# Engineering the Requirements

Youssef Ghatas

# Acknowledgement


- Presentation is adapted from Ian Sommerville “Software Engineering 9th ed.”

# Agenda

- Software Requirements Concepts & Process
- Functional vs. Non-functional
- Use case model
- Development Constraints
- Operational Environment

# TMS: Case study

- Traffic Management Systems (TMS) use a variety of technologies to manage traffic flows and the effects of congestion on the roading network. Traffic Management Systems do this by addressing the traffic management effects of accidents and slow moving or queuing vehicles, planned events and extreme weather.

# Software Requirements

The IEEE Standard Systems and Software Engineering Vocabulary [IEEE 24765] defines a software requirement as:

– A software capability needed by a user to solve a problem to achieve an objective.

– A software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

- System requirements are the requirements for the system as a whole. For example, a Traffic Management System (TMS) would have requirements for the various system elements:
  - people (e.g., requirements for the traffic police that help control traffic in the TMS)
  - hardware (e.g., requirements for the signage and traffic signals in the TMS)
  - software (e.g., requirements for the software subsystem of the TMS that helps manage and control traffic)
- One common way to classify requirements is as “functional” or “non-functional“.

# Software Requirements

- One common way to classify requirements is as "functional" or "non-functional". From SWEBOK 2014:
  - ✓ **Functional requirements** describe the functions that the software must execute.
    (e.g., "The TMS shall provide the capability for a Traffic Manager to set the duration of individual traffic signal states using the following duration ranges: Red [30–120 seconds], Green [30–120 seconds], and Amber [5–10 seconds]".)
  - ✓ **Non-functional requirements** are the ones that act to constrain the solution.
    (e.g., "The TMS software subsystem shall not fail more often than once in every 24,000 hours".)

- A **Quality Attribute** is a characteristic that is related to the performance or use of the software, which is not related to specific functionality of the software. Example attributes include the following: availability, reliability, usability, reusability, maintainability, efficiency (in use of computing resources), safety, and security.
- An **Emergent Requirement** is a requirement which cannot be addressed by a single component, but which depends for its satisfaction on how all the software components interoperate (e.g., "The TMS shall provide for at least a 20% increase in traffic throughput during the weekday traffic period of 4:30 pm to 6:30 pm".).

# Documentation-Specifying Requirements-SRS Outline

# Software Requirements

The following are characteristics of an effective set of software requirements

- Correct – every specified requirement is one that the software must satisfy
- Clear/Unambiguous – requirements are expressed in a precise, unambiguous, easy to understand format; there is only one interpretation of a requirement
- Complete – includes all requirements agreed upon by the system stakeholders
- Concise – describes a single property, without excessive verbiage
- Consistent – no two requirements contradict each other
- Feasible – realizable within resource and schedule constraints
- Traceable – can be **traced backward** to sources (e.g. end user, customer, government, or company policies) and **forward to** software artifacts developed from the requirements (e.g., design components, code, test cases)
- Verifiable – has a clear, testable criterion, and a cost-effective way to check that a requirement has been properly implemented
- Prioritized – ranked for importance and/or stability (e.g., for importance: “essential”, “conditional”, “optional”)

# GUIDELINES FOR WRITING REQUIREMENTS

## How does one go about writing a good set of requirements?

Since most requirements efforts use a natural language style to specify requirements and natural languages lack the precision and rigor of formal languages, great care must be taken to ensure that natural language requirements are clear, concise, correct, consistent, verifiable, etc. In the following list, we offer a set of guidelines on writing good natural language requirements.

Write requirements in complete sentences in an "active voice".

– Example: Write "The TMS shall provide the capability for a Traffic Manager to set the duration of individual traffic signal states, …" rather than "The duration of individual traffic signal states may be set by the TMS Traffic Manager, such that …".

Be precise, avoid ambiguity.

– Example: The statement "The TMS shall have high reliability" is imprecise. The statement "The TMS software subsystem shall not fail more often than once in every 20,000 hours" is clear about what is meant by reliability.

Requirements need to be verifiable.

– Example: The statement "The new TMS system shall result in more efficient traffic flow in the intersection" is not testable. The statement "The new TMS system shall reduce the overall wait time at the intersection by 15%" is more precise and easier to test.

Keep requirements concise, avoid wordiness.

– Example: Write "A TMS Manager shall be able to initiate video recording of traffic at a designated traffic signal" instead of "If it is desired to perform a video recording of traffic at a traffic signal, a TMS Traffic Manager designates a traffic light and then starts the video recorder for that traffic signal and stops it when she is finished".

Write requirements in a consistent fashion.

– Example: Use the same names for the same objects – in the TMS SRS, do not refer separately to "Traffic Light", "Traffic Signal", "Stop Light", or "Traffic Control Device", if they are synonyms.

Each requirement statement represents a single requirement, avoid compound statements.

– Example: The statement "The TMS Traffic Manager shall be able to view traffic at a designated traffic signal or request a TMS report" represents two separate requirements.

Avoid duplication.– Repeating the same requirement at different places in an SRS can cause confusion and create an inconsistency, if the requirement is reworded or changed (in one place and not where it is repeated).

# Examples

- Avoid **Non-Requirement Statements:**
  - General Information: Statements that provide context, background information, or general knowledge about the system, but do not specify any particular functionality.

    Facts or Definitions: Statements that define terms or relationships but do not describe system behavior.

    Negative Statements: Statements that describe what the system is not or relationships that don't exist within the system.

    "The system is designed for educational institutions."
  - "The application is not compatible with older operating systems."
  - "A professor is not a student."
  - "The company's mission is to provide quality customer service."

- **Examples of Functional Requirements:**
  - "The system must allow users to create and edit profiles."
  - "The system shall generate a monthly financial report."
  - "Users will be able to reset their passwords via email verification.“
  - They are actionable: use verbs like "must," "shall," "can," "will be able to," or other action-oriented terms.

# Examples

- Non-functional requirements define the quality attributes and constraints of a system, while functional requirements describe specific system functionalities.

  **Functional answer the what? And non-functional answer the How?**

- Examples of non-functional requirements
  - Pressing the switch, the room shall get lightened in less than one second
  - After two minutes that the room is empty the light must switch off

# Examples

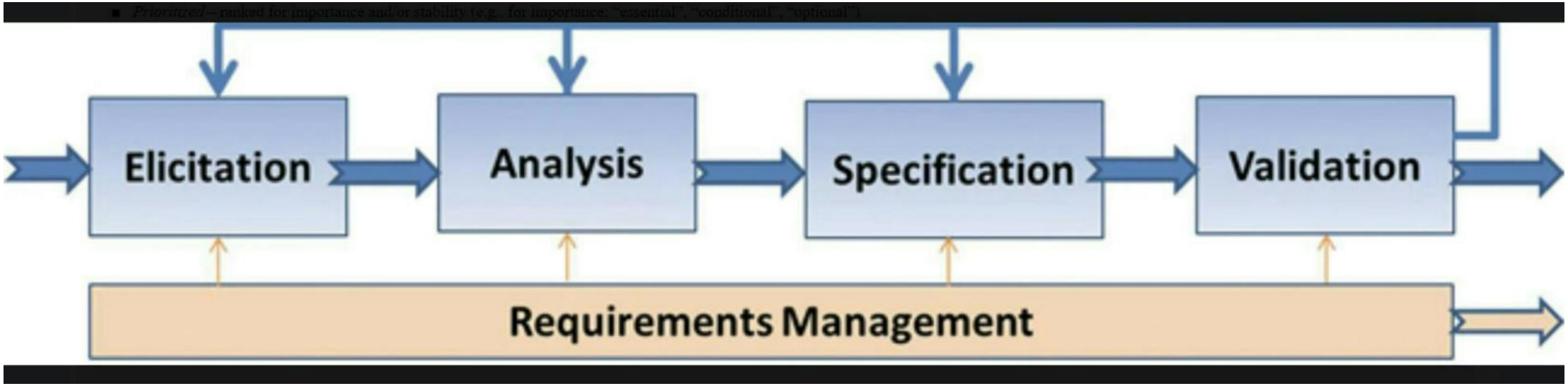
This requirement is not well written. Why?

- The output of the program shall usually be given within 10 seconds

1. The requirement contains a general word “usually”.
   Is there any condition in which this requirement should not be the case?
2. The requirement is also incomplete:
   The output of the program is not specified.
3. Also, when do you start counting the 10 seconds?
   The Requirement cannot be implemented or tested as stated.

- When performing calculations the software shall produce correct results.
- This requirement is not necessary. This type of requirement is inherent to correctness should not need to be specified, unless it refers to a specific (complex) calculation and what is considered as correct.

# Requirements Process



# Requirements Process

- Requirements Elicitation – **determine where software requirements come from (identifying stakeholders)** and collect them (by apprenticeship, prototyping, interviews,..)
- Requirements Analysis – **study, analyze, and model the problem to be solved**
- Requirements Specification – **define and document the requirements** in an organized and precise manner
- Requirements Validation – **ensure that the requirements** are properly understood and confirm that the requirements conform to applicable standards and regulations, and that they are understandable, correct, consistent, and complete.

Requirement Management – for a complex software system, the above processes much be managed **throughout the life of the software**, because the requirements will change – new ones added and modifications made to improve understanding or correct errors.

- Requirements development is not strictly linear: work in one phase of the process may lead to a return to an earlier phase. For example, during analysis of the TMS requirement on the duration of different stoplight states (red, green, amber), an analyst may question whether the periods are long enough for some situations. In this case, the engineer may need to go back and discuss this with traffic officials; hence, reentering the Elicitation Phase.

# Analyzing Requirements

After the Team completed its initial requirements elicitation activities, Team leader led a discussion of the next task listed in the Development Process: “analyze requirements”.

■ The purpose of requirements analysis is to **take information provided by the stakeholders, and analyze and model that information** so that

– we better understand the stakeholder needs and requirements;

– the functional and non-functional requirements can be specified clearly, correctly, and completely;

– and, there is a solid foundation for the design, construction, and testing of the software.

■ During the requirements analysis phase, we will build a “conceptual model” for the system that will support effective communication between potential users, management, and the team.

The Conceptual Model consists of the multiple elements including:

– Conceptual Design – an initial view of the system architecture with a high-level view of the principal components and their relationship.

– Use Case Model – a model that describes the various ways (use cases) that system users can interact with a system.

