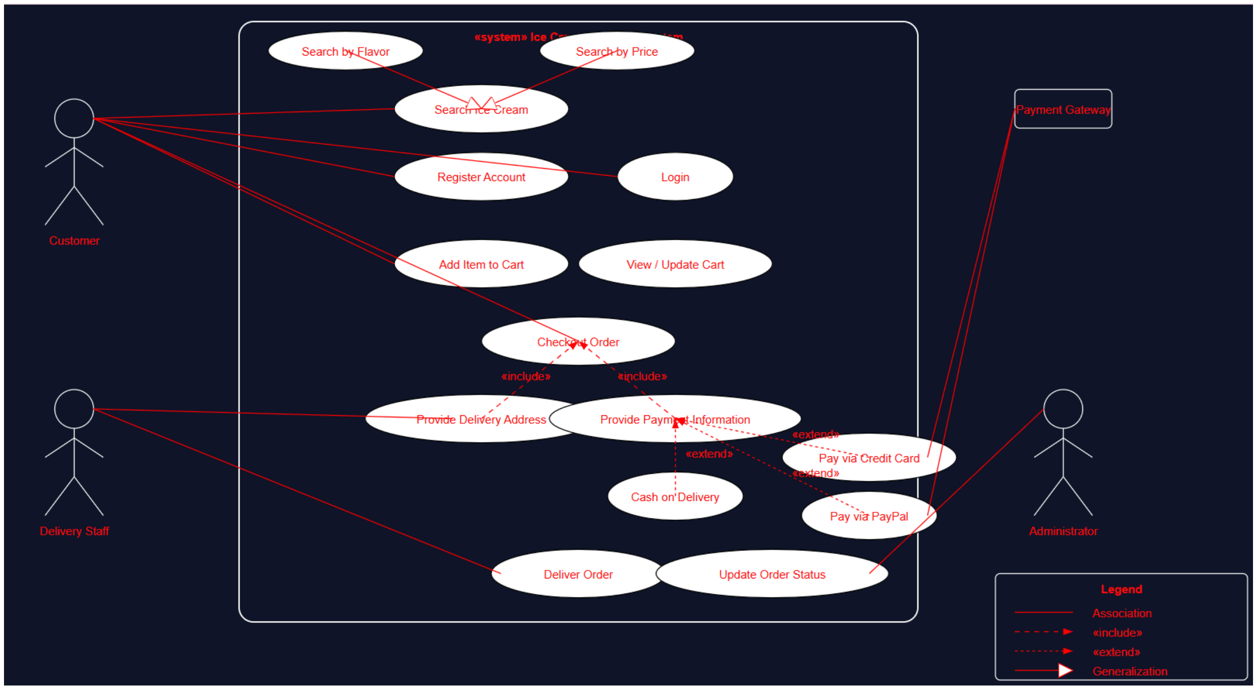
Unified Modeling Language (UML)

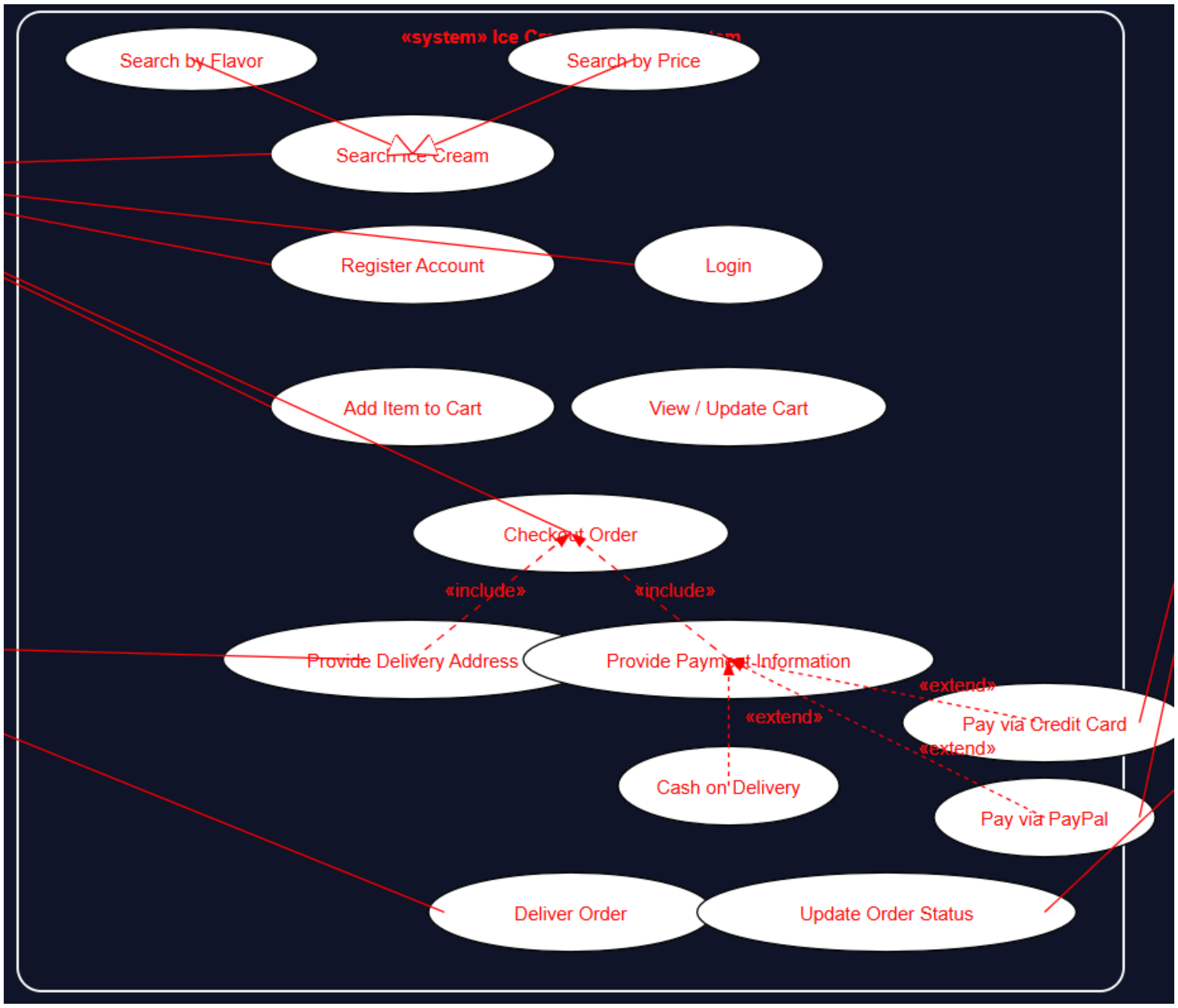
- Visual language for specifying, constructing, documenting systems.
- Can be used as a **sketch, blueprint, or programming language.**
- **Perspectives**: Conceptual, Specification, Implementation.
- **Not a silver bullet**: Supports but doesn't guarantee quality.

USE CASE DIAGRAM:

THE ICE CREAM ORDERING SYSTEM

Q: The ICE Cream System allows the user of a web browser to order ICE Cream for home delivery. To place an order, a customer searches to find items to purchase, adds items one at a time to a shopping cart, and possibly searches again for more items. When all items have been chosen, the customer provides a delivery address. If not paying with cash, the customer can also provides credit card information. The system has an option for customer to register with the ICE Cream shop. They can then save their name and address information, so that they do not have to enter this information every time that they place an order.


«system» Ice Cr
Search by Flavor
Search by Price
Search Ice Cream
Register Account
Login
Add Item to Cart
View / Update Cart
Checkout Order
«include»
«include»
Provide Delivery Address
Provide Payment Information
«extend»
«extend»
«extend»
Pay via Credit Card
Cash on Delivery
Pay via PayPal
Deliver Order
Update Order Status
Customer
Delivery Staff
Payment Gateway
Administrator
Legend
Association
«include»
«extend»
Generalization



UML CLASS DIAGRAMS:

Class Name

Attributes

Methods/Operations

```
-----------------------
|  Student          |
-----------------------
```

```
| - name : String     |
| - rollNo : int      |
-----------------------
| + getDetails()      |
| + updateRecord()    |
-----------------------
+ → public
- → private
# → protected
```

Relationships

- **Association (solid line)**: "uses/has a" relationship.

  Example: `Teacher —— Student` (teaches/learns).

- **Multiplicity**: Defines how many objects participate (1, 0..*, 1..*, etc.).

  Example: A `Teacher` can teach `0..* Students`.

- **Aggregation (hollow diamond)**: "whole-part" relationship but part can exist independently.

  Example: `Library ◇— Book`.

- **Composition (solid diamond)**: Stronger aggregation; part cannot exist without whole.

  Example: `House ◆— Room`.

- **Inheritance / Generalization (triangle arrow)**: "is-a" relationship.

  Example: `Person —▷ Student`.

