Rules for Drawing UML Class Diagram

1. **Define the Scope and Context**:
   - Clearly specify the system or subsystem being modeled (e.g., "Timetabling System Data Model").
   - Focus on the key entities and relationships relevant to the system's requirements or domain.
2. **Identify Classes**:
   - Represent classes as **rectangles** divided into three compartments: class name (top), attributes (middle), and operations (bottom).
   - Name each class with a singular, descriptive noun or noun phrase (e.g., "Student," "Schedule") reflecting a system entity.
   - Ensure classes correspond to significant domain entities identified during requirements elicitation.
3. **Specify Attributes**:
   - List attributes in the middle compartment of the class rectangle, using the format: `[visibility] name: type`.
   - Use visibility indicators: + (public), - (private), # (protected), or ~ (package).
   - Example: `-studentID: String` or `+name: String`.
   - Include only relevant attributes that reflect the class's data properties, avoiding implementation details.
4. **Specify Operations**:
   - List operations (methods) in the bottom compartment, using the format: `[visibility] name(parameter: type): returnType`.
   - Example: `+viewTimetable(): Schedule` or `-updateStatus(status: String): void`.
   - Focus on key behaviors or services tied to functional requirements, excluding trivial operations.
5. **Model Relationships**:
   - Represent relationships between classes using specific connectors:
     - **Association**: A **solid line** indicating a general relationship (e.g., "Student enrolls in Course"). Label with a verb phrase if needed.
     - **Aggregation**: A **solid line with an open diamond** at the whole class, indicating a "has-a" relationship (e.g., "Department has Faculty").
     - **Composition**: A **solid line with a filled diamond** at the whole class, indicating a strong "owns-a" relationship where parts cannot exist without the whole (e.g., "Schedule owns TimeSlot").

        - **Generalization**: A **solid line with a hollow triangle** pointing to the parent class, indicating inheritance (e.g., “Lecturer inherits from Person”).
    - Include **multiplicity** (e.g., `1`, `0..*`, `1..n`) at the ends of associations to specify how many instances are involved.

6. **Maintain Simplicity**:
    - Focus on **key classes and relationships** relevant to the system’s requirements, avoiding excessive detail or low-level implementation.

7. **Ensure Consistency with Requirements**:
    - Align classes, attributes, and operations with **functional and non-functional requirements** from the requirements engineering process.
    - Verify that classes reflect domain entities identified during elicitation (e.g., stakeholders like “Student” or “Admin”).
    - Exclude non-functional details (e.g., performance metrics) unless they define class constraints.