Software Engineering Bootcamp

Hyperiondev

Object-Oriented Design: SOLID Principles

Welcome

Your Lecturer for this session

Armand Le Roux

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://hyperionde.wufoo.com/forms/zsgv4m40ui4i0g/

Objectives

1. Recap on
    a. Context diagrams
    b. Sequence diagrams
2. SOLID Principles
    a. What they are.
    b. Why we use them.
    c. Different principles.
    d. How each principle is applied.

Github Repository - Lecture Examples

https://github.com/HyperionDevBootcamps/C4_SE_lecture_examples

Platform to create context & sequence diagrams

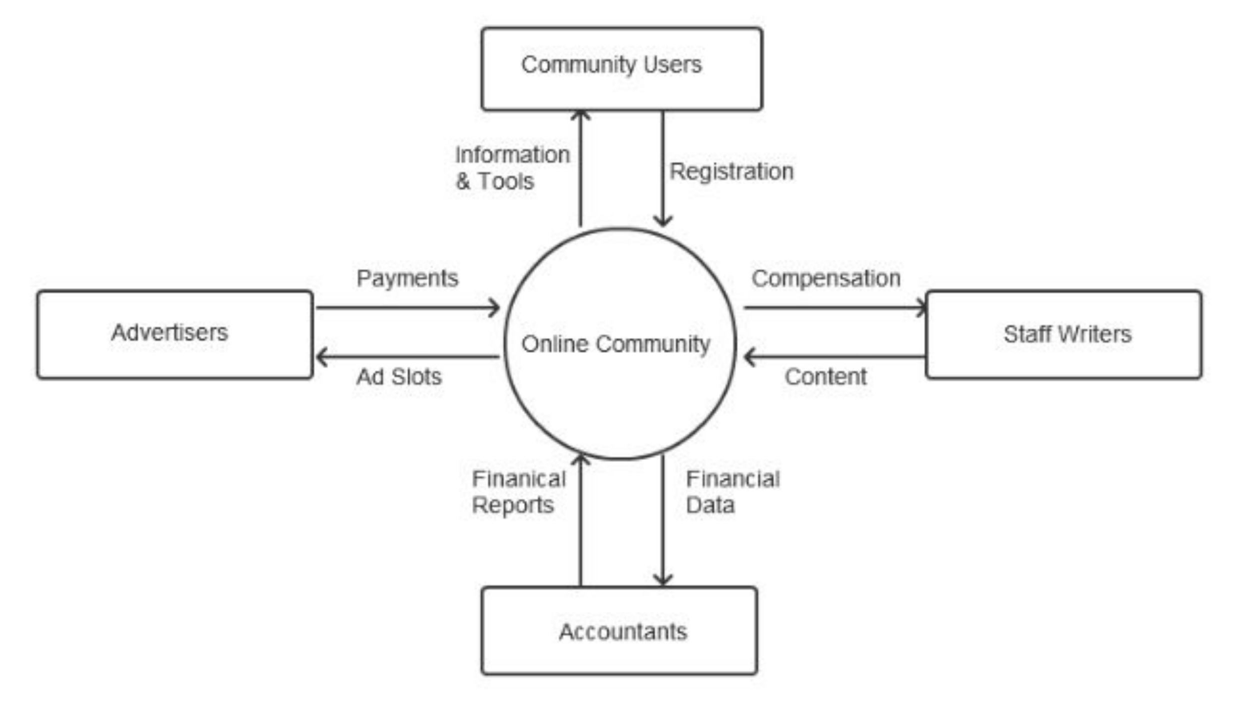
https://www.lucidchart.com/pages/uml-class-diagram