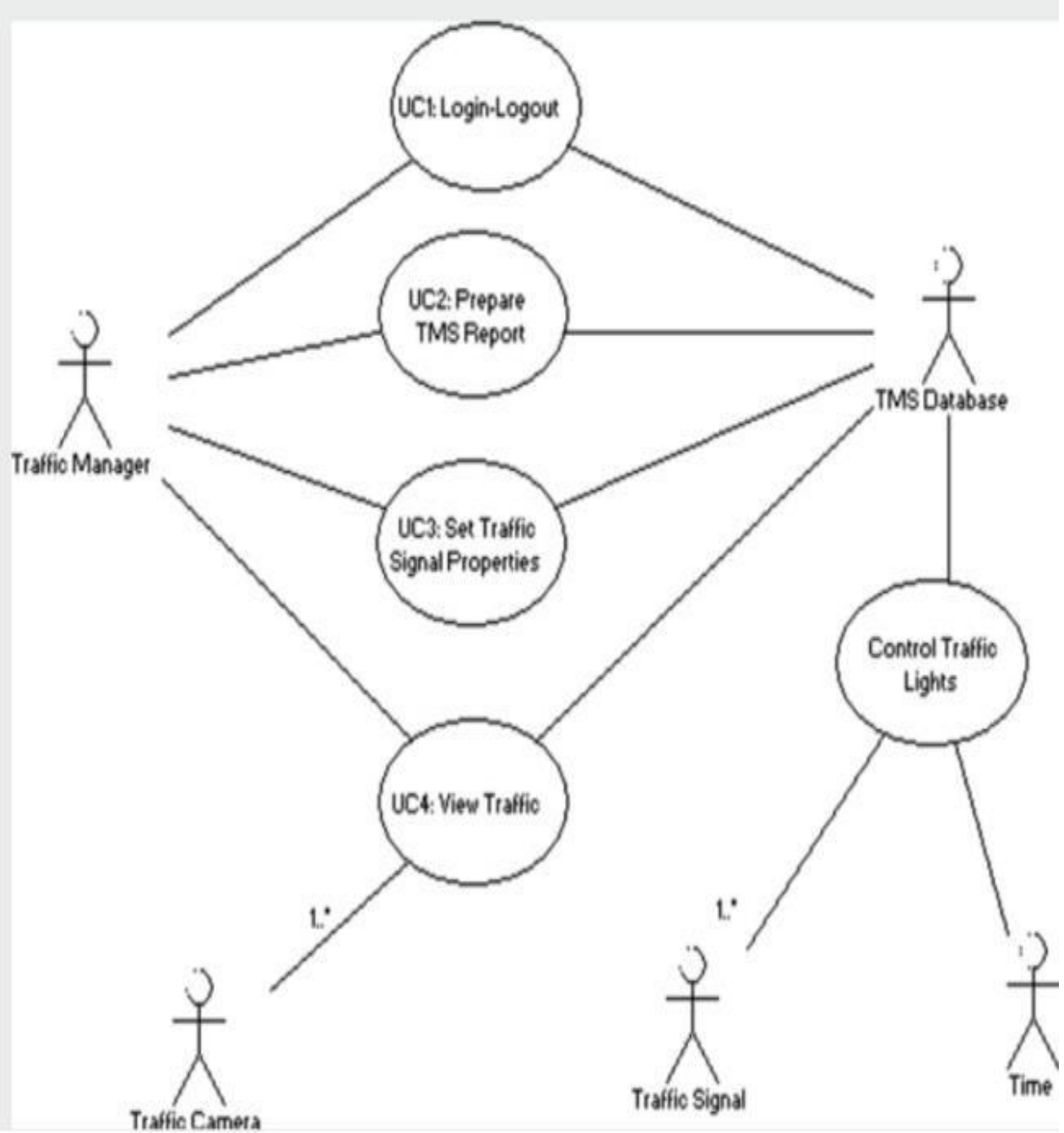
# Outputs from Requirements Engineering

## Use Case Model

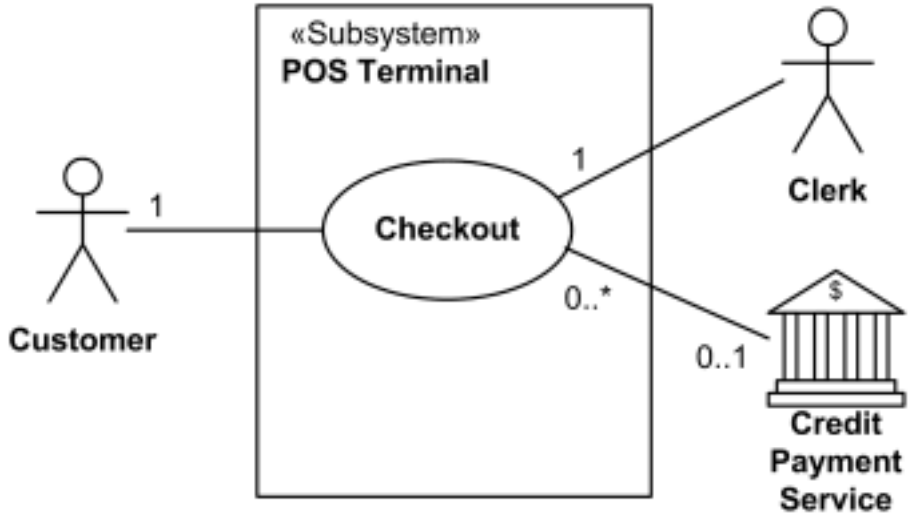
- A model that describes the various ways (use cases) that system users can interact with a system.
- A use case actor is a user or some other external entity (e.g., another system such as a database or an alarm system) that interacts with the system being analyzed.
- A use case includes a scenario, which provides a sequence of interactions between the system being modeled and its external actors.
- Each use case scenario provides a picture of how the user interacts with the system, an initial version can be used to elicit user response that will help clarify, expand on, or correct the system functionality being modeled.
- Use cases can also be used in system test planning: use case scenarios can be converted into test case scenarios that guide the engineer performing the test.
- There are different opinions, approaches, techniques, and styles associated with use case modeling.

# Use Case Model

- Unified modeling language (UML) includes a diagram, a Use Case Diagram, for depicting actors, uses case, and their relationships.
- Figure shows a use case diagram for the Traffic Management System (TMS).
    1. The stick figures represent the actors
    2. The ovals represent the use cases
    3. The connectors represent the relationships between the actors and the use cases.



# Multiplicity




*"Checkout use case requires Customer actor, hence the 1 multiplicity of Customer. The use case may not need Credit Payment Service (for example, if payment is in cash), thus the 0..1 multiplicity."*
https://www.uml-diagrams.org/use-case-actor-association.html

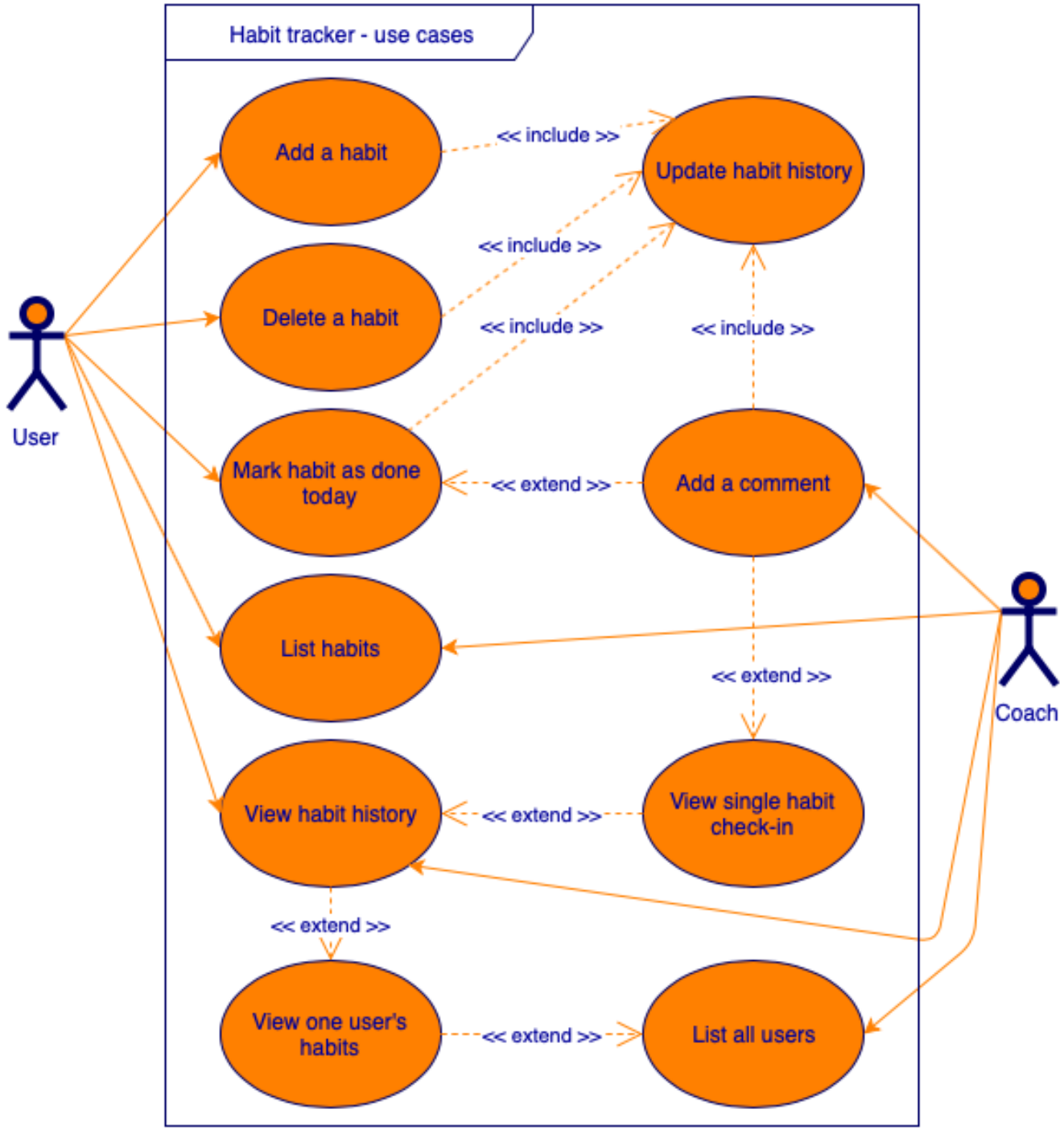
# Use Case Activity: Question

- **Requirements:**

1. **Actors:**
    1. **User**: The primary individual who uses the habit tracker.
    2. **Coach**: A secondary actor who can access additional functionalities.
2. **Use cases (Actions):**
    1. **Add a habit**: Allows the user to add a new habit to track.
    2. **Delete a habit**: Enables the user to remove an existing habit.
    3. **Mark habit as done today**: Lets the user mark a habit as completed for the day.
    4. **List habits**: Displays all the habits the user is tracking.
    5. **View habit history**: Shows the historical data of a specific habit.
    6. **View one user's habits**: Allows the coach to view the habits of a specific user.
    7. **List all users**: Enables the coach to see all users in the system.
    8. **Update habit history**: A necessary function invoked whenever a habit is added, delete or marked as done. The history should also be updated if the coach added a comment.
    9. **Add a comment**: The coach can add comments on a user’s habit check-in (when a habit is marked as done).
    10. **View single habit check-in**: Allows the coach to see specific check-in details for a habit.
3. **Relationships:**
    1. **Include Relationships**: Use cases that are always performed as part of another use case.
        1. For example, "Update habit history" is included when the user marks a habit as done.
    2. **Extend Relationships**: Use cases that may be optional or may be invoked based on certain conditions.
        1. For instance, "Add a comment" is an extension of "Mark habit as done today."

# Use Case Activity: Solution

Ref: https://drawio-app.com/blog/uml-use-case-diagrams-with-draw-io/



# Use Case Scenario

A Use Case Model includes a scenario for each use case: a sequence of steps describing the interaction between the actor(s) and the system. There is no standard for writing use case scenarios. Some are less structured, just listing the scenarios steps; and others have different or additional components. A good example scenario can Have the following features:

- a use case Goal
- a list of Actors
- a pre-condition (Pre), which states **what must be true before** carrying out the use case scenario.
- a post-condition (Post), which states **what must be true when** the use case scenario is completed.
- a Main Success Scenario that describes a set of steps which will lead to achievement of the uses case Goal. The steps are numbered and may include control structures such as selection (e.g., steps 13A and 13B) and repetition (e.g., Go to Step 1).
- UC GUIs (Use case GUIs) that are referenced in the scenario.
- Exceptions (Alternative Scenarios of Failure) that describe what happens when an exceptional situation occurs, which would interrupt the Main Success Scenario and require an alternate scenario. In such a situation, the use case Goal may not be achieved.
- Use Cases Utilized in the scenario. A scenario can call (or “include”) another use case, which is equivalent to inserting the referenced use case scenario steps into the main scenario.

