

**Software Engineering
Bootcamp**

Hyperiondev

Object-Oriented Design: Recap

Lecture – Housekeeping

- ❑ The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
- ❑ No question is daft or silly - **ask them!**
- ❑ There are Q/A sessions midway and at the end of the session, should you wish to ask any follow-up questions.
- ❑ You can also submit questions here:
<http://hyperiondev.com/sbc4-se-questions>
- ❑ For all non-academic questions, please submit a query:
www.hyperiondev.com/support
- ❑ Report a safeguarding incident:
<http://hyperiondev.com/safeguardreporting>
- ❑ We would love your feedback on lectures:
<https://hyperiondev.wufoo.com/forms/zsgv4m40ui4i0g/>

Objectives

1. Recap on
 - a. Class diagrams
 - b. Context and Sequence diagrams
 - c. SOLID Principles

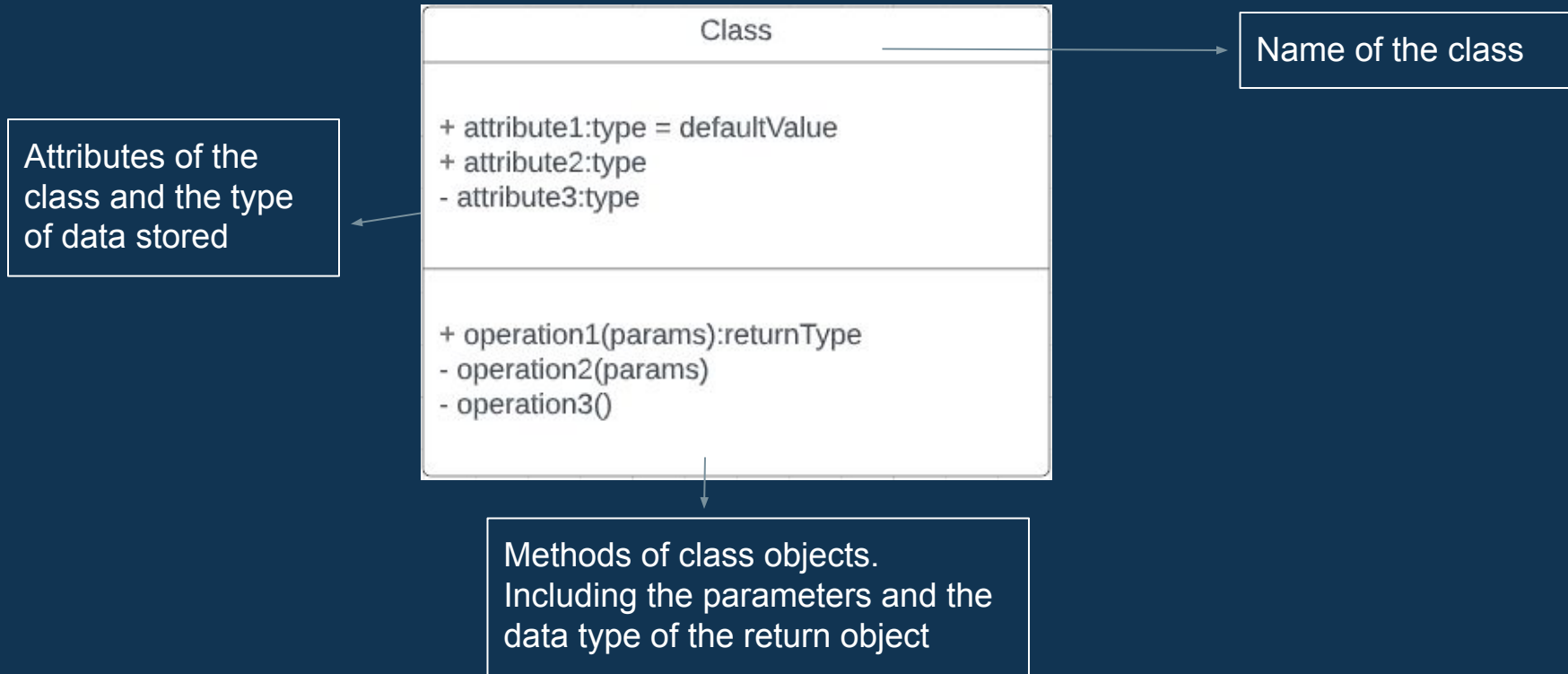
Github Repository – Lecture Examples/Slides

https://github.com/HyperionDevBootcamps/C4_SE_lecture_examples

Platform to create context & sequence diagrams

<https://www.lucidchart.com/pages/uml-class-diagram>

The Basics of Class Diagrams



Class diagrams

A class notation consists of three parts:

1. **Class Name**

- The name of the class appears in the first partition.





