LECTURE 8 OBJECT-ORIENTED PROGRAMMING FUNDAMENTALS AND DESIGN PHASE

LEK HSIANG HUI

LEARNING OBJECTIVES

At the end of this lecture, you should understand:

- The activities of the design phase
- How to transform analysis models to design models
- Object-Oriented Programming Fundamentals
- How to draw complete class diagrams
- How to draw sequence diagrams

OBJECT-ORIENTED PROGRAMMING FUNDAMENTALS

OBJECT-ORIENTED PROGRAMMING (OOP) RECAP

Major concepts (Covered in IT5001/IT5003)

- Classes and Instances
- Attributes
- Methods
- Constructors
- Inheritance

OBJECT-ORIENTED PROGRAMMING (OOP)

Other concepts

- Dynamic Typing vs Static Typing
- Access Rights
- Static (aka “class-level”) Attributes/Methods
- Method Overloading
- Methods Overriding
- Multiple Constructors

CLASSES

Class:

- Defines a type of object
- Can be thought of as blueprint of a type of object
  - Set of attributes and methods


```
class Cat:
    def __init__(self, name):
        self.name = name

    def makeSound(self):
        print("meow")
```

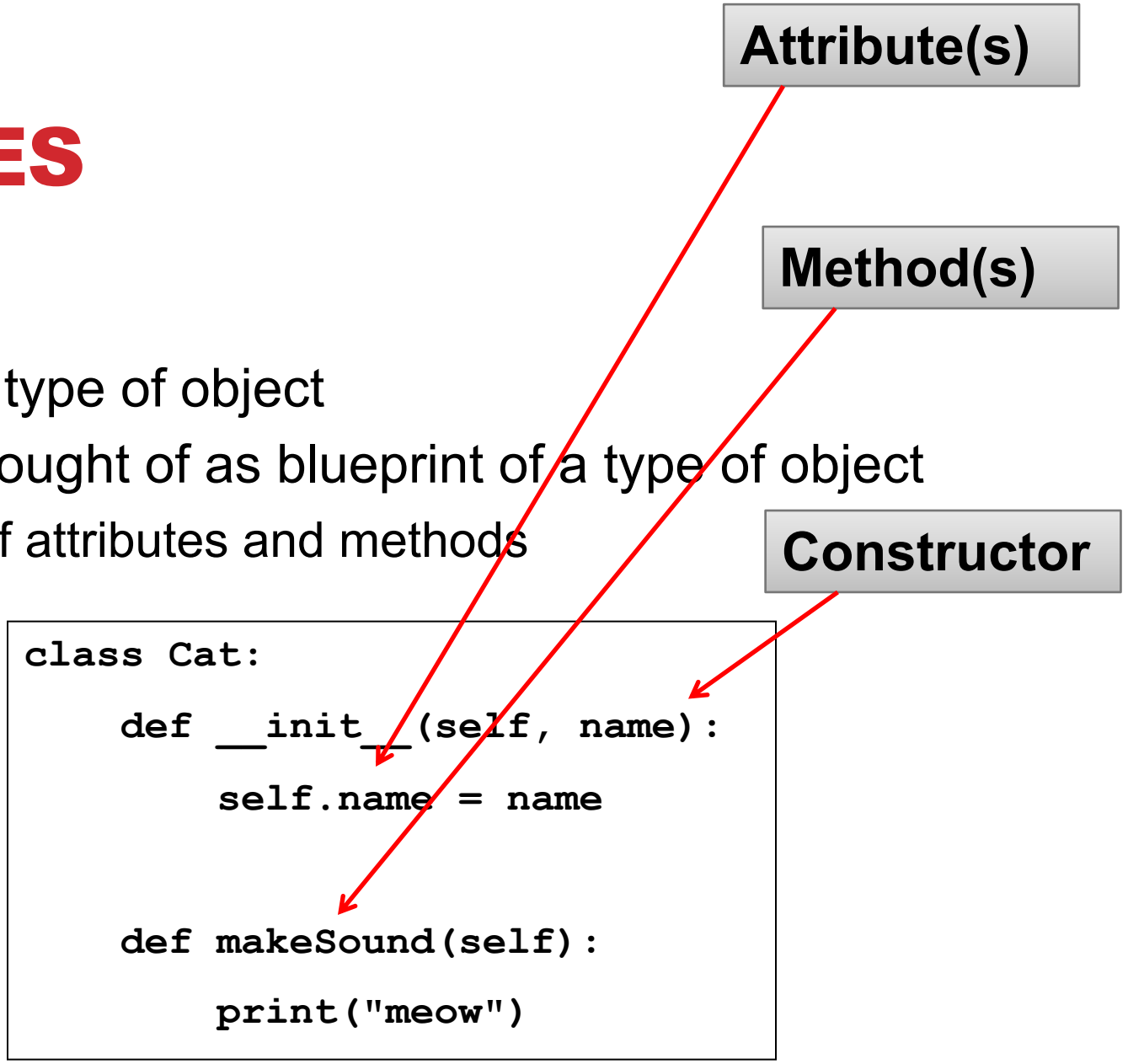
```
#instantiate a new cat
cat1 = Cat("Tom")
cat1.makeSound()

#instantiate a new cat
cat2 = Cat("Puss")
cat2.makeSound()
```

CLASSES

Class:

- Defines a type of object
- Can be thought of as blueprint of a type of object
  - Set of attributes and methods



PYTHON VS JAVA

STATIC/DYNAMIC -TYPING

Static-typing (Java)
i.e. need to define before hand
Python is **Dynamic-typing**

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

```
#instantiate a new cat
Cat cat1 = new Cat("Tom")
cat1.makeSound()

#instantiate a new cat
Cat cat2 = new Cat("Puss")
cat2.makeSound()
```

STATIC-TYPING

When to specify types?

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

```
#instantiate a new cat
Cat cat1 = new Cat("Tom")
cat1.makeSound()

#instantiate a new cat
Cat cat2 = new Cat("Puss")
cat2.makeSound()
```

STATIC-TYPING

When to specify types?

When you declare variables

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

```
#instantiate a new cat
Cat cat1 = new Cat("Tom")
cat1.makeSound()
```

When to specify types?

When you declare attributes

STATIC-TYPING

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }

  public String getName(){
    return name;
  }
}
```

When to specify types?

Return type of methods

When to specify types?

void methods do not return anything

STATIC-TYPING

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }

  public String getName(){
    return name;
  }
}
```

When to specify types?

Method parameters
(one for each parameter)

"NEW" KEYWORD

new keyword
Used to instantiate objects
A better idea IMO (less confusion whether it is calling a function or instantiating an object)

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

```
#instantiate a new cat
Cat cat1 = new Cat("Tom")
cat1.makeSound()

#instantiate a new cat
Cat cat2 = new Cat("Puss")
cat2.makeSound()
```

VERBOSE?

```
class Cat:
    def __init__(self, name):
        self.name = name

    def makeSound(self):
        print("meow")
```

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

Python is **Less Verbose**

VERBOSE?

```
class Cat:
    def __init__(self, name):
        self.name = name

    def makeSound(self):
        print("meow")
```

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

But…
Need to define **self** for all the methods/constructor

ATTRIBUTES

```
class Cat:
    def __init__(self, name):
        self.name = name
    ...

cat1 = Cat("Tom")
cat1.weight = 2
```

Fields can be defined “on the fly”

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

Fields needs to be declared before using

METHOD OVERLOADING (PYTHON)

```
s = Shape()
```

```
class Shape:
    def compute(self, length, width = None, height = None):
        """
        Compute either the length or area or volume
        """
        if width is None:
            return length
        elif height is None:
            return length * width
        else:
            return length * width * height
```

```
print(s.compute(2))
```

```
2
```

```
print(s.compute(2, 2))
```

```
4
```

```
print(s.compute(2, 2, 3))
```

```
12
```

METHOD OVERLOADING (PYTHON)

```
class Shape:
    def compute(self, length, width = None, height = None):
        """
        Compute either the length or area or volume
        """
        if width is None:
            return length
        elif height is None:
            return length * width
        else:
            return length * width * height
```

Probably make more sense to instantiate different objects and do compute by running **obj.compute()** instead!
(will talk about this shortly)

METHOD OVERLOADING (JAVA)

```
class Shape {
  public int compute(int length) {
    return length;
  }

  public int compute(int length, int width) {
    return length * width;
  }

  public int compute(int length, int width, int height) {
    return length * width * height;
  }
}
```

Multiple method definitions

CONSTRUCTORS

Constructors can be thought off as special “methods” for constructing objects

```
class Cat:
    def __init__(self, name):
        self.name = name
    ...

cat1 = Cat("Tom")
cat1.weight = 2
```

The constructor is defined using __init__()

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

Constructors* are defined using the same name of the class

MULTIPLE CONSTRUCTORS (JAVA)

```
class Shape2 {
  private int length = 0, width = 0, height = 0;

  public Shape2(int length) {
    this.length = length;
  }

  public Shape2(int length, int width) {
    this.length = length;
    this.width = width;
  }

  public Shape2(int length, int width, int height) {
    this.length = length;
    this.width = width;
    this.height = height;
  }
  …
}
```

Same idea as methods except now for constructors

MULTIPLE CONSTRUCTORS (JAVA)

```
class Shape2 {
  …
  public int compute() {
    if (width > 0) {
      if (height > 0) {
        return length * width * height;
      } else {
        return length * width;
      }
    } else {
      return length;
    }
  }
}
```

"MULTIPLE" CONSTRUCTORS APPROACH FOR PYTHON

```
s2 = Shape2(2)
s3 = Shape2(2, 2)
s4 = Shape2(2, 2, 3)
```

```
class Shape2:
    def __init__(self, length, width = None, height = None):
        self.length = length
        self.width = width
        self.height = height

    def compute(self):
        #same implementation as Shape
        ...
```

```
print(s2.compute())
```

2

```
print(s3.compute())
```

4

```
print(s4.compute())
```

12

INHERITANCE (PYTHON)

```
class Parent:
    def __init__(self, name):
        self.name = name

class Child1(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age
```

Child inherits all the attributes and methods from parent

```
c = Child1("John", 20)
print(c.name)
```
John
```
print(c.age)
```
20

Can define it within ()

Call parent class constructor using **super().__init__()**

INHERITANCE (JAVA)

```
class Parent{
  private String name;
  public Parent(String name){
    this.name = name;
  }
}

class Child1 extends Parent {
  private int age;
  public Child1(String name, int age) {
    super(name);
    this.age = age;
  }
}
```

Slightly different syntax
More verbose but cleaner IMO

INHERITANCE (PYTHON)

```
class Parent:
    …

class AnotherParent:
    …

class child2(Parent, AnotherParent):
    …
```

