SOLID principles - SRP and OCP

Key terms

SOLID principles

> SOLID is a mnemonic acronym for five design principles intended to make object-oriented designs more understandable, flexible, and maintainable.

Single responsibility principle

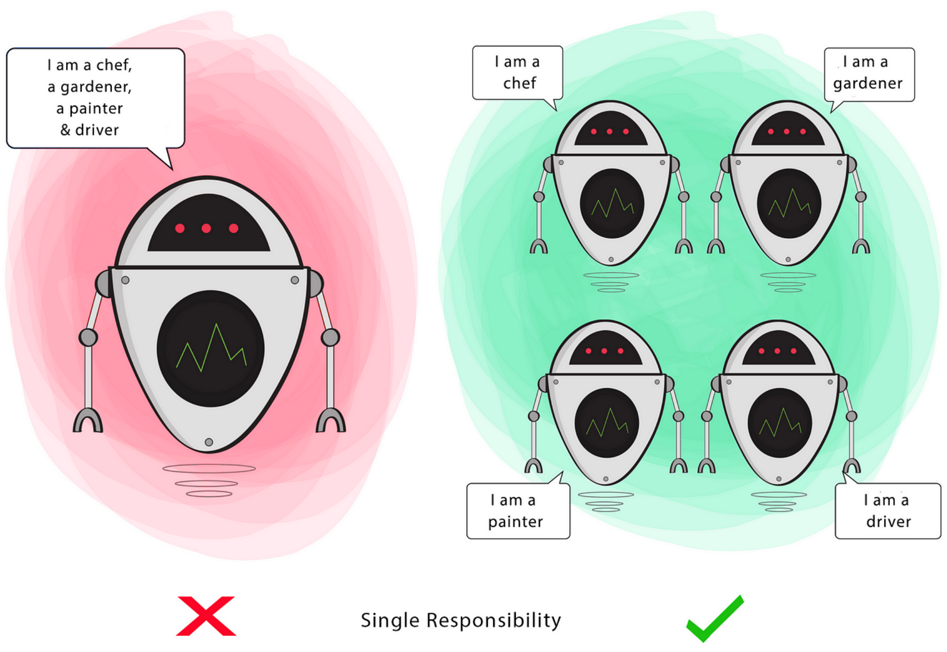
> There should never be more than one reason for a class/code unit to change. Every class should have only one responsibility.

Open/closed principle

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

Single responsibility principle

> When designing our classes, we should aim to put related features together, so whenever they tend to change they change for the same reason. And we should try to separate features if they will change for different reasons.



The Single Responsibility Principle states that a class should do one thing, and therefore it should have only a single reason to change. Only one potential change (database logic, logging logic, and so on.) in the software's specification should be able to affect the specification of the class.

This means that if a class is a data container, like a Book class or a Student class, and it has some fields regarding that entity, it should change only when we change the data model.

Case study - Design a bird

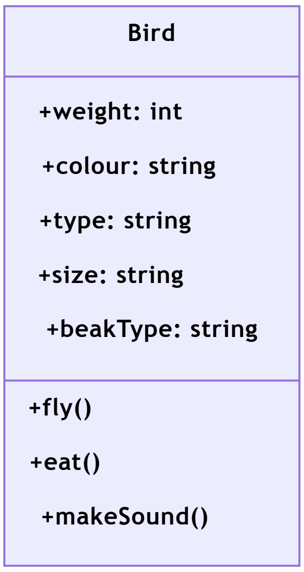
To understand the SOLID principles, let us take the help of a bird. A bird is a living creature that can fly, eat, and make a sound. How can we design a bird?

The simplest solution would be to create a `Bird` class with different attributes and methods. A bird could have the following attributes:

- Weight
- Colour
- Type
- Size
- BeakType

A bird would also exhibit the following behaviours:

- Fly
- Eat
- Make a sound



The `Bird` class would look as follows:

```
from dataclasses import dataclass

@dataclass
class Bird:
    weight: int
    colour: str
    bird_type: str
    size: str
    beak_type: str

    def fly(self) -> None:
        pass

    def eat(self) -> None:
        pass

    def make_sound(self) -> None:
        pass
```

- We are using a dataclass to create a class with attributes. A dataclass is a class that is typically used to store data.
- It provides a concise way to create classes that store data. The `@dataclass` decorator automatically adds special methods such as `__init__`, `__repr__`, `__eq__`, etc. to the class.
- You can read more about dataclasses here (https://docs.python.org/3/library/dataclasses.html).

