Assignment #1 - Class modelling

Task #1 : System description

Banking - Communication Services System (Email & SMS)

The *Banking – Communication Services System* is a **secure and auditable platform** used by a bank to create, schedule, and deliver customer-facing messages through **Email** and **SMS** channels.
It supports both **transactional messages** (account alerts, OTPs, etc.) and **marketing campaigns**, offering features such as message templating, multi-recipient campaign management, delivery tracking per recipient, retry policies, and full audit logging to satisfy banking compliance requirements.
The system is **role-based**, allowing different user types - such as customers, bank employees, and administrators - to perform specific actions within their assigned permissions. Its design emphasizes modularity, traceability, and sustainability while ensuring that all communications remain secure, regulated, and measurable in environmental impact.

Core domain classes :
Total = 11 classes + enumeration. The model contains an abstract class, generalization, association class, derived attributes, and qualifiers.

1. User (abstract)
"User" is a generic actor in the system. All user roles inherit from User, so common data like identifiers and contact info are stored once.

Attributes:
- userId: String
- name: String
- surname: String
- emailAddress: String
- phoneNumber: String

2. Customer (inherits User)
"Customer" represents a bank customer who receives messages.

Attributes:
- customerId: String

3. BankStaff (inherits User)
"Bank Staff" prepares and manages campaigns, they are the author of the messages.

Attributes:
- bankStaffId: String
- departmentName: String

4. ChannelAdmin (inherits User)
"Channel Admin" manages the technical configuration of messaging channels (Email/SMS gateways).

Attributes:
- channelAdminId: String

5. SystemAdmin (inherits User)
“System Admin” manages system security, roles and audits.

Attributes:
- systemAdminId: String

6. Message (abstract)
“Message” is the generic template for all message types.

Attributes:
- messageId: String
- subject: String
- body: String
- createdAt: DateTime
- scheduledAt: DateTime (nullable)
- status: MessageStatus (enum: DRAFT, SCHEDULED, SENT, DELIVERED, FAILED)
- /deliveryRate: Double

7. EmailMessage (inherits Message)
“Email Message” is the specialisation for emails; supports file attachments.

Attributes:
- attachments: String[]

8. SMSMessage (inherits Message)
“SMS Message” is the specialisation for SMS; derived field counts how many 160-char segments the text occupies.

Attributes:
- /smsParts: Integer

9. Campaign
“Campaign” groups messages under a single marketing or information initiative.

Attributes:
- campaignId: String
- name: String
- startDate: Date
- endDate: Date

10. Delivery (association class between Message and Customer)
“Delivery” represents “each message to customer delivery” event; stores delivery-specific data.

Attributes:
- deliveryId: String
- status: MessageStatus
- attempts: Integer

11. SustainabilityMetric
“Sustainability Metric” records the environmental footprint of a campaign.

Attributes:
- metricId: String
- energyKWh: Double
- co2Kg: Double

Enumeration:
MessageStatus { DRAFT, SCHEDULED, SENT, DELIVERED, FAILED } : defines the life-cycle of any message or delivery.

Derived Data and Business Constraints :

- Derived data:
- Message./deliveryRate = successful deliveries / total deliveries.
- SMSMessage./smsParts = calculated from message length and encoding.

