SIT232 - OBJECT ORIENTED DEVELOPMENT

Session 7. Unified Modeling Language

Outline

- Session 07. UML
  - Objectives
  - Introduction to UML
  - Use Case Diagrams
  - Class Diagrams
  - State Machine Diagrams
  - Activity Diagrams
  - Interaction Diagrams

SESSION 7. UML

Objectives

- At the end of this session you should:
  - Understand the **developments leading to the creation of the UML** and its significance in modern IT;
  - Be able to **read, understand, and prepare the following UML diagrams**: use case diagrams, class diagrams, state machine diagrams, activity diagrams, and sequence diagrams.

Introduction to UML

- Buildings are **not constructed randomly**, they are *constructed from a plan/blueprint that has been carefully designed*
  - Software is **no different**, you must *carefully analyse a problem and design a solution before writing the code*
  - The task of *programming represents only the translation of a design into code*
    - **Many learning developers make the critical mistake of skipping the analysis/design phase** – don't be one of them!

Introduction to UML

- Object-oriented analysis and design developed substantially in the 1980s, leading to three primary methods:
  - Booch Method by Grady Booch
  - Object Modeling Technique by James Rumbaugh
  - Objectory by Ivar Jacobson
- In the 1990s they all ended up working for the same company by the name – Rational
  - Since purchased by IBM

Introduction to UML

- The authors began merging their different systems into one, technique known as
  - The (Rational) Unified Process
- This required a single graphical notation, for which the authors created the Unified Modeling Language (UML)
  - Currently at Version 2.0
  - Defines graphical notation and an optional language (Object Constraint Language / OCL)
  - Notation is extensible

Use Case Diagrams

- Use **case diagrams** are useful for capturing functional requirements of a system
  - **What** does the *system actually have to do*?
  - **How** should the *system behave given different scenarios*?
- Before we consider use case diagrams we must first consider what a use case is!
  - *Use cases are not covered by UML*
  - Describe *one or more scenarios related to a single goal*

Use Case Diagrams

Withdraw Funds

Level: User Goal

Main Success Scenario:

1. Customer inserts ATM card.
2. Customer enters correct PIN.
3. Customer selects withdrawal option.
4. Customer fills in withdrawal amount.
5. System authorises withdrawal.
6. System dispenses cash.
7. System returns ATM card.

Extensions:

2a: Customer enters incorrect PIN

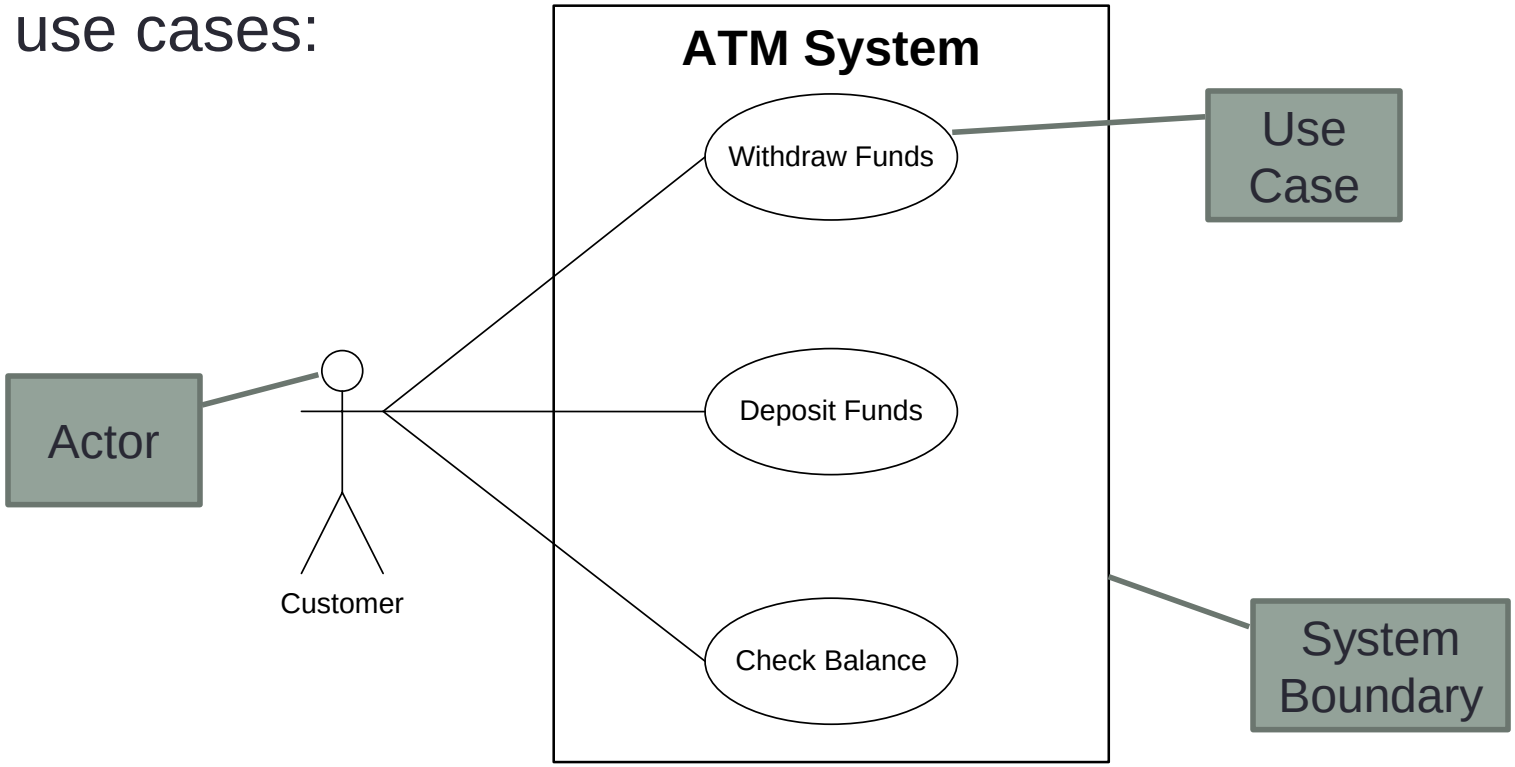
.1: Customer may reenter PIN number

5a: System fails to authorise withdrawal

.1: Customer may reenter withdrawal amount or cancel transaction

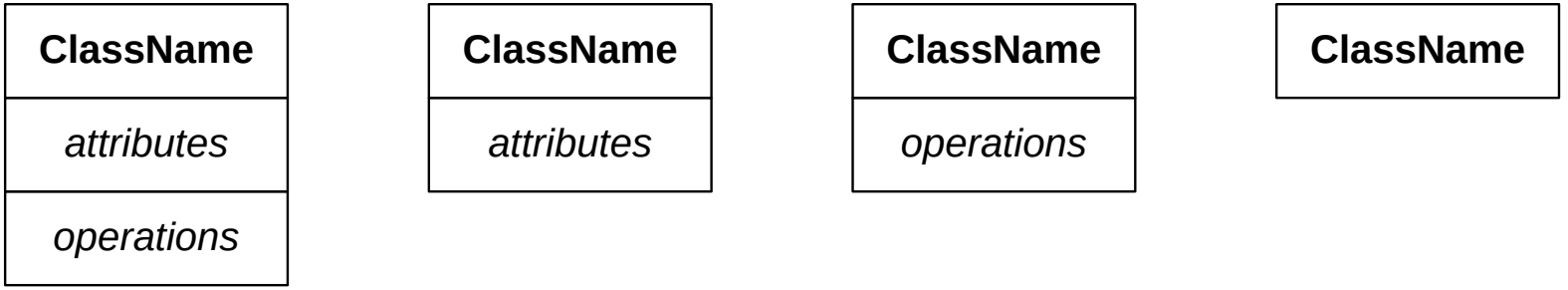
Use Case Diagrams

- The Use Case Diagram is then effectively a contents page for use cases:



Class Diagrams

- Class diagrams are the most common type of diagram
  - Classes are represented in UML by a box, which can have up to three sections:
    - **Class name (required, in bold);**
    - **Attributes (optional); and**
    - **Functions (optional).**



  - For *abstract classes*, also *italicise the class name*

Class Diagrams

- Attributes:

  visibility name : *type multiplicity* = *default_value* {*property*}

  - Static attributes are underlined
  - *visibility* – indicates the visibility (access modifier) of the attribute using one of four symbols:
    - + for public;
    - # for protected;
    - – for private; or
    - ~ for package (such as an assembly).
  - *name* – the name of the attribute
  - *type* – data type for the attribute (simple type or custom type)
  - *multiplicity* – optional, indicating how many instances the attribute refers to (usually one unless referring to a collection)
  - *default_value* – optional, an equals symbol (=) followed by the attribute's default value
  - *property* – optional, surrounded by braces ('{' and '}'), indicates any additional properties about the attribute, e.g., readOnly

Class Diagrams

- Operations:

  visibility name(parameters) : return_type {properties}

  - Parenthesis are mandatory
  - Static operations are underlined
  - Abstract operations are *italicised*
  - *visibility* and *property* are the same as for attributes
  - *name* – the name of the operation
  - *parameters* – optional, the parameters to the operation use a similar syntax to attributes, as follows:

    direction parameter_name : type multiplicity = default_value {property}

    - *direction* represents: input parameters (in), output parameters (out), or reference parameters (inout)
  - *return_type* – indicates the data type for any
    - Blank for ***no return value (void)***

Class Diagrams

- UML does not include any notation for properties, here are possible ways to do them:

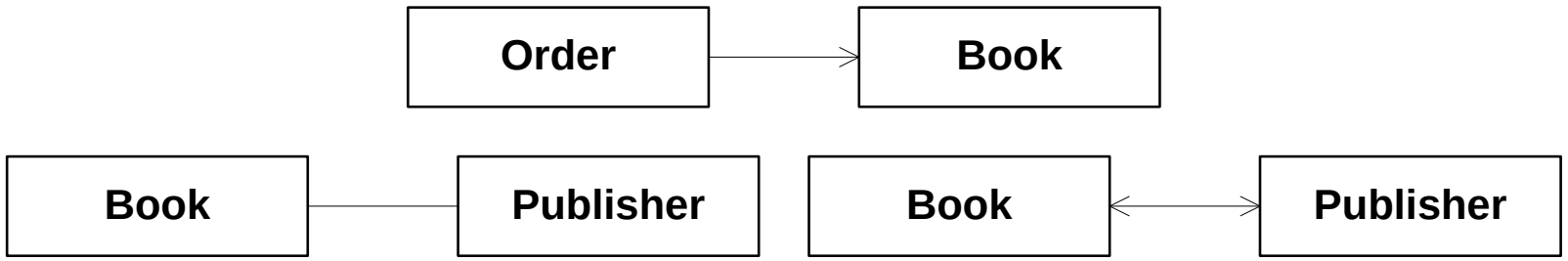
Student
+ ID : string {readOnly} + Name : string {readOnly} + Course : string {readOnly}
+ Student(id: string, name: string, course: string) + ChangeCourse(newCourse: string) + SendFeesInvoice(amount: decimal) + Enrol(unit: string): bool + Withdraw(unit: string): bool

Student
– _ID : string + «property» ID : string {readOnly} – _Name : string + «property» Name : string {readOnly} – _Course : string + «property» Course : string {readOnly}
+ Student(id: string, name: string, course: string) + ChangeCourse(newCourse: string) + SendFeesInvoice(amount: decimal) + Enrol(unit: string): bool + Withdraw(unit: string): bool

Student
– _ID : string – _Name : string – _Course : string
+ Student(id : string, name : string, course : string) + GetID() : string + GetName() : string + GetCourse() : string + ChangeCourse(newCourse : string) + SendFeesInvoice(amount : decimal) + Enrol(unit : string) : bool + Withdraw(unit : string) : bool

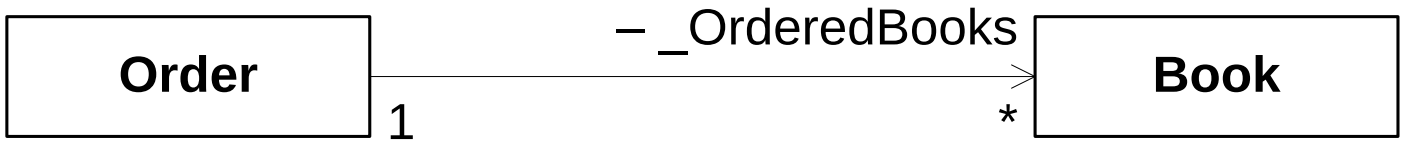
Class Diagrams

- Indicating association:



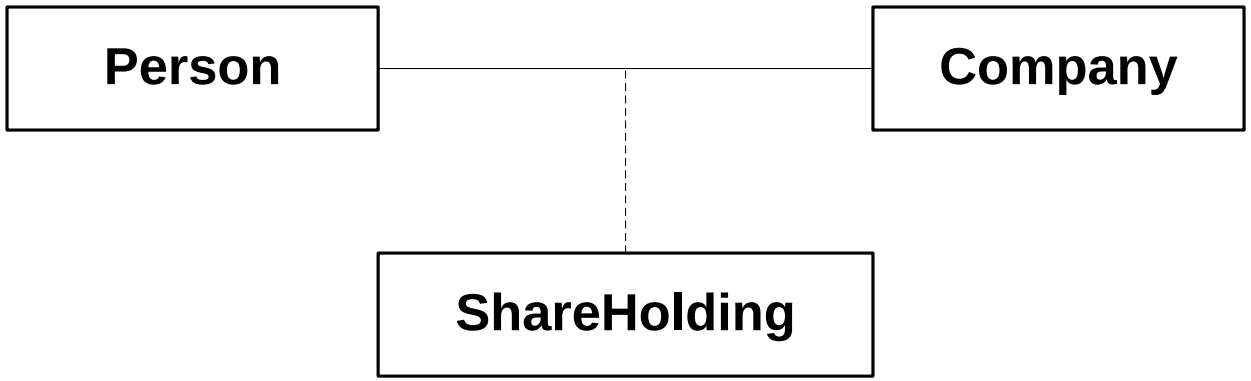
Class Diagrams

- **Adding roles** and multiplicity:
  - A range of values separate by two periods, i.e., *start..finish;*
  - Single values or an asterisk ('*')
    - * means zero or more when shown on its own
  - If same value for *start* and *finish* shown, e.g.,
    - Instead of '1..1', just show '1'; and
  - Separate several different multiplicities with commas.