Possible to do multiple inheritance in Python

Multiple inheritance not allowed in Java

METHOD OVERRIDING (PYTHON)

```
class Parent:
    def foo(self):
        print("from parent")

    def bar(self):
        print("from parent")

class Child1(Parent):
    def foo(self):
        print("from child")

    def bar(self):
        super().bar()
```

Child's method by default will override parent's method

```
c = Child1()
c.foo()
```
from child

```
c.bar()
```
from parent

METHOD OVERRIDING (JAVA)

```
class Parent{
  public void foo(){
    System.out.println("from parent");
  }
  public void bar(){
    System.out.println("from parent");
  }
}
class Child1 extends Parent {
  public void foo(){
    System.out.println("from child");
  }
  public void bar(){
    super.bar();
  }
}
```

Same idea but slightly different syntax

STATIC (CLASS-LEVEL) ATTRIBUTES VS INSTANCE ATTRIBUTES

```
class SomeClass:
    field1 = 10

    def __init__(self, field2):
        self.field2 = field2
```

static attributes are attributes on the class-level (i.e. can be thought of as attributes belonging to a class)

instance attributes are on the instance-level(i.e. each individual instance would have its own copy of that attribute)

STATIC (CLASS-LEVEL) ATTRIBUTES VS INSTANCE ATTRIBUTES

```
class SomeClass:
    field1 = 10

    def __init__(self, field2):
        self.field2 = field2
```

```
print(SomeClass.field1)
10
sc1 = SomeClass(11)
print(sc1.field1)
10
print(sc1.field2)
11


sc2 = SomeClass(22)
print(sc2.field1)
10
print(sc2.field2)
22
```

static attributes

Instance attributes

STATIC (CLASS-LEVEL) ATTRIBUTES

```
class SomeClass:
    field1 = 10

    def __init__(self, field2):
        self.field2 = field2
```

```
SomeClass.field1 = 20
print(sc1.field1)
20

sc2 = SomeClass(22)
print(sc2.field1)
20
```

static attributes are attributes on the class-level (i.e. can be thought of as attributes belonging to a class)

STATIC (CLASS-LEVEL) ATTRIBUTES

```
class SomeClass:
    field1 = 10

    def __init__(self, field2):
        self.field2 = field2
```

```
SomeClass.field1 = 20
sc1 = SomeClass(11)
sc1.field1 = 30
print(SomeClass.field1)
```
20
(not 30)

```
sc2 = SomeClass(22)
print(sc2.field1)
```
20
(not 30)

Gotcha in Python's implementation of **static** attributes
(by right there should only be one copy of `field1` since it is on class-level)

STATIC (CLASS-LEVEL) ATTRIBUTES (JAVA)

```
class SomeClass{

  public static int field1 = 10;

}
```

Java implementation of **static** attributes is more correct

```
System.out.println(SomeClass.field1);
10

SomeClass sc1 = new SomeClass();
System.out.println(sc1.field1);
10

sc1.field1 = 20;
System.out.println(SomeClass.field1);
20

SomeClass.field1 = 30;
System.out.println(sc1.field1);
30
```

STATIC (CLASS-LEVEL) METHODS

```
class SomeClass:
    field1 = 10

    def foo():
        print("foo")
```

static methods in python:

these methods does not have the usual **self** argument parameter

STATIC (CLASS-LEVEL) METHODS (JAVA)

static methods in Java is again much clearer

```
class SomeClass{
  public static int field1 = 10;

  public static void foo() {
    System.out.println("foo " + field1);
  }
}
```

Note that static methods can only access static attribute (or use parameters supplied by the caller)

It can't access instance attributes

ACCESS RIGHTS (JAVA)

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

Most OOP languages also come with different types of access rights:

- **private**
- **public**
- **protected**

ACCESS RIGHTS (JAVA)

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

private

Not accessible outside of this class (only allowed within the class)

I.e.

```
Cat cat1 = new Cat("Tom")
```

this is not allowed!

```
cat1.name
```

ACCESS RIGHTS (JAVA)

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

public
Can be accessible anywhere

Pretty much the case for python (python **does not** support **private** access rights)

ACCESS RIGHTS (JAVA)

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

protected
Not accessible outside of this class unless the class is a subclass (slightly less restrictive compared to **private**)

ACCESS RIGHTS (JAVA)

```
class Cat {
  private String name;

  public Cat(String name) {
    this.name = name;
  }

  public void makeSound() {
    System.out.println("meow");
  }
}
```

Most OOP languages also come with different types of access rights:

- `private`
- `public`
- `protected`

AND THE WINNER IS...

JAVA

WHY?

Java

- Even though more verbose
- More support for Object-Oriented Programming (OOP)
- OOP in Python feels a bit limited

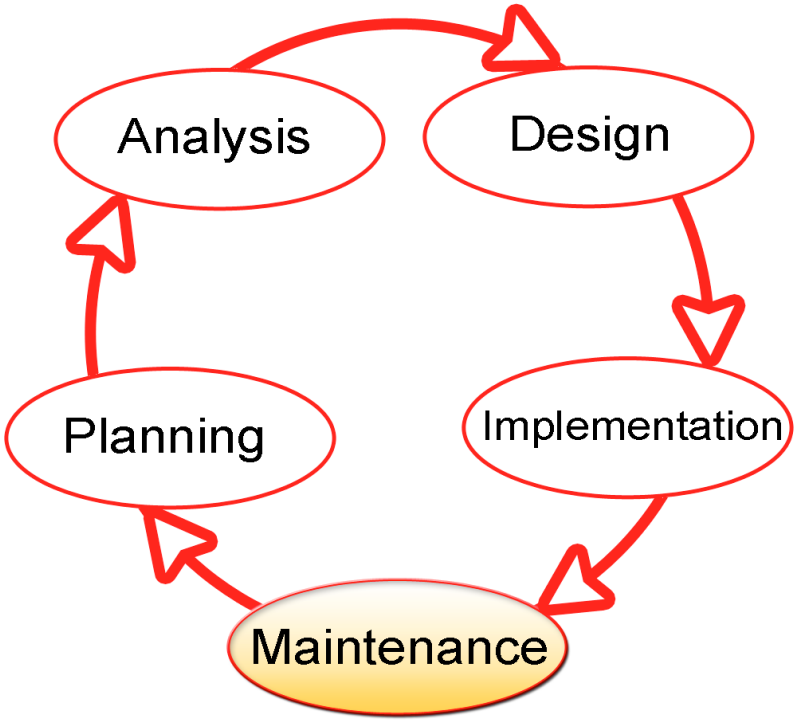
Having said that, we are only comparing with regards to the OOP features

- Python has its own benefits (e.g. being multi-paradigm, simple, etc)

Will focus on Java-like of OOP implementation for lecture

INTRODUCTION TO DESIGN PHASE

DESIGN PHASE



We can generalize SDLC into 5 different phases:

1. Planning
2. Analysis
3. **Design**
4. Implementation
5. Support/Maintenance

ANALYSIS PHASE (REQUIREMENTS)

Focuses on what the system should do

DESIGN PHASE

Oriented towards how the system will be built

Structural components

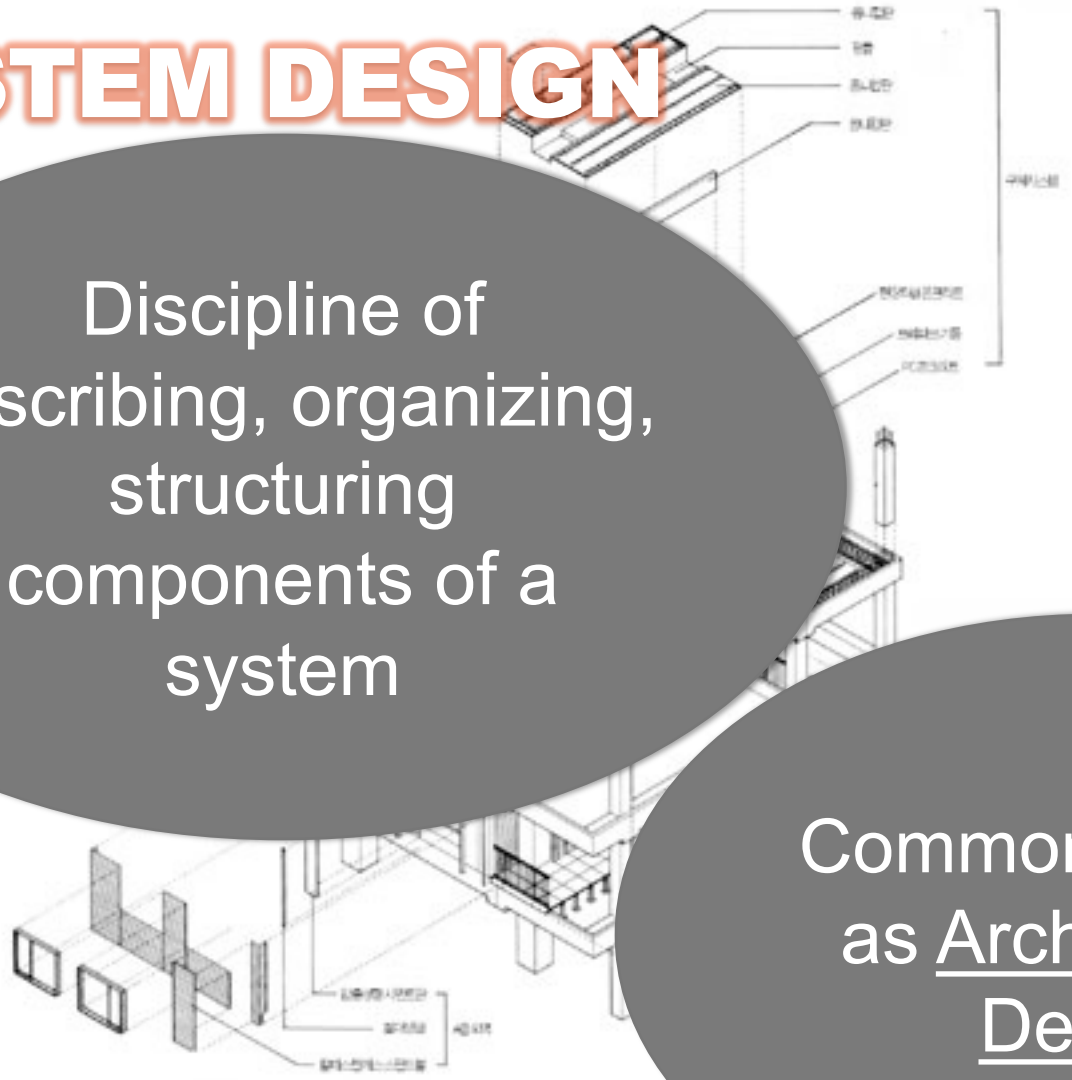
Dynamic interactions

DESIGN PHASE

This is the phase when the business model is turned into a design specification

This also prepares for the implementation phase (where actual construction of system takes place)