# Example

**Use Case ID**: UC3
**Use Case Name**: Set Traffic Signal Properties
**Goal**: Set/Change traffic signal properties
**Primary Actors**: Traffic Manager
**Secondary Actors**: TMS Database
**Pre:**
- Traffic Manager is logged into his/her account.
- TMS Traffic Signal Configuration Page is displayed on User Display Device.

**Post:**
- All newly entered traffic signal properties have been stored in the TMS Database.
- TMS Traffic Signal Configuration Page is displayed on User Display Device.

**Main Success Scenario:**

| Step | Actor Action | Step | System Reaction |
|---|---|---|---|
| 1 | Select "Set Traffic Signal Properties" | 2 | Display "Select the traffic signal(s) for which you wish to set properties". |
| 3 | Select one or more traffic signals. | 4 | Display "Enter Green Light duration (in seconds)". |
| 5 | Enter seconds of Green Light duration. | 6 | Store Green Light duration for selected traffic signals in the TMS database. |
| | | 7 | Display "Enter Red Light duration (in seconds)". |
| 8 | Enter seconds of Red Light duration. | 9 | Store Red Light duration for selected traffic signals in the TMS database. |
| | | 10 | Display "Enter Amber Light duration (in seconds)". |
| 11 | Enter seconds of Amber Light duration. | 10 | Store Amber Light duration for selected traffic lights in the TMS database. |
| | | 12 | Display "Do you wish to set the properties for other traffic signals? YES/NO?" |
| 13A | Enter "YES" | 14A | Go to Step 1 |
| 13B | Enter "NO" | 14B | Display TMS Traffic Signal Configuration Page |

**UC GUIs:** DH Configuration Page, DH Default Parameter Configuration Page
**Exceptions (Alternative Scenarios of Failure):**
- At any time, User fails to make an entry.
  a. Timeout after 5 minutes and logout from the system.
- System detects a failure (loss of internet connection, power loss, hardware failure, etc.).
  a. Display an error message on the User Display Device.
  b. System restores system data from TMS Database.
  c. Logoff user.

**Use Cases Utilized**: none
**Notes and Issues**: The duration of individual traffic signal states must be in the following duration ranges: Red [30 to 120 sec], Green [30 to 120 sec], and Amber [5 to 10 sec]

# Dev. Constraints Example

1. Technology Constraints:
   - The software shall be developed using Java programming language.
   - It must be compatible with Internet Explorer 11 and later versions.
2. Resource Constraints:
   - The development team consists of part-time developers.
   - Server infrastructure is limited to 100 GB of storage space.
3. Regulatory Constraints:
   - The software must comply with GDPR regulations for data privacy.
   - It should adhere to industry-specific security standards (e.g., ISO 27001).
4. Time Constraints:
   - The software release is scheduled for March 1, 2028.
   - It must be ready for the holiday shopping season.
5. Scope Constraints:
   - The software will initially support Windows and macOS only.

# Op. Env. Example

1. Hardware Requirements:
   - Minimum: 4GB RAM, dual-core processor
   - Recommended: 8GB RAM, quad-core processor
2. Software Dependencies:
   - MySQL Database Server (version 5.7 or later)
   - Apache Tomcat (version 9.0)
   - Java Runtime Environment (JRE 8 or later)
3. Network Requirements:
   - An internet connection is required for real-time updates.
   - The software communicates with external APIs over HTTPS.
4. Security Considerations:
   - Data encryption using TLS/SSL.
   - Access control based on user roles and permissions.
5. Scalability and Performance:
   - The software should be designed to scale horizontally to accommodate increased user loads.
   - Performance testing should be conducted to ensure responsive user experience.

# SDS

## Engineering the Requirements

## From Requirements to Design

# Agenda

- SDS Overview
- Class Diagram
- Sequence Diagram

# SDS: General Chapters

- Introduction
- Architectural Design
  - UML Class Diagram
  - UML Sequence Diagram
  - **Coding style guidelines****
- UI Design
- Data Management
- Hardware Design

# Case Study: DigitalHome (DH)

- New homeowners, are interested in living in houses that have "smart" features that automate and facilitate energy efficiency, entertainment delivery, communication, household chores, and home security.
- The design and construction of such dwellings rely on the latest technology (devices with smart capabilities, distributed computing, wireless and web communications, intelligent agents, etc.)

# What is UML and Why we use UML?

- UML → “Unified Modeling Language”

➢Language: express idea, not a methodology

➢Modeling: Describing a software system at a high level of abstraction

➢Unified: UML has become a world standard

Object Management Group (OMG): www.omg.org

# What is UML and Why we use UML?

- More description about UML:

➢ It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems

➢ The UML uses mostly graphical notations to express the OO analysis and design of software projects.

➢ Simplifies the complex process of software design

# What is UML and Why we use UML?

- Why we use UML?

➢ Use graphical notation?

[Balance Point] more clearly than natural language (imprecise) and code (too detailed).

➢ Help acquire an **overall view** of a system.

➢ UML is ***not*** **dependent on any one language** or technology.

➢ UML is a powerful language for use in software architecture and design, as it allows for the modeling of system's structure and behavior.

➢ Structure (static) modeling in UML:
Example: class diagrams

➢ The interaction (dynamic) modeling of the system or individual components is accomplished
Example: sequence diagrams

# Overview of UML Diagrams

## Structural

: element of spec. irrespective of time

- Class
- Component
- Deployment
- Object
- Composite structure
- Package

## Behavioral

: behavioral features of a system / business process

- Activity
- State machine
- Use case
- Interaction

## Interaction

: emphasize object interaction

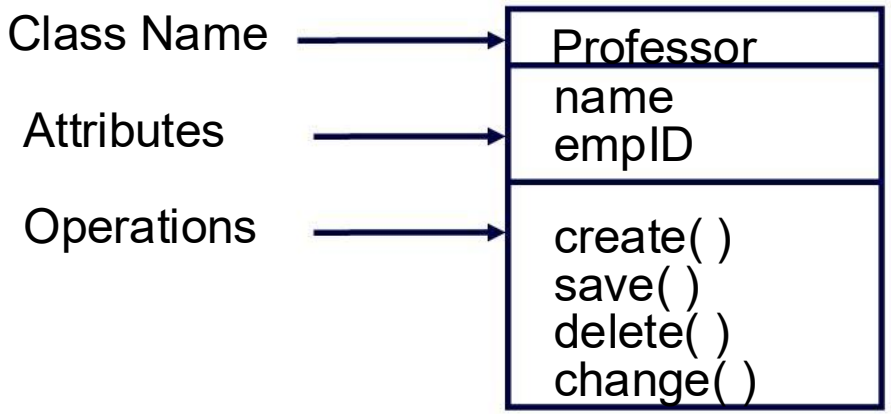
- Communication (collaboration)
- Sequence
- Interaction overview
- Timing

# Basic Concepts of Object Orientation and how to model them using UML

- Object
- Class
- Attribute
- Operation
- Interface (Polymorphism)
- Component
- Package
- Relationships

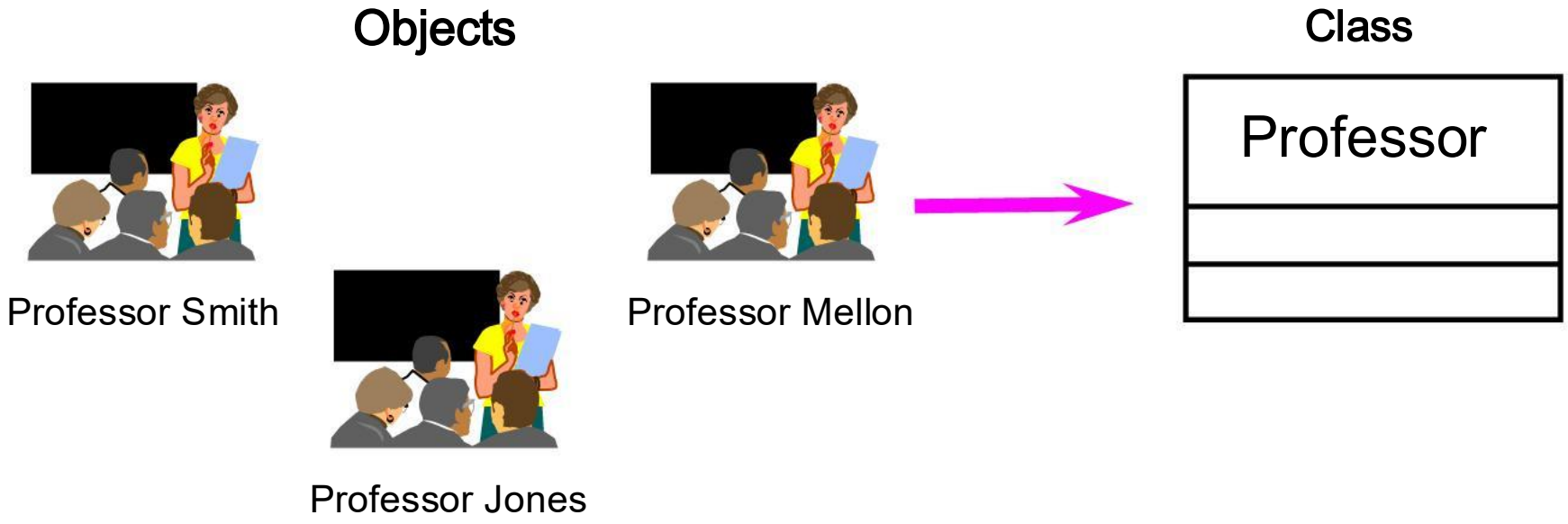
# A Class

- A class is comprised of three sections
  - The first section contains the class name
  - The second section shows the structure (attributes)
  - The third section shows the behavior (operations)