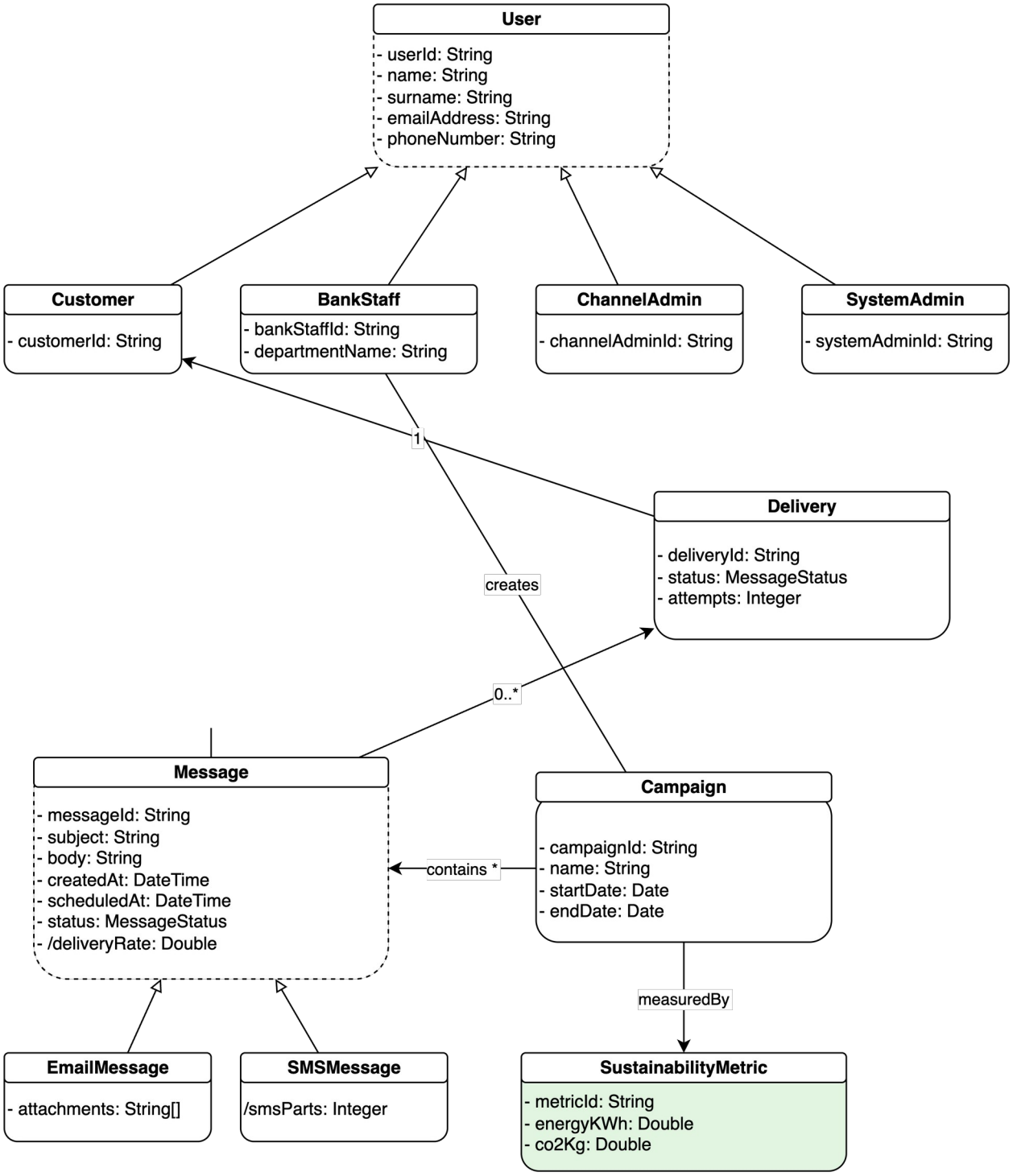
- Constraints:
- Delivery.attempts ≤ 5; after five failures, status = FAILED.
- Only **BankStaff** can create or schedule campaigns.
- **ChannelAdmin** may modify channel configurations but cannot edit content.
- **SysAdmin** manages user roles and security settings.
- **Customer** can only receive messages, not create them.
- Each Campaign must record its sustainability data for annual reporting.

Roles and Responsibilities :
1. Customer : Receives Email/SMS notifications; interacts passively with messages.
2. BankStaff : Creates Campaigns and Messages; monitors delivery outcomes; ensures compliance.
3. ChannelAdmin : Manages channel configurations and providers; oversees throughput and reliability.
4. SystemAdmin : Controls system settings, roles, and audit processes; ensures regulatory compliance.