2. **Class Attributes**

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

3. **Class Operations** (Methods)

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters is shown after the colon following the parameter name.
- Operations map onto class methods in code

Relationships Between Classes

Inheritance	A child class inherits attributes and methods from a parent class.	
Association	A non-dependent relationship just a basic association relationship eg. siblings	
Aggregation	A specific type of association where the one class can exist without the other.	
Composition	A specific type of association where the one class cannot exist without the other.	

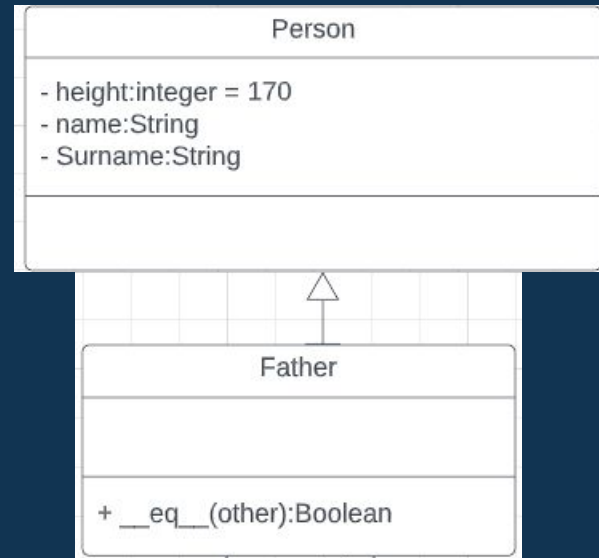
Drawing you own class diagram

- Let's have a look at one of our previous inheritance examples.

```
# Create a parent class called person
class Person:
    height = 170
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
```

```
# Create a child class called Father
class Father(Person):
    def __init__(self, name, surname = "Reeds"):
        super().__init__(name, surname)
        self.height = super().height - 10
```

```
    def __eq__(self, other):
        if self.height == other.height:
            return True
        else:
            return False
```



- We also had two child classes of Father class

```
# Create a child class of Father called Son
class Son(Father):
```

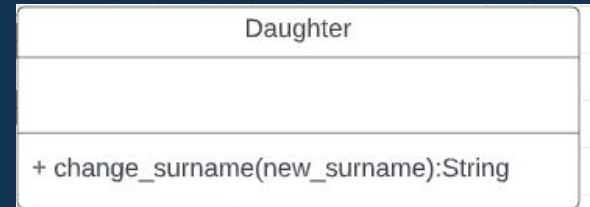
```
    def __init__(self, name):
        super().__init__(name)
        self.height = super().height + 10
```

```
    # Create a method to change the height of the son
    def set_height(self, new_height):
        self.height = new_height
```