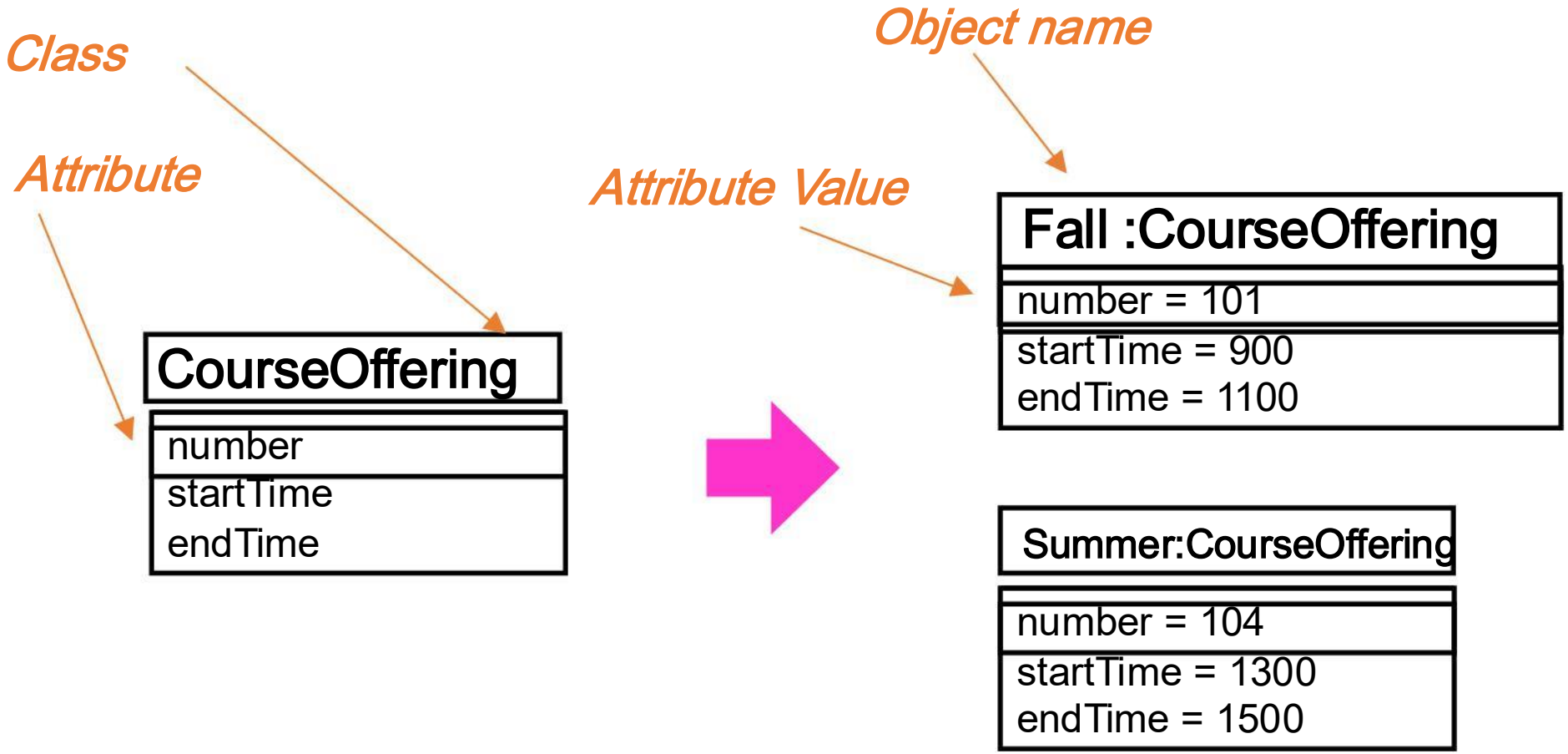


# The Relationship Between Classes and Objects

- A class is an abstract definition of an object
  - It defines the structure and behavior of each object in the class
  - It serves as a template for creating objects

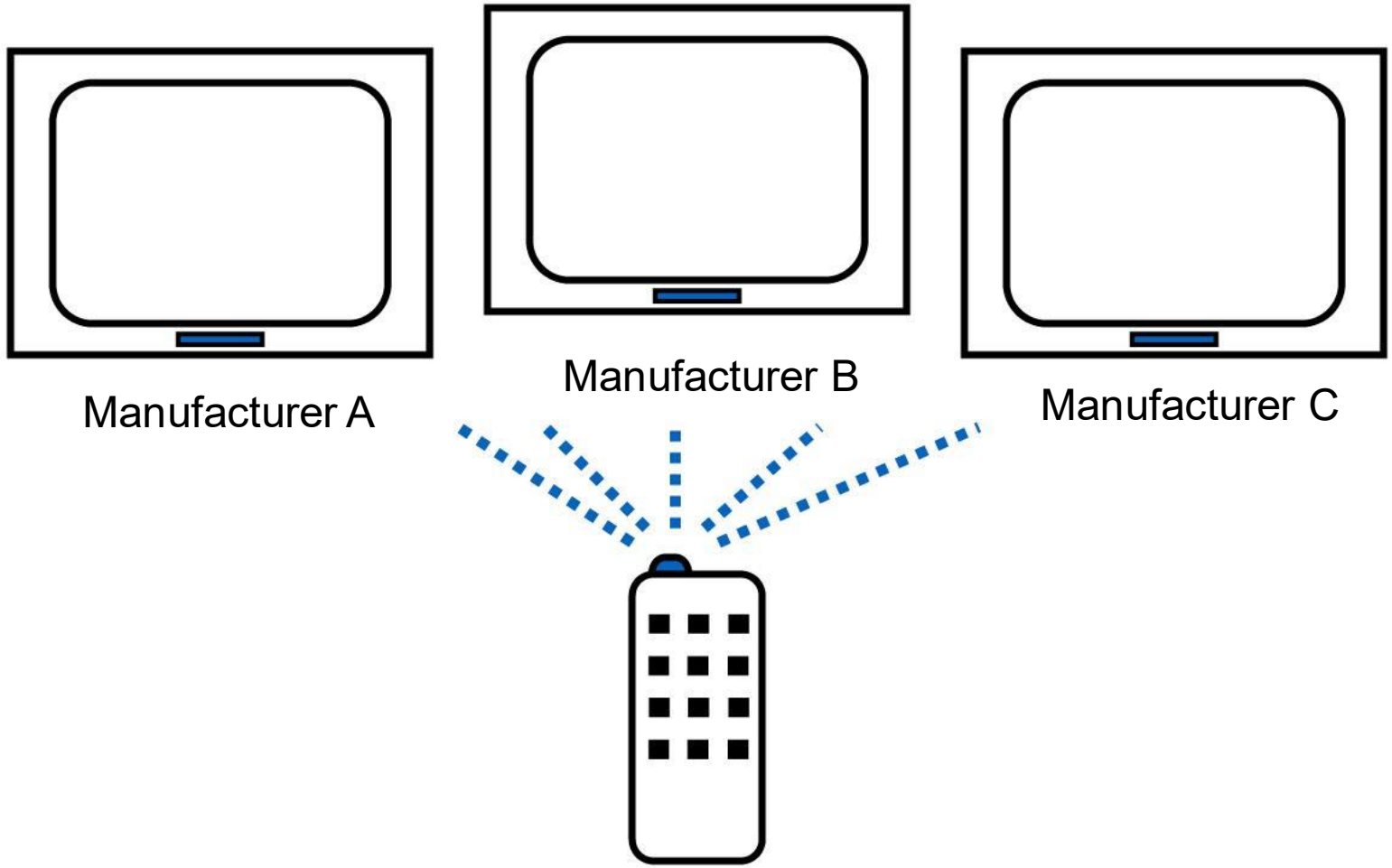


# What is an Attribute?



# What is Polymorphism?

- The ability to hide many different implementations behind a single interface



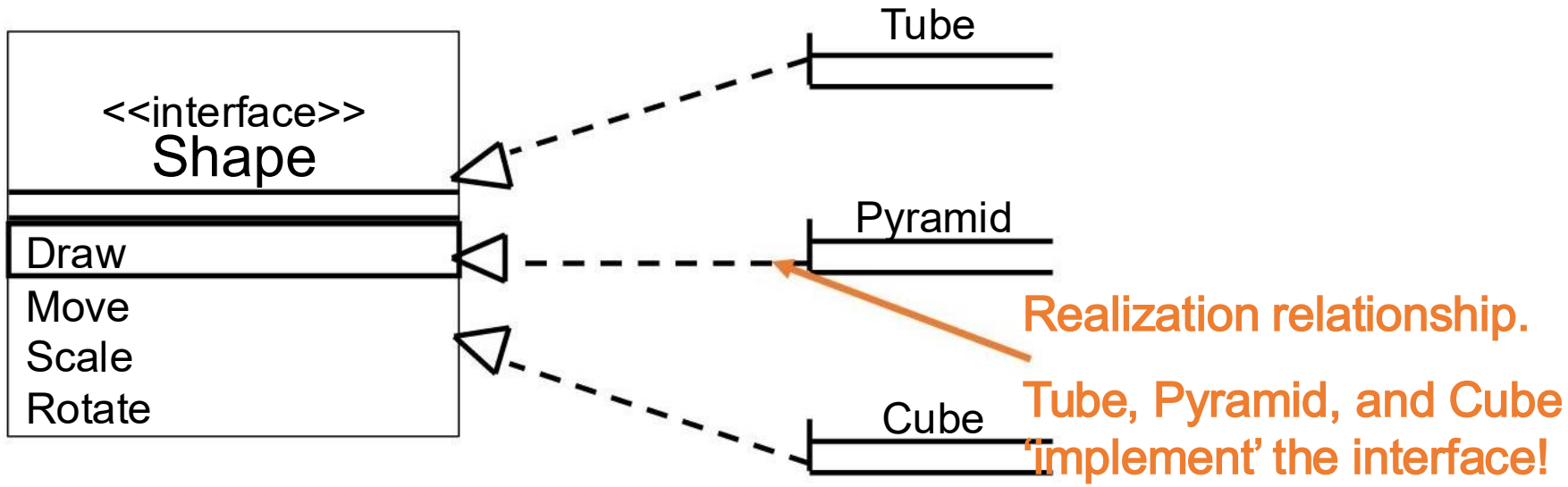
# Implementation details of Interfaces

- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
- A class is made abstract by declaring at least one of its functions as **pure virtual** function.
- A pure virtual function is specified by placing "= 0“ or empty body.

- Question: What is the difference between abstracts and interfaces?