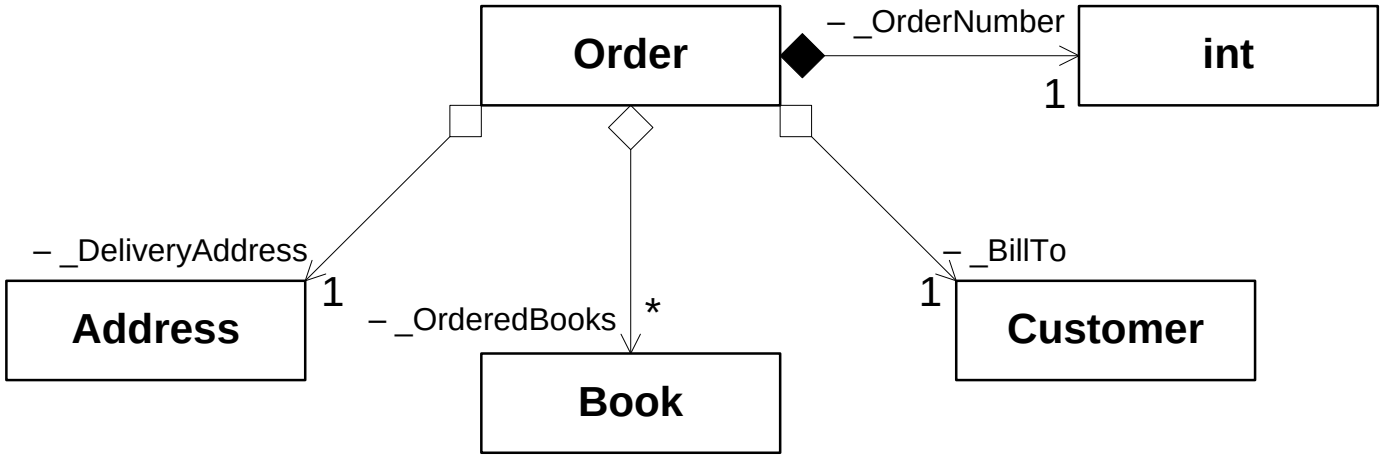
Class Diagrams

- **Association classes**
  - For data that does not fit in either the client object or supplier object
    - Data is relevant to the relationship itself



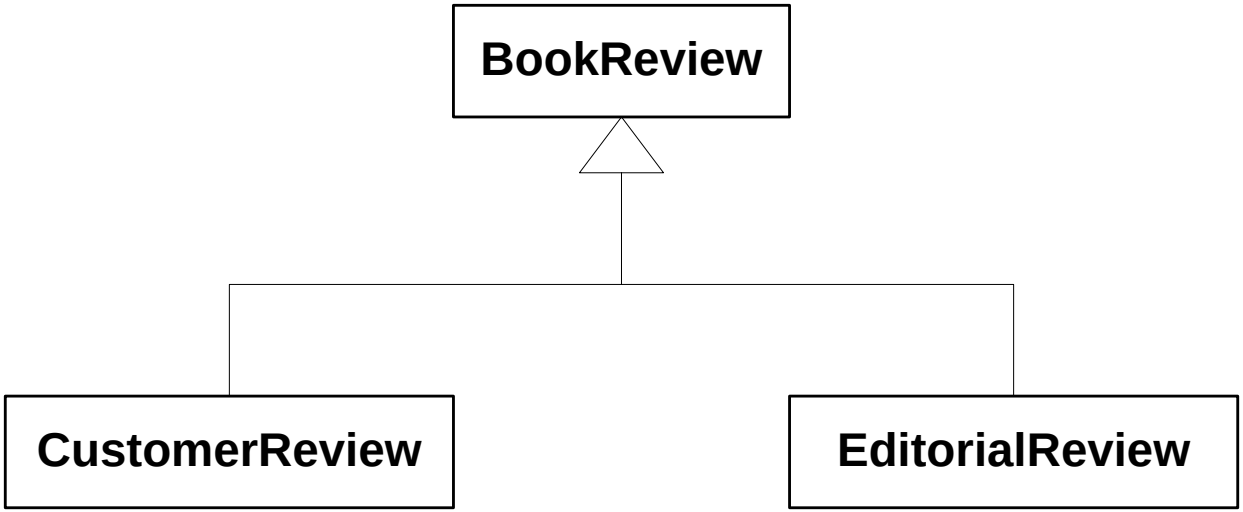
Class Diagrams

- Aggregation
  - Don’t bother, in general use uni-directional association instead
- Composition
  - Only really relevant to value-types for C#
    - These are usually shown as attribute types instead!