Recap

Context Diagrams

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

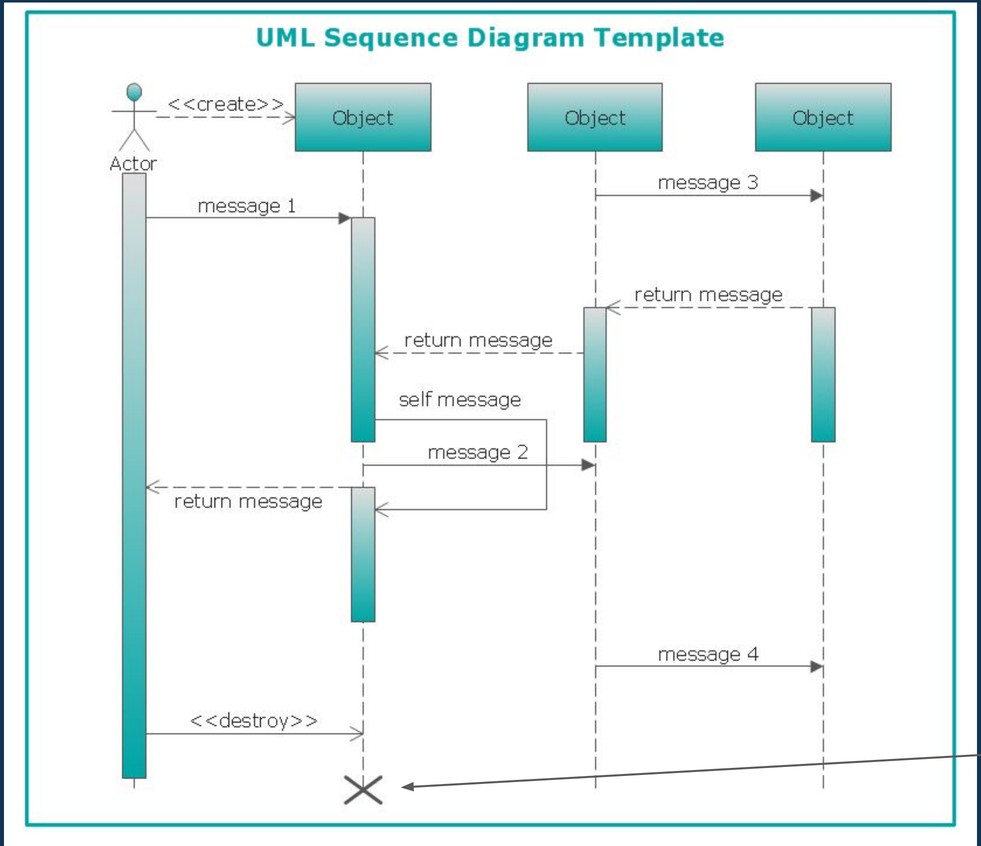
Context Diagrams



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.

Sequence Diagram



Delete message symbol: This message destroys an object.

Hyperiondev

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.

Where do the SOLID principles come from?

- Introduced by Robert J. Martin in a paper he wrote in 2000
- Also known as “Uncle Bob”

- Although he introduced the principles the SOLID acronym was introduced later by Michael Feathers.

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.