SYSTEM DESIGN



Discipline of describing, organizing, structuring components of a system

Commonly known as Architectural Design

HIGH/LOW LEVEL INFORMATION SYSTEM

Helpful to break down the design discipline tasks into high and low levels

HIGH-LEVEL INFORMATION SYSTEM DESIGN

Example of high-level design include hardware, network, system infrastructure, etc

LOW-LEVEL INFORMATION SYSTEM DESIGN

Example of low-level design include user interface, classes, methods, etc

TRANSFORMING ANALYSIS MODELS TO DESIGN MODELS

USE CASE REALIZATION

Use-case realization is the design of software that implements each use case
(i.e. use case description → design)

DESIGN SOFTWARE ARCHITECTURE

"Now that you have an overview of the system, we're ready for a little more detail"

Analysis phase results in domain model class diagram

Architectural design extends it to a design class diagram

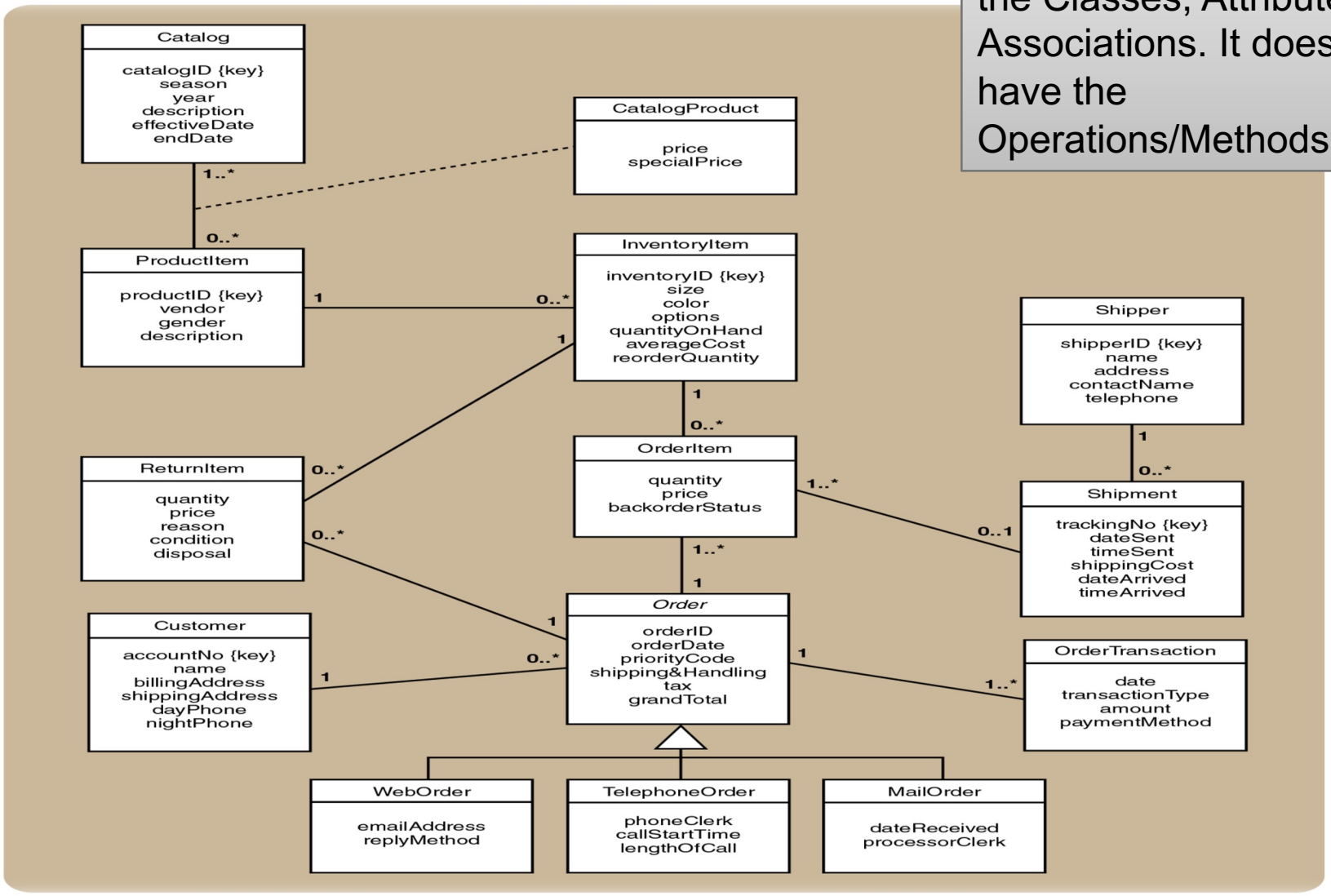
DESIGN CLASS DIAGRAM

Design class diagram (often just simply called class diagram) extends domain model class diagram:

- Methods
- Method parameters
- Access rights (of attributes and methods)
- Constructors
- etc

STARTING POINT: DOMAIN MODEL CLASS DIAGRAM

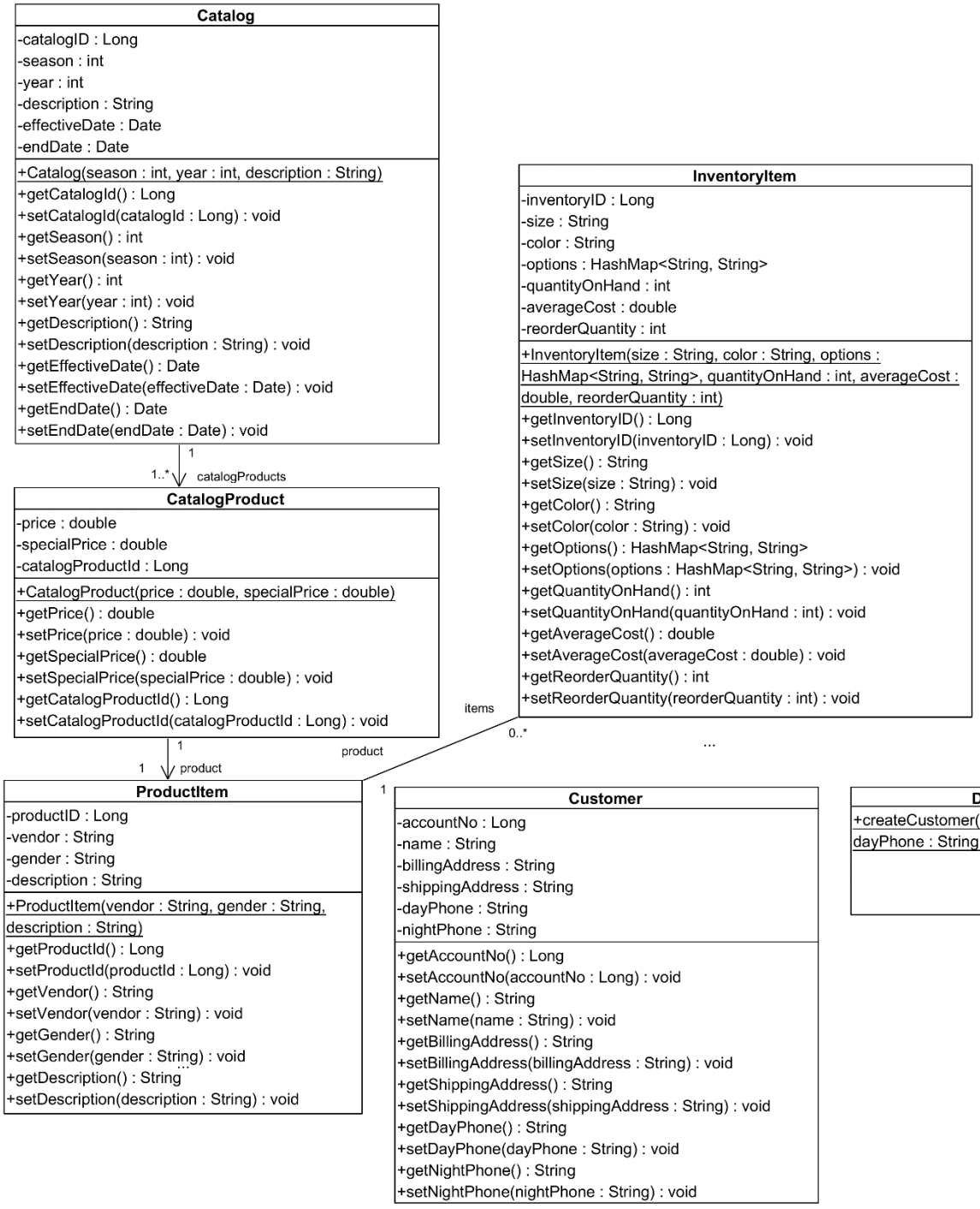
Domain Model Class Diagram only contains the Classes, Attributes, Associations. It doesn't have the Operations/Methods



STARTING POINT: FLOW OF EVENT (USE CASE DESCRIPTION)

Use case name	Create new customer	
…		
Flow of events	**Actor**	**System**
	1. Click on the Register link 2. Enter name, address, contact number (day/night phone numbers)	1.1 Display the registration form 2.1 Validate fields 2.2 Create Customer record and save into Database 2.3 Display registration confirmation message

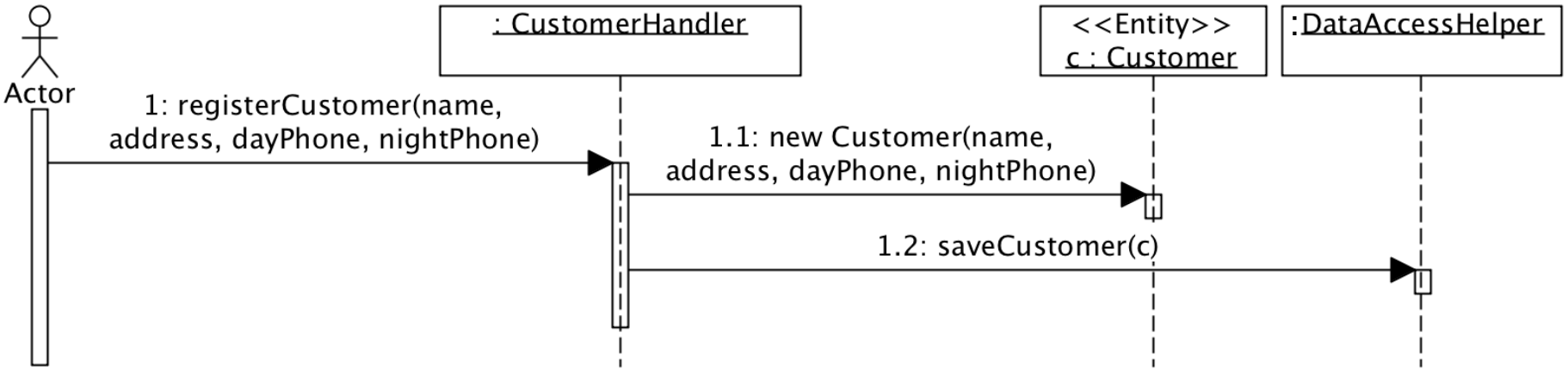
End Product 1: Design Class Diagram (partially completed)



While UML is programming language agnostic, we will focus on a static-typing language such as Java

END PRODUCT 2: SEQUENCE DIAGRAM

Create new customer use case



SEQUENCE DIAGRAM

1 sequence diagram for each use case

Sequence diagram contains:

- Method calls
- Interaction between 2 different objects

It is used to discover the methods in the design class diagram

- Why is this needed?
- Often it is difficult to imagine (exhaustively) what methods are required
- By analyzing the method calls needed for a use case, we are less likely to miss out methods and the details of a method

OBJECT-ORIENTED DESIGN PROCESS

The process of creating the class diagram and sequence diagram happen concurrently

Very difficult to accurately complete one diagram after another

There will be a lot of back and forth editing!