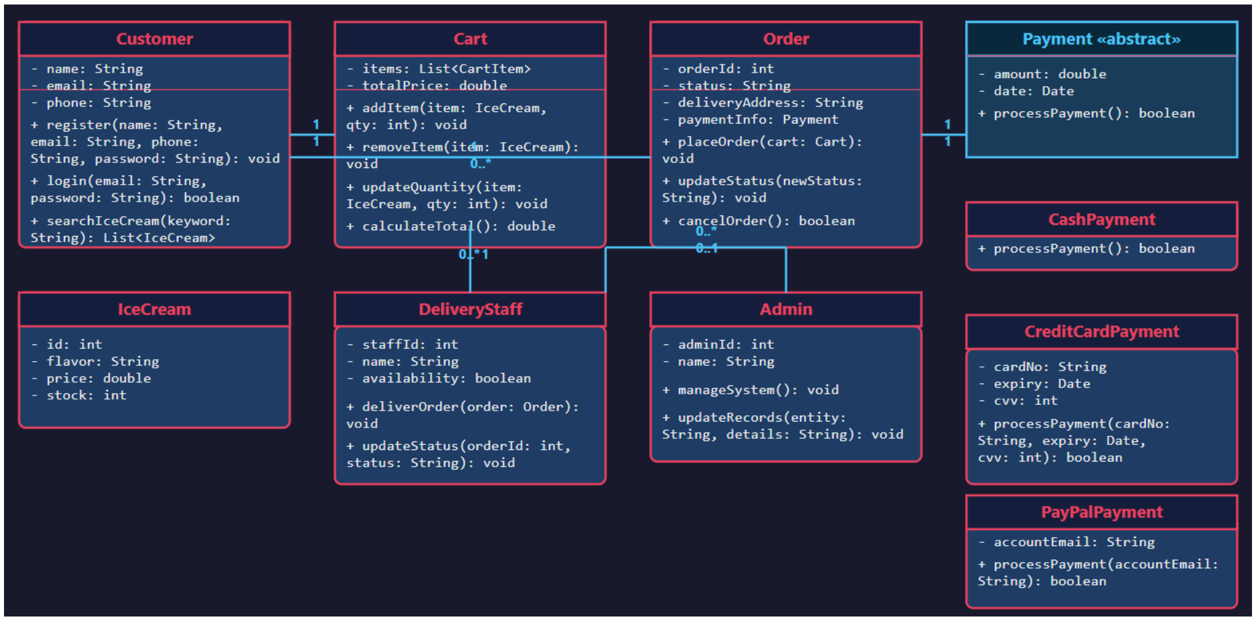
- **Dependency (dashed arrow)**: One class depends on another temporarily.

  Example: `Car – - -▷ Mechanic`.

Why Use Class Diagrams?

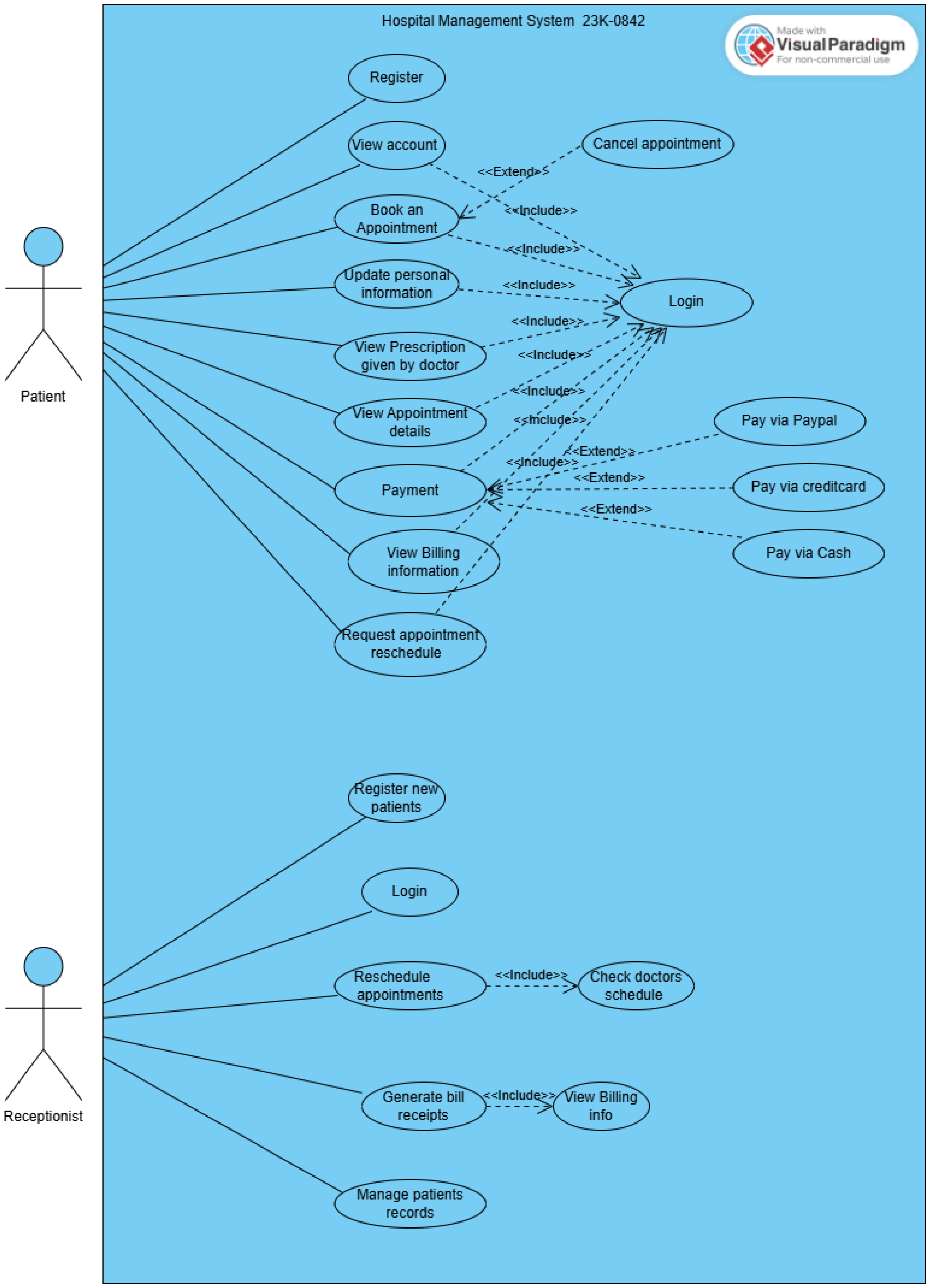
- Blueprint for **object-oriented design**.
- Helps understand system structure before coding.
- Useful for communication between designers, developers, and stakeholders.

Class Diagram of Above Icecream Example:


Customer
- name: String
- email: String
- phone: String
+ register(name: String, email: String, phone: String, password: String): void
+ login(email: String, password: String): boolean
+ searchIceCream(keyword: String): List<IceCream>
Cart
- items: List<CartItem>
- totalPrice: double
+ addItem(item: IceCream, qty: int): void
+ removeItem(item: IceCream): void
+ updateQuantity(item: IceCream, qty: int): void
+ calculateTotal(): double
Order
- orderId: int
- status: String
- deliveryAddress: String
- paymentInfo: Payment
+ placeOrder(cart: Cart): void
+ updateStatus(newStatus: String): void
+ cancelOrder(): boolean
Payment «abstract»
- amount: double
- date: Date
+ processPayment(): boolean
CashPayment
+ processPayment(): boolean
IceCream
- id: int
- flavor: String
- price: double
- stock: int
DeliveryStaff
- staffId: int
- name: String
- availability: boolean
+ deliverOrder(order: Order): void
+ updateStatus(orderId: int, status: String): void
Admin
- adminId: int
- name: String
+ manageSystem(): void
+ updateRecords(entity: String, details: String): void
CreditCardPayment
- cardNo: String
- expiry: Date
- cvv: int
+ processPayment(cardNo: String, expiry: Date, cvv: int): boolean
PayPalPayment
- accountEmail: String
+ processPayment(accountEmail: String): boolean
1
1
0..*
1
1
0..*
0..1
0..*1

CLASS ACTIVITY:

Q: Draw a use case diagram for the hospital reception system. In this system, receptionist can schedule patient appointment and patient hospital admission after the patient registration. Both types of patients i.e. outpatient and inpatient can be admitted in the hospital. Receptionist also checks the insurance and claim forms and put them in file. Patient medical report is also filed by the receptionist.


Hospital Management System 23K-0842
Made with Visual Paradigm For non-commercial use
Patient
Register
View account
Cancel appointment
Book an Appointment
Update personal information
View Prescription given by doctor
View Appointment details
Payment
View Billing information
Request appointment reschedule
Login
Pay via Paypal
Pay via creditcard
Pay via Cash
<<Extend>>
<<Include>>
Receptionist
Register new patients
Login
Reschedule appointments
Check doctors schedule
Generate bill receipts
View Billing info
Manage patients records

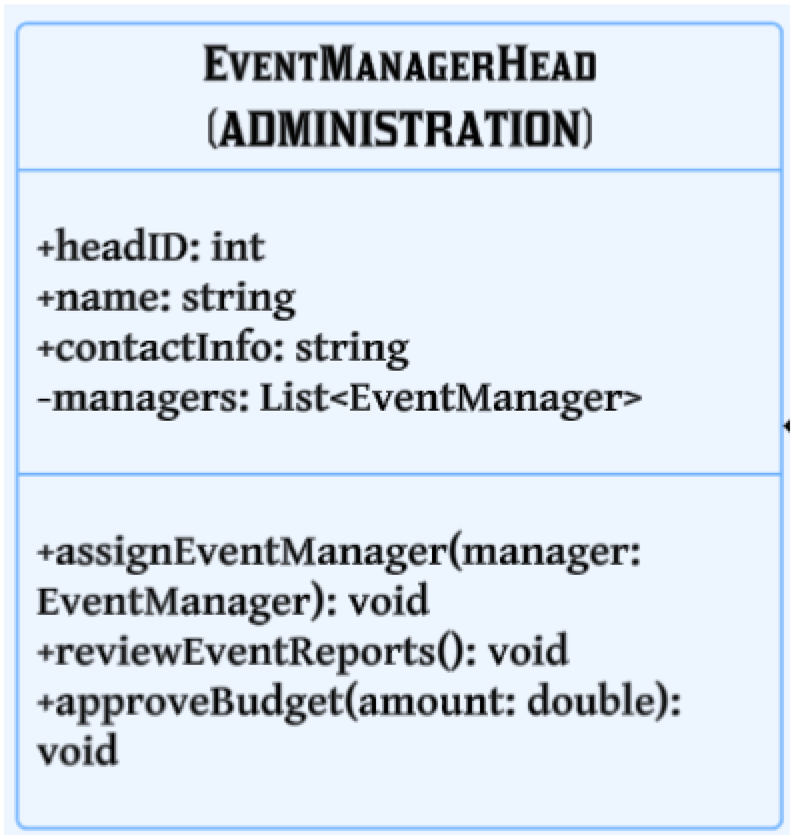
Q: Imagine you’re building a software system for “ Events Management System”, a company that organizes weddings, corporate parties, birthdays, and conferences.

