

1 Lecture 1: Class Diagrams

Course: Software Engineering and Systems Analysis

Topic: Class Diagram with UML

Recommended duration: 90 minutes

1.1 Learning objectives

By the end of this lecture, students should be able to explain what a class diagram represents, distinguish a class from an object, identify attributes and operations, model associations with multiplicity, and produce a readable class diagram from a short problem statement.

1.2 1. Overview

A **class diagram** is a structural model that describes the important types in a software system and the relationships among those types. It does not show the order of events. Instead, it provides a stable view of the system's vocabulary, data, and responsibilities.

A class diagram is useful during analysis because it exposes missing concepts and ambiguous relationships before implementation begins. During design, it becomes a bridge between requirements and source code. The model should be detailed enough to communicate structure, but not so detailed that it merely duplicates every line of code.

1.3 2. Core concepts

A **class** is a description of objects that share structure and behavior. A **class name** identifies the concept. **Attributes** describe information held by an object. **Operations** describe behavior that the object offers. A typical class is written as a rectangle divided into three compartments:

```
+-----+
| Customer          |
+-----+
| - customerId: Integer |
| - name: String       |
| - email: String      |
+-----+
| + register(): void   |
| + updateEmail(): void |
+-----+
```

The visibility symbols are commonly interpreted as follows: + means public, - means private, # means protected, and ~ means package visibility. Visibility is a design decision, so it should not be added mechanically when the model is still conceptual.

1.4 3. Associations and multiplicity

An **association** states that instances of two classes are connected. Multiplicity describes how many instances may participate in the relationship. Common multiplicities include 1, 0..1, 0..*, and 1..*. For example, a customer may place many orders, while each order belongs to exactly one customer:

```
Customer 1 ----- 0..* Order
           places
```

The role names clarify the meaning of each end. A strong model also states important constraints, such as whether an order must contain at least one order line.

1.5 4. Generalization, aggregation, and composition

Generalization represents an “is-a” relationship. If `CreditCardPayment` and `CashPayment` are specialized kinds of `Payment`, both may inherit common properties from `Payment`.

Aggregation represents a whole–part relationship in which the part can conceptually exist independently.

Composition is stronger: the lifetime of the part is controlled by the whole. An `Order` and its `OrderLine` objects are often modeled with composition because an order line has no useful identity outside its order. Use these relationships carefully. A relationship should express a business or design rule, not simply make the diagram look complete.

1.6 5. Deriving a diagram from requirements

Start by extracting candidate nouns and verbs from the requirements. Nouns may become classes or attributes. Verbs may become operations or associations. Then remove synonyms, implementation details, and concepts that have no responsibility in the system.

Consider the statement: “A library member searches the catalog, borrows books, and returns them. A loan records the borrowing date and the due date.” A reasonable first model contains `Member`, `Book`, `Catalog`, and `Loan`. The dates belong to `Loan`, not to `Book`, because the dates describe a particular borrowing event. The modeling process is iterative. Ask whether every class has a clear purpose, whether every relationship has a meaningful name, and whether multiplicities reflect the requirements. Validate the result with at least one realistic scenario.

1.7 6. Common mistakes

The first common mistake is confusing a class with an instance. `Book` is a class, while “the copy with barcode B104” is an object. The second mistake is placing every noun in a separate class. A noun should become a class only when it has identity, behavior, or important relationships.

Another mistake is using inheritance for code reuse alone. Inheritance should communicate a genuine specialization. Finally, avoid connecting every class to every other class. Unnecessary associations make the model difficult to understand and often indicate that responsibilities have not been assigned clearly.

1.8 7. Worked example: online bookstore

The bookstore contains customers, books, orders, and order lines. A customer can create several orders. Each order contains one or more order lines. Each line refers to one book and stores the quantity and selling price at the time of purchase.

```
Customer 1 ----- 0..* Order
                  ♦ 1..* OrderLine
                      * ----- 1 Book
```

```
Order
- orderNumber: String
- orderDate: Date
+ calculateTotal(): Money

OrderLine
- quantity: Integer
- unitPrice: Money
+ lineTotal(): Money
```

The `unitPrice` belongs to `OrderLine` rather than `Book` because the price may change after an order is placed. This is a small example of preserving historical business facts in the correct class.

1.9 8. Quality checklist

A useful class diagram has a clear scope, meaningful names, consistent notation, and correct multiplicities. It distinguishes structural facts from implementation choices. Each class has a responsibility, and each association answers a question about how objects collaborate.

Before finalizing the diagram, test it against normal, exceptional, and boundary cases. Ask whether the model can represent a customer with no orders, an order with multiple lines, and a discontinued book that still appears in an old order.

1.10 9. Practice activity

Model a university course registration system. Students enroll in courses, courses are taught by instructors, and each enrollment stores the registration date and final grade. Identify classes, attributes, operations, associations, and multiplicities. Explain why `Enrollment` should be modeled as a class rather than as a simple association.

1.11 Summary

A class diagram is a structural explanation of a system. Its most important decisions concern the system's concepts, their responsibilities, and their relationships. Good diagrams remain readable because they model essential meaning instead of every possible implementation detail.

1.12 References