DESIGN CLASSES AND DESIGN CLASS DIAGRAMS

The design class diagrams and the sequence diagrams use each other as inputs for design, and are developed at the same time

Design Class Diagram

- Start with **domain model class diagram**
- Add in the **methods** for each class
- Refine on these methods while you are **developing the sequence diagrams**
- Update the Class Diagram and Sequence Diagram
- Repeat this process until the **flow** and **structure** is logical

DESIGN CLASSES AND DESIGN CLASS DIAGRAMS

During analysis, we do not care about the details of the classes

During design, it is important to encode the details of the classes

- Access rights of attributes and methods (public, private, etc)
- Type of the attributes
- Methods
- Arguments/Parameters of methods
- Return value of methods and type of arguments

DESIGN CLASSES AND DESIGN CLASS DIAGRAMS

In Analysis phase, we only consider Entity classes

To build a complete system, we must also identify other classes

- e.g. Input window classes and Database helper classes are additional classes that must be defined

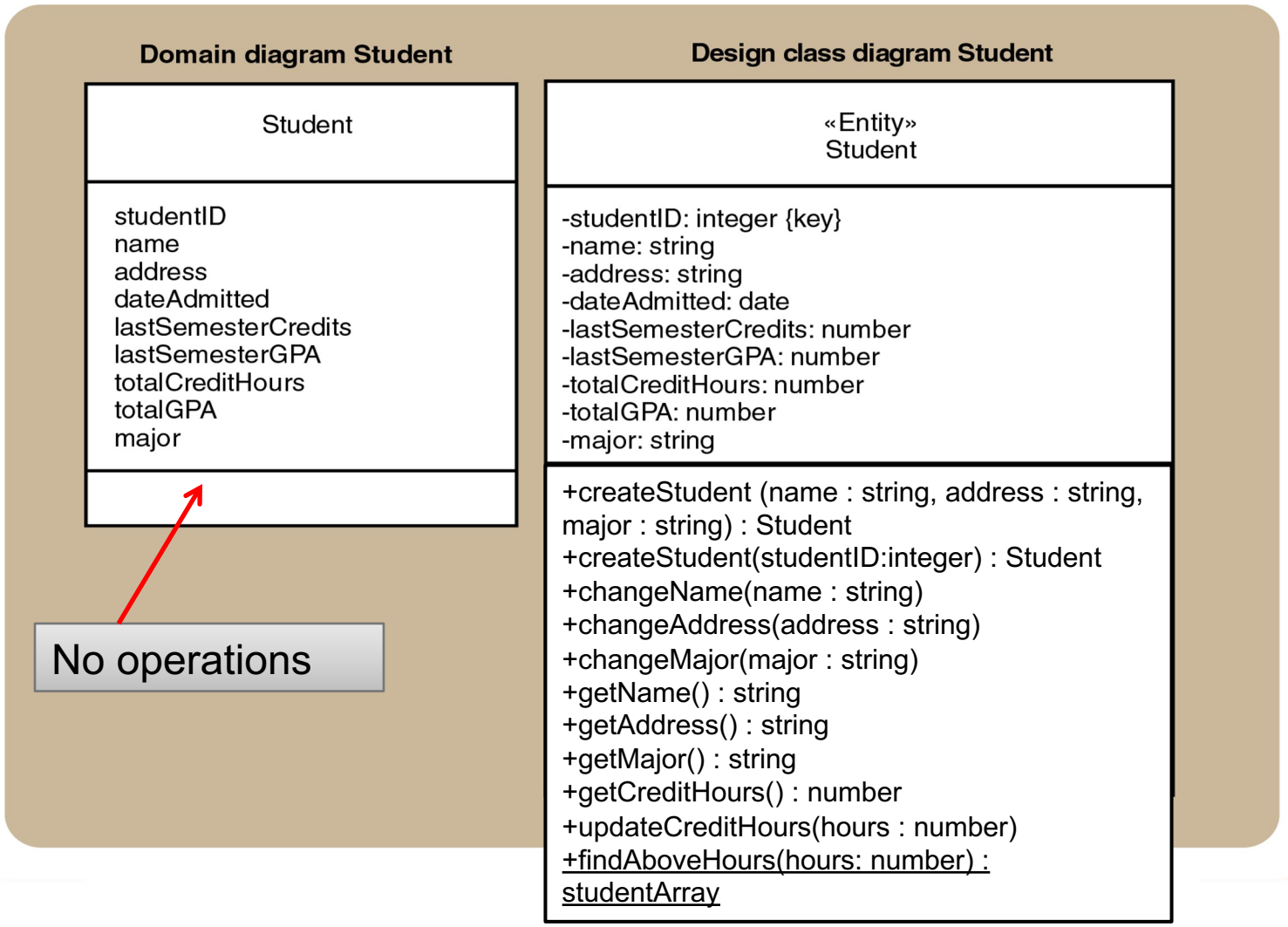
REALIZATION OF USE CASES

The end result of the development of these design models is called realization of use cases

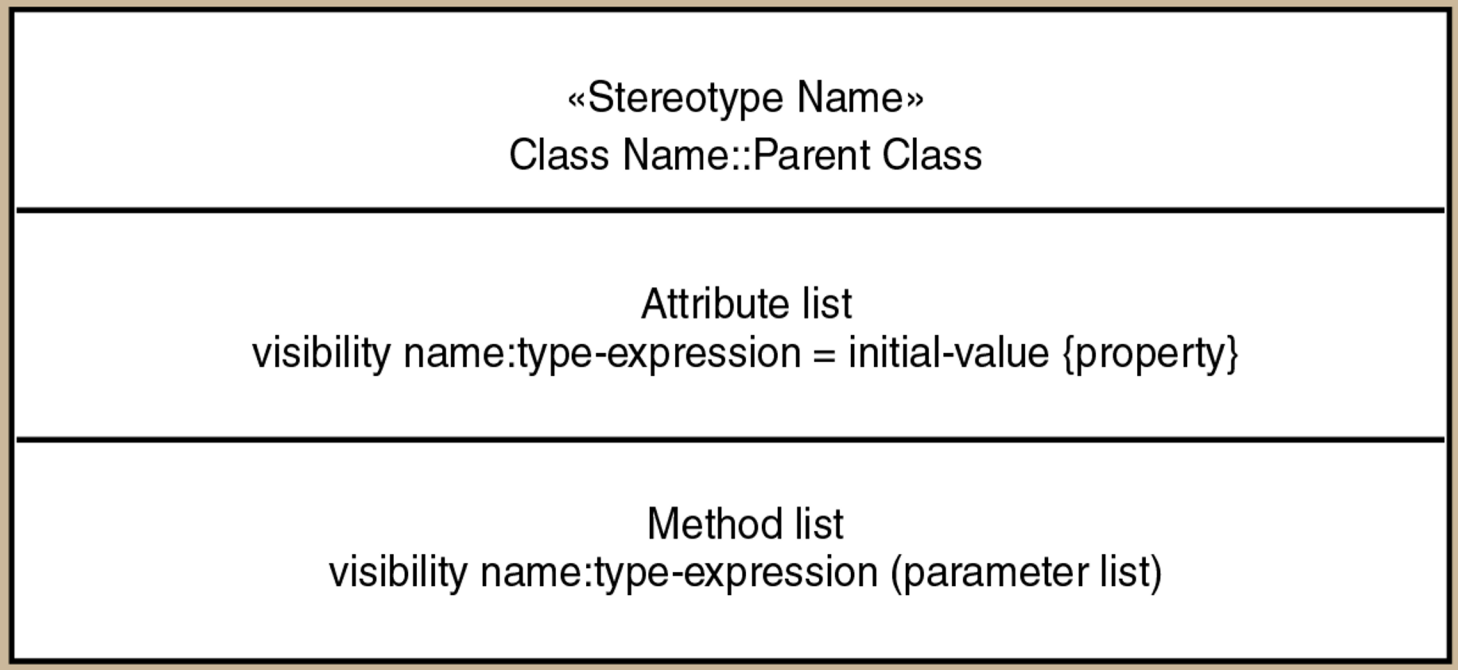
- "Realization" is the specification of the detailed processing that the system must do to carry out the use case i.e. make a set of software blueprints

DESIGN CLASS DIAGRAM NOTATION

DOMAIN MODEL CLASS DIAGRAM VS DESIGN CLASS DIAGRAM



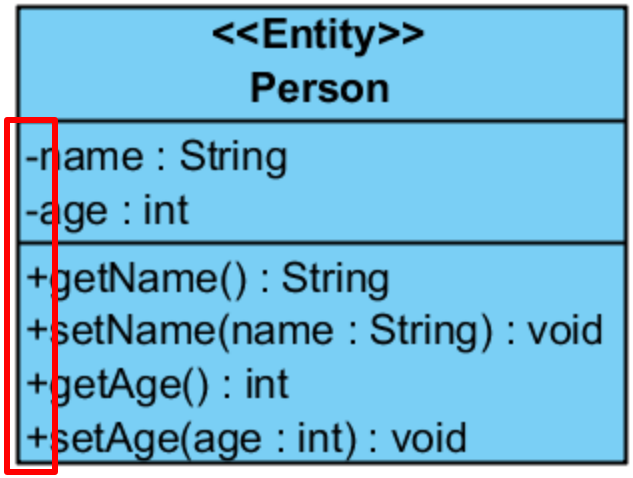
DESIGN CLASS NOTATION



DESIGN CLASS NOTATION

Visibility – denotes the access rights information

- Attribute visibility
- Operation visibility
- **+** : public access rights
- **-** : private access rights
- **#** : protected access rights



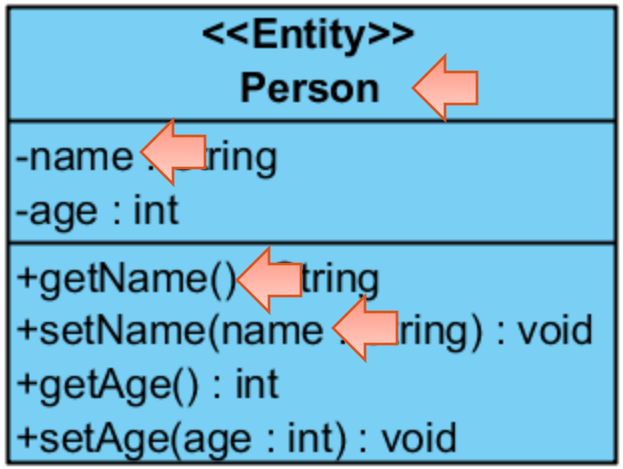
DESIGN CLASS NOTATION

Naming

Class, Attribute, Operation, Parameter

Make sure you use the actual names to be used in the actual development!

