Good Engineering Principles

General Engineering Principles

- **DRY (Don't Repeat Yourself)**:
  - Avoid duplication of code
    - Abstract common functionalities into reusable components or functions.
  - Emphasis on modularity and maintainability.

- **KISS (Keep It Simple, Stupid)**:
  - Complex solutions should be avoided in favor of simpler, more straightforward ones whenever possible.

- **YAGNI (You Aren't Gonna Need It)**:'
  - Only implement features that are necessary based on current requirements, and do not "over engineer"

- **Separation of Concerns (SoC)**:
  - Divide the software into distinct sections, with each section addressing a separate concern or responsibility.
  - Promotes modularity, maintainability, reusability, flexibility, and loose coupling between components

- **Single Source of Truth (SSOT)**:
  - Store each piece of information (like config files) in the system in a single location.
  - Reduces the risk of data inconsistencies.

- **Fail-Fast Principle**:
  - Identify and report errors as soon as they occur rather than allowing them to propagate and potentially cause more significant issues later on.
  - Helps in diagnosing and fixing problems quickly.

Dependency Injection Design Pattern

> If a class uses an object of a certain type, we are NOT also responsible for creating the object.

> Providing the objects that an object needs (its dependencies) instead of having it construct them itself.

Dependency injection (Inversion of Control Technique) is a principle that helps to decrease coupling and increase cohesion.

What is coupling and cohesion?

Coupling and cohesion are about how tough the components are tied.

- **High coupling**. If the coupling is high it's like using superglue or welding. No easy way to disassemble.
- **High cohesion**. High cohesion is like using screws. Quite easy to disassemble and re-assemble in a different way. It is an opposite to high coupling.

Cohesion often correlates with coupling. Higher cohesion usually leads to lower coupling and vice versa.

Low coupling brings flexibility. Your code becomes easier to change and test.

    - Dependency:
        - When an object type and its class is coupled (has-a relationship). Ex.
            - Class could depend on another class could it has an attribute of that type
            - Object of that type is passed as a parameter to a method
            - Class inherits from another class, the strongest dependency since it uses
        - **Flexibility**. The components are loosely coupled. You can easily extend or change the functionality of a system by combining the components in a different way. You even can do it on the fly.
        - **Testability**. Testing is easier because you can easily inject mocks instead of real objects that use API or database, etc.
        - **Clearness and maintainability**. Dependency injection helps you reveal the dependencies. Implicit becomes explicit. And "Explicit is better than implicit" (PEP 20 - The Zen of Python). You have all the components and dependencies defined explicitly in a container. This provides an overview and control of the application structure. It is easier to understand and change it.

Before:

```
import os



class ApiClient:
```

```
    def __init__(self) -> None:
        self.api_key = os.getenv("API_KEY")  # <-- dependency
        self.timeout = int(os.getenv("TIMEOUT"))  # <-- dependency


class Service:

    def __init__(self) -> None:
        self.api_client = ApiClient()  # <-- dependency


def main() -> None:
    service = Service()  # <-- dependency
    ...


if __name__ == "__main__":
    main()
```

After:

```
import os


class ApiClient:

    def __init__(self, api_key: str, timeout: int) -> None:
        self.api_key = api_key  # <-- dependency is injected
        self.timeout = timeout  # <-- dependency is injected


class Service:

    def __init__(self, api_client: ApiClient) -> None:
        self.api_client = api_client  # <-- dependency is injected


def main(service: Service) -> None:  # <-- dependency is injected
    ...
```

```
if __name__ == "__main__":
    main(
        service=Service(
            api_client=ApiClient(
                api_key=os.getenv("API_KEY"),
                timeout=int(os.getenv("TIMEOUT")),
            ),
        ),
    )
```

`ApiClient` is decoupled from knowing where the options come from. You can read a key and a timeout from a configuration file or even get them from a database.

`Service` is decoupled from the `ApiClient`. It does not create it anymore. You can provide a stub or other compatible object.

Function `main()` is decoupled from `Service`. It receives it as an argument.

Flexibility comes with a price.