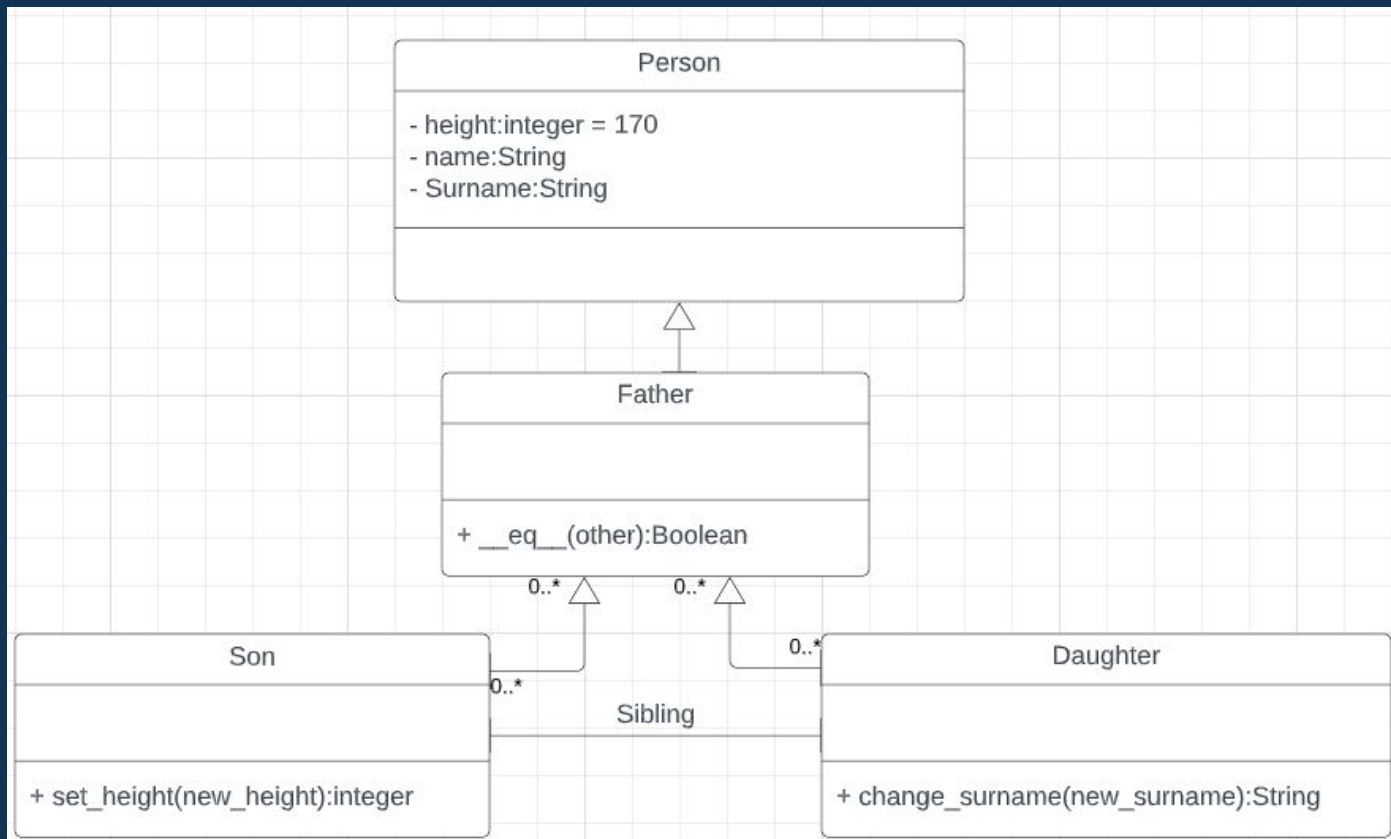
```
# Create another child class of Father called daughter
class Daughter(Father):
```

```
    def __init__(self, name):
        super().__init__(name)
```

```
    # Create methods to change the surname of the daughter
    def change_surname(self, new_surname):
        self.surname = new_surname
```

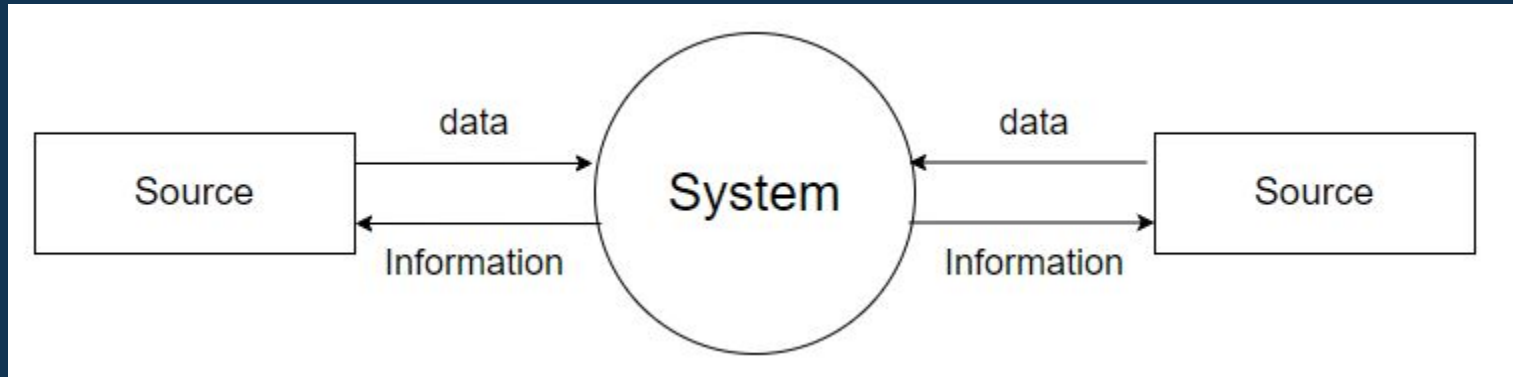


The relationship between classes



Context Diagrams

- We use context diagrams to display the flow of data between our system and sources.