Class Diagrams

- Lastly, inheritance
  - Known as generalisation in UML



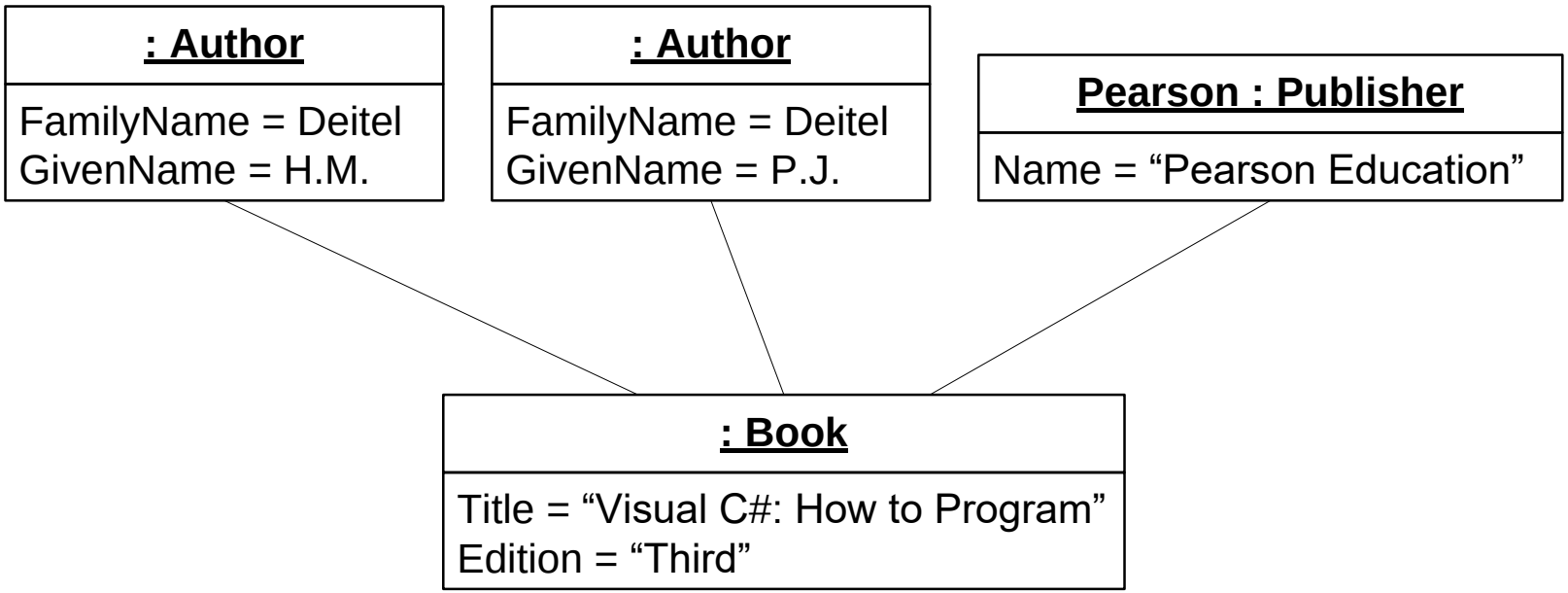
Object Diagrams

- UML Object Diagrams are a sub-type of the UML Class Diagram
  - Often called **Instance Diagrams**
  - *Show a snapshot of an active system at a point in time*:
    - Instances of classes (objects) that exist at that time; and
    - Relationships between those objects at that time

Object Diagrams

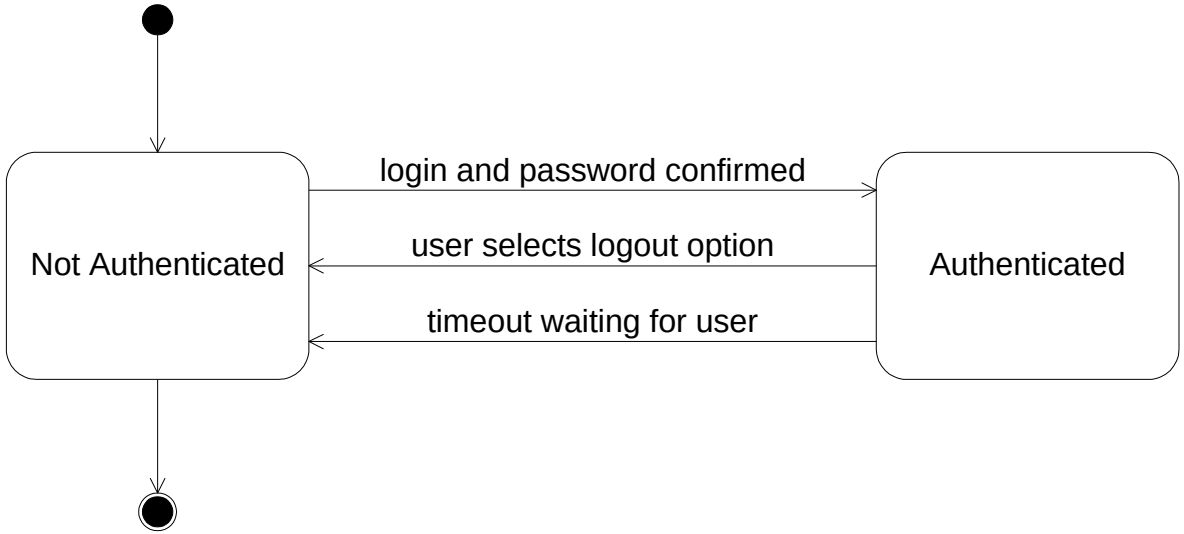
- Notation *slightly from class diagrams* as follows:
  - The class name takes the form '**instance name : class name**', including the underline e.g., John Doe : Person
  - Both the instance name and class name are optional, but the colon should always be shown with the class name, i.e., John Doe: Person, John Doe, or **: Person**;
  - *Actual values are shown for attributes*, rather than classes, and you do not need to show all attributes; and
  - Operations are not shown.