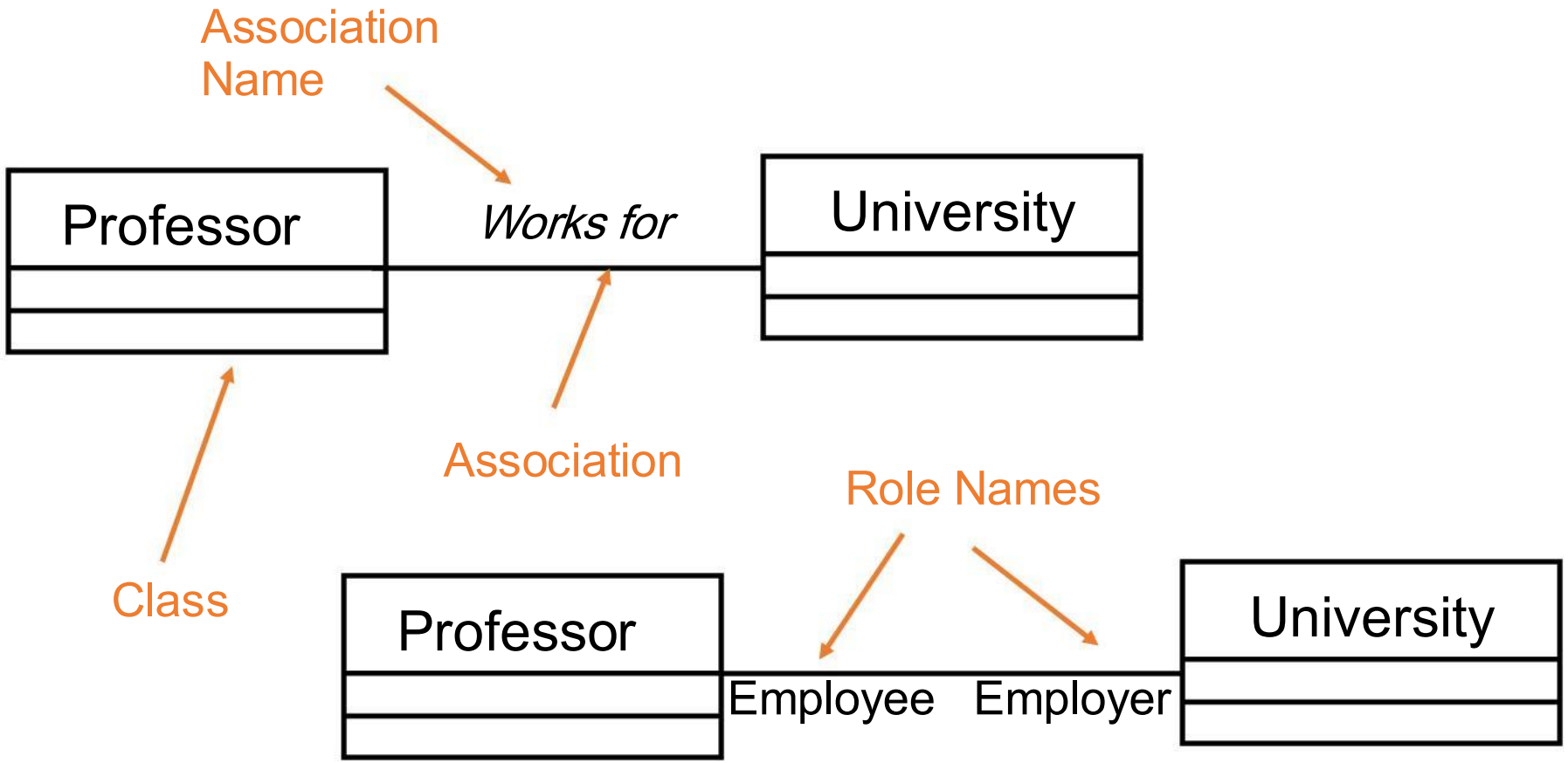
# Interface Representations

Canonical (Class/Stereotype) Representation



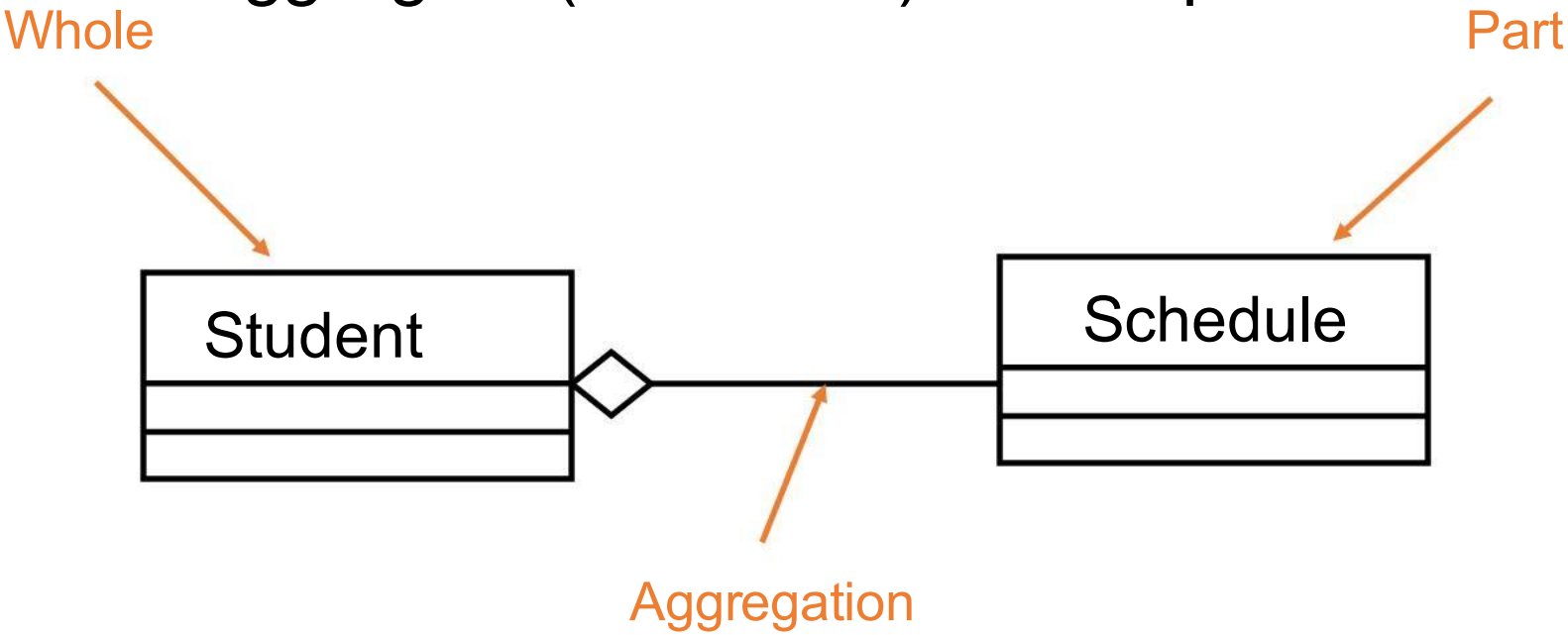
# Relationships: Association

- Models a semantic connection among classes



# Relationships: Aggregation

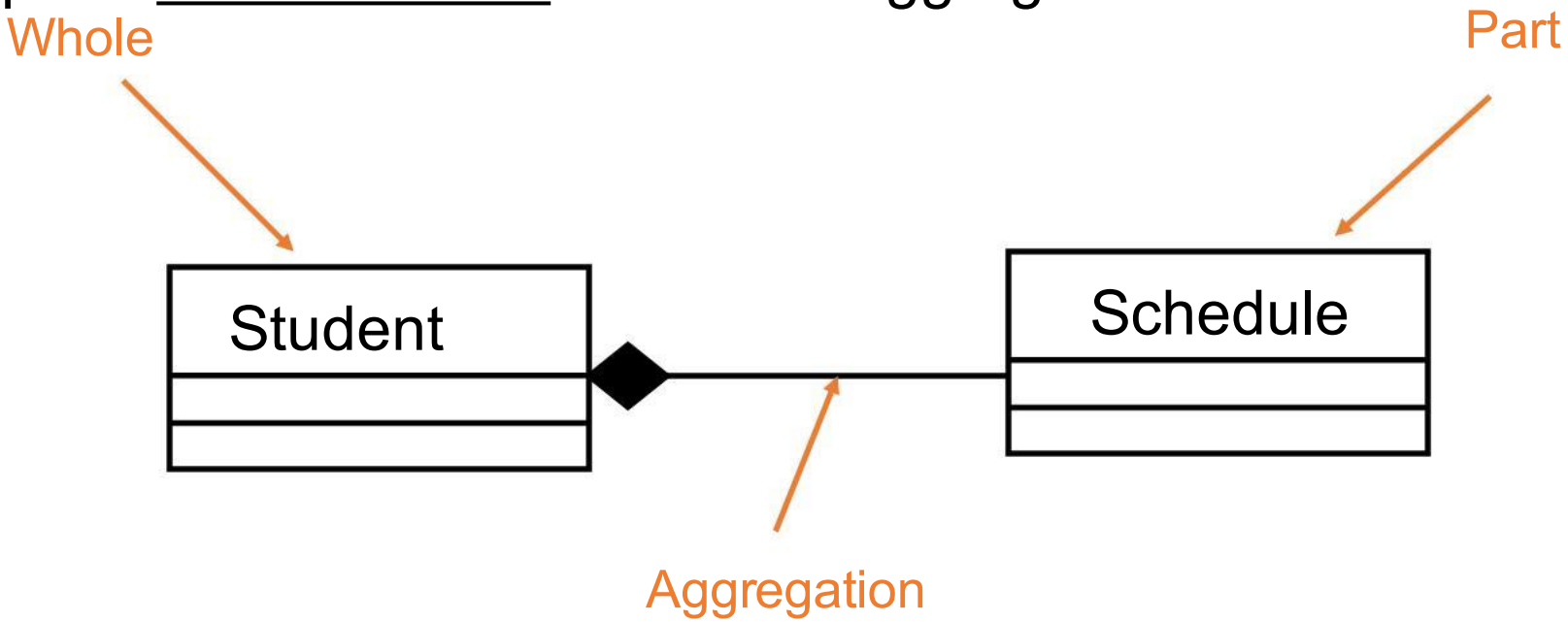
- A special form of **association** that models a whole-part relationship between an aggregate (the whole) and its parts




This is sometimes called a ‘has_a’ relationship

# Relationships: Composition

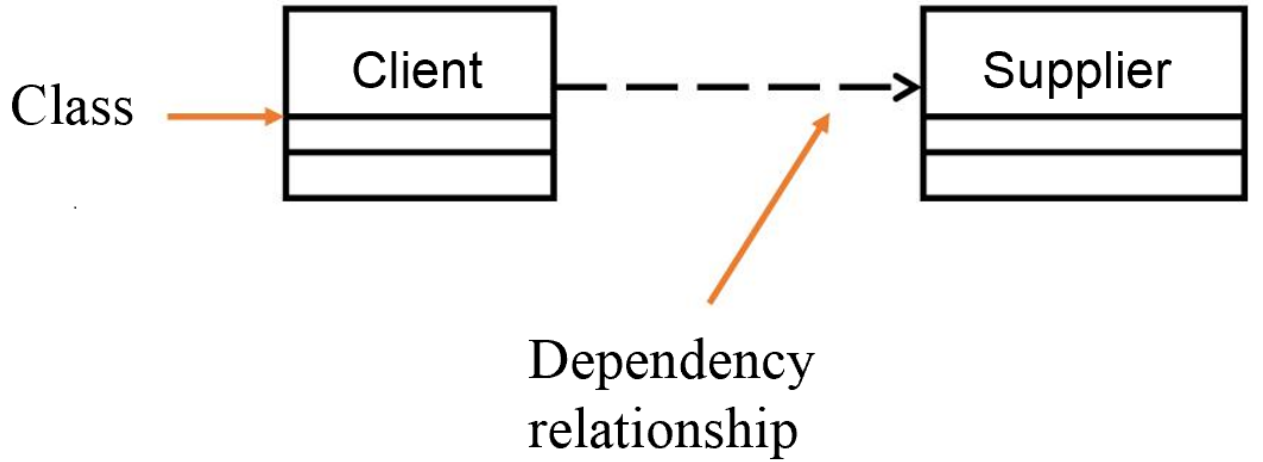
- A form of **aggregation** with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate



# Relationships: Dependency

- A relationship between two model elements where a change in one **may** cause a change in the other
- Non-structural, “using” relationship
- Can actually say the Client ‘uses’ (is dependant on) the Supplier.
- Weaker than association relationship:

**Example:** A method in one class may take an object of another class as an argument, or call a method of another class.