Now you need to assemble and inject the objects like this:

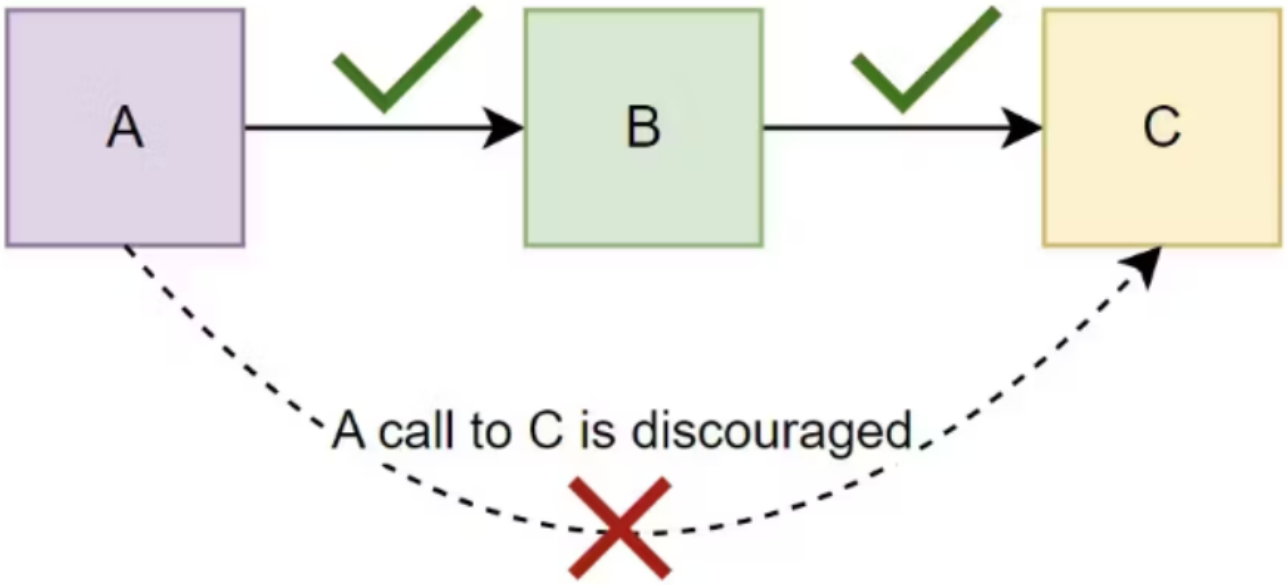
```
main(
    service=Service(
        api_client=ApiClient(
            api_key=os.getenv("API_KEY"),
            timeout=int(os.getenv("TIMEOUT")),
        ),
    ),
)
```

The assembly code might get duplicated and it’ll become harder to change the application structure.

Law of Demeter / Principle of Least Knowledge:

- Module should have limited knowledge about other modules.
- Encourages encapsulation and loose coupling by restricting the interaction between objects.
- Method in an object should only call:

- Itself
- Its parameters
- Object it creates
- Its direct component objects



Violates Law of Demeter due to nesting:

```
data = {
    "containers": [
        {"scoops": [{"flavor": "chocolate"}, {"flavor": "vanilla"}]},
        {"scoops": [{"flavor": "strawberry"}, {"flavor": "mint"}]}
    ]
}

flavor = data["containers"][0]["scoops"][0]["flavor"]
```

Fix:

1. Create a class that represents the container structure. And,
2. Provide methods to access our inner data.

```
class Scoop:
    def __init__(self, flavor:str):
        self.flavor = flavor

    def get_flavor(self):
```

```
        return self.flavor

class Container:
    def __init__(self):
        self.scoops = []

    def add_scoop(self,flavor:str):
        self.scoops.append(Scoop(flavor))

    def get_flavor_of_scoop(self, index:int):
        return self.scoops[index].get_flavor()

data = Container()
data.add_scoop("chocolate")
data.add_scoop("vanilla")
flavor = data.get_flavor_of_scoop(0)
```

Object Oriented Design Principles

Solid Principles

Single Responsibility Principle (SRP):

- A class should have only one reason to change.
- The following handles both read/write of files and encryption, which violates SRP

```
class FileManager:
    def __init__(self, file_path):
        self.file_path = file_path

    def read_file(self):
        pass

    def write_file(self, data):
        pass

    def encrypt_data(self, data):
        pass

    def decrypt_data(self, data):
        pass
```

- In this refactored version, the `FileManager` class now focuses solely on file management operations.

```
class FileManager
    def __init__(self, file_path):
        self.file_path = file_path

    def read_file(self):
        pass

    def write_file(self, data):
        pass



class DataEncryptor:
    def encrypt_data(self, data):
        pass

    def decrypt_data(self, data):
        pass
```

Open/Closed Principle (OCP):

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

The function animal_sound does not conform because it cannot be closed against new kinds of animals. If we add a new animal, Snake, We have to modify the animal_sound function. For every new animal, a new logic is added to the animal_sound function.