- Case sensitivity
- No spaces etc



DESIGN CLASS NOTATION

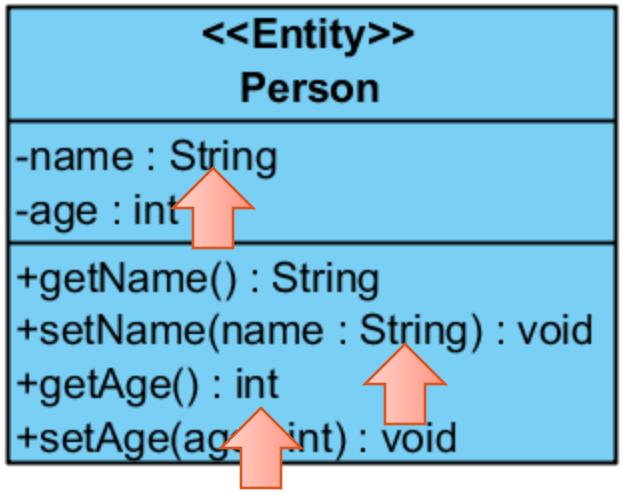
Type-expression – denotes the type of the attribute, parameter, return value

Again, make sure you use the actual names to be used in the actual development!

- Case sensitivity
- No spaces etc

void is used to denote methods with no return value

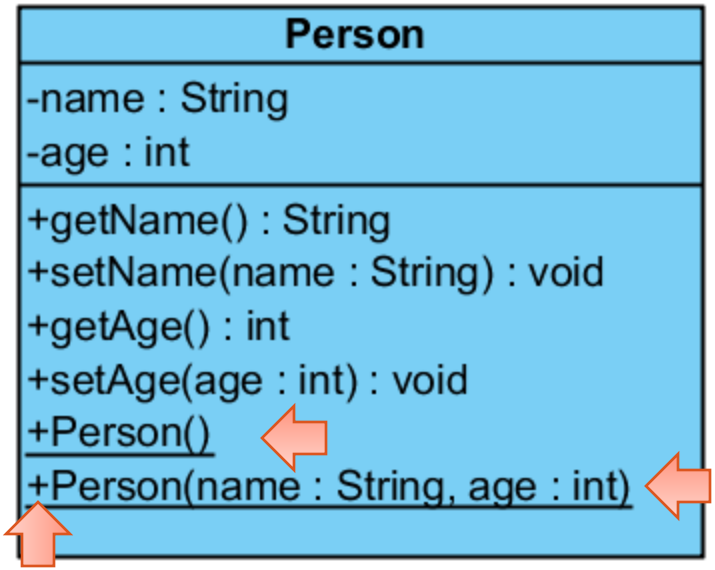
- You can omit writing void for this case
- E.g.
  setName(name: String) : void
  =>
  setName(name: String)



DESIGN CLASS NOTATION

Constructor

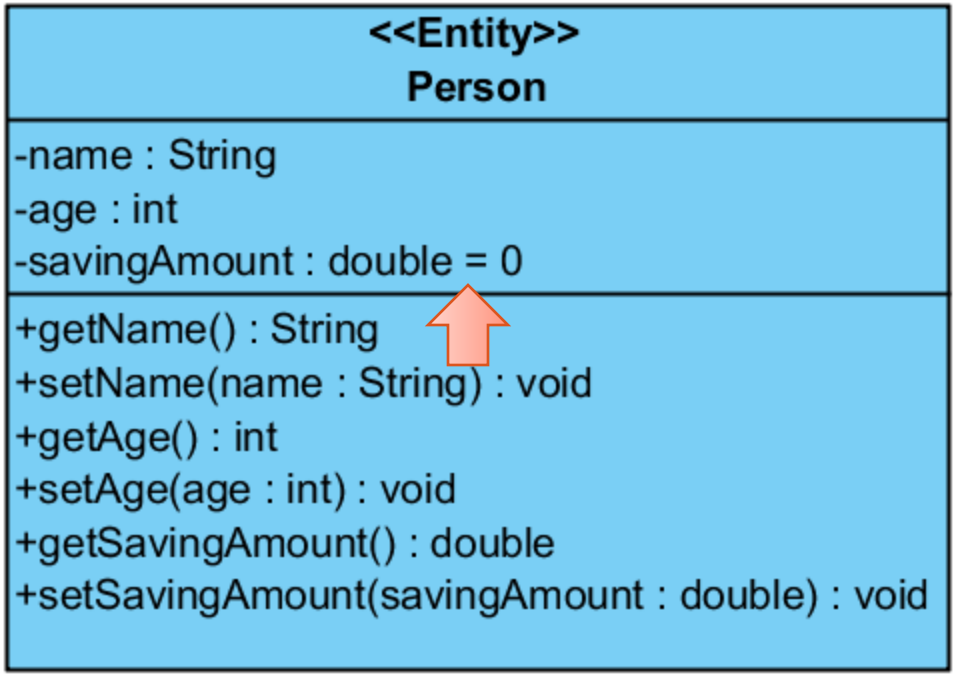
- Similar to a method except that the “operation” name is the same as the class name
- Multiple constructors can be created
- Since the “return type” is supposed to be the class itself, you do not have to include the type
- Constructors usually have public or protected access rights



DESIGN CLASS NOTATION

Initial Value

- It is also possible to define an initial value
- savingAmount is set to 0 when a Person object is created

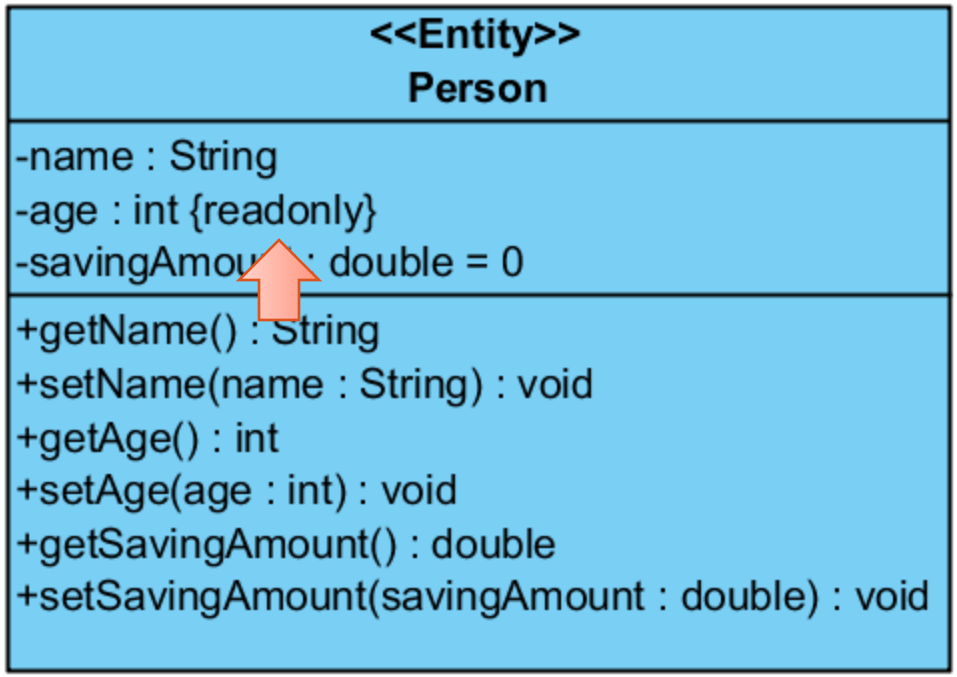


DESIGN CLASS NOTATION

Property – properties are placed within curly braces

- E.g. {readonly}, {key}

This is to provide additional information about the attribute

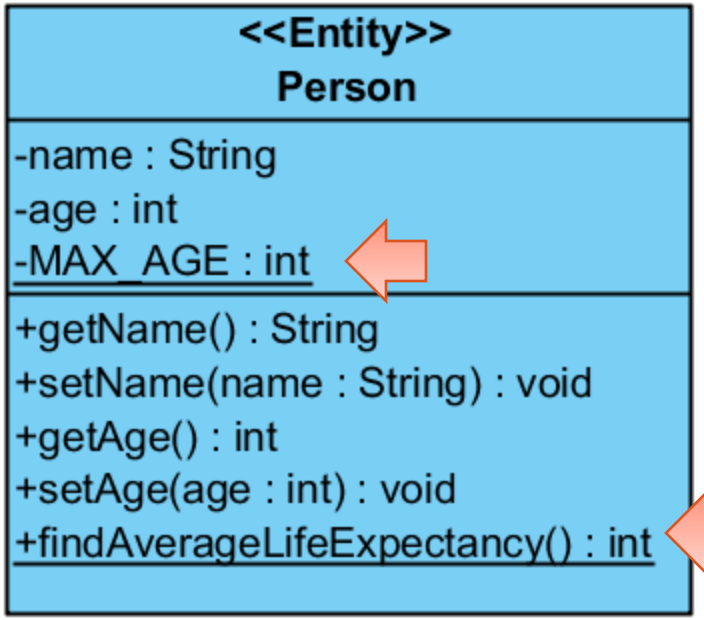


DESIGN CLASS NOTATION

Class attribute/operation vs Instance attribute/operation

- Class attribute/operations are underlined
- class method/attribute => **static**