Object Diagrams



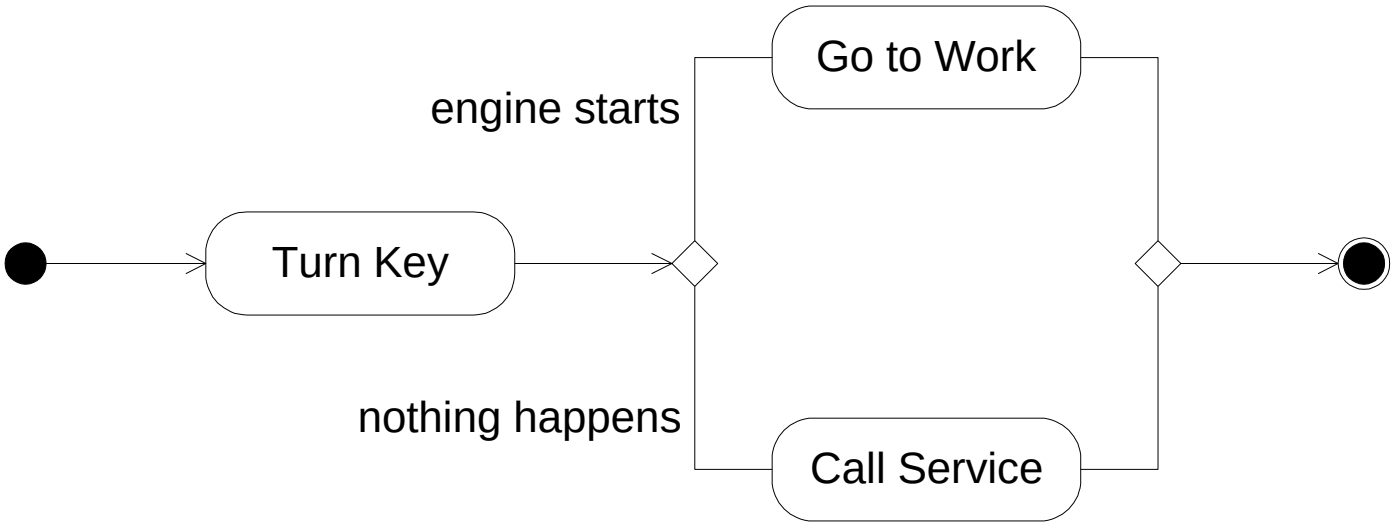
State Machine Diagrams

- Many computer/software systems can be modelled as a state machine
  - Clearly *defined states where some operations are possible* while others are only available in other states
  - Clearly ***indicated transitions which require certain conditions to be met to occur***



Activity Diagrams

- Similar to traditional flow charts
  - ***Illustrate the logic of the system***
  - One activity is *broken down into some number of actions*
    - Each action is roughly the result of one method call



Interaction Diagrams

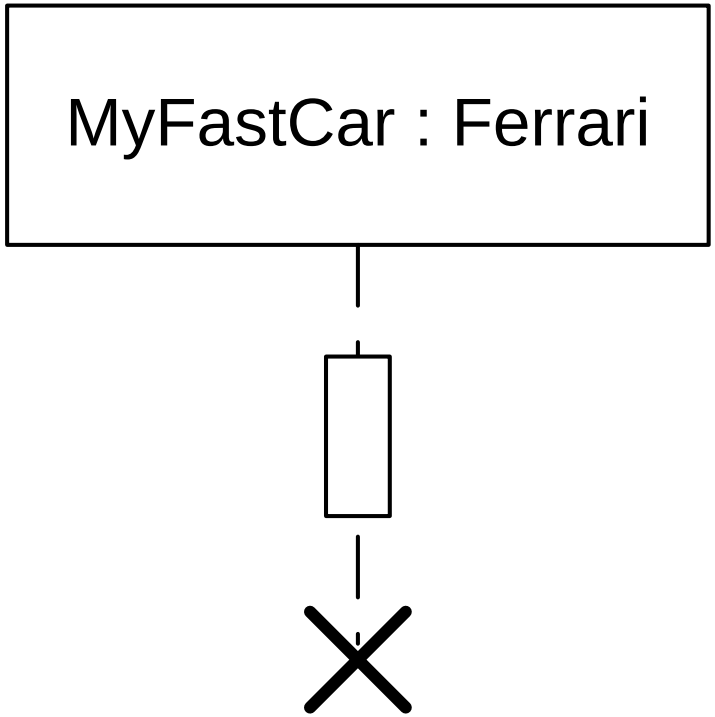
- Interaction diagrams
  - Show the ***exchange of messages between objects to implement some behaviour*** within some context
  - Two alternative diagrams
    - **Sequence diagrams** – *focus on order of events*
    - **Communication diagrams** – *focus on organisation of objects*
      - Usually less expressive
  - We will only consider use sequence diagrams in this unit