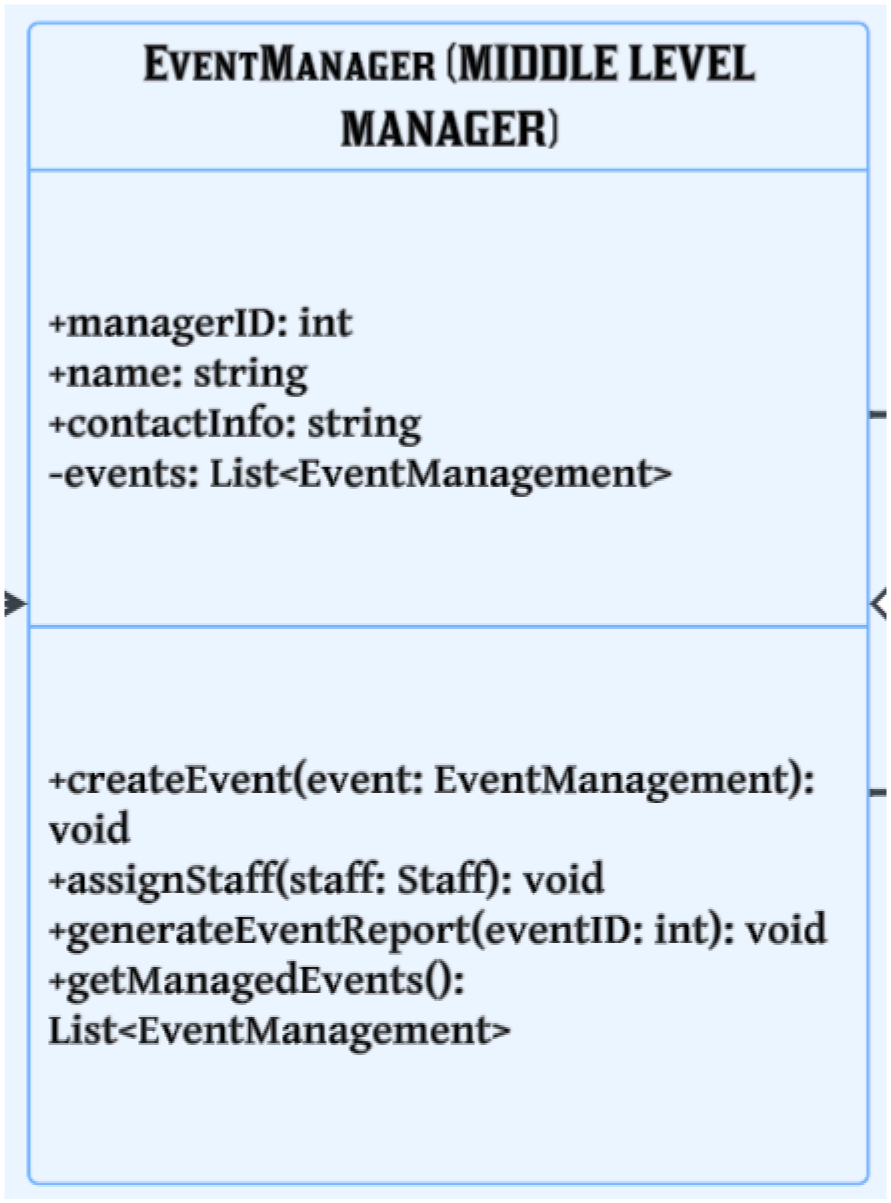
1. **Whenever a client books an event, the company has to manage multiple things:**
2. **Venue – The client selects a hall, banquet, or outdoor space. The system should store venue details, check availability, and reserve it.**
3. **Invitation Cards – The system keeps track of design options, printing cost, and delivery status.**
4. **Valet Parking – If required, the system books parking staff, manages number of vehicles, and calculates service charges.**
5. **Decoration – Clients choose decoration theme (Floral, Lights, Luxury setup, etc.). The system records theme, material cost, and workers needed.**
6. **Finance – Handles overall billing, payment installments, discounts, and profit calculation.**
7. **Administration – Oversees bookings, approves budgets, manages staff, and ensures smooth execution**

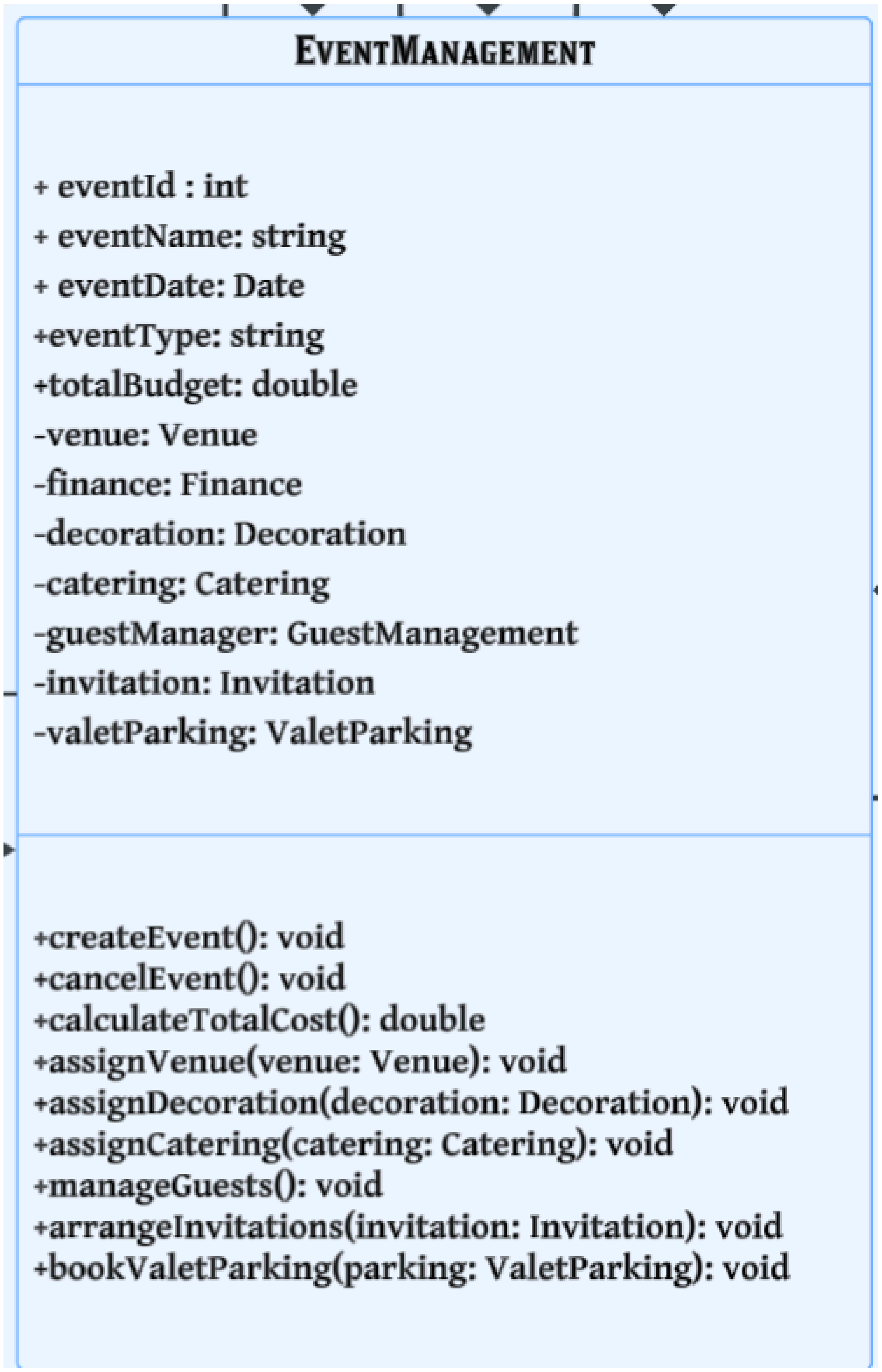
Draw Classes Diagram for each of above classes with Attributes (Field with Types) & Functions Modules (you can add any new class or change existing , for EVENT MANAGEMENT System)

<u>EVENT MANAGEMENT SYSTEM</u>

These are the screenshots of classes so that its easier to understand and see what are the attributes and methods in each class. I have also attached a link and pic which shows the whole EMS and the relationships between each class.