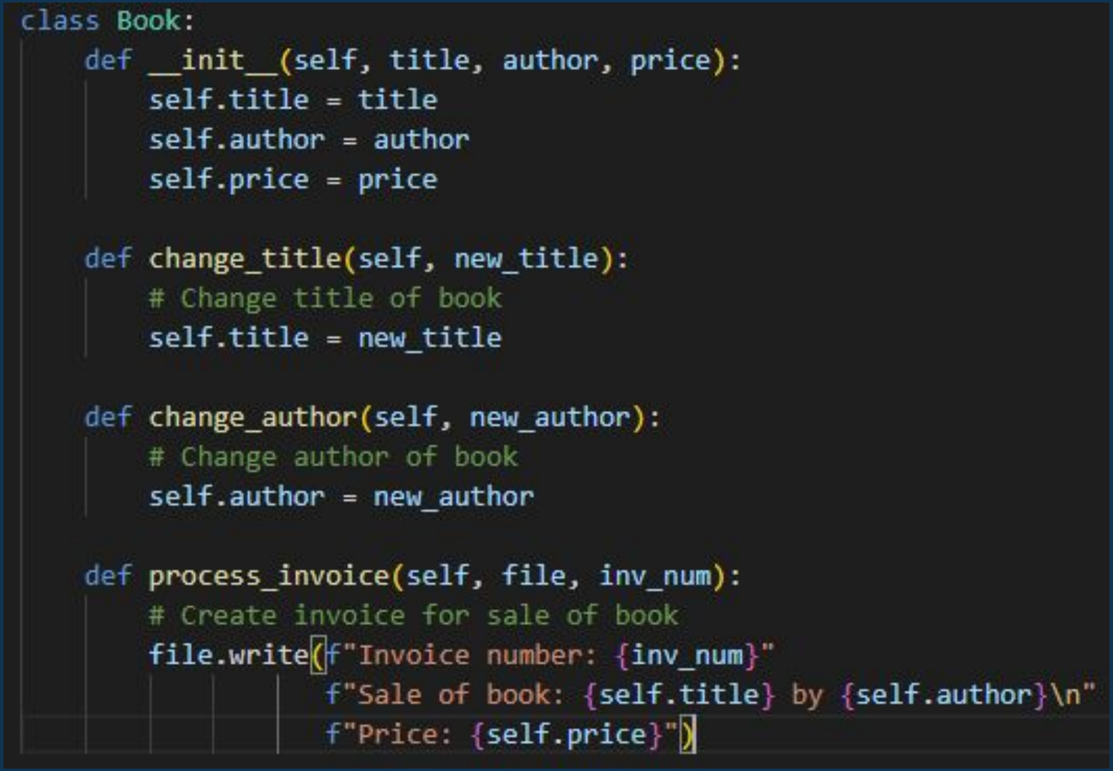
S - Single Responsibility Principle

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

- A class should only have a single purpose
- If a class has to many responsibilities it increases the possibility of bugs as changing one responsibility might affect the others.

S - Single Responsibility Principle

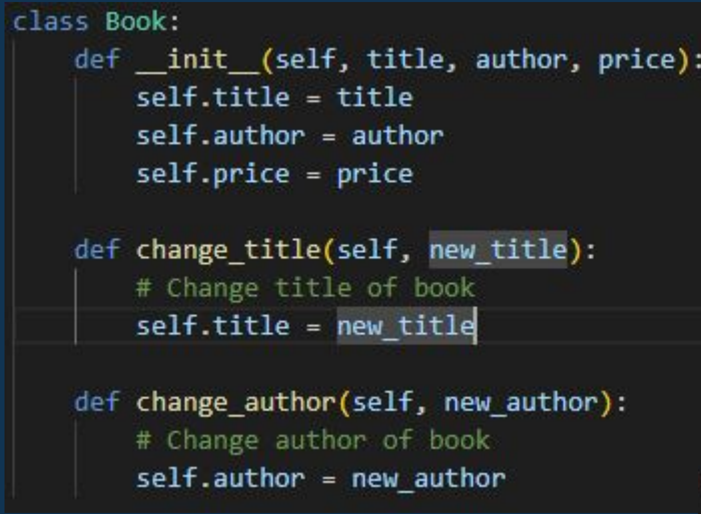
A class should have one, and only one, reason to change.



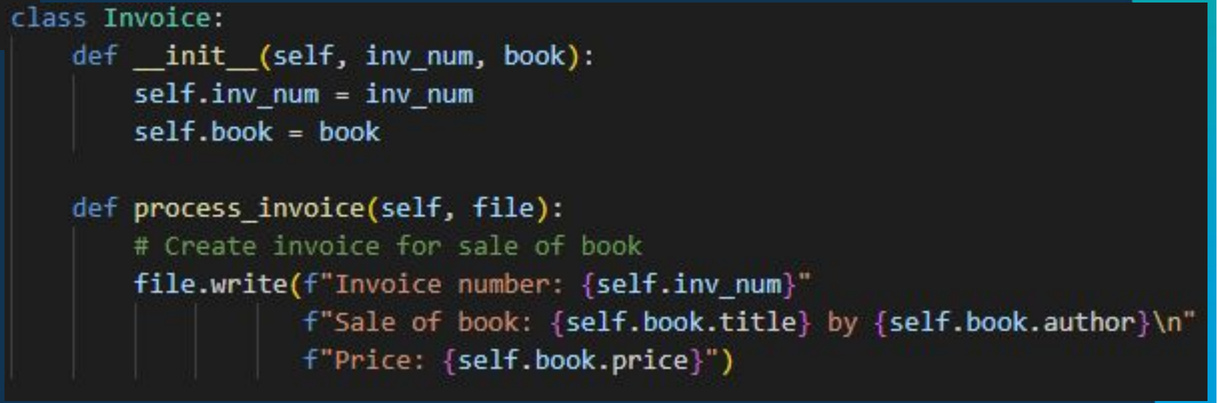
```python
class Book:
    def __init__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price

    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.