Interaction Diagrams

- Why bother?
  - Show us how the system really works
    - If you saw a bicycle for the very first time, would you know what it did? how it works?
  - Can *demonstrate system operation to clients* before coding (during analysis)
  - Can *confirm that the necessary pathways will exist* in the code (during analysis/design)

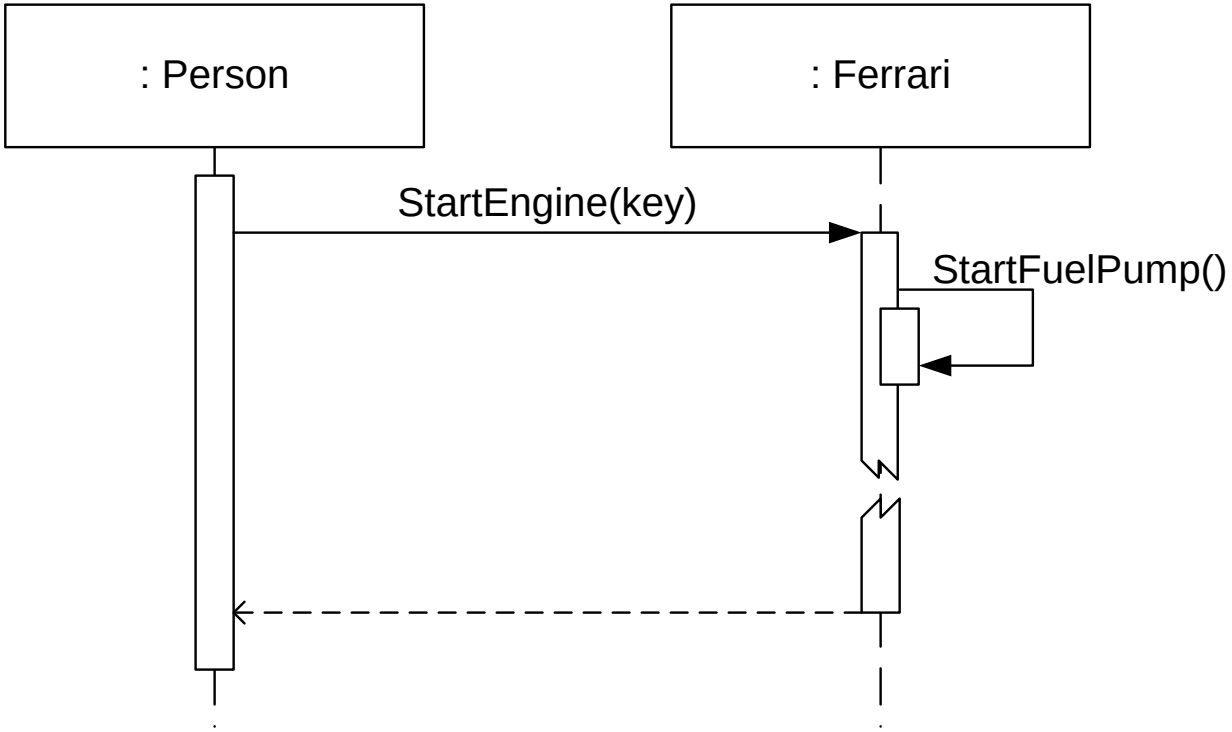
Interaction Diagrams

- Lifelines, activation bars, and destruction:



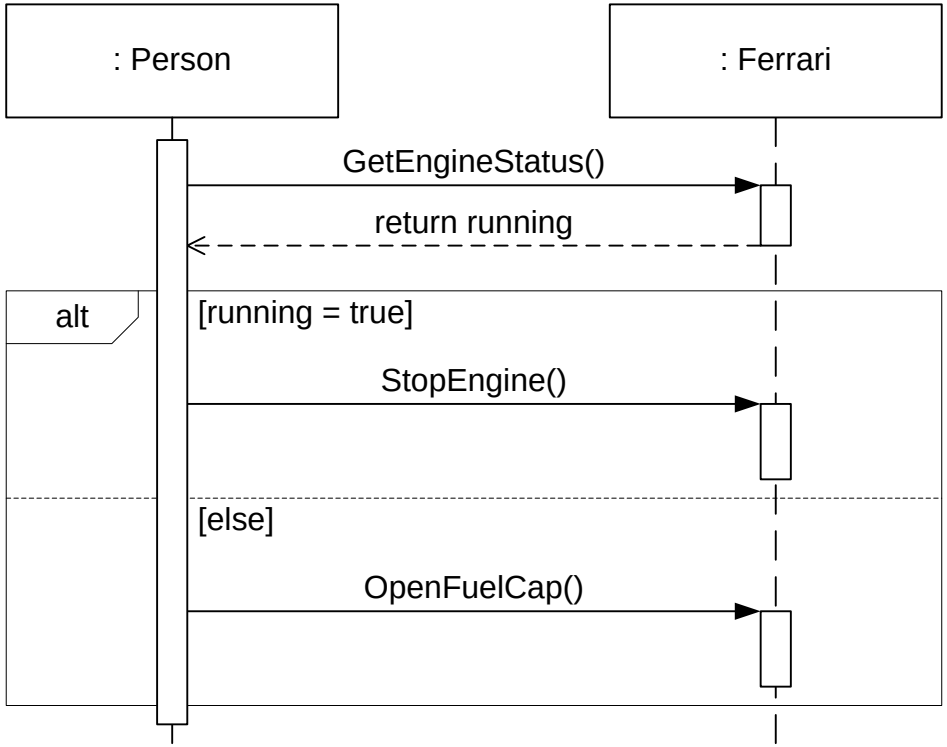
Interaction Diagrams

- Exchanging messages:



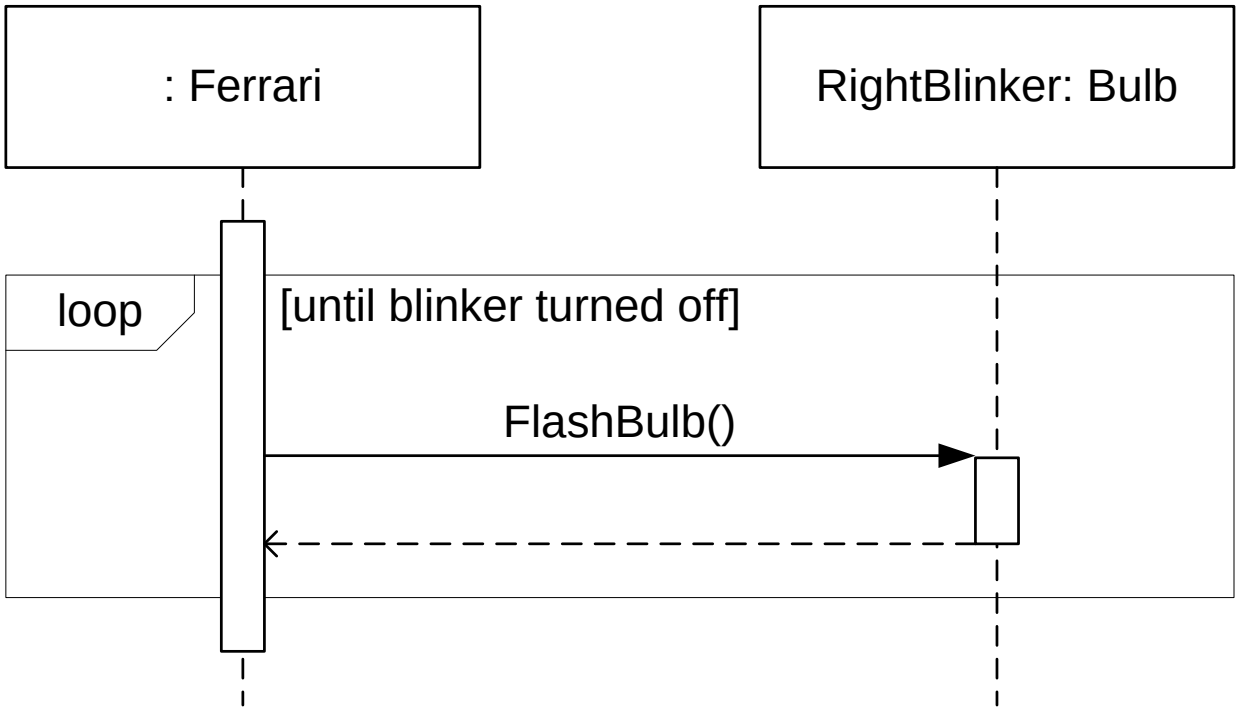
Interaction Diagrams

- Making decisions:



Interaction Diagrams

- Looping:



Summary

- Session 07. UML
  - Objectives
  - Introduction to UML
  - Use Case Diagrams
  - Class Diagrams
  - State Machine Diagrams
  - Activity Diagrams
  - Interaction Diagrams

Summary

- Training Videos:
  - Visio: Class Diagrams in Visio