# Relationships: Dependency

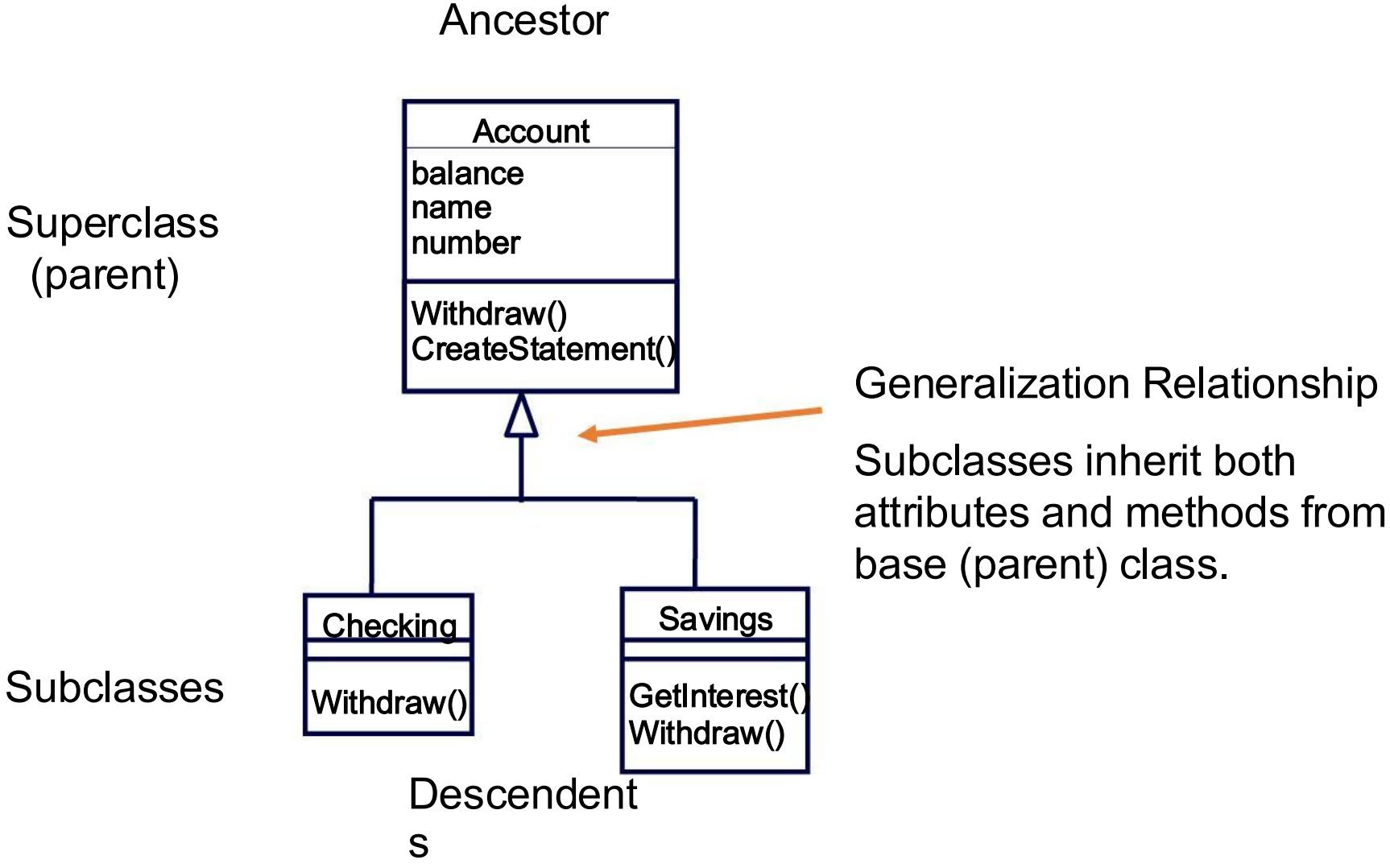
```
// Supplier Class: PaymentGateway
class PaymentGateway {
public:
    void processPayment(double amount) {
        // Logic for processing payment
    }
};

// Client Class: OrderService
class OrderService {
public:
    void submitOrder(double paymentAmount) {
        PaymentGateway paymentGateway; // Using the supplier class
        paymentGateway.processPayment(paymentAmount)
    }
};
```

# Relationships: Generalization

- A relationship among classes where one class shares the structure and/or behavior of one or more classes
- Defines a hierarchy of abstractions in which a subclass **inherits** from one or more superclasses
  - Single inheritance
  - Multiple inheritance
- Generalization is an “is-a-kind of” relationship, or simply, “is_a” relationship.

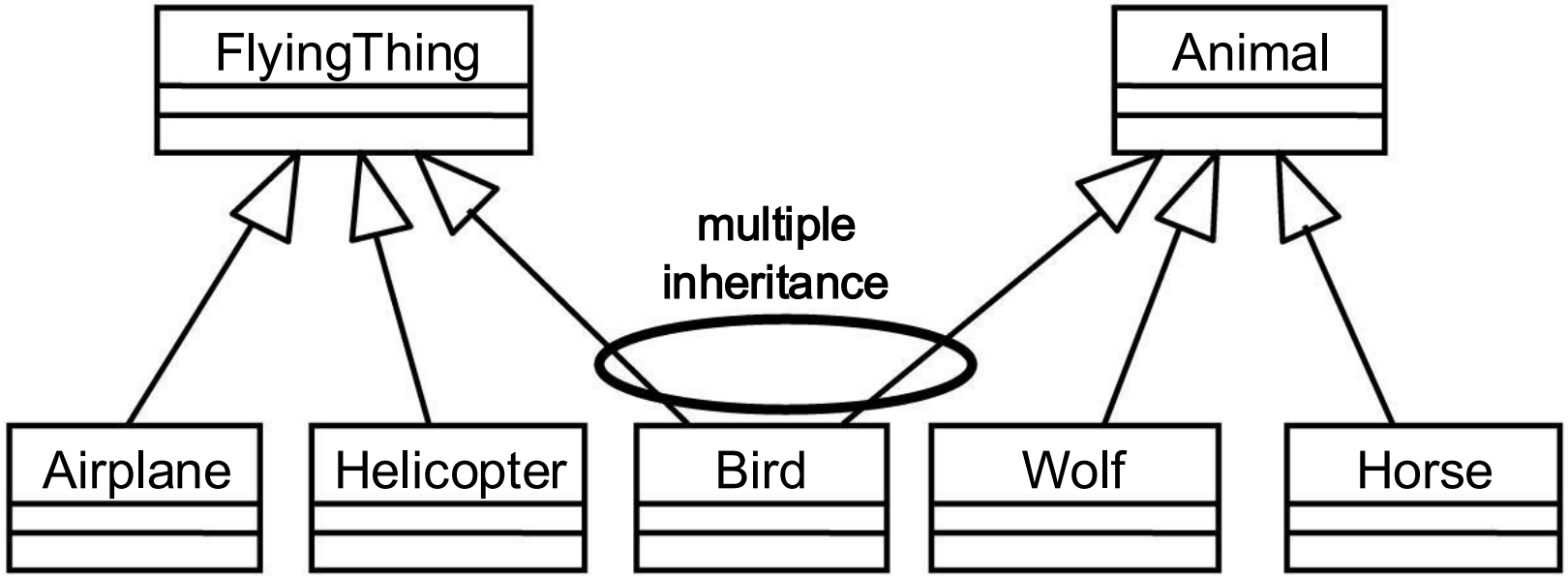
# Example: Single Inheritance

- One class inherits from another



# Example: Multiple Inheritance

- A class can inherit from several other classes




***Use multiple inheritance only when needed, and always with caution !***

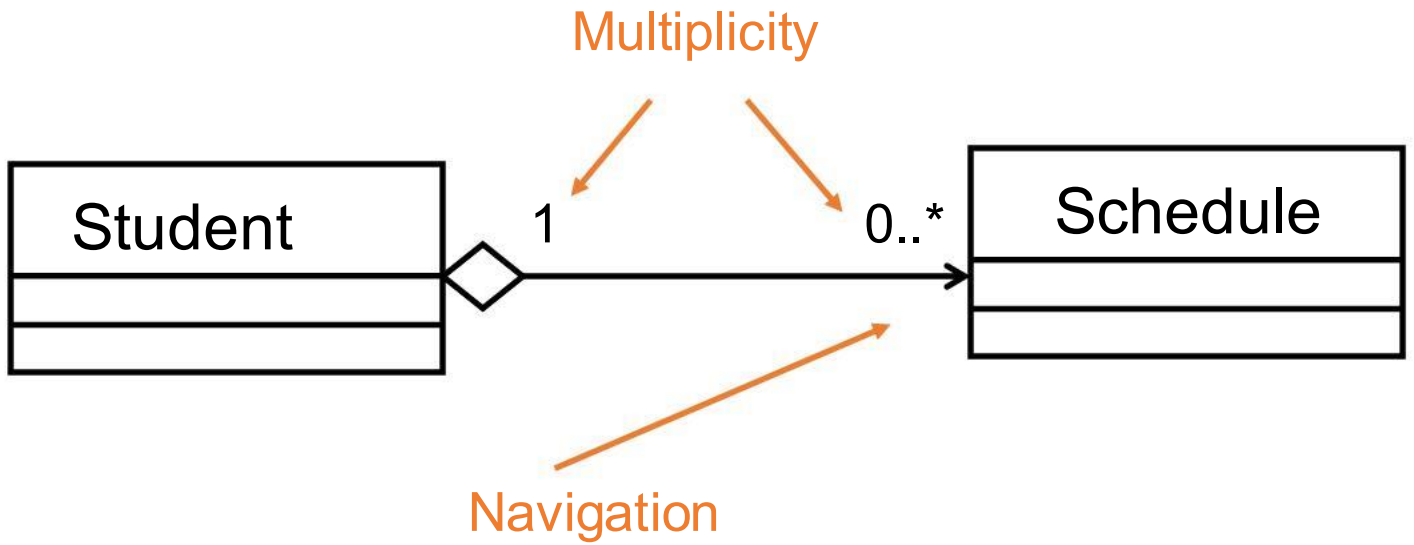
# Association: Multiplicity and Navigation

- Multiplicity defines how many objects participate in a relationship
  - The number of instances (that is, 'objects') of one class related to ONE instance of another class
  - Specified for each end of the association
- Associations and aggregations are bi-directional by default, but it is often desirable to restrict navigation to one direction
  - If navigation is restricted, an arrowhead is added to indicate the direction of the navigation

# Association: Multiplicity

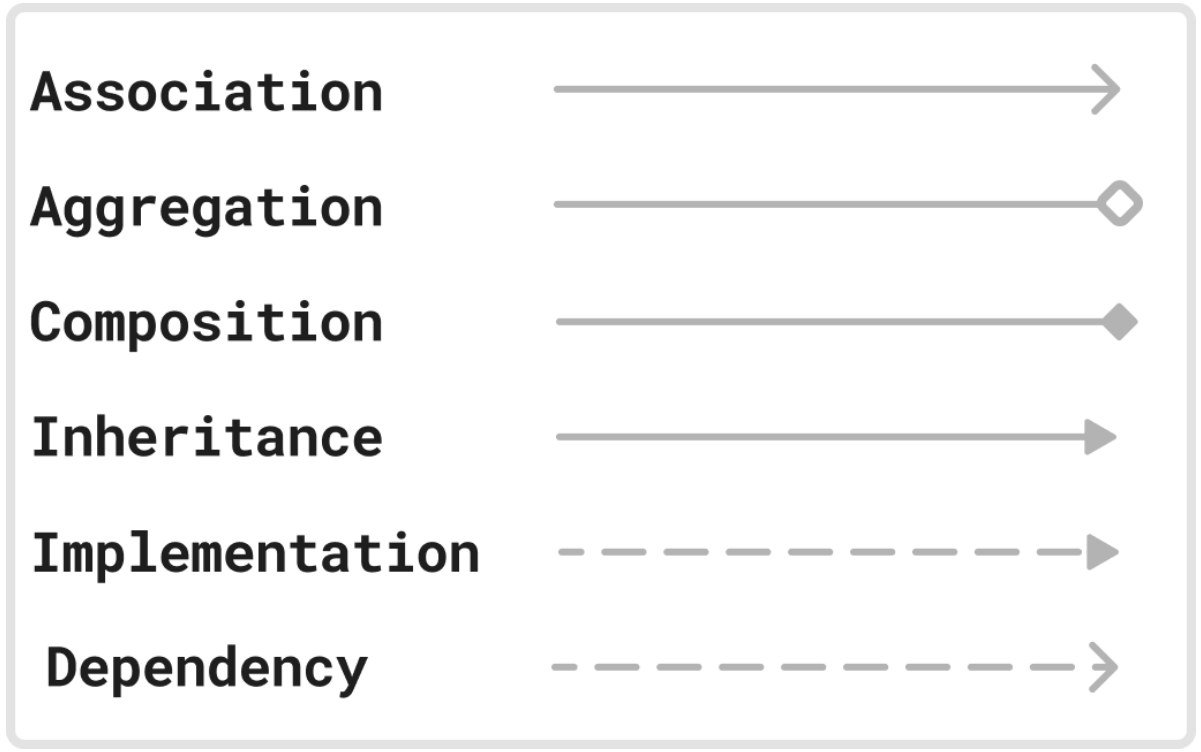
| | |
|---|---|
| • Unspecified | |
| • Exactly one | 1 |
| • Zero or more (many, unlimited) | 0..* |
| | * |
| • One or more | 1..* |
| • Zero or one | 0..1 |
| • Specified range | 2..4 |
| • Multiple, disjoint ranges | 2, 4..6 |

# Example: Multiplicity and Navigation




**A student has zero or more schedules. (Multiplicity)**

# Summary




https://blog.algomaster.io/p/uml-class-diagram-explained-with-examples

# Example

A university library system manages publications (books and journals) with the following notes:

1. Books have ISBN numbers, page counts, and editions.
2. Journals have ISSN numbers, volume numbers, and issue numbers.
3. Both books and journals have titles, authors, and publication dates.
4. Students have names, student IDs, and email addresses.
5. Students can borrow books and journals.
6. Create a UML class diagram to represent this system

# Example

A university library system manages publications (books and journals) with the following notes:

1. Books have ISBN numbers, page counts, and editions.
2. Journals have ISSN numbers, volume numbers, and issue numbers.
3. Both books and journals have titles, authors, and publication dates.
4. Students have names, student IDs, and email addresses.
5. Students can borrow books and journals.
6. Create a UML class diagram to represent this system

HINT 1: What is the relation between books and journals?