```python
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
```



```python
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

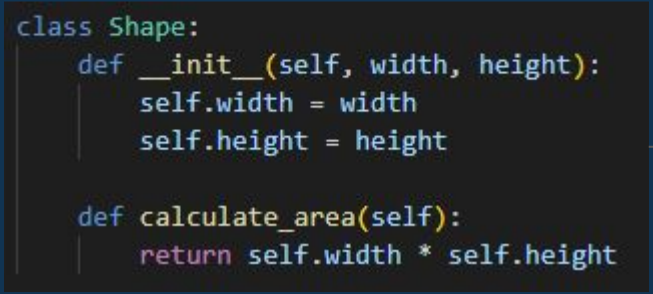
You should be able to extend a classes behavior, without modifying it.

- Open for extension, meaning that the class’s behavior can be extended.

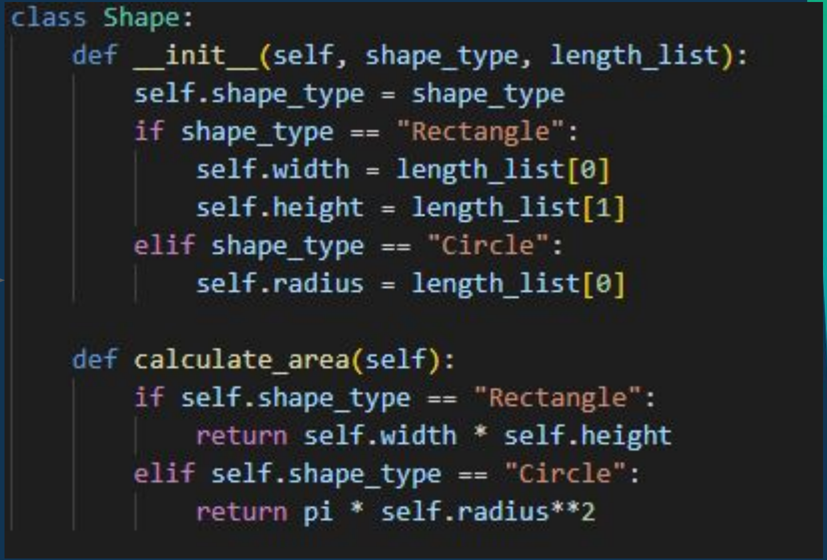
- Closed for modification, meaning that the source code is set and cannot be changed.