Relationships and multiplicities :

Relationship	Multiplicity	Explanation
User → subclasses	1 generalisation → {Customer, BankStaff, ChannelAdmin, SystemAdmin}	Different user roles in the system
BankStaff → Campaign	1 staff creates 0..* campaigns	Each campaign has exactly one creator
Campaign → Message	1 campaign contains 1..* messages	Every campaign includes at least one message
Message → Delivery → Customer	1 message produces 0..* deliveries / each delivery targets 1 customer	Association class captures per-customer delivery details (status, attempts)
Customer → Message (qualifier)	[messageSeq] qualifier	Each customer's messages are indexed by a sequential number (messageSeq) for easy audit
Campaign → SustainabilityMetric	0..1 metric per campaign	Each campaign optionally references its sustainability impact data

Task #2 : Class Model



Sustainability Dimension :
The sustainability perspective is integrated at **three levels**:

1. **Artifact / Model Level :**
   The SustainabilityMetric class is explicitly linked to Campaign, storing quantitative data

(energyKWh, co2Kg). This allows each campaign's environmental footprint to be modeled directly.

2. **Process Level :**
   When a campaign is executed, the system automatically estimates resource consumption (e.g., power used by SMS/email servers) and stores it in SustainabilityMetric. Reports comparing different channels (SMS vs Email) can guide low-impact choices.

3. **Application Level :**
   A dashboard can visualize sustainability trends across all campaigns. High-impact campaigns can trigger optimization alerts for ChannelAdmin or SystemAdmin. Integrating sustainability promotes awareness and aligns the bank's digital communication with ESG (Environmental, Social & Governance) objectives.

Task #3 : Modelling Environment and Tool Evaluation

The class diagram for the *Banking – Communication Services System* was designed using **draw.io** - a free, browser-based visual modelling environment.
Draw.io supports the creation of UML diagrams such as class diagrams, use-case diagrams, and sequence diagrams, and allows exporting models in multiple formats including .drawio, .png, .pdf, and .xml.
The choice of draw.io was motivated by several decisive factors:

1. **Accessibility and Collaboration :**
   Draw.io is entirely web-based and does not require installation, making it convenient for remote teamwork. Diagrams can be stored and versioned in Google Drive, OneDrive, or GitHub.
2. **Ease of Use :**
   The interface provides drag-and-drop UML shapes and connectors, allowing for fast model construction without steep learning curves.
3. **Customization and Flexibility :**
   Custom labels (e.g., for multiplicities, derived attributes /deliveryRate, or qualifiers [messageSeq]) can be easily added, which is critical for accurate class modelling.
4. **Compatibility and Export Options :**
   The .drawio format can be converted to PlantUML or XML, making it possible to integrate into other UML tools if deeper modelling or code generation is required later.
5. **Open Source and Lightweight :**
   Unlike heavy UML suites such as Enterprise Architect, draw.io is open-source, free, and lightweight, which makes it suitable for academic use.

In conclusion, Draw.io was chosen because it is **free, intuitive, collaborative, and flexible**, making it ideal for early-stage system modelling.

While draw.io is primarily a diagramming tool, it supports model-to-code transformation indirectly by allowing diagrams to be exported as PlantUML XML files, which can then be processed by external code generation tools such as PlantUML, Visual Paradigm, or GenMyModel.

Although draw.io does not provide native automatic code generation, its compatibility with PlantUML and other export formats enables seamless integration into broader model-driven development workflows. In later project stages, the models designed in draw.io can be migrated into code-centric environments like Visual Paradigm or StarUML to generate Java, Python, or C# class skeletons that reflect the designed structure.

Therefore, draw.io effectively supports the conceptual design phase of the *Banking - Communication Services System* while maintaining portability and future readiness for code generation and model-based software engineering.

Observations regarding generated code:

- **Mapping accuracy:**
  When exported through the PlantUML format, class names, attributes, and inheritance structures (User, Message, and their subclasses) are preserved correctly.
  Associations (e.g., Message → Delivery → Customer) are generated as references or list relationships in the target code.
- **Weaknesses:**
  - Some UML semantics such as *derived attributes (/deliveryRate)*, *qualifiers ([messageSeq])*, and *association classes (Delivery)* are not automatically translated into programming-language constructs.
  - Code generators often omit visibility modifiers or derived field formulas, requiring manual refinement after generation.
- **Model–Code alignment:**
  The overall structure of the generated code matches the class hierarchy and relationships in the designed model, but semantic details (e.g., multiplicities and OCL constraints) need to be re-added in code manually.
- **Improvement reflection:**
  This experiment highlights that model clarity (precise class naming, attributes, and multiplicities) is essential for clean code generation.
  Some refinements—like converting derived data into calculated methods—were necessary to make the model implementation-ready.