EventManagerHead
(Administration)
+headID: int
+name: string
+contactInfo: string
-managers: List<EventManager>
+assignEventManager(manager: EventManager): void
+reviewEventReports(): void
+approveBudget(amount: double): void

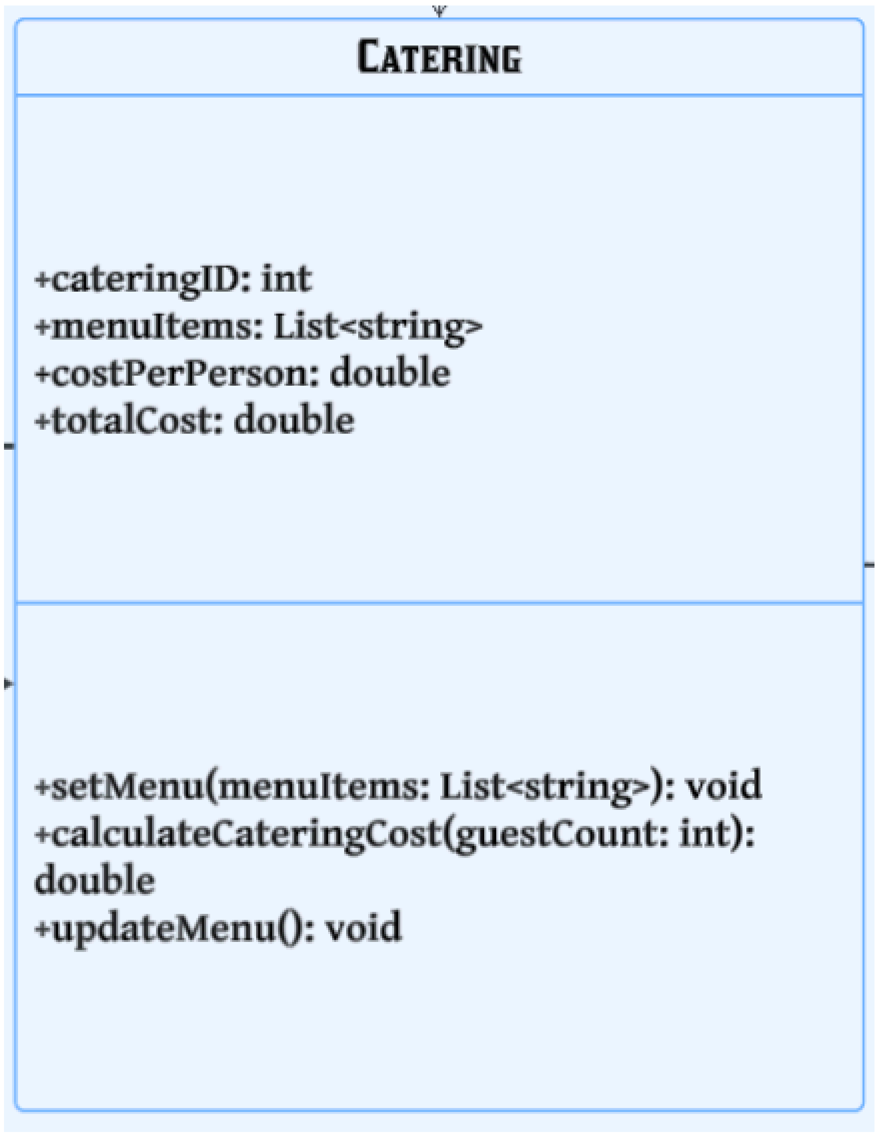

EventManager (Middle Level Manager)
+managerID: int
+name: string
+contactInfo: string
-events: List<EventManagement>
+createEvent(event: EventManagement): void
+assignStaff(staff: Staff): void
+generateEventReport(eventID: int): void
+getManagedEvents(): List<EventManagement>



EventManagement

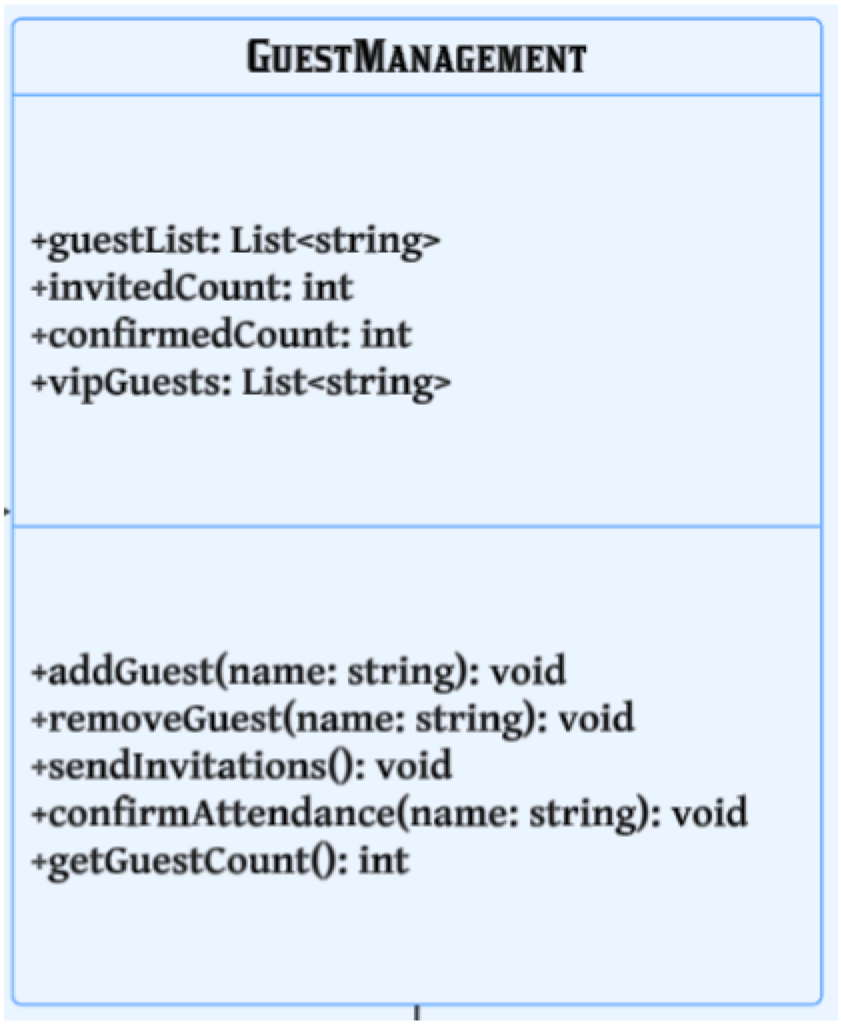
+ eventId : int
+ eventName: string
+ eventDate: Date
+eventType: string
+totalBudget: double
-venue: Venue
-finance: Finance
-decoration: Decoration
-catering: Catering
-guestManager: GuestManagement
-invitation: Invitation
-valetParking: ValetParking

+createEvent(): void
+cancelEvent(): void
+calculateTotalCost(): double
+assignVenue(venue: Venue): void
+assignDecoration(decoration: Decoration): void
+assignCatering(catering: Catering): void
+manageGuests(): void
+arrangeInvitations(invitation: Invitation): void
+bookValetParking(parking: ValetParking): void