O - Opened Closed Principle

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



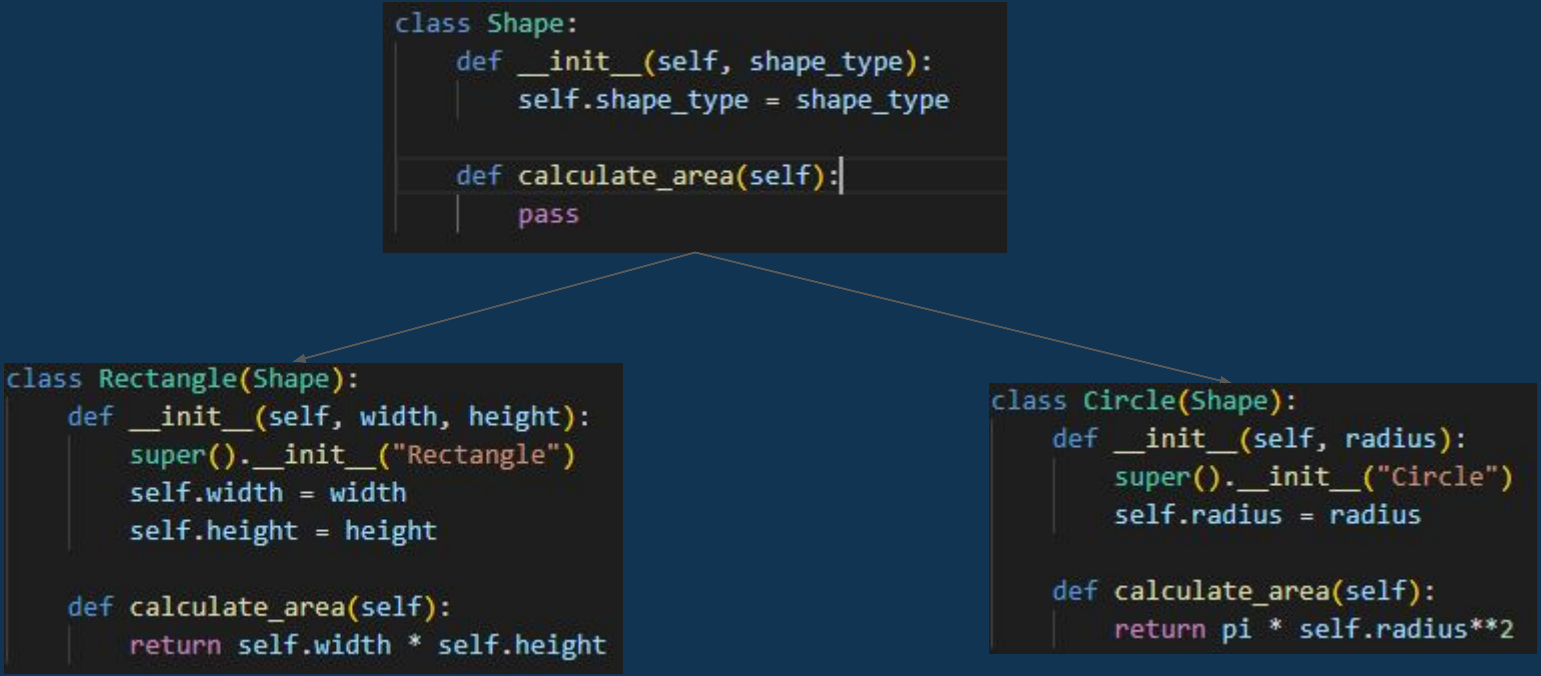
```
class Shape:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    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.



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

    def calculate_area(self):
        pass
```

```python
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
```

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

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

Hyperiondev

L - Liskov Substitute Principle

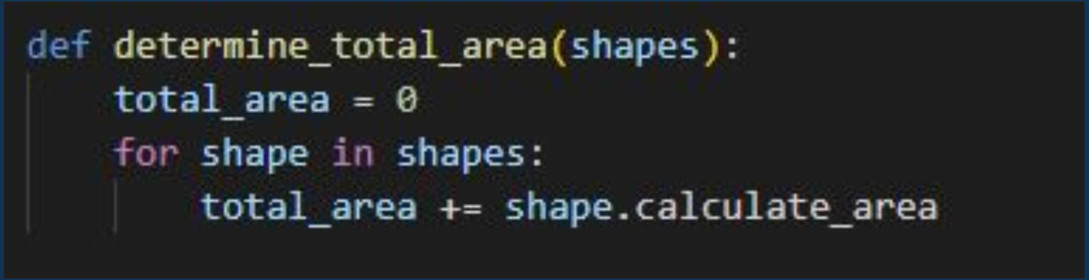
Derived classes must be substitutable for their base classes.