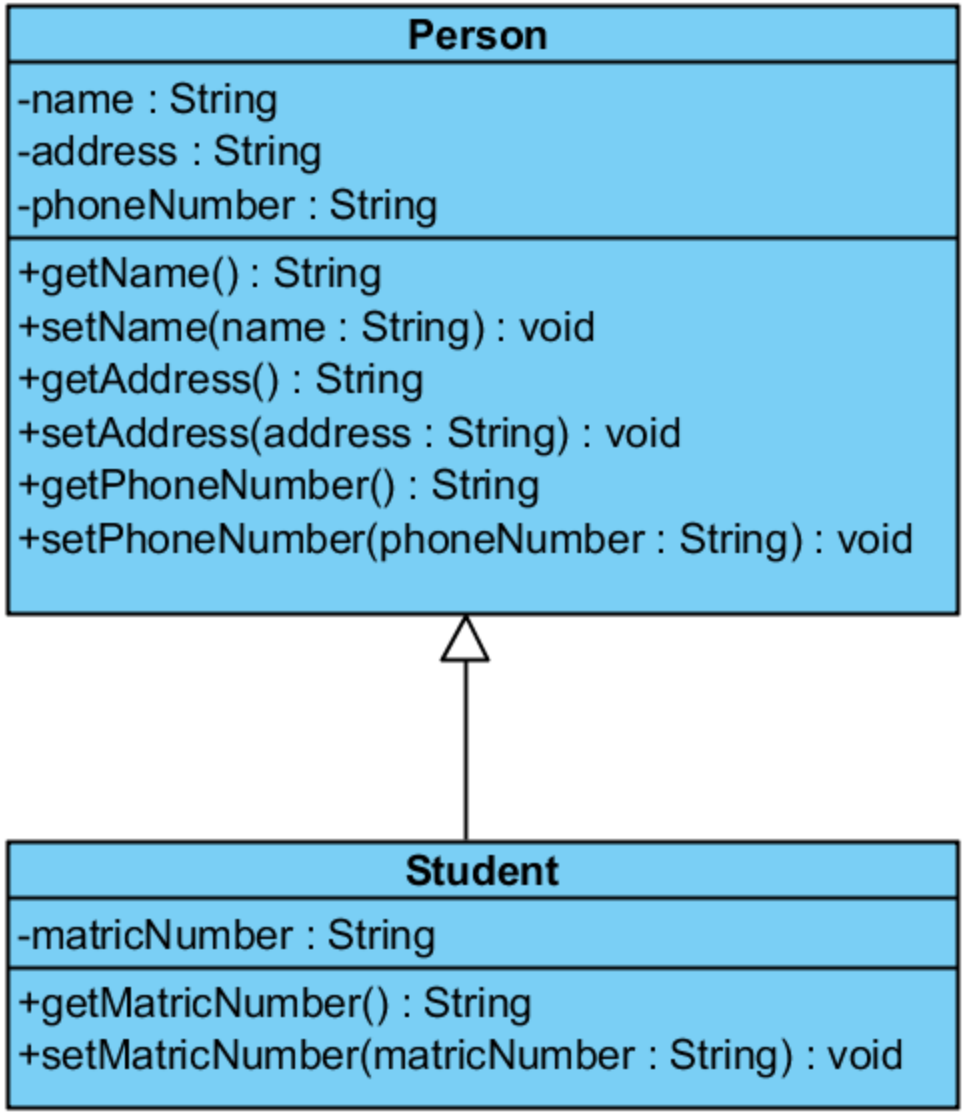
Do you know what's the difference between class and instance?



INHERITANCE

For the case of inheritance, you do not include the attributes/methods in the subclass (unless they are meant to be overridden)

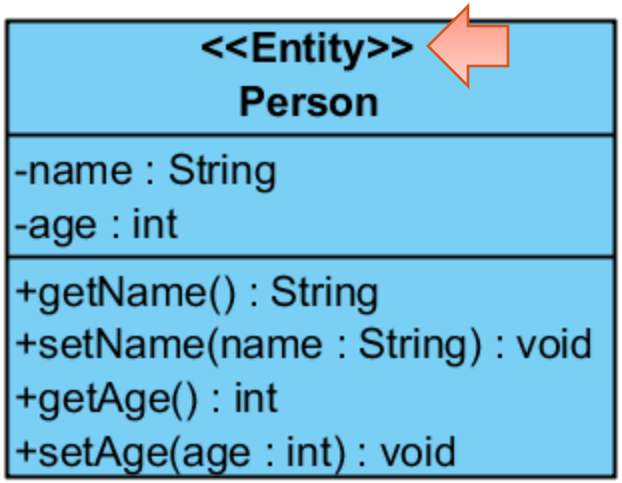
- Similar to actual coding



DESIGN CLASS NOTATION

Stereotype – denotes the different types of design classes

- E.g. <<Entity>>, <<Boundary>>, <<Controller>> etc

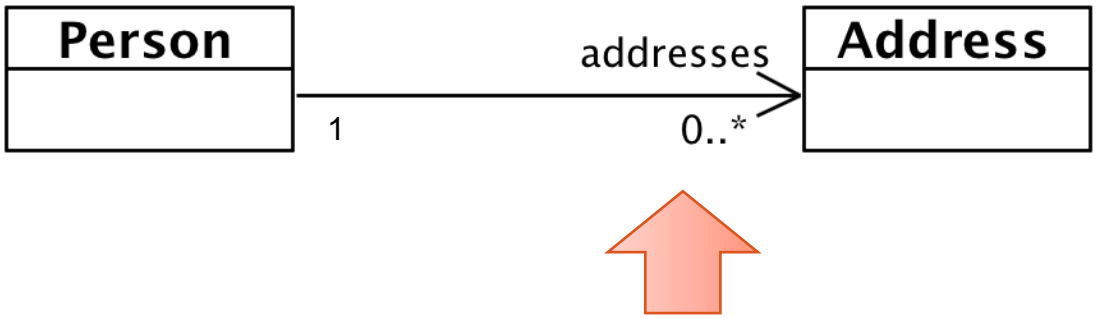


DESIGN CLASS NOTATION

Navigation Arrows

Role names

Multiplicity



STANDARD TYPES OF DESIGN CLASSES

<<Entity>> : denotes the business objects classes

- Often used as a “container” for storing data and for transferring data

<<Boundary>> : a class that is specifically designed to live on the system’s automation boundary

- User Interface class that allows user to interact with the system
- Usually we omit this in the sequence diagram

STANDARD TYPES OF DESIGN CLASSES

<<Controller>> : a class that mediates between the boundary classes and domain model classes

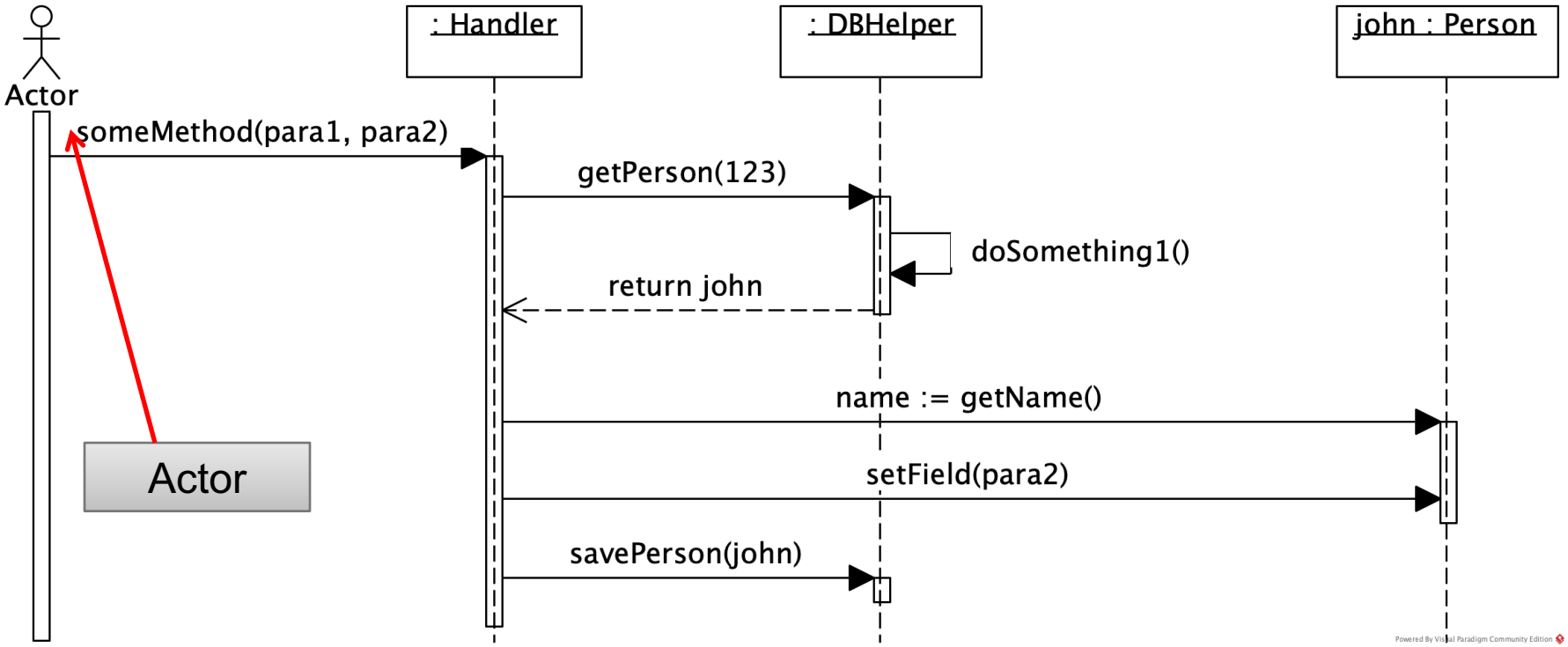
- Its responsibility is to catch the messages from the boundary class objects and send them to the correct domain model objects which will do the necessary actions
- E.g. Controller directs a login() request to MemberHandler class which will do the authentication