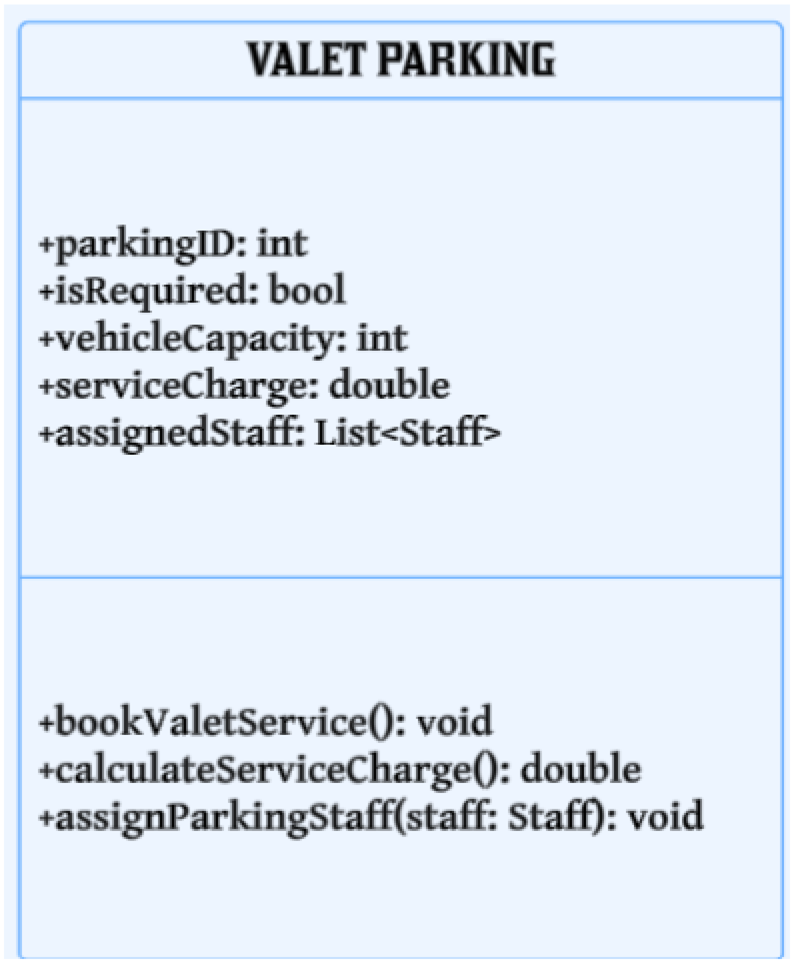


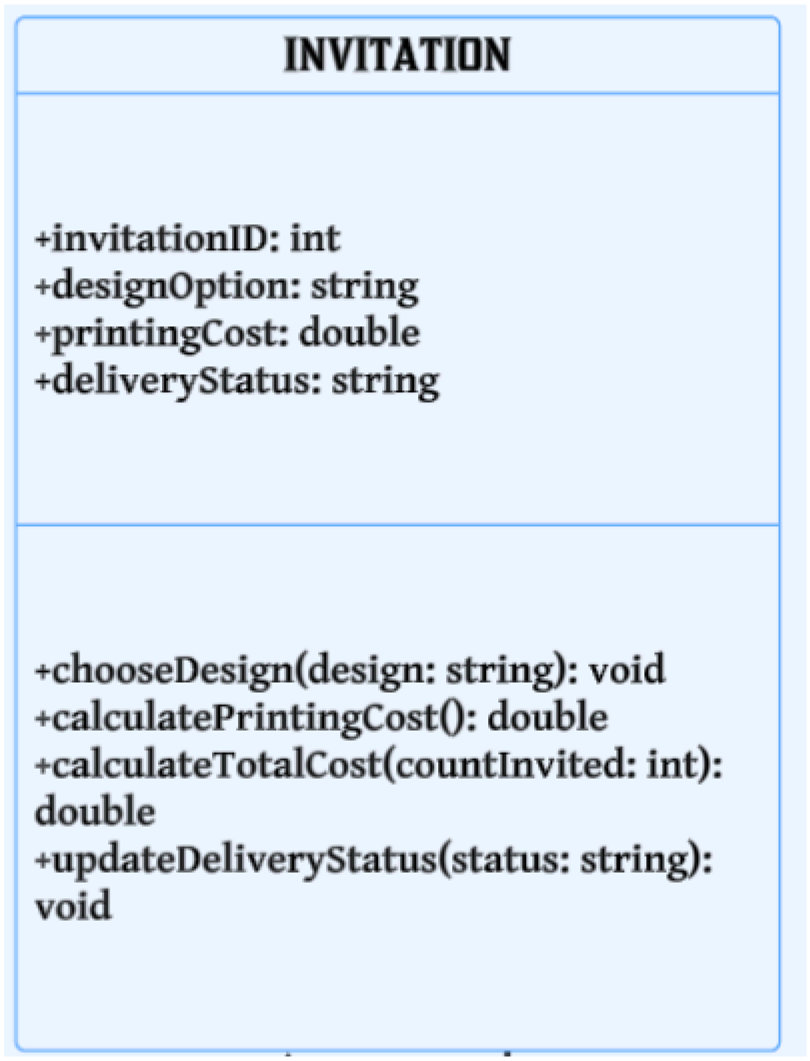
+cateringID: int
+menuItems: List<string>
+costPerPerson: double
+totalCost: double

+setMenu(menuItems: List<string>): void
+calculateCateringCost(guestCount: int): double
+updateMenu(): void



GuestManagement
+guestList: List<string>
+invitedCount: int
+confirmedCount: int
+vipGuests: List<string>
+addGuest(name: string): void
+removeGuest(name: string): void
+sendInvitations(): void
+confirmAttendance(name: string): void
+getGuestCount(): int

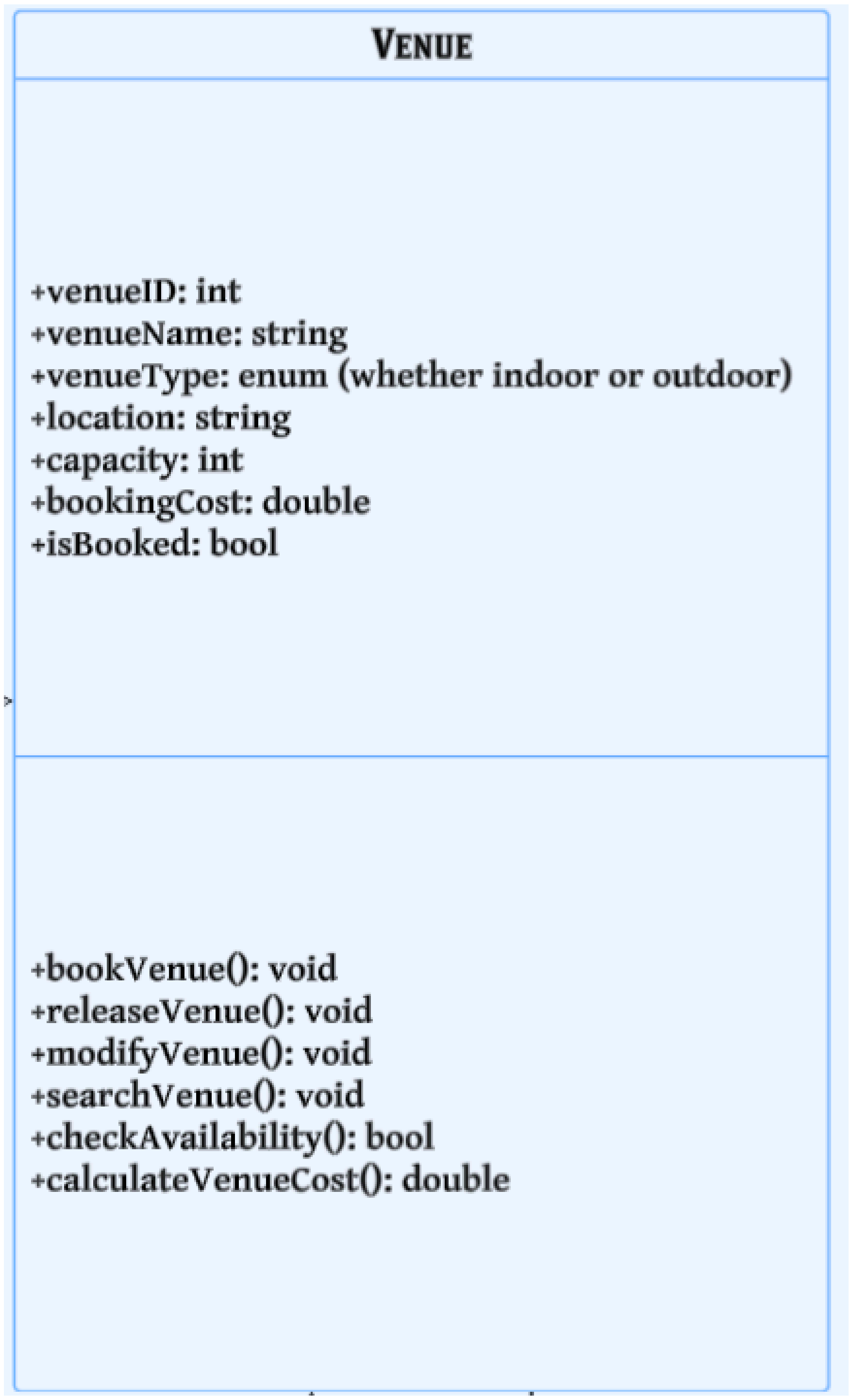

VALET PARKING
+parkingID: int
+isRequired: bool
+vehicleCapacity: int
+serviceCharge: double
+assignedStaff: List<Staff>
+bookValetService(): void
+calculateServiceCharge(): double
+assignParkingStaff(staff: Staff): void