In order to understand the design further, let us try to implement the `fly` method. Since each bird has a different method of flying, we would have to implement conditional statements to check the type of the bird and then call the appropriate method.

```
def fly(self) -> None:
    if self.bird_type == "eagle":
        fly_like_eagle()
    elif self.bird_type == "penguin":
        fly_like_penguin()
    elif self.bird_type == "parrot":
        fly_like_parrot()
```

The above code exhibits the following problems:

- `Readability` - The code is not readable. It is difficult to understand what the code is doing.
- `Testing` - It is difficult to test the code. We would have to test each type of bird separately.
- `Reusability` - The code is not reusable. If we want to re-use the code of specific type of bird, we would have to change the above code.
- `Parallel development` - The code is not parallel development friendly. If multiple developers are working on the same code, they could face merge conflicts.
- `Multiple reasons to change` - The code has multiple reasons to change. If we want to change the way a type of bird flies, we would have to change the code in the `fly` method.

Reasons to follow SRP

Apart from overcoming the problems mentioned above, there are other reasons to follow the SRP:

- Maintainability - Smaller, well-organized classes are easier to search than monolithic ones.
- Ease of testing – A class with one responsibility will have far fewer test cases.
- Lower coupling – Less functionality in a single class will have fewer dependencies.

How/Where to spot violations of SRP?

- A method with multiple `if-else` statements. An example would be the `fly` method of the `Bird` class. This is not a silver bullet, but it is a good indicator. There can be other reasons for multiple `if-else` statements such as business logic e.g. calculating the tax, checking access rights, etc.
- `Monster methods` or `God classes` - Methods that are too long and doing much more than the name suggests. This is a good indicator of a violation of SRP.

```
def save_to_database(self) -> None:
    # Connect to database
    db = Database(url="localhost", port=3306)
    db.connect()

    connection = db.create_connection()
    connection.set_database("users")

    # Create a query
    query = "SELECT * FROM users"

    # Execute the query
    result = db.execute(query)

    # Create a user defined object from the result
    user = [User(row) for row in result]

    # Close the connection
    db.close()
```

The above method is doing much more than the name suggests. It is connecting to the database, creating a query, executing the query, creating a user defined object, and closing the connection. This method violates the SRP. It should be split into multiple methods such as `connect_to_database` , `create_query` , `execute_query` , `create_user_object` , and `close_connection` .

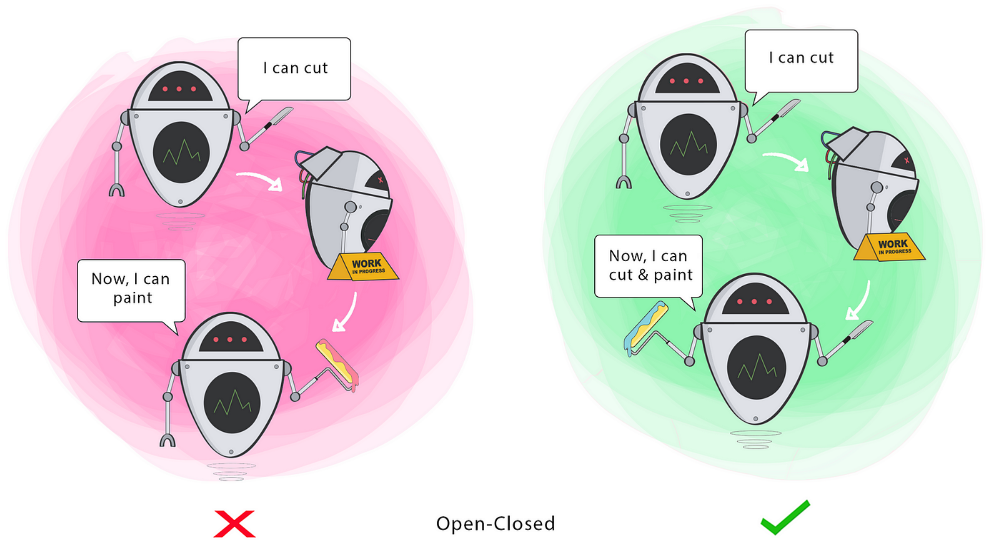
- `Utility classes` - Utility classes are classes that contain only static methods which are used to perform some utility functions. Have a look at the utility package of Java here (https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html). There is just way too many responsibilities of this package.

Open/closed principle

We identified a bunch of problems with the `Bird` class. Let us see the fly method again to spot another problem.

```
def fly(self) -> None:
    if self.bird_type == "eagle":
        fly_like_eagle()
    elif self.bird_type == "penguin":
        fly_like_penguin()
    elif self.bird_type == "parrot":
        fly_like_parrot()
```

In the above code, we are checking the type of the bird and then calling the appropriate method. If we want to add a new type of bird, we would have to change the code in the `fly` method. This is a violation of the Open/Closed Principle.