- The base class should be able to be replaced with a derived class without the code breaking.

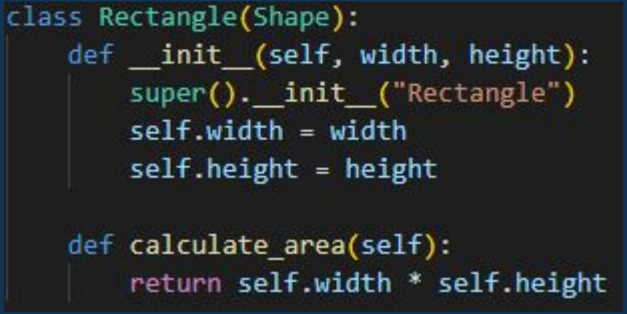
- This is an extension to the open-closed principle as it is also ensuring the derived classes extend the base class without changing behavior

L - Liskov Substitute Principle

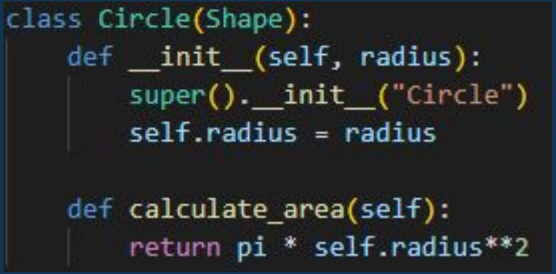
Derived classes must be substitutable for their base classes.



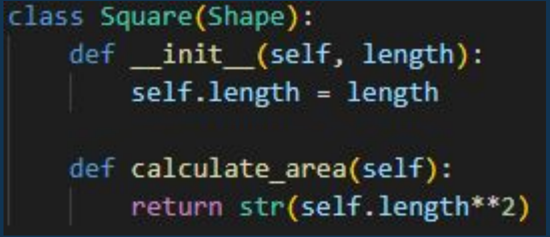
```python
def determine_total_area(shapes):
    total_area = 0
    for shape in shapes:
        total_area += shape.calculate_area
```



```python
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
```