Both should inherit from a super class

# Example

A university library system manages publications (books and journals) with the following notes:

1. Books have ISBN numbers, page counts, and editions.
2. Journals have ISSN numbers, volume numbers, and issue numbers.
3. Both books and journals have titles, authors, and publication dates.
4. Students have names, student IDs, and email addresses.
5. Students can borrow books and journals.
6. Create a UML class diagram to represent this system

HINT 2: What is the relation between students & publications?

Association, cardinality??

# Example

A university library system manages publications (books and journals) with the following notes:

1. Books have ISBN numbers, page counts, and editions.
2. Journals have ISSN numbers, volume numbers, and issue numbers.
3. Both books and journals have titles, authors, and publication dates.
4. Students have names, student IDs, and email addresses.
5. Students can borrow books and journals.
6. Create a UML class diagram to represent this system

HINT 3: What is the relation between the library & publications?

Aggregation

# Another Dynamic Diagram: A Sequence Diagram

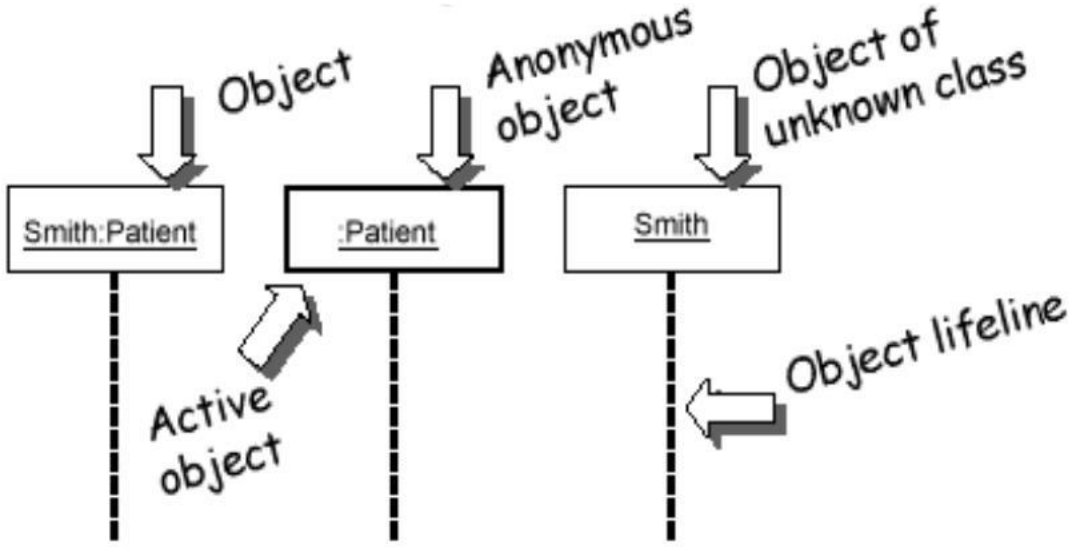
Key parts of a sequence diag.:

- **participant**: an object or entity that acts in the sequence diagram
  - sequence diagram starts with an unattached "found message" arrow
- **message**: communication between participant objects

- the **axes** in a sequence diagram:
  - horizontal: which object/participant is acting
  - vertical: time (down -> forward in time)

# Representing objects

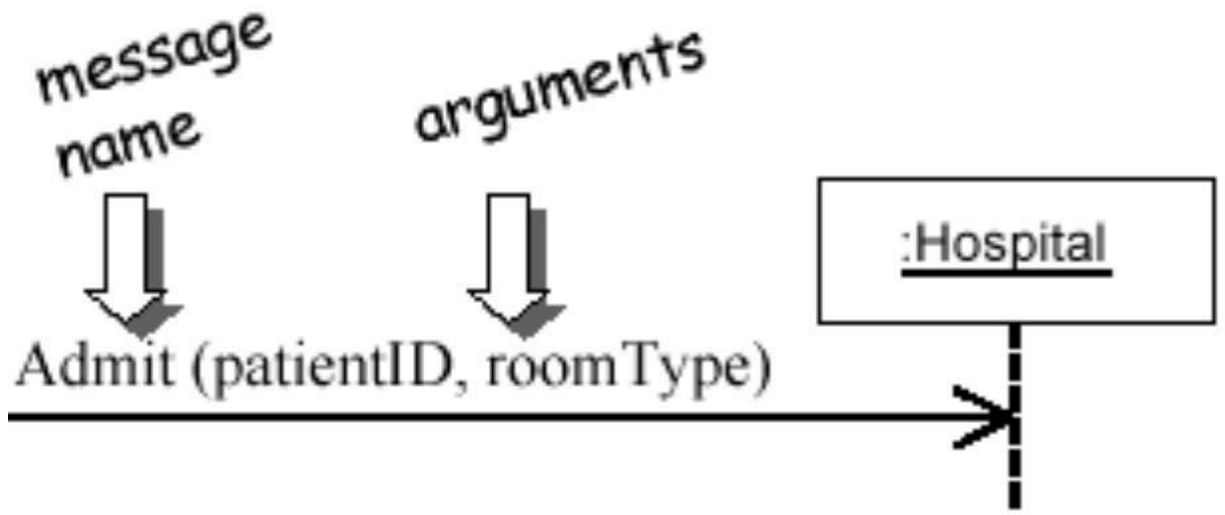
- Squares with object type, optionally preceded by object name and colon
  - write object's name if it clarifies the diagram
  - object's "lifeline" represented by dashed vert. line




**Name syntax:** <objectname>:<classname>

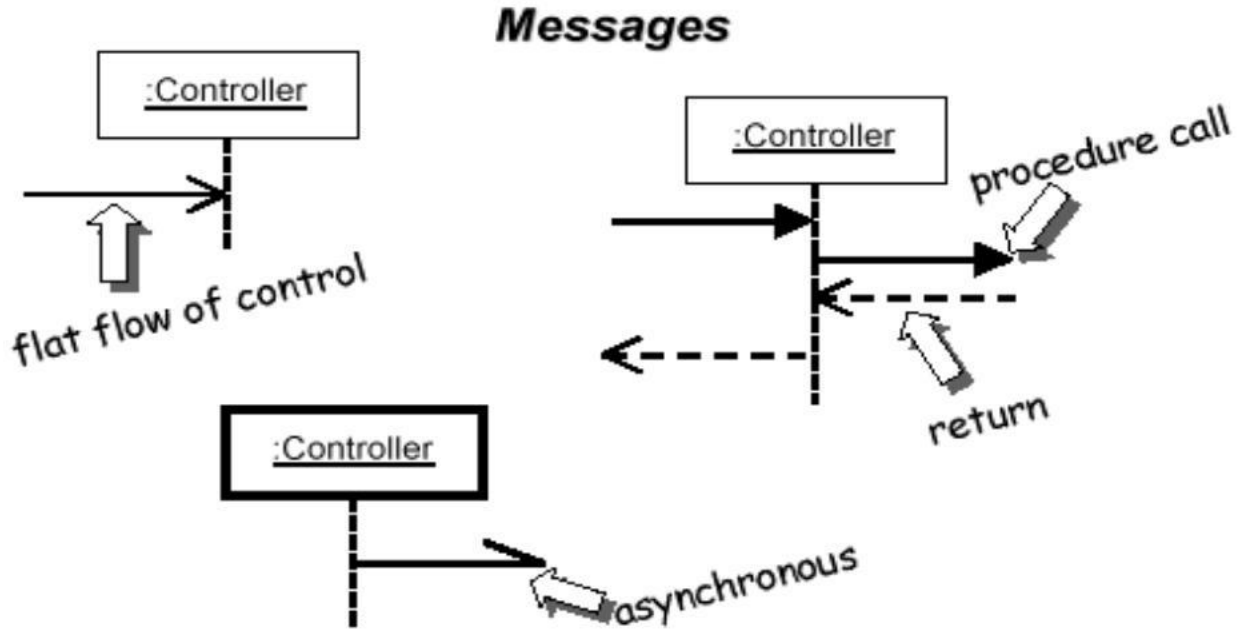
# Messages between objects

- message (method call) indicated by horizontal arrow to other object
  - write message name and arguments above
  arrow



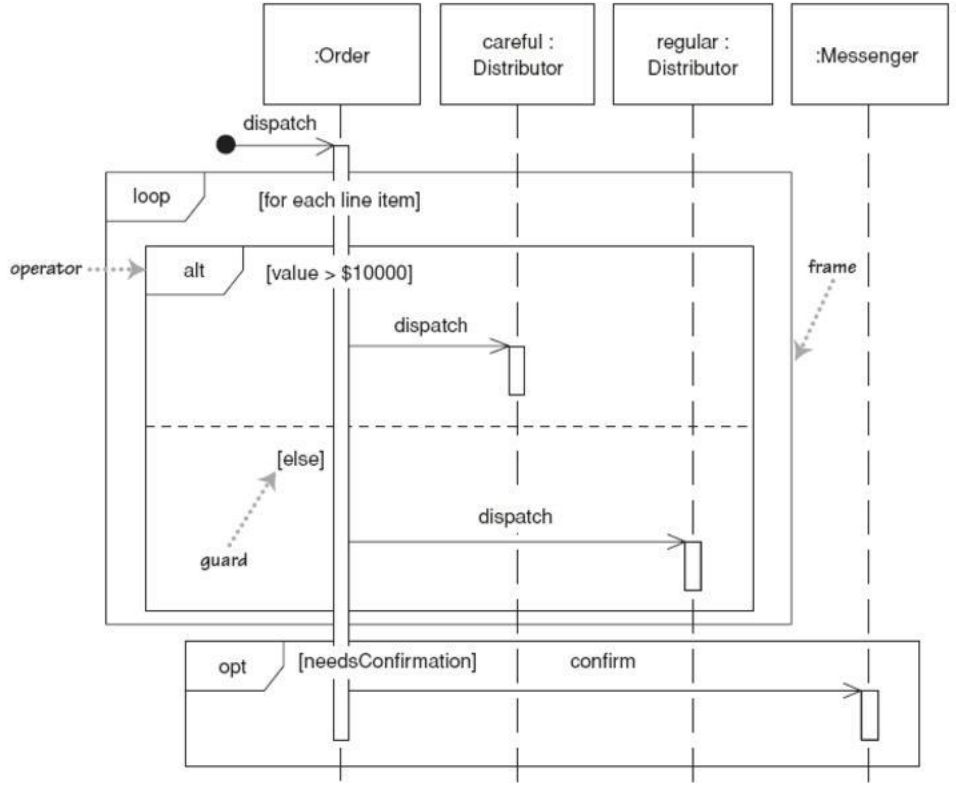
# Messages, continued

- message (method call) indicated by horizontal arrow to other object
  - dashed arrow back indicates return
  - different arrowheads for normal / concurrent (asynchronous) methods