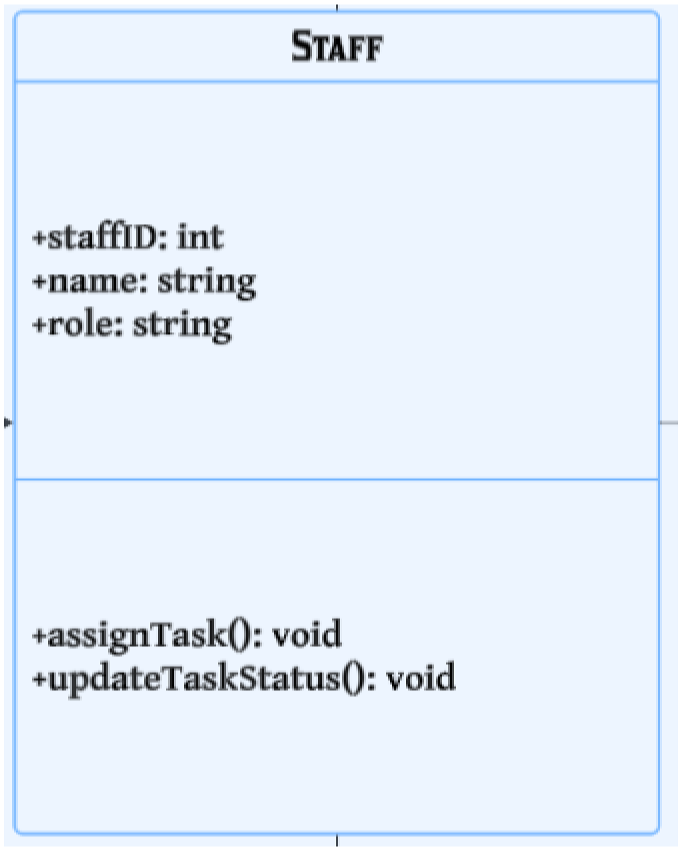


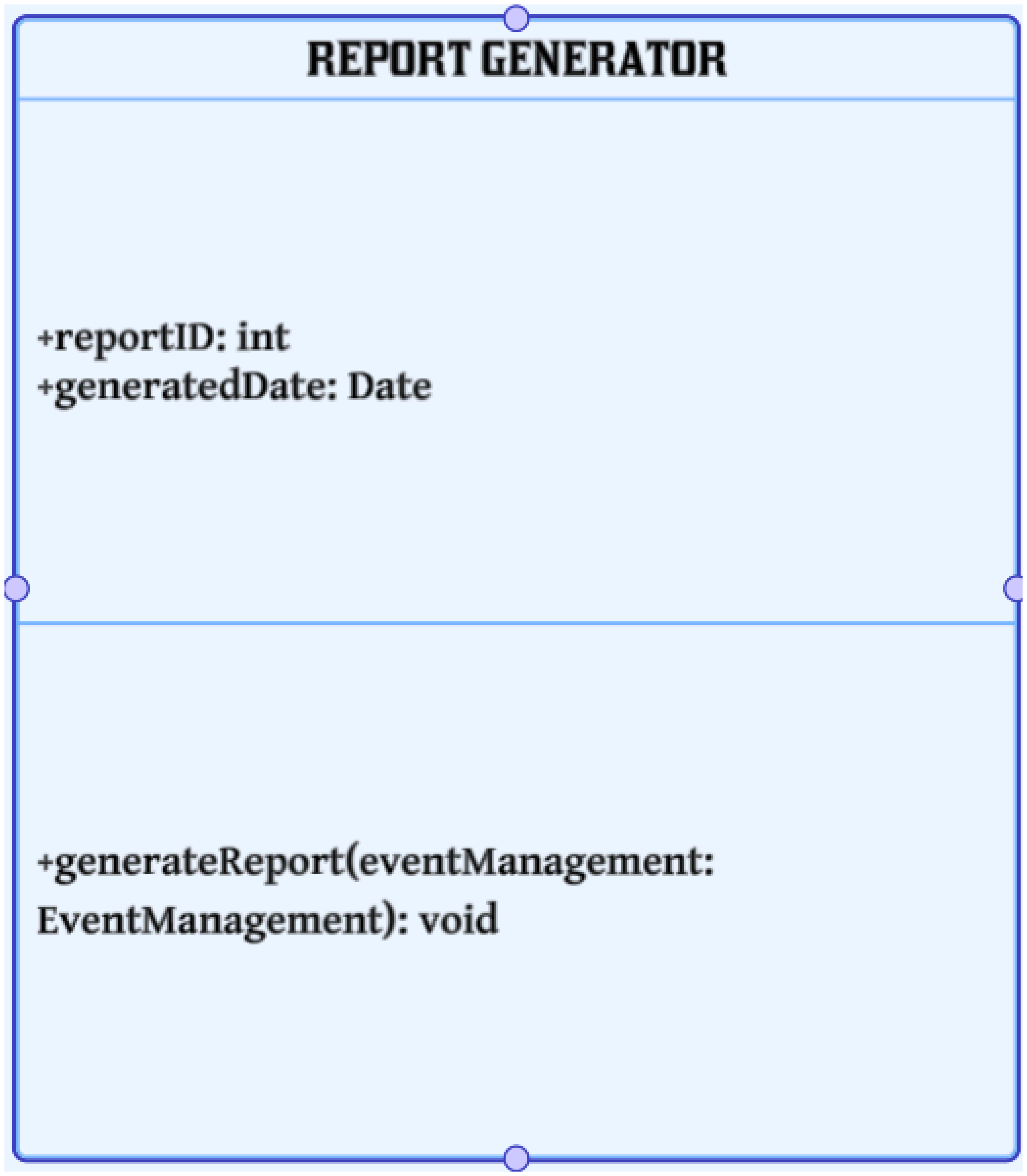
INVITATION

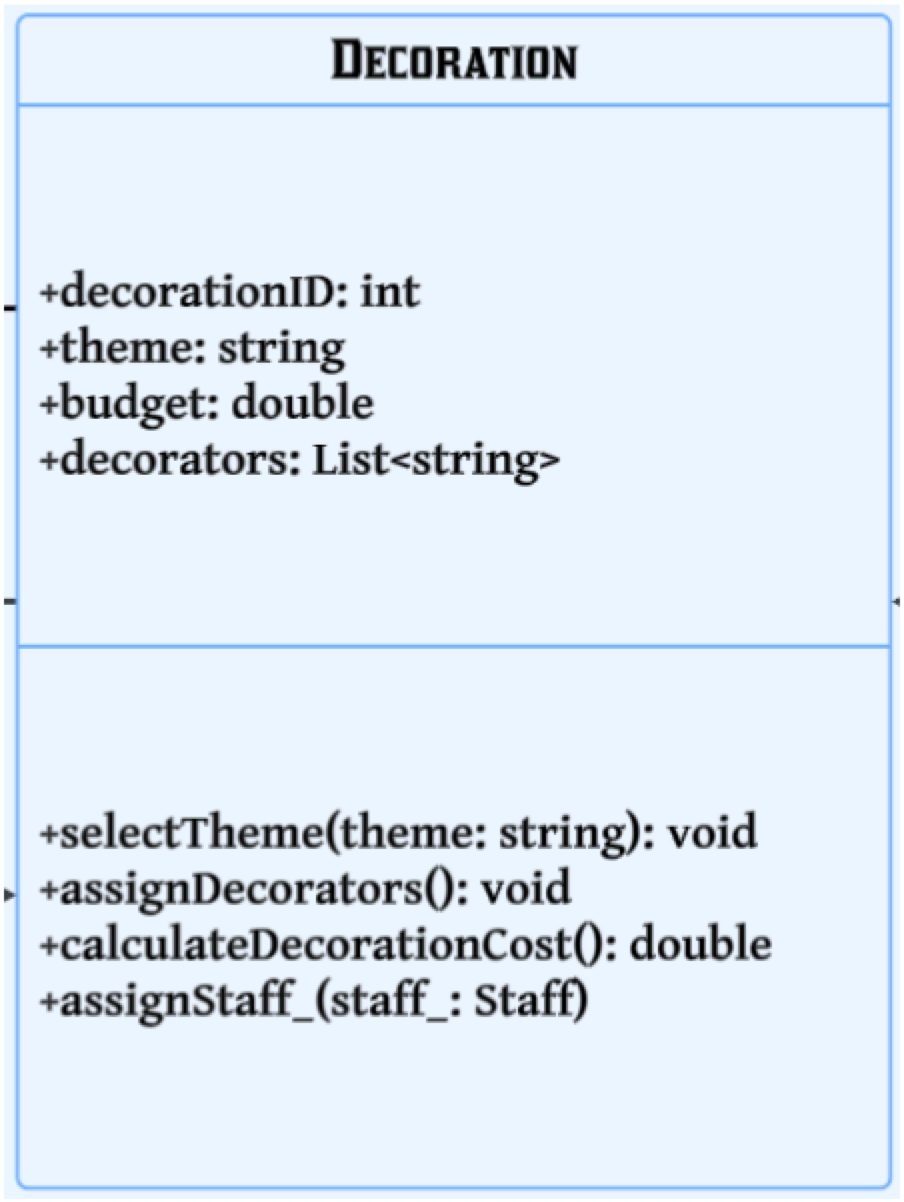
+invitationID: int
+designOption: string
+printingCost: double
+deliveryStatus: string

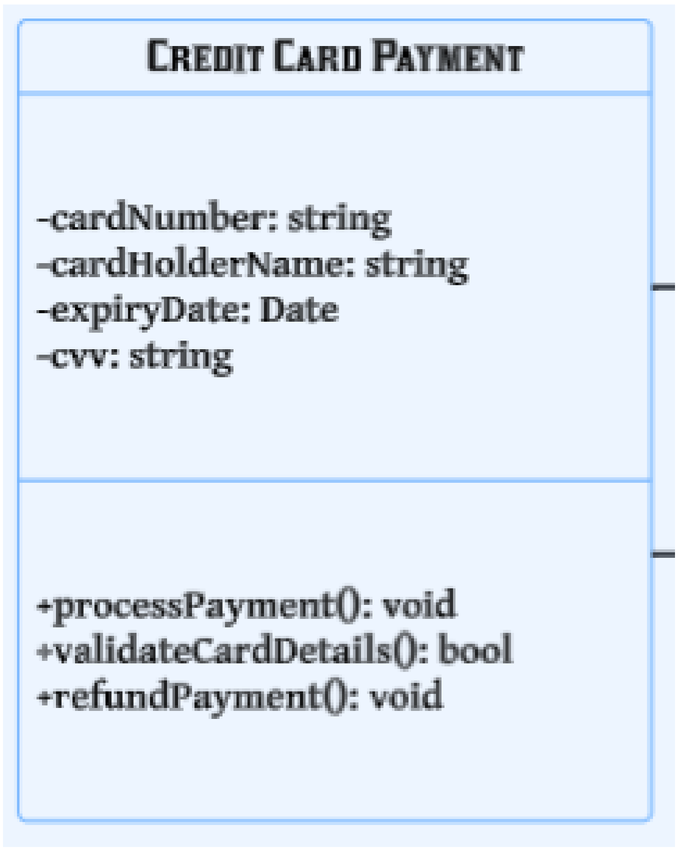
+chooseDesign(design: string): void
+calculatePrintingCost(): double
+calculateTotalCost(countInvited: int): double
+updateDeliveryStatus(status: string): void