When your application grows in complexity, the if statement would be repeated in the animal_sound function each time a new animal is added, all over the application. We want to decrease amount of if else statements.

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

    def get_name(self) -> str:
        pass


animals = [
    Animal('lion'),
    Animal('mouse')
]


def animal_sound(animals: list):
    for animal in animals:
        if animal.name == 'lion':
            print('roar')
```

```
        elif animal.name == 'mouse':
            print('squeak')



animal_sound(animals):
```

The Animal class has been enhanced with the addition of the **make_sound** method. Each animal class extends the Animal class and provides its own implementation of the **make_sound** method, defining how it produces its unique sound.

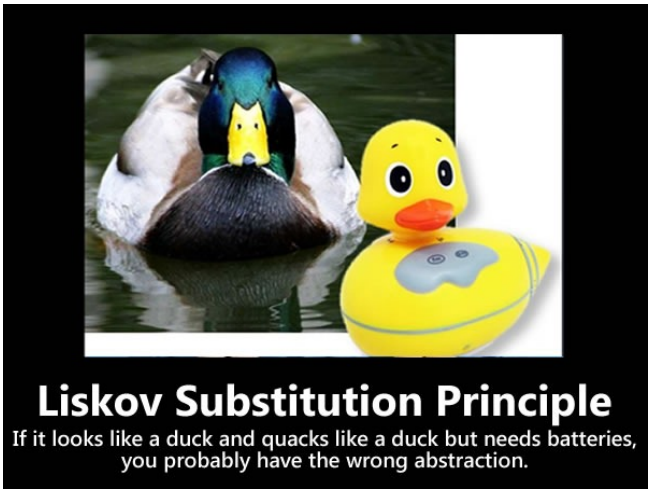
In the **animal_sound** function, we iterate through the array of animals and simply invoke their respective **make_sound** methods. The **animal_sound** function remains unchanged even when new animals are introduced. We only need to include the new animal in the animal array.

This adherence to the Open-Closed Principle ensures that the code is extensible without requiring modifications to existing code.

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

    def get_name(self) -> str:
        pass

    def make_sound(self):
        pass

class Lion(Animal):
    def make_sound(self):
        return 'roar'

class Mouse(Animal):
    def make_sound(self):
        return 'squeak'

class Snake(Animal):
    def make_sound(self):
        return 'hiss'

def animal_sound(animals: list):
    for animal in animals:
```

```
        print(animal.make_sound())


animal_sound(animals):
```

Liskov Substitution Principle:



Make your subclasses behave like their base classes without breaking anyone’s expectations when they call the same methods.

This implementation violates the Liskov Substitution Principle because you can't seamlessly replace instances of Rectangle with their Square counterparts.

Imagine someone expects a rectangle object in their code. Naturally, they would assume that it exhibits the behavior of a rectangle, including separate **width** and **height** attributes. Unfortunately, the Square class in your codebase violates this assumption by altering the expected behavior defined by the object's interface.

To apply the Liskov Substitution Principle, introduce a base Shape class and make both Rectangle and Square inherit from it:

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

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

class Square(Rectangle)
    def __init__(self, side):
        super().__init__(side, side)
```

```
    def __setattr__(self, key, value):
        super().__setattr__(key, value)
        if key in ("width", "height"):
            self.__dict__["width"] = value
            self.__dict__["height"] = value:
```

Introducing a common base class (Shape), ensures that objects of different subclasses can be seamlessly interchanged wherever the superclass is expected. Both Rectangle and Square are now siblings, each with their own set of attributes, initializer methods, and potentially more separate behaviors. The only shared aspect between them is the ability to calculate their respective areas

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

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

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

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def calculate_area(self):
        return self.side ** 2d
```

With this implementation in place, you can use the `Shape` type interchangeably with its `Square` and `Rectangle` subtypes when you only care about their common behavior:

```
from shapes_lsp import Rectangle, Square

def get_total_area(shapes):
  return sum(shape.calculate_area() for shape in shapes)
```

```
get_total_area([Rectangle(10, 5), Square(5)])
75
```

Three Violation of LSK Principle:

1. **Violating the Contract**: Derived classes should not violate the contracts defined by the base class. If a derived class modifies or ignores the requirements specified by the base class, it can lead to inconsistencies and unexpected behaviors.