Context Diagrams



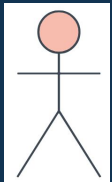


Sequence Diagram




- We use sequence diagrams to help us see how and in what order a group of objects will interact with each other within our program.

Basic symbols & components


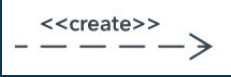

To understand what a sequence diagram is, you should be familiar with its symbols and components. Sequence diagrams are made up of the following icons and elements:

Name	Symbol	Description
Object symbol		Represents a class or object in UML. The object symbol demonstrates how an object will behave in the context of the system. Class attributes should not be listed in this shape.
Activation box		Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.
Actor Symbol		Shows entities that interact with or are external to the system.

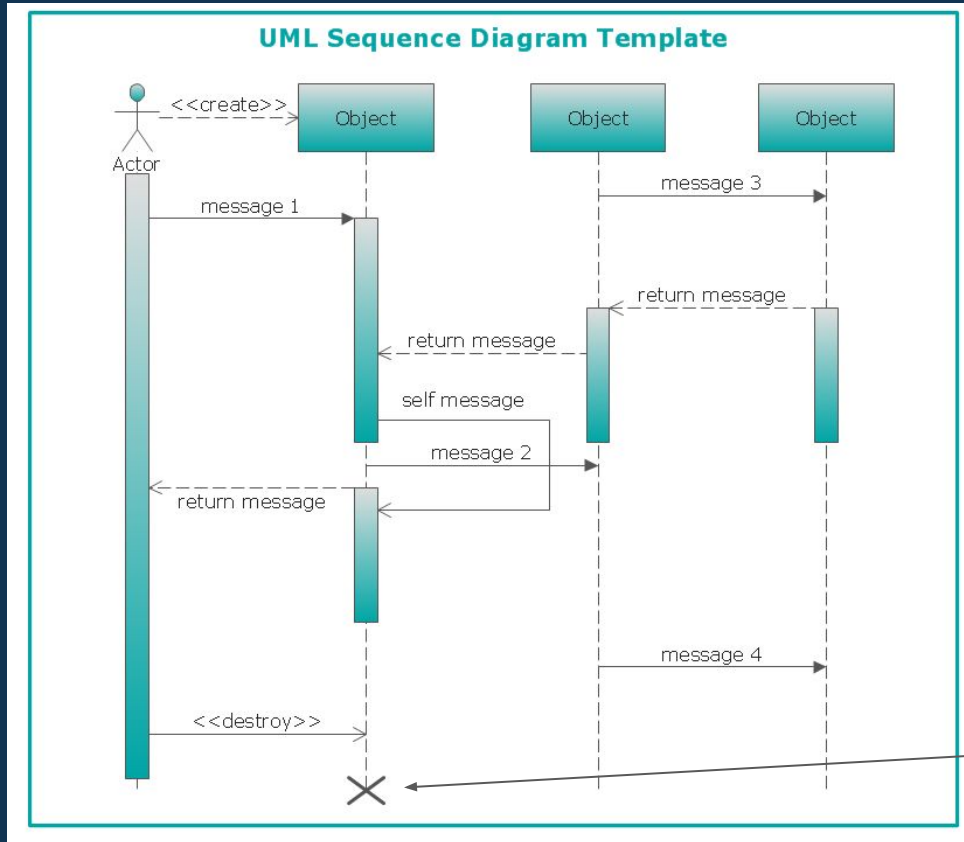
Common message symbols

Name	Symbol	Description
Synchronous message symbol		Represented by a solid line with a solid arrowhead. This symbol is used when a sender must wait for a response to a message before it continues. The diagram should show both the call and the reply.
Reply message symbol		Represented by a dashed line with a lined arrowhead, these messages are replies to calls.
Asynchronous message symbol		Represented by a solid line with a lined arrowhead. Asynchronous messages don't require a response before the sender continues. Only the call should be included in the diagram.