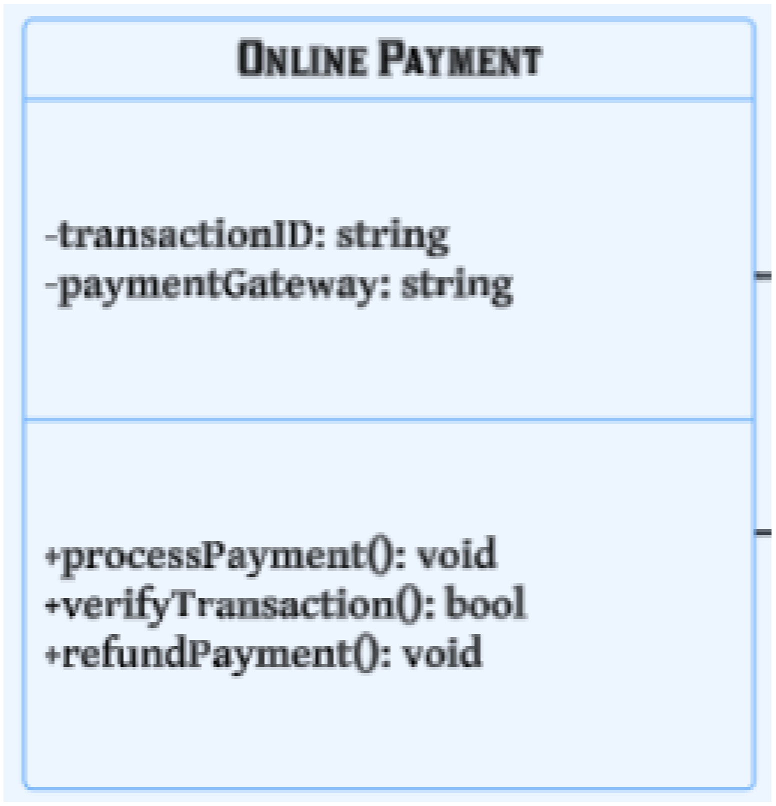

VENUE
+venueID: int
+venueName: string
+venueType: enum (whether indoor or outdoor)
+location: string
+capacity: int
+bookingCost: double
+isBooked: bool
+bookVenue(): void
+releaseVenue(): void
+modifyVenue(): void
+searchVenue(): void
+checkAvailability(): bool
+calculateVenueCost(): double

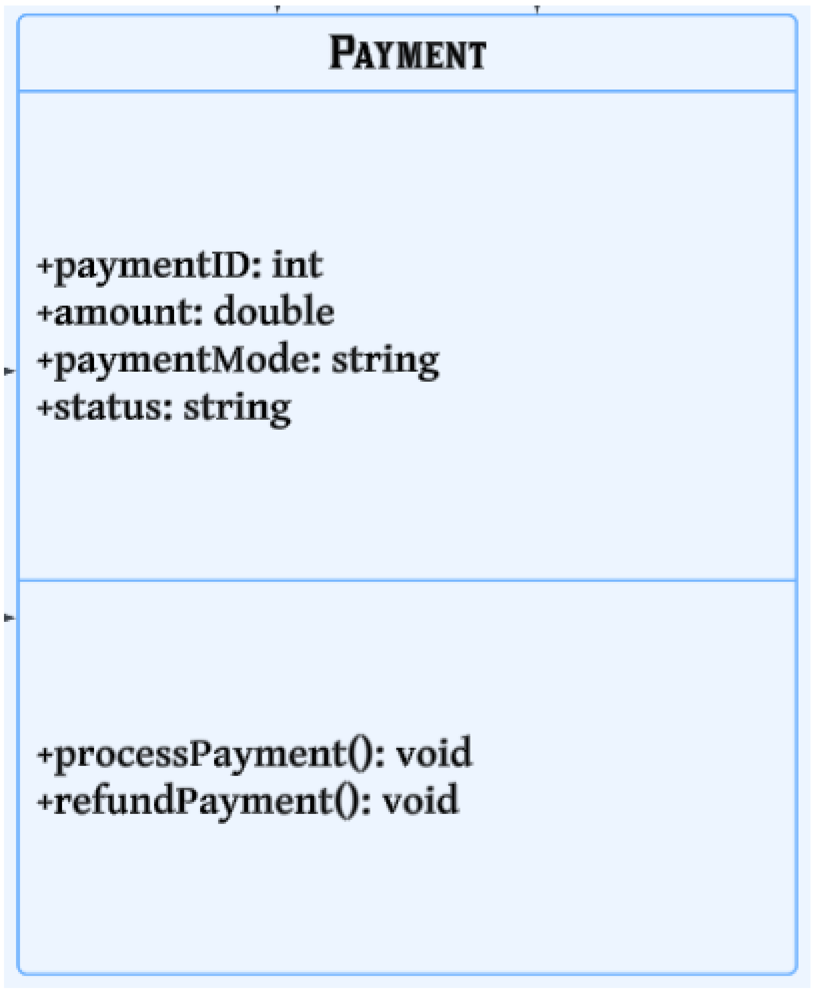

Staff
+staffID: int
+name: string
+role: string
+assignTask(): void
+updateTaskStatus(): void


REPORT GENERATOR
+reportID: int
+generatedDate: Date
+generateReport(eventManagement: EventManagement): void


Decoration
+decorationID: int
+theme: string
+budget: double
+decorators: List<string>
+selectTheme(theme: string): void
+assignDecorators(): void
+calculateDecorationCost(): double
+assignStaff_(staff_: Staff)


Credit Card Payment
-cardNumber: string
-cardHolderName: string
-expiryDate: Date
-cvv: string
+processPayment(): void
+validateCardDetails(): bool
+refundPayment(): void


Online Payment
-transactionID: string
-paymentGateway: string
+processPayment(): void
+verifyTransaction(): bool
+refundPayment(): void



EXPLANATION:

1. EventManagerHead (Top-level Controller)

Role:

Supervises the entire event management system and manages multiple event managers.

- **Attributes:** headID, name, contactInfo
- **Methods:** Assign managers, review reports, approve budgets.
- **Relationship:**
  - **Aggregation** with EventManager → One head manages multiple managers (1..*).

2. EventManager (Middle-level Manager)

Role:

Handles multiple events under the supervision of the head and manages staff.

● **Attributes:** managerID, name, contactInfo, events

● **Methods:** Create events, assign staff, generate event reports.

● **Relationship:**

○ **Aggregation** with EventManagement → One manager manages multiple events.

○ **Association** with Staff → One manager supervises multiple staff members.

3. EventManagement (Core Class — Main Entity)

Role:

Represents a single event and integrates all event-related components.

● **Attributes:** Event details + associated entities (venue, finance, catering, etc.) ● **Methods:** Create/cancel events, assign resources, calculate costs.

● **Relationship:**

○ **Composition** with Venue, Finance, Decoration, Catering, GuestManagement, Invitation, and ValetParking → Each event **owns** these components.

○ **Association** with Staff → Multiple staff members can work on multiple events.

4. GuestManagement

Role:

Handles guest invitations, confirmations, and VIP guest tracking.

● **Attributes:** Guest list, invited count, confirmed count, VIP list. ● **Methods:** Add/remove guests, send invitations, confirm attendance.

● **Relationship:**

○ **Composition** with EventManagement → Each event has its own guest list. ○ **Association** with Catering → Catering cost depends on the number of confirmed guests.

5. Finance

Role:

Manages budgeting, expenses, and financial reports.

- **Attributes:** financeID, totalBudget, spentAmount, remainingBudget.
- **Methods:** Allocate budget, record expenses, calculate remaining funds.
- **Relationship:**
  - **Composition** with EventManagement.
  - **Association** with Payment → One finance record can have multiple payment transactions.

6. Payment

Role:

Base class for handling all payment transactions.

- **Attributes:** paymentID, amount, paymentMode, status.
- **Methods:** Process and refund payments.
- **Relationship:**
  - **Association** with Finance.
  - **Inheritance:** Extended by CreditCardPayment and OnlinePayment.

7. Decoration

Role:

Manages event decoration themes and resources.

- **Attributes:** decorationID, theme, budget, decorators.
- **Methods:** Select theme, assign decorators, calculate decoration costs.
- **Relationship:**
  - **Composition** with EventManagement.
  - **Association** with Staff.

8. Catering

Role:

Handles food, beverages, and catering budgets.

- **Attributes:** cateringID, menuItems, costPerPerson, totalCost.
- **Methods:** Set menu, calculate catering cost, update menu items.

● **Relationship:**

○ **Composition** with EventManagement.

○ **Association** with GuestManagement → Total cost depends on guest count.

9. Venue

Role:

Manages booking, cost, and capacity for event venues.

● **Attributes:**venueID, venueName, location, capacity, bookingCost.

● **Methods:** Book/release venue, check availability, calculate cost.

● **Relationship:**

○ **Composition** with EventManagement → Venue belongs to the event.