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

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

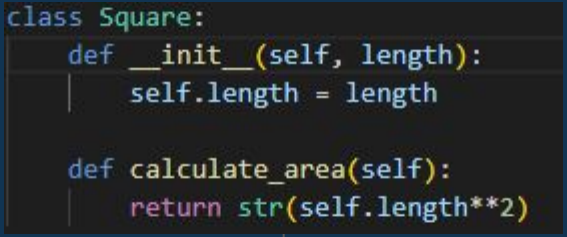


```python
class Square(Shape):
    def __init__(self, length):
        self.length = length

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

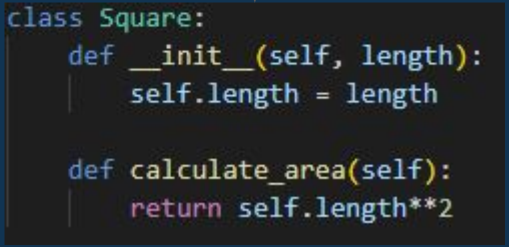
Hyperiondev

L - Liskov Substitute Principle

Derived classes must be substitutable for their base classes.



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

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



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

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

I - Interface Segregation Principle

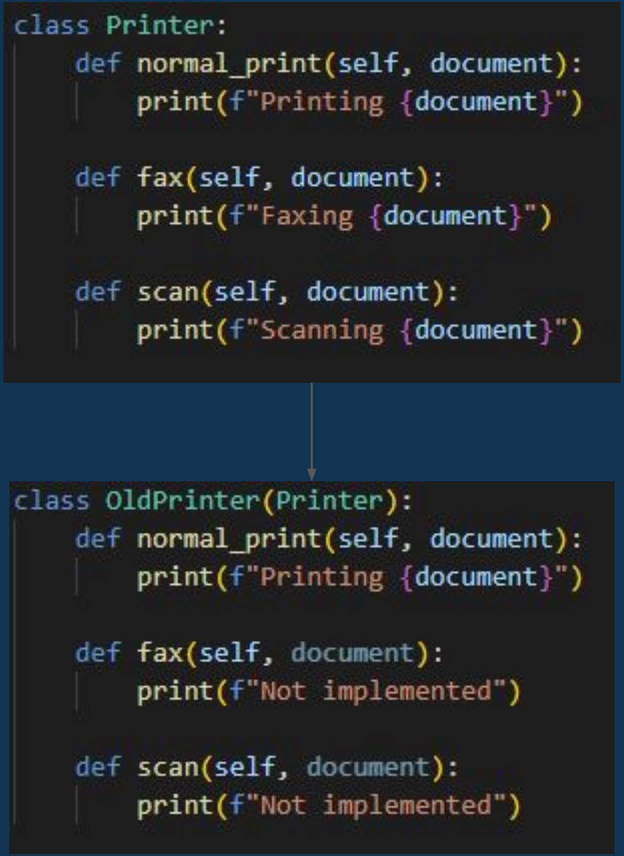
A class should not be forced to inherit a function in will not use.

- A derived class should only inherit functions it will be using and should not be forced to inherit any extra functions.

- This help us avoid the temptation of having one big, general-purpose class.

I - Interface Segregation Principle

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

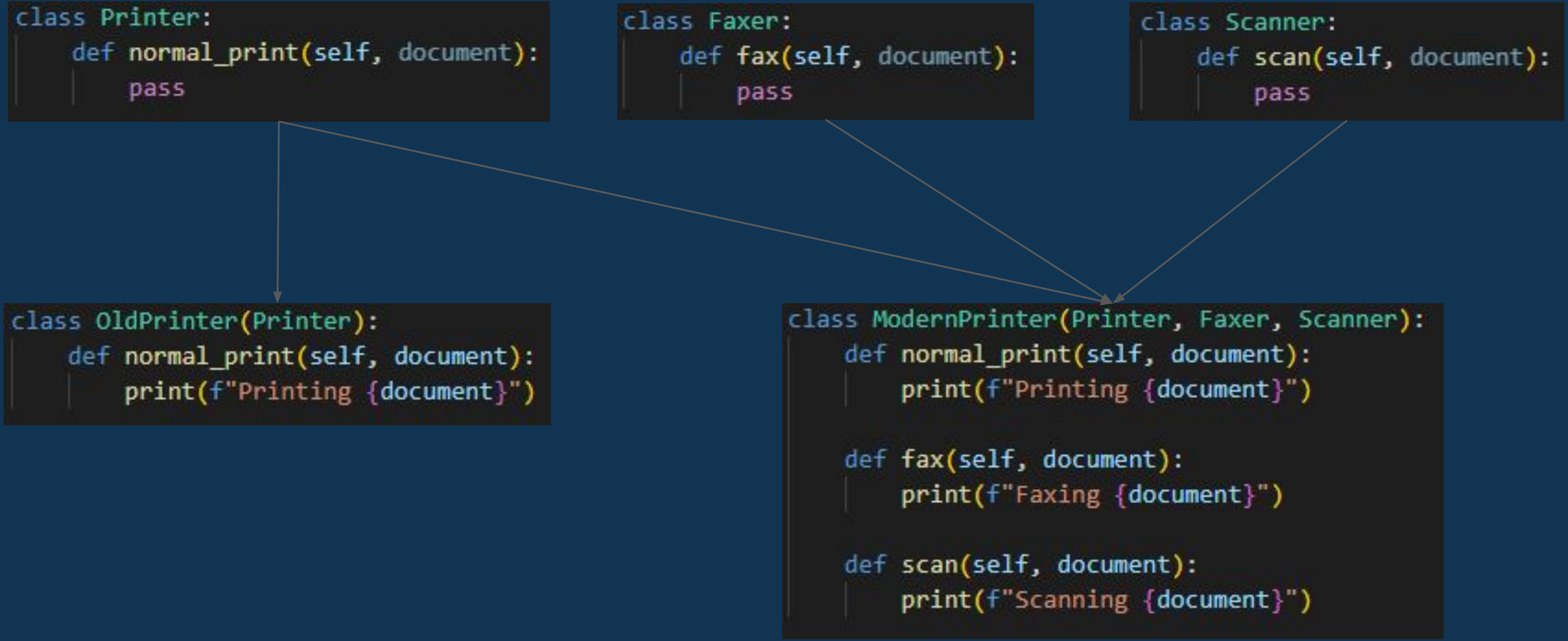


```
class Printer:
    def normal_print(self, document):
        print(f"Printing {document}")

    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 in will not use.