Common message symbols

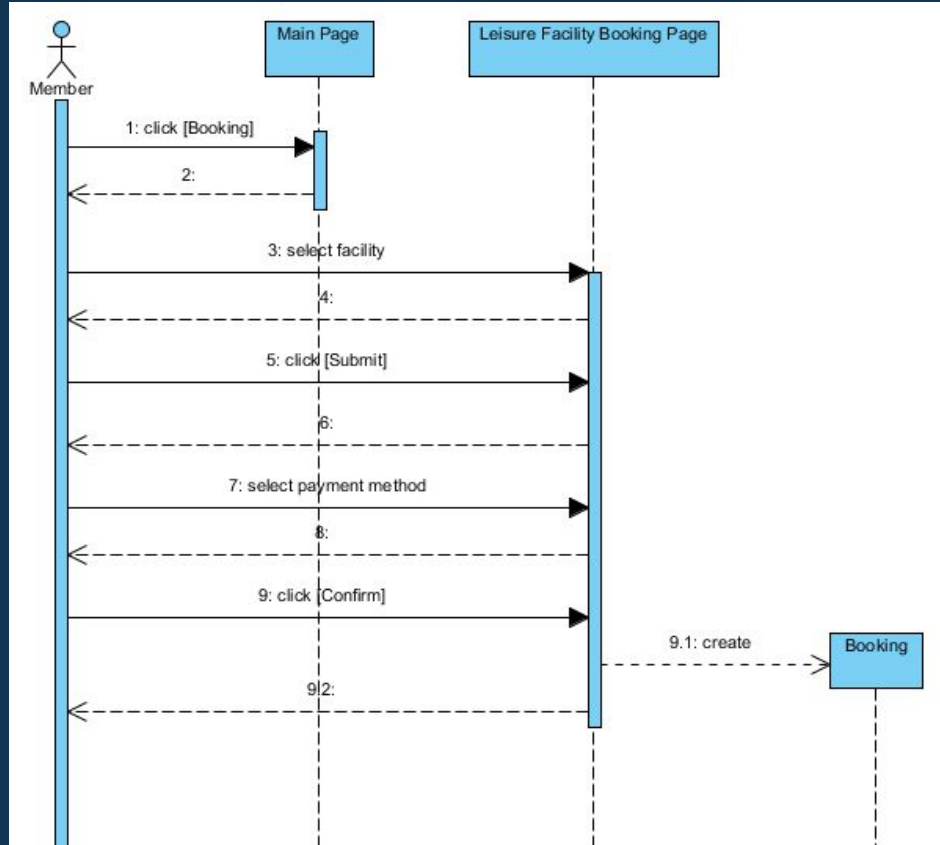
Name	Symbol	Description
Asynchronous return message symbol		Represented by a dashed line with a lined arrowhead.
Asynchronous create message symbol		Represented by a dashed line with a lined arrowhead. This message creates a new object.
Delete message symbol		Represented by a solid line with a solid arrowhead, followed by an X. This message destroys an object.

Sequence Diagram



Delete message symbol:
This message destroys
an object.

Sequence Diagram Example 2



The SOLID Principles

What are the SOLID principles?

- Five principles of object oriented class design.
- They are a set of rules and best practices to follow while designing a class structure.
- These principles will help us understand the need for certain design patterns.

Why do we use the SOLID principles?

- Helps to reduce dependencies.
- Engineers can change one area without impacting others.
- Makes designs easier to understand, maintain and extend.

What are the 5 principles?

- **S – Single Responsibility Principle**
A class should have one, and only one, reason to change.
- **O – Opened Closed Principle**
You should be able to extend a classes behavior, without modifying it.
- **L – Liskov Substitute Principle**
Derived classes must be substitutable for their base classes.
- **I – Interface Segregation Principle**
A class should not be forced to inherit a function in will not use.
- **D – Dependency Inversion Principle**
Depend on abstractions, not on concretions.