<<DataAccess>> : denotes the data access class

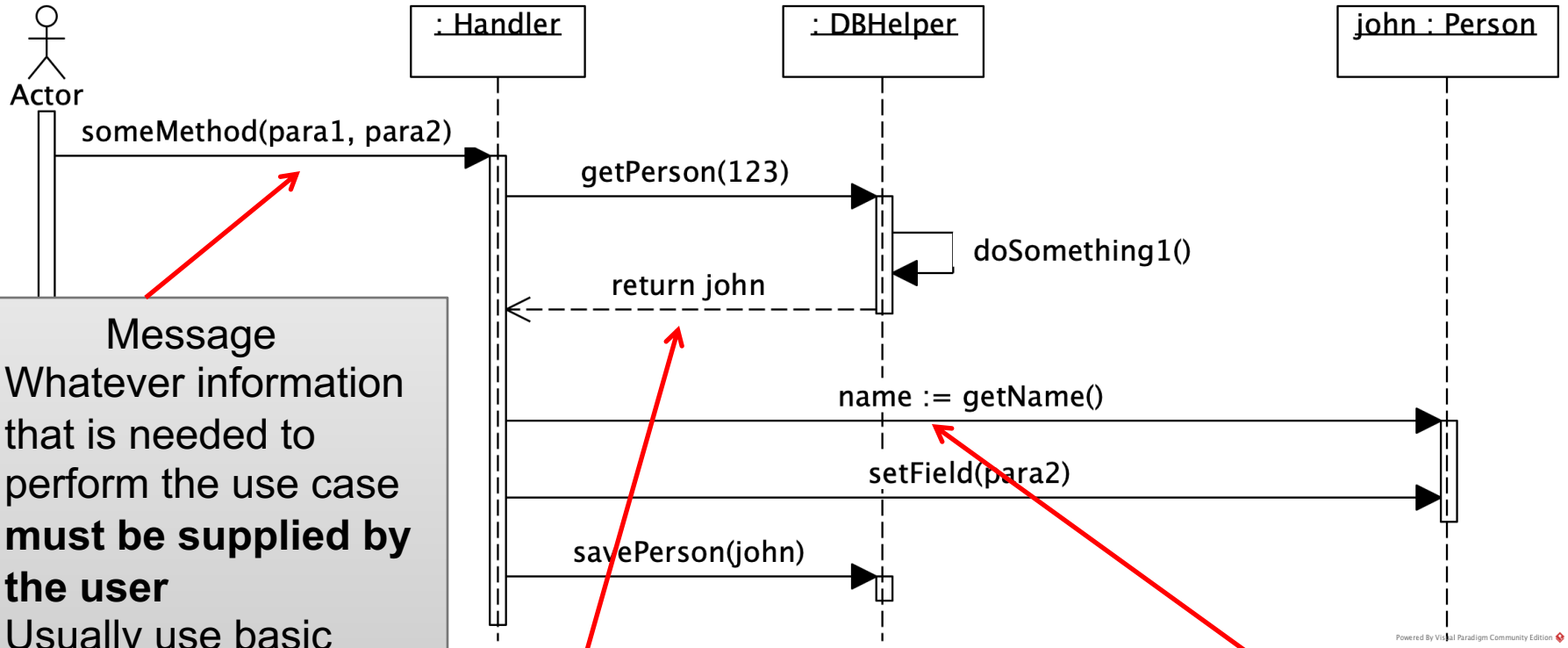
- Communication is usually facilitated using entity classes

SEQUENCE DIAGRAM NOTATION

SEQUENCE DIAGRAM NOTATION



SEQUENCE DIAGRAM NOTATION



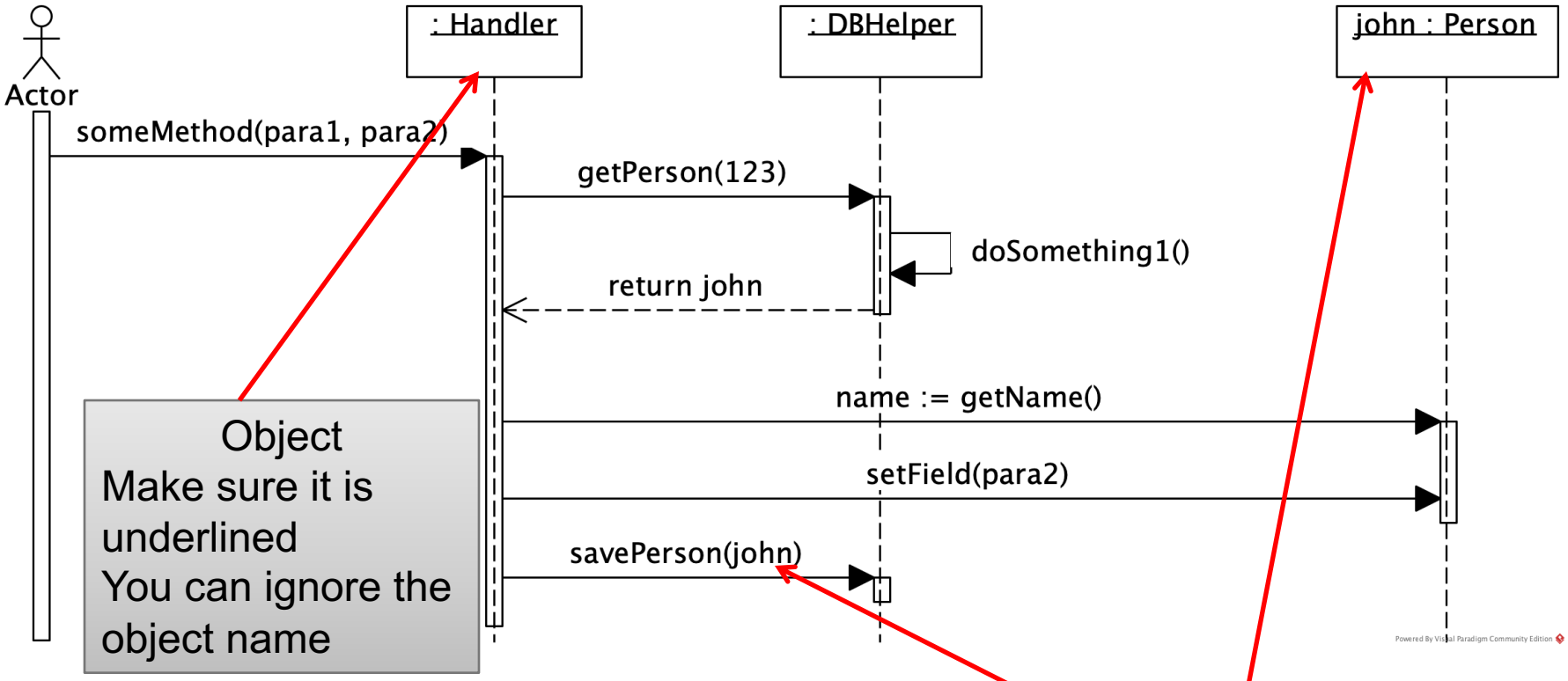
Message

- Whatever information that is needed to perform the use case **must be supplied by the user**
- Usually use basic datatypes (rather than entities) so that we know what parameters are needed

Return Value
Explicitly indicates when the method returns the value

Return Value (alternative)
Do not care when the method returns the value

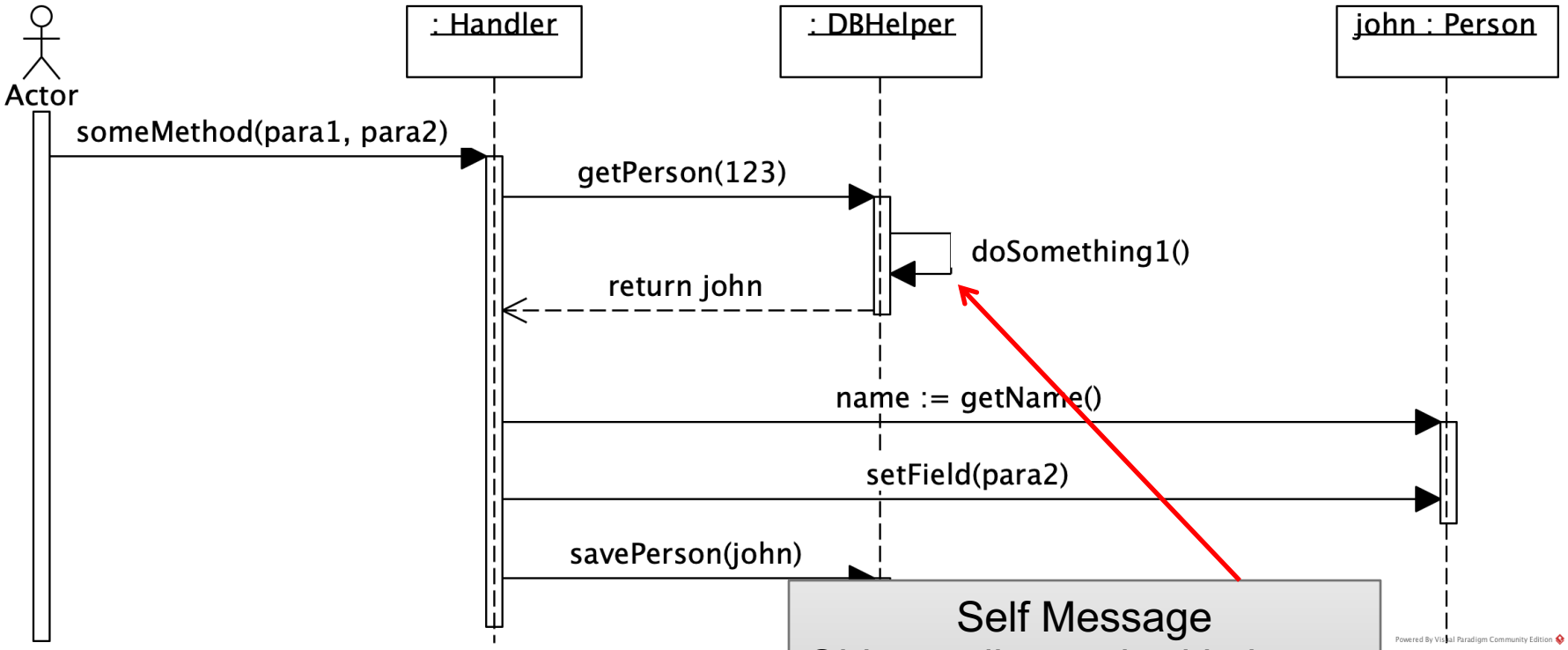
SEQUENCE DIAGRAM NOTATION



Object
Make sure it is underlined
You can ignore the object name

Object
Sometimes it is useful to name the objects if you need to refer to an object (as return value or as parameters for another method call)