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

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

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

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

```python
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}")
```

Hyperiondev

D - Dependency Inversion Principle

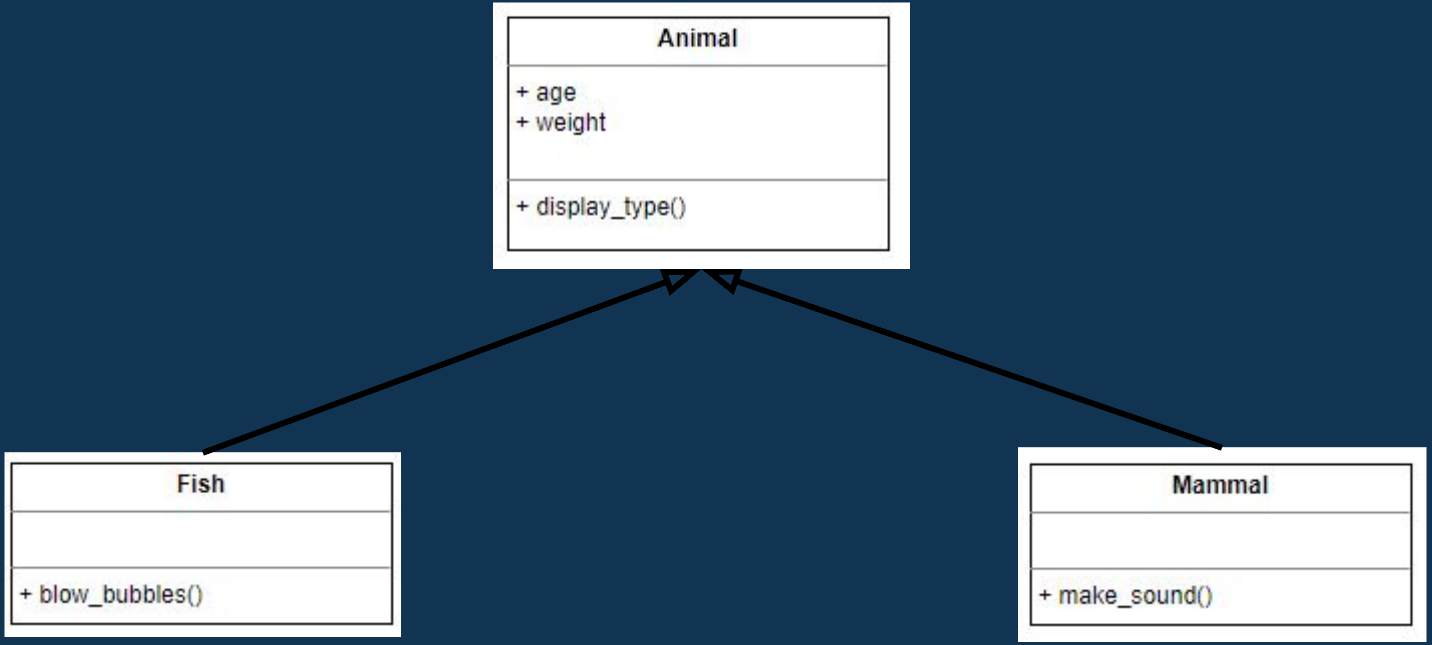
Depend on abstractions, not on concretions.

- When creating classes we try to rely on abstraction to stay focused on the classes and what they do rather than how they do it.

- We first determine what the class will do then we worry about the implementation later.

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 :)