S – Single Responsibility Principle

A class should have one, and only one, reason to change.

```
class Book:
    def __init__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price

    def change_title(self, new_title):
        # Change title of book
        self.title = new_title

    def change_author(self, new_author):
        # Change author of book
        self.author = new_author

    def process_invoice(self, file, inv_num):
        # Create invoice for sale of book
        file.write(f"Invoice number: {inv_num}"
                  f"Sale of book: {self.title} by {self.author}\n"
                  f"Price: {self.price}")
```

S – Single Responsibility Principle

A class should have one, and only one, reason to change.

```
class Book:
    def __init__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price

    def change_title(self, new_title):
        # Change title of book
        self.title = new_title

    def change_author(self, new_author):
        # Change author of book
        self.author = new_author
```

```
class Invoice:
    def __init__(self, inv_num, book):
        self.inv_num = inv_num
        self.book = book

    def process_invoice(self, file):
        # Create invoice for sale of book
        file.write(f"Invoice number: {self.inv_num}"
                  f"Sale of book: {self.book.title} by {self.book.author}\n"
                  f"Price: {self.book.price}")
```


O – Opened Closed Principle

You should be able to extend a classes behavior, without modifying it.

```
class Shape:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height
```

```
class Shape:
    def __init__(self, shape_type, length_list):
        self.shape_type = shape_type
        if shape_type == "Rectangle":
            self.width = length_list[0]
            self.height = length_list[1]
        elif shape_type == "Circle":
            self.radius = length_list[0]

    def calculate_area(self):
        if self.shape_type == "Rectangle":
            return self.width * self.height
        elif self.shape_type == "Circle":
            return pi * self.radius**2
```

O – Opened Closed Principle

You should be able to extend a classes behavior, without modifying it.

```
class Shape:
    def __init__(self, shape_type):
        self.shape_type = shape_type

    def calculate_area(self):
        pass
```

```
class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height
```

```
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def calculate_area(self):
        return pi * self.radius**2
```

L – Liskov Substitute Principle

Derived classes must be substitutable for their base classes.

```
def determine_total_area(shapes):  
    total_area = 0  
    for shape in shapes:  
        total_area += shape.calculate_area
```

```
class Rectangle(Shape):  
    def __init__(self, width, height):  
        super().__init__("Rectangle")  
        self.width = width  
        self.height = height  
  
    def calculate_area(self):  
        return self.width * self.height
```

```
class Circle(Shape):  
    def __init__(self, radius):  
        super().__init__("Circle")  
        self.radius = radius  
  
    def calculate_area(self):  
        return pi * self.radius**2
```


```
class Square(Shape):  
    def __init__(self, length):  
        self.length = length  
  
    def calculate_area(self):  
        return str(self.length**2)
```

L – Liskov Substitute Principle

Derived classes must be substitutable for their base classes.

```
class Square:
    def __init__(self, length):
        self.length = length

    def calculate_area(self):
        return str(self.length**2)
```



```
class Square:
    def __init__(self, length):
        self.length = length

    def calculate_area(self):
        return self.length**2
```


I – Interface Segregation Principle

A class should not be forced to inherit a function it will not use.

```
class Printer:
    def normal_print(self, document):
        print(f"Printing {document}")

    def fax(self, document):
        print(f"Faxing {document}")

    def scan(self, document):
        print(f"Scanning {document}")
```



```
class OldPrinter(Printer):
    def normal_print(self, document):
        print(f"Printing {document}")

    def fax(self, document):
        print(f"Not implemented")

    def scan(self, document):
        print(f"Not implemented")
```

I – Interface Segregation Principle

A class should not be forced to inherit a function it will not use.

```
class Printer:  
    def normal_print(self, document):  
        pass
```

```
class Faxer:  
    def fax(self, document):  
        pass
```