Every Bird subclass should be able to use the methods of the abstract Bird class.

```
class Bird:
    def fly(self):
        pass

class Ostrich(Bird):
    def fly(self):
        raise NotImplementedError("Ostriches cannot fly!")

bird = Bird()
bird.fly()  # Output: (no implementation)

ostrich = Ostrich()
ostrich.fly()  # Raises NotImplementedError
```

A better solution is to further abstract the method from "fly" to "move," allowing the Ostrich to run when move method is called, and the other birds (ducks) can fly

2. Overriding Essential Behavior: Overriding crucial methods in a way that changes the fundamental behavior defined by the base class can break the LSP. Derived classes should extend or specialize the behavior rather than completely altering it.

```
class Vehicle:
    def start_engine(self):
        print("Engine started.")

class ElectricVehicle(Vehicle):
    def start_engine(self):
        print("Engine cannot be started for an electric vehicle.")

vehicle = Vehicle()
```

```
vehicle.start_engine()  # Output: Engine started.

electric_vehicle = ElectricVehicle()
electric_vehicle.start_engine()  # Output: Engine cannot be started for an electric vehicle.
```

The solutions is to change method name from "start_engine" to "start," since electric and gas vehicles both have to start.

3. **Tight Coupling with Implementation Details**: Relying heavily on implementation details of derived classes in client code can lead to tight coupling and hinder the flexibility of the LSP. Aim for loose coupling and focus on interacting with objects through their defined interfaces.

```
class DatabaseConnector:
    def connect(self):
        pass

class MySQLConnector(DatabaseConnector):
    def connect(self):
        print("Connecting to MySQL database...")

class PostgreSQLConnector(DatabaseConnector):
    def connect(self):
        print("Connecting to PostgreSQL database...")

# Tight coupling with concrete class instantiation
connector = MySQLConnector()  # Specific to MySQL

connector.connect()  # Output: Connecting to MySQL database...
```

In this example, the client code tightly couples with the concrete class `MySQLConnector` when instantiating the `connector` object. This direct dependency on the specific class limits the flexibility to switch to other database connectors, such as `PostgreSQLConnector`. To follow the Liskov Substitution Principle, it is better to interact with objects through their common base class interface (`DatabaseConnector` in this case) and use polymorphism to instantiate objects based on runtime configuration or user input. The following fixes the issue:

```
class DatabaseConnector:
    def connect(self):
        pass
```

```
class MySQLConnector(DatabaseConnector):
    def connect(self):
        print("Connecting to MySQL database...")

class PostgreSQLConnector(DatabaseConnector):
    def connect(self):
        print("Connecting to PostgreSQL database...")

# Dependency injection with a generic database connector
def use_database_connector(connector):
    connector.connect()

# Usage example
if __name__ == "__main__":
    mysql_connector = MySQLConnector()
    postgresql_connector = PostgreSQLConnector()

    use_database_connector(mysql_connector)  # Output: Connecting to MySQL database...
    use_database_connector(postgresql_connector)  # Output: Connecting to PostgreSQL database...
```

Interface Segregation Principle (ICP)

The Interface Segregation Principle revolves around the idea that clients should not be forced to rely on methods they do not use. To achieve this, the principle suggests creating specific interfaces or classes tailored to the needs of individual clients.

In this example, the base class **Printer** defines an interface that its subclasses are required to implement. However, the **OldPrinter** subclass doesn't utilize the **fax()** and **scan()** methods because it lacks support for these functionalities.

Unfortunately, this design violates the ISP as it forces **OldPrinter** to expose an interface that it neither implements nor requires.

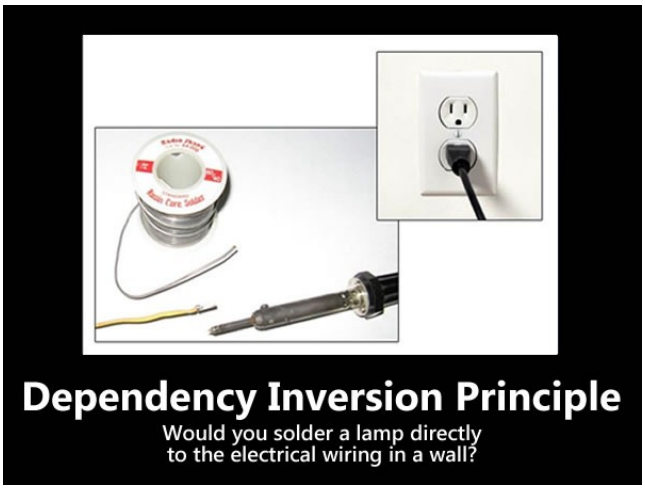
```
from abc import ABC, abstractmetho

class Printer(ABC):
    @abstractmethod
    def print(self, document):
        pass

    @abstractmethod
    def fax(self, document):
        pass

    @abstractmethod
    def scan(self, document):
        pass

class OldPrinter(Printer):
    def print(self, document):
        print(f"Printing {document} in black and white...")

    def fax(self, document):
        raise NotImplementedError("Fax functionality not supported")

    def scan(self, document):
        raise NotImplementedError("Scan functionality not supported")

class ModernPrinter(Printer):
    def print(self, document):
        print(f"Printing {document} in color...")

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

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

In this revised design, the base classes—**Printer**, **Fax**, and **Scanner**—provide distinct interfaces, each responsible for a single functionality. The **OldPrinter** class only inherits the **Printer** interface, ensuring that it doesn't have any unused methods. On the other hand, the **NewPrinter**

class inherits from all the interfaces, incorporating the complete set of functionalities. This segregation of the **Printer** interface enables the creation of various machines with different combinations of functionalities, enhancing flexibility and extensibility.