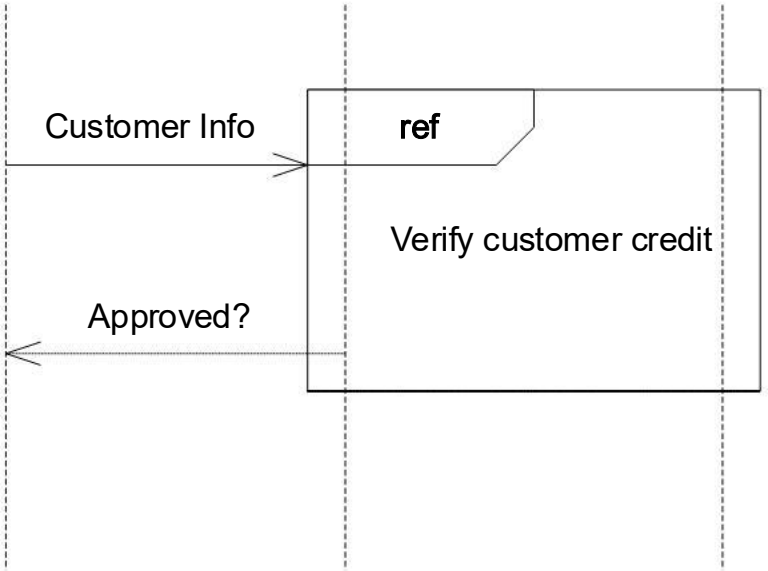
# Indicating selection and loops

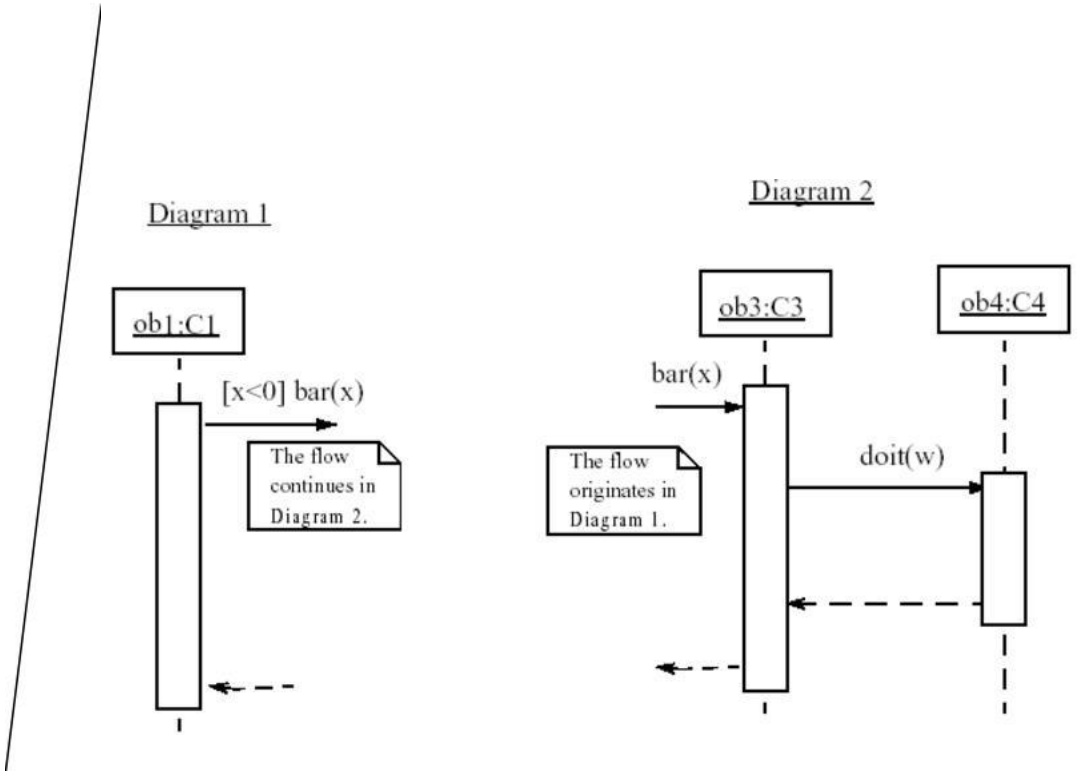
- frame: box around part of a sequence diagram to indicate selection or loop
  - if -> (opt) [condition]
  - if/else -> (alt) [condition], separated by horizontal dashed line
  - loop -> (loop) [condition or items to loop over]



# linking sequence diagrams

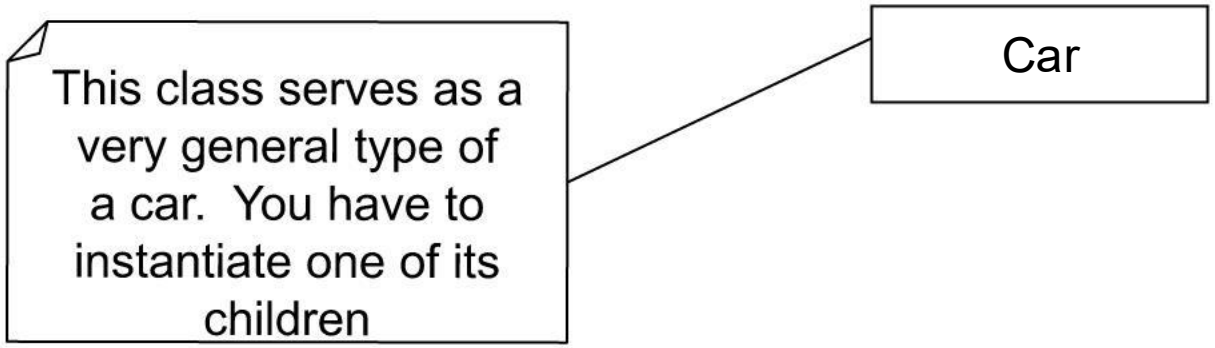
- if one sequence diagram is too large or refers to another diagram, indicate it with either:
  - an unfinished arrow and comment
  - a "ref" frame that names the other diagram



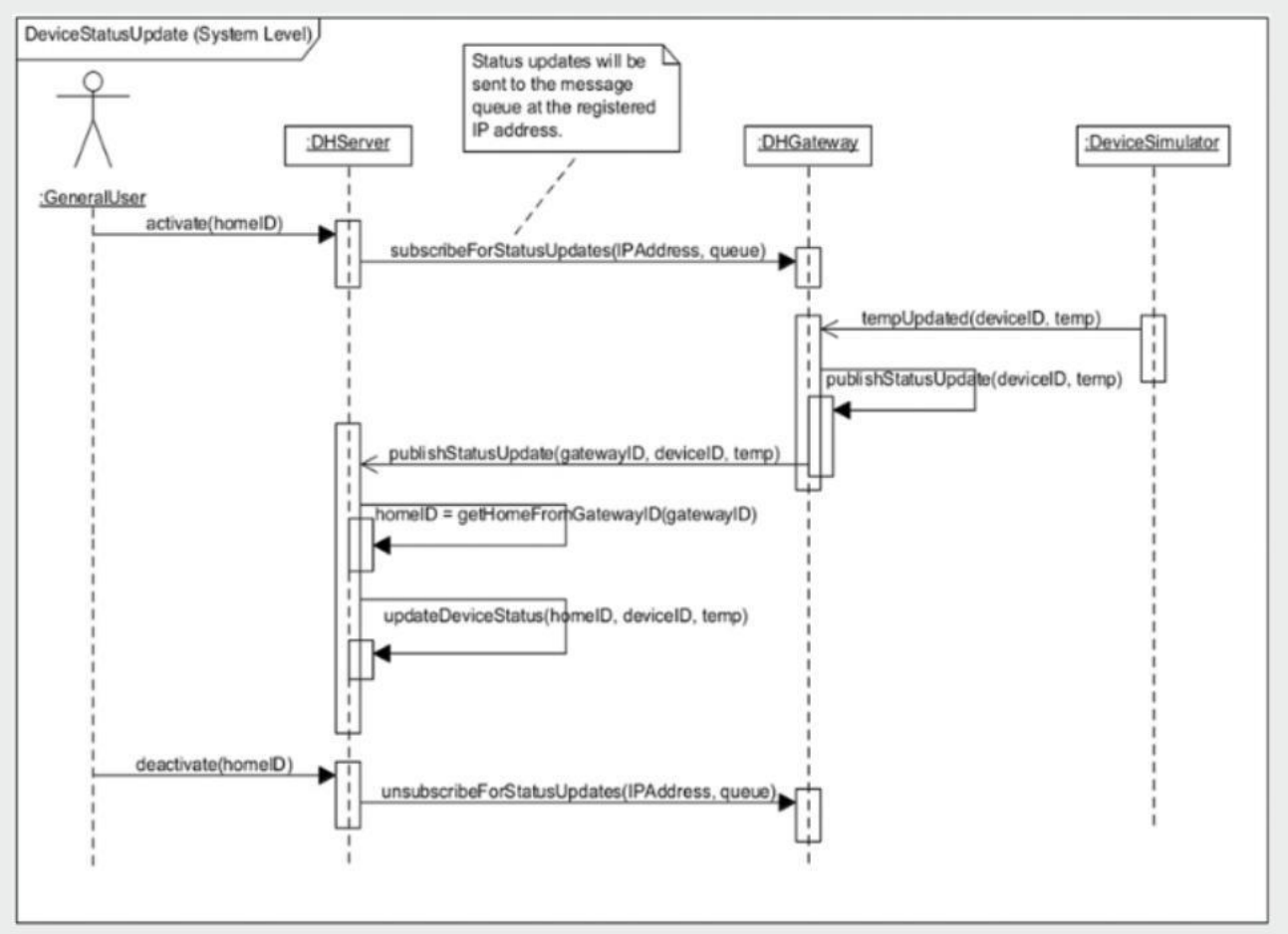
# Annotations

- Annotations, or comments, may be added to any part of any UML diagram
- Notation:



# Example- DH Sequence Diagram of DeviceStatusUpdate Use Case Scenario

- The diagram shows the behavior associated with updating a DH device's status.
- In the diagram, there are four objects; GeneralUser, DHServer, DHGateway, and DeviceSimulator.
- The diagram shows that the GeneralUser object initiates this activity of updating a device status by calling the method activate with a particular home ID.
- This triggers the work of the DHServer object which calls the method subscribeForStatusUpdate.
- This passing of messages and method invocations continues in order from left to right and top to bottom, until the value of the device is updated by the DHServer (by calling undateDeviceStatus).



# Why not just code it?

• Sequence diagrams can be somewhat close to the code level. So why not just code up that algorithm rather than drawing it as a sequence diagram?

- a good sequence diagram is still a bit above the level of the real code (not all code is drawn on diagram)
- sequence diagrams are language-agnostic (can be implemented in many different languages
- non-coders can do sequence diagrams
- easier to do sequence diagrams as a team
- can see many objects/classes at a time on same page (visual bandwidth)

# Sequence diagram exercise

- Let's do a sequence diagram for the following casual use case, *Add Calendar Appointment* :

  The scenario begins when the user chooses to add a new appointment in the UI. The UI notices which part of the calendar is active and pops up an Add Appointment window for that date and time.

  The user enters the necessary information about the appointment's name, location, start and end times. The UI will prevent the user from entering an appointment that has invalid information, such as an empty name or negative duration. The calendar records the new appointment in the user's list of appointments. Any reminder selected by the user is added to the list of reminders.

  If the user already has an appointment at that time, the user is shown a warning message and asked to choose an available time or replace the previous appointment. If the user enters an appointment with the same name and duration as an existing group meeting, the calendar asks the user whether he/she intended to join that group meeting instead. If so, the user is added to that group meeting's list of participants.