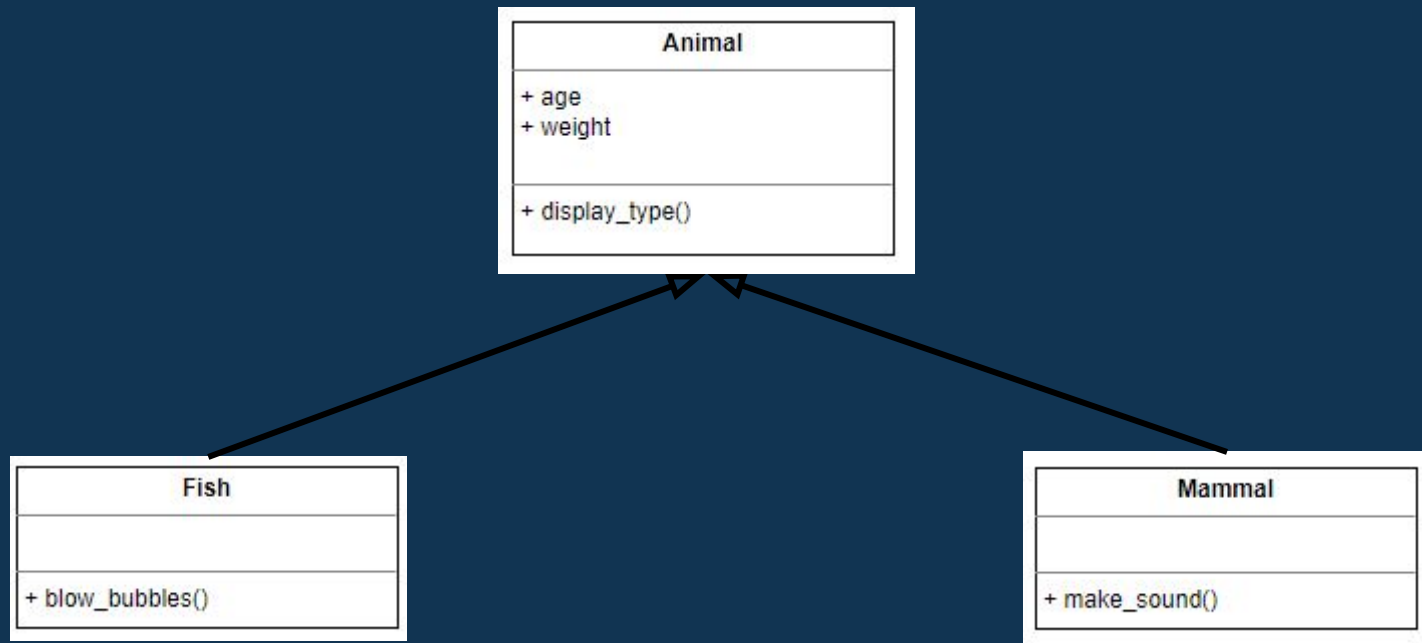
```
class Scanner:  
    def scan(self, document):  
        pass
```

```
class OldPrinter(Printer):  
    def normal_print(self, document):  
        print(f"Printing {document}")
```

```
class ModernPrinter(Printer, Faxer, Scanner):  
    def normal_print(self, document):  
        print(f"Printing {document}")  
  
    def fax(self, document):  
        print(f"Faxing {document}")  
  
    def scan(self, document):  
        print(f"Scanning {document}")
```

D - Dependency Inversion Principle

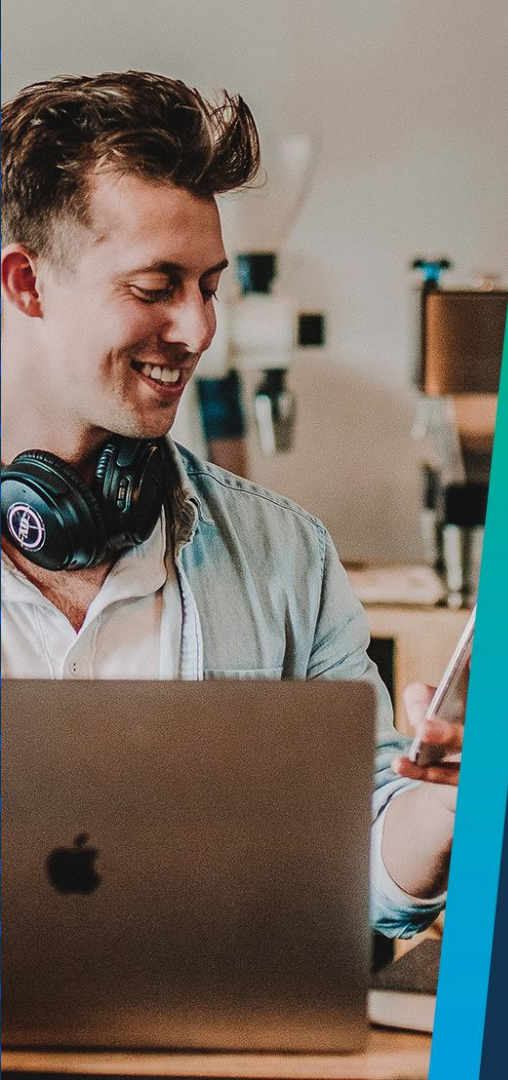
Depend on abstractions, not on concretions.



Hyperiondev

Q & A Section

Please use this time to ask any questions relating to the topic explained, should you have any



Hyperiondev

Thank you for joining us

**Take regular breaks.
Stay hydrated.
Avoid prolonged screen time.
Remember to have fun :)**