The Open/Closed Principle states that a class should be open for extension but closed for modification. This means that we should be able to add new functionality to the class without changing the existing code. To add a new feature, we should ideally create a new class or method and have very little or no changes in the existing code. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application. We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

- A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
- A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

Abstract classes and interfaces

An abstract class in Python is a class that contains one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. Subclasses of an abstract class in Python are not required to implement abstract methods of the parent class. An abstract class can have both abstract and non-abstract (concrete) methods.

- In Python, an abstract class is created by inheriting the `ABC` class from the `abc` module.
- The `ABC` class is also known as the `metaclass` since it is used to create classes.
- The `ABC` class is an abstract class that provides the infrastructure for defining abstract methods.
- The `abstractmethod` decorator is used to define abstract methods.

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class Animal(ABC):
    name: str
    age: int

    @abstractmethod
    def make_sound(self):
        pass

    def eat(self):
        print("Eating...")
```

Interface

An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a class. A Java interface contains static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction.

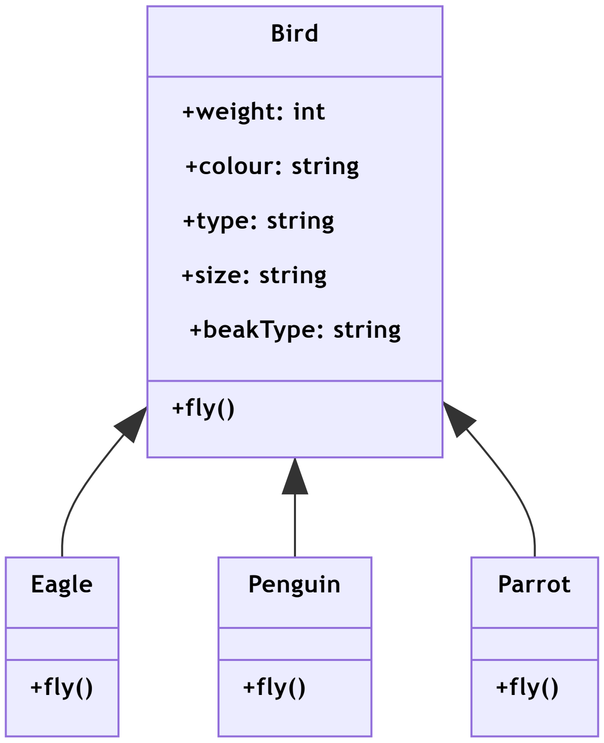
However, Python does not have interfaces.

Why does Python not have interfaces?

- Python, like many modern programming languages, does not enforce explicit interfaces because it follows a principle called “Duck typing.”
- Duck typing means that the type or the class of an object is less important than the methods it defines. If an object has a particular behavior (methods), it’s considered to be of a certain type, regardless of its actual class or inheritance hierarchy.
- Python supports multiple inheritance, which allows a class to inherit from more than one base class. This approach allows for a more dynamic and adaptable system, where classes can inherit methods and behaviors from multiple sources.

Fixing OCP violation in the `Bird` class

Now that we have learnt about abstract classes and interfaces, let us fix the SRP and OCP violation in the `Bird` class. In order to fix the SRP violations, we would consider having a parent class `Bird` and child classes `Eagle` , `Penguin` , and `Parrot` . Since, different birds have the same attributes and behaviours, we would want to use classes. An instance of the `Bird` class does not make sense, hence we would use an abstract class. We can't use an interface since we would want to have instance variables. We would also want to have a fixed contract for the subclasses to implement the common functionalities. Hence, we would use an abstract class.
Now, our `Bird` class would look like this.



The `Bird` class would look as follows:

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class Bird(ABC):
    weight: int
    colour: str
    bird_type: str
    size: str
    beak_type: str

    @abstractmethod
    def fly(self) -> None:
        pass
```

The `Eagle` class would look as follows:

```python
from bird import Bird

class Eagle(Bird):
    def fly(self) -> None:
        print("Flying like an eagle...")
```

Reading List

- SOLID vs CUPID vs GRASP (https://www.boldare.com/blog/solid-cupid-grasp-principles-object-oriented-design/#what-is-solid-and-why-is-it-more-than-just-an-acronym?-solid-vs.-cupid---is-the-new-always-better?)
- Principles of OOD (http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod)
- SRP and Python (https://sobolevn.me/2019/03/enforcing-srp)