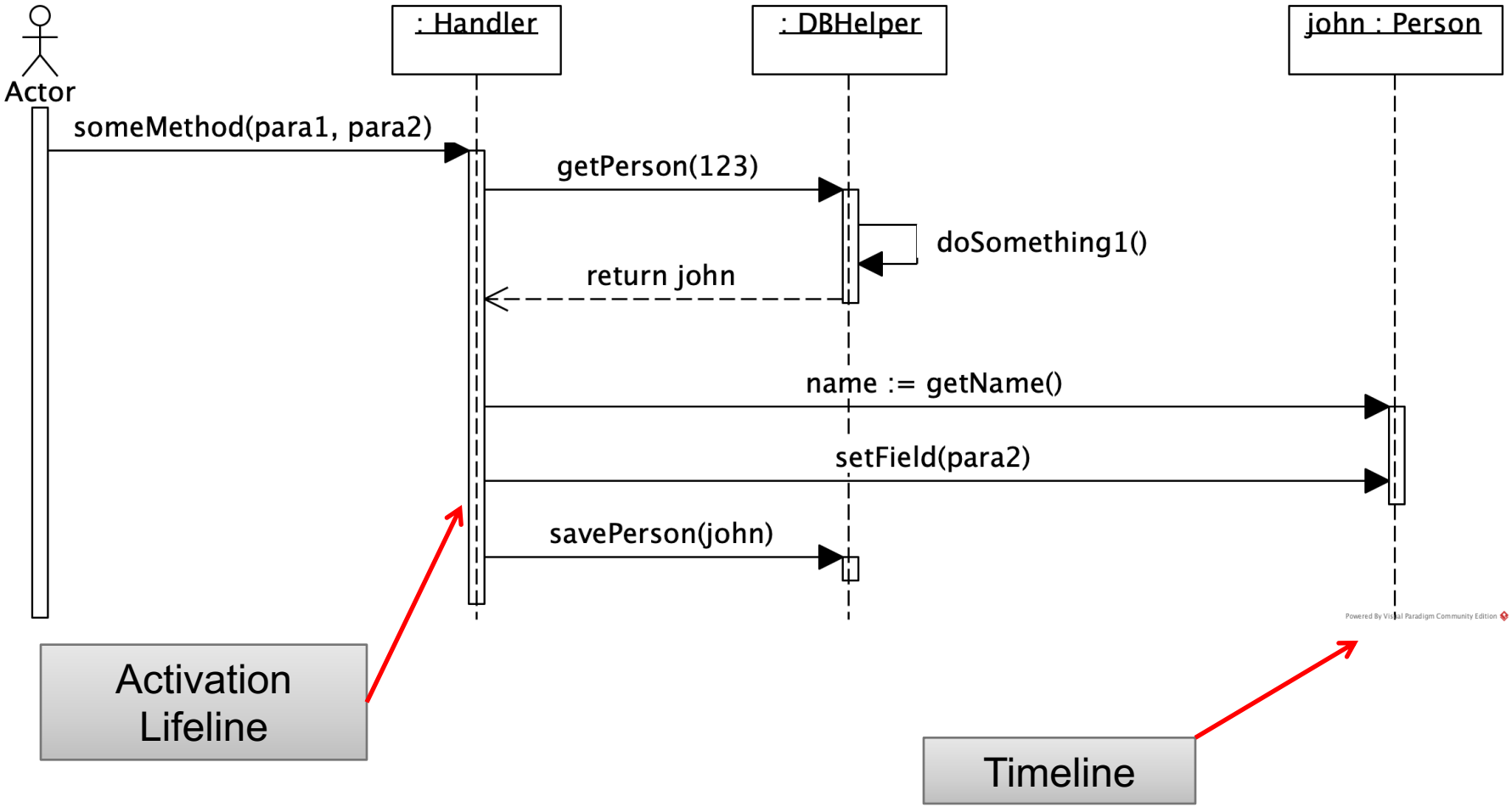
SEQUENCE DIAGRAM NOTATION



Self Message

Object call a method in its own class to perform some task required for the original method call

SEQUENCE DIAGRAM NOTATION



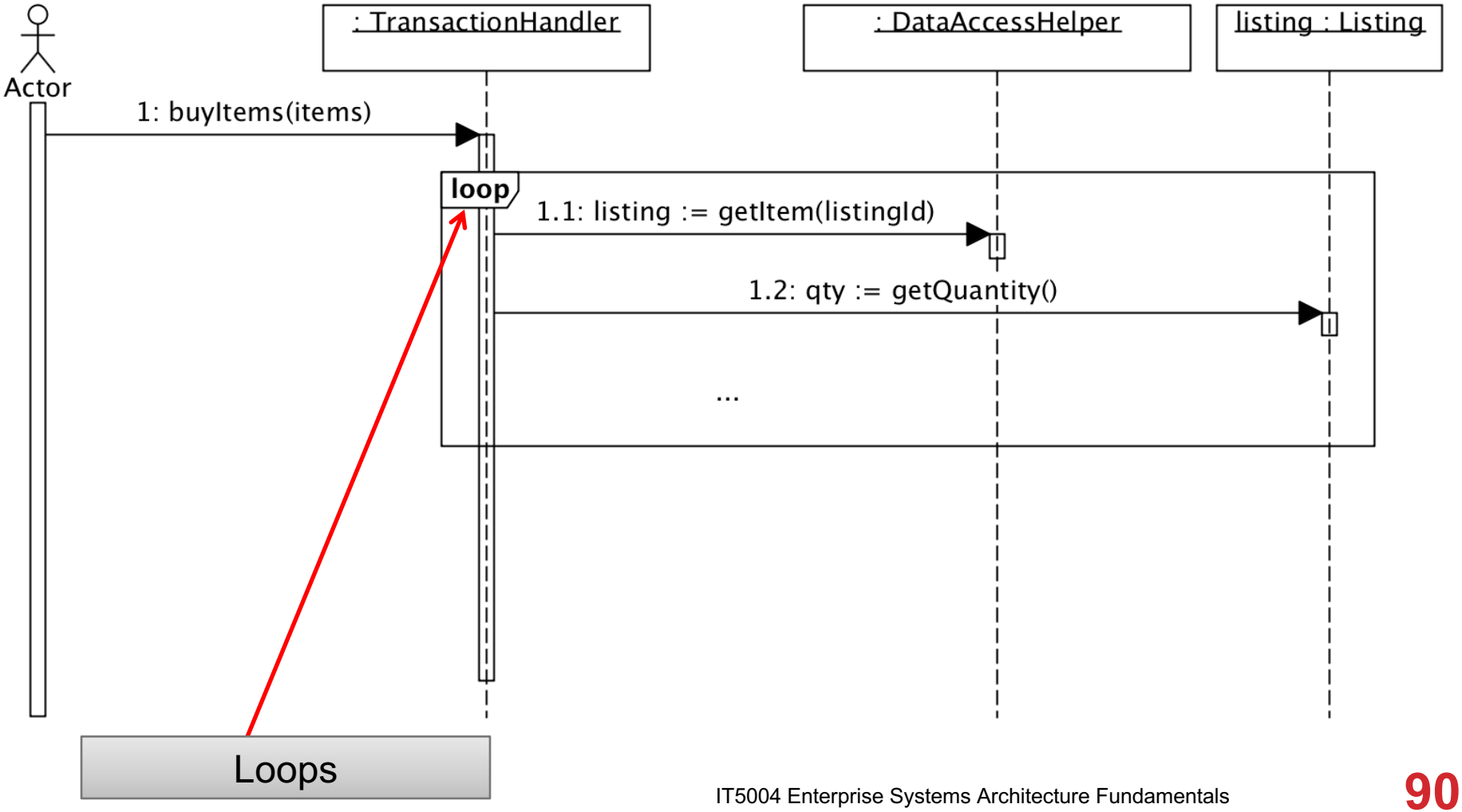
ACTIVATION LIFELINE

Activation lifelines/boxes/Focus of Control
indicates when an object is executing a method

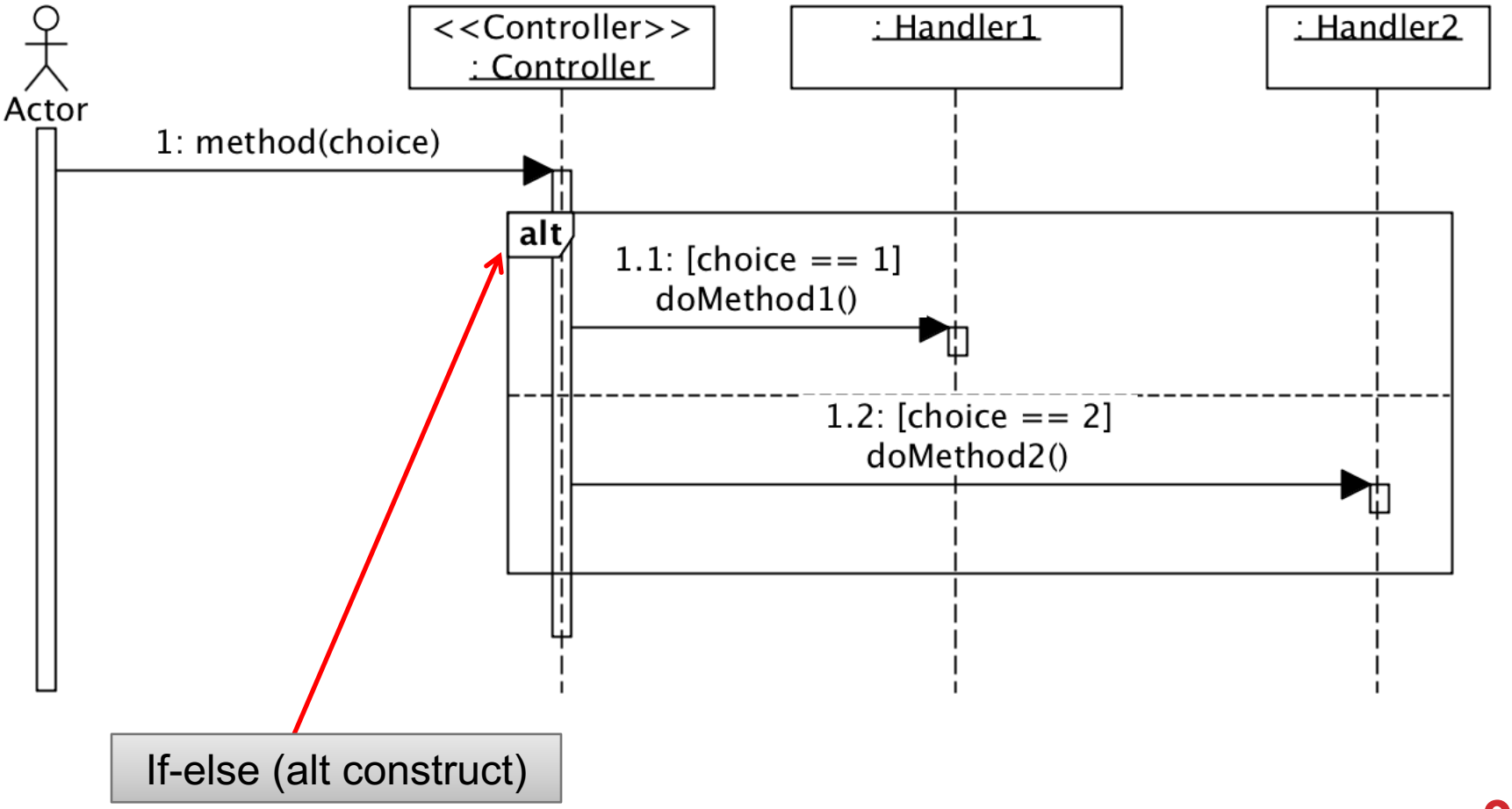
- This is useful to determine which object is in charge of a message sequence and the time duration

Some of the diagrams (in the book) did not include this but you should include this

SEQUENCE DIAGRAM NOTATION



SEQUENCE DIAGRAM NOTATION



SUMMARY

The main focus of design phase is to transform analysis model into design models (design class diagram and sequence diagram)

Design class diagram:

- Notation is similar to domain model class diagram
- Comes with access rights, methods, method parameters, constructors, navigation arrow
- Blueprint of a class (without the actual implementation)

SUMMARY

Sequence diagram

- Method calls are shown in the form of messages
- Helps in discovering classes/methods/method signatures

Design class diagram and sequence diagrams are developed concurrently

WHAT'S NEXT?

Fundamental design principles

Enterprise System Architecture Design Styles