```
from abc import ABC, abstractmetho

class Printer(ABC):
    @abstractmethod
    def print(self, document):
        pass

class Fax(ABC):
    @abstractmethod
    def fax(self, document):
        pass

class Scanner(ABC):
    @abstractmethod
    def scan(self, document):
        pass

class OldPrinter(Printer):
    def print(self, document):
        print(f"Printing {document} in black and white...")

class NewPrinter(Printer, Fax, Scanner):
    def print(self, document):
        print(f"Printing {document} in color...")

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

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

Dependency Inversion Principle (DIP)



The Dependency Inversion Principle focuses on managing dependencies between classes. It states that

- Dependencies should be based on *abstractions* rather than concrete implementations.
- Abstractions should not rely on implementation details; instead, details should depend on abstractions.

Without dependency injection, there is no dependency inversion

```
class PaymentProcessor
    def process_payment(self, payment):
        if payment.method == 'credit_card':
            self.charge_credit_card(payment)
        elif payment.method == 'paypal':
            self.process_paypal_payment(payment)



    def charge_credit_card(self, payment):
        # Charge credit card



    def process_paypal_payment(self, payment):
        # Process PayPal payment
```

In this updated code, we introduced the PaymentMethod abstract base class, which declares the process_payment() method. The PaymentProcessor class now depends on the PaymentMethod abstraction through its constructor, rather than directly handling the payment logic. The specific payment methods, such as CreditCardPayment and PayPalPayment, implement the PaymentMethod interface and provide their own implementation for the process_payment() method.

By following this structure, the PaymentProcessor class is decoupled from the specific payment methods, and it depends on the PaymentMethod abstraction. This design allows for greater flexibility and easier extensibility, as new payment methods can be added by creating new classes that implement the PaymentMethod interface without modifying the PaymentProcessor class.

```
from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    @abstractmethod
    def process_payment(self, payment):
        pass

class PaymentProcessor:
    def __init__(self, payment_method):
        self.payment_method = payment_method

    def process_payment(self, payment):
        self.payment_method.process_payment(payment)

class CreditCardPayment(PaymentMethod):
    def process_payment(self, payment):
        # Code to charge credit card

class PayPalPayment(PaymentMethod):
    def process_payment(self, payment):
        # Code to process PayPal payment
```

Composition Over Inheritance:

Prefer composition (building objects by assembling smaller, reusable components) over inheritance (creating new classes by extending existing ones). This approach leads to more flexible and maintainable code.

- **Inheritance reduces encapsulation:** we want our classes and modules to be loosely coupled to the rest of the codebase.
  - A child class, instead, is strongly coupled to its parent. When a parent changes, the change will ripple through all of its children and might break the codebase.
- **Testability:** Reduced encapsulation results in classes being harder to test.

Composition involves using other classes to build more complex classes, there is no parent/child relationship exists in this case. Objects are composed of other objects, through a **has-a**

relationship, not a **belongs-to** relationship. This means that we can **combine** other objects to reach the behavior we would like, thus avoid the subclasses explosion problem. In Python, we can leverage a couple of mechanisms to achieve composition.

Use Inheritance sparingly. Abstract Base Class or Protocol are good examples of clean inheritance.

Make Classes Data-Oriented or Behavior-Oriented

Use @dataclass for data-oriented classes, and consider just having a separate module with functions for Behavior-Oriented classes

---

Revision #20
Created 14 February 2024 23:51:26 by victor
Updated 31 March 2024 04:12:53 by victor