10. Staff

Role:

Performs various event-related tasks like decoration, catering, and parking.

● **Attributes:** staffID, name, role.

● **Methods:** Assign tasks, update task status.

● **Relationship:**

○ **Aggregation** with EventManager.

○ **Association** with Decoration, Catering, Venue, and ValetParking.

11. Invitation

Role:

Handles invitation card design, printing, and delivery tracking.

● **Attributes:** invitationID, designOption, printingCost, deliveryStatus.

● **Methods:** Choose design, calculate printing cost, update delivery status.

● **Relationship:**

○ **Composition** with EventManagement → Each event has its own invitations.

12. ValetParking

Role:

Manages valet parking services for events.

● **Attributes:** parkingID, isRequired, vehicleCapacity, serviceCharge, assigned staff ● **Methods:** Book valet service, calculate service charges, assign parking staff.

● **Relationship:**

○ **Composition** with EventManagement.

○ **Aggregation** with Staff → Parking staff exist independently.

13. ReportGenerator

Role:

Generates various reports for events, finances, and guests.

● **Attributes:** reportID, eventID, generatedDate.

● **Methods:** Generate reports.

● **Relationship:**

○ **Dependency** on EventManagement → Uses its data temporarily just to generate reports.

14. CreditCardPayment (Extends Payment)

Role:

Handles payments made using credit cards.

● **Attributes:** cardNumber, cardHolderName, expiryDate, cvv.

● **Methods:** Process payments, validate card details.

● **Relationship:**

○ **Inheritance** → Extends Payment.

○ **Implements** the processPayment() method specifically for credit cards.

15. OnlinePayment (Extends Payment)

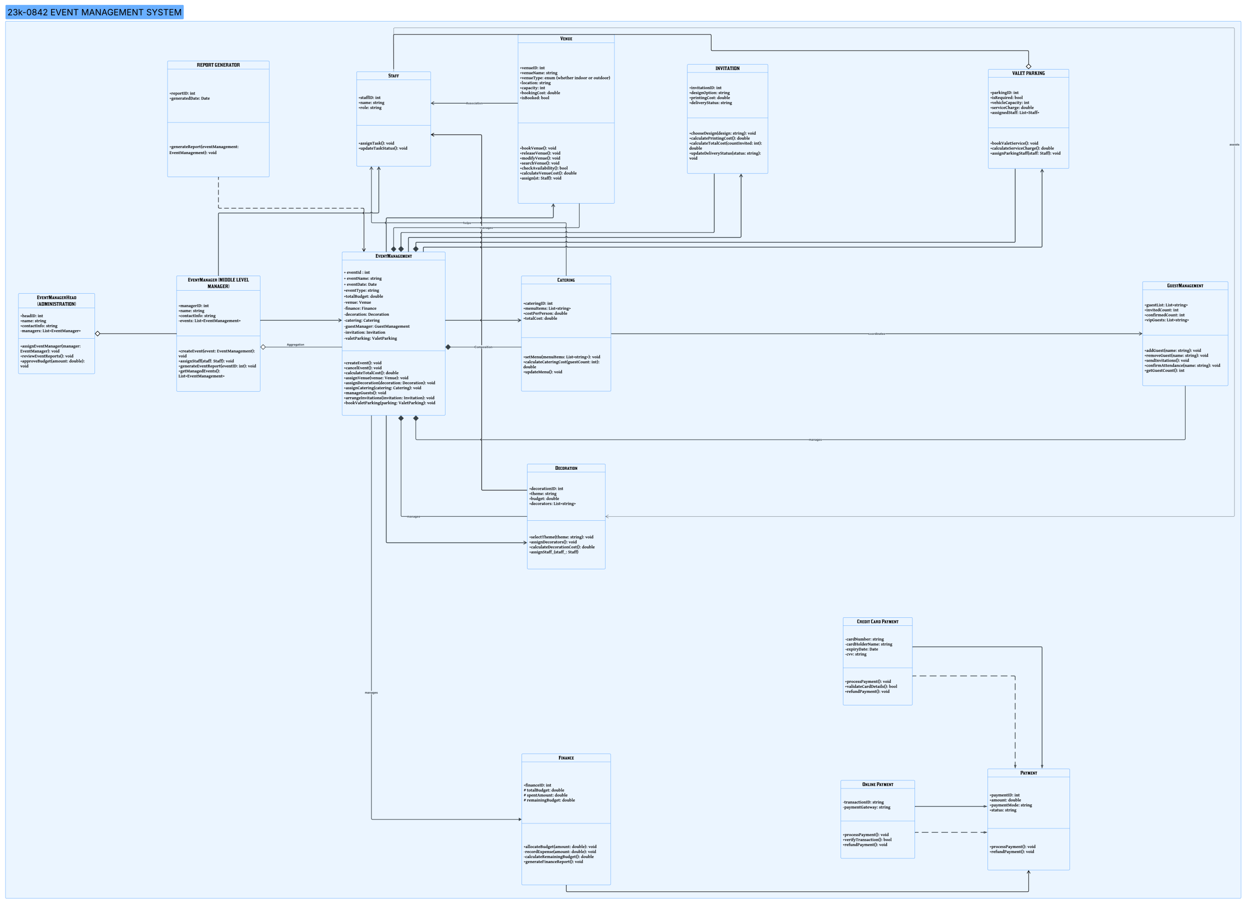
Role:

Handles online payment gateways like PayPal, Easypaisa, or JazzCash.

● **Attributes:** transactionID, paymentGateway.

● **Methods:** Process payments, verify transactions.

- **Relationship:**
  - **Inheritance** → Extends Payment.
  - **Implements** the processPayment() method for online transactions.


23k-0842 EVENT MANAGEMENT SYSTEM
REPORT GENERATOR
+reportID: int
+generatedDate: Date
+generateReport(eventManagement: EventManagement): void
Staff
+staffID: int
+name: string
+role: string
+assignTask(): void
+updateTaskStatus(): void
Venue
+venueID: int
+venueName: string
+venueType: enum (whether indoor or outdoor)
+location: string
+capacity: int
+bookingCost: double
+isBooked: bool
+bookVenue(): void
+releaseVenue(): void
+modifyVenue(): void
+searchVenue(): void
+checkAvailability(): bool
+calculateVenueCost(): double
+assign(st: Staff): void
INVITATION
+invitationID: int
+designOption: string
+printingCost: double
+deliveryStatus: string
+chooseDesign(design: string): void
+calculatePrintingCost(): double
+calculateTotalCost(countInvited: int): double
+updateDeliveryStatus(status: string): void
VALET PARKING
+parkingID: int
+isRequired: bool
+vehicleCapacity: int
+serviceCharge: double
+assignedStaff: List<Staff>
+bookValetService(): void
+calculateServiceCharge(): double
+assignParkingStaff(staff: Staff): void
assists
Association
helps
manages
EventManagerHead (Administration)
+headID: int
+name: string
+contactInfo: string
-managers: List<EventManager>
+assignEventManager(manager: EventManager): void
+reviewEventReports(): void
+approveBudget(amount: double): void
EventManager (MIDDLE LEVEL MANAGER)
+managerID: int
+name: string
+contactInfo: string
-events: List<EventManagement>
+createEvent(event: EventManagement): void
+assignStaff(staff: Staff): void
+generateEventReport(eventID: int): void
+getManagedEvents(): List<EventManagement>
Aggregation
EventManagement
+ eventId : int
+ eventName: string
+ eventDate: Date
+eventType: string
+totalBudget: double
-venue: Venue
-finance: Finance
-decoration: Decoration
-catering: Catering
-guestManager: GuestManagement
-invitation: Invitation
-valetParking: ValetParking
+createEvent(): void
+cancelEvent(): void
+calculateTotalCost(): double
+assignVenue(venue: Venue): void
+assignDecoration(decoration: Decoration): void
+assignCatering(catering: Catering): void
+manageGuests(): void
+arrangeInvitations(invitation: Invitation): void
+bookValetParking(parking: ValetParking): void
Composition
Catering
+cateringID: int
+menuItems: List<string>
+costPerPerson: double
+totalCost: double
+setMenu(menuItems: List<string>): void
+calculateCateringCost(guestCount: int): double
+updateMenu(): void
coordinates
GuestManagement
+guestList: List<string>
+invitedCount: int
+confirmedCount: int
+vipGuests: List<string>
+addGuest(name: string): void
+removeGuest(name: string): void
+sendInvitations(): void
+confirmAttendance(name: string): void
+getGuestCount(): int
manages
Decoration
+decorationID: int
+theme: string
+budget: double
+decorators: List<string>
+selectTheme(theme: string): void
+assignDecorators(): void
+calculateDecorationCost(): double
+assignStaff_(staff_: Staff)
manages
manages
Credit Card Payment
-cardNumber: string
-cardHolderName: string
-expiryDate: Date
-cvv: string
+processPayment(): void
+validateCardDetails(): bool
+refundPayment(): void
Finance
+financeID: int
# totalBudget: double
# spentAmount: double
# remainingBudget: double
+allocateBudget(amount: double): void
-recordExpense(amount: double): void
-calculateRemainingBudget(): double
+generateFinanceReport(): void
Online Payment
-transactionID: string
-paymentGateway: string
+processPayment(): void
+verifyTransaction(): bool
+refundPayment(): void
Payment
+paymentID: int
+amount: double
+paymentMode: string
+status: string
+processPayment(): void
+refundPayment(): void