OOP / SOLID

☰ Tags	Python Language and Ecosystem
≡ Description	SOLID, inheritance, polymorphism, encapsulation, abstraction, composition, MRO, mixins, dataclasses
≡ Topic	OOP

Sources:

- https://realpython.com/python-classes/
- https://realpython.com/solid-principles-python/
- https://realpython.com/python-interface/
- https://realpython.com/inheritance-composition-python/
- https://realpython.com/instance-class-and-static-methods-demystified/
- https://realpython.com/python-super/
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- https://medium.com/@m.nusret.ozates/solid-principles-with-python-245e45f9b1f8

SOLID in Python

1. **S**ingle-responsibility principle (SRP)
2. **O**pen–closed principle (OCP)
3. **L**iskov substitution principle (LSP)
4. **I**nterface segregation principle (ISP)
5. **D**ependency inversion principle (DIP)

SRP

> *A class should have only one reason to change. - Robert C. Martin*

> *Birlikte iş yapmak üzere toplanan kişiler çok olursa her kafadan bir ses çıkar, anlaşmazlıklar belirir, iş yapmak güçleşir [**Turkish**]*

> *If there are many people gathering together to do business, everyone will have a voice, disagreements will arise, and doing business will become difficult. [**Translate**]*

This means that a class should have only one **responsibility**, as expressed through its methods. If a class takes care of more than one task, then you should separate those tasks into separate classes.

A design that fails to conform to the SRP would look like this:

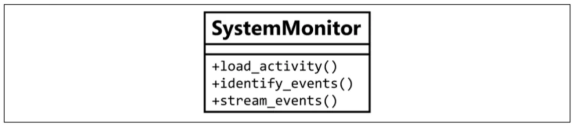

Figure 4.1: A class with too many responsibilities

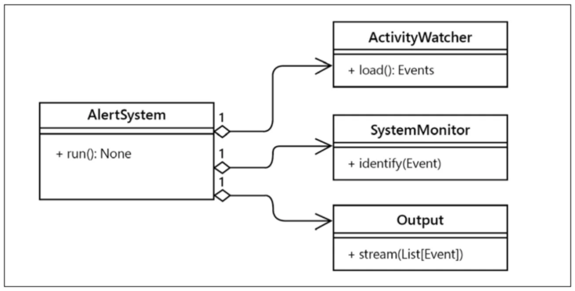

Figure 4.2: Distributing responsibilities throughout classes

BAD

```
from pathlib import Path
from zipfile import ZipFile

class FileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def read(self, encoding="utf-8"):
        return self.path.read_text(encoding)

    def write(self, data, encoding="utf-8"):
        self.path.write_text(data, encoding)

    def compress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="w") as archive:
            archive.write(self.path)

    def decompress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="r") as archive:
            archive.extractall()
```

GOOD

```
from pathlib import Path
from zipfile import ZipFile

class FileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def read(self, encoding="utf-8"):
        return self.path.read_text(encoding)

    def write(self, data, encoding="utf-8"):
        self.path.write_text(data, encoding)

class ZipFileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def compress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="w") as archive:
            archive.write(self.path)

    def decompress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="r") as archive:
            archive.extractall()
```

The idea of **responsibility** in this context can be subjective. It's not about the number of methods, but the core task your class handles. Despite the subjectivity, strive to use the Single Responsibility Principle (SRP).

OCP

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

BAD

```
class Employee:

    def __init__(self, name: str, salary: str):
        self.name = name
        self.salary = salary


class Tester(Employee):

    def __init__(self, name: str, salary: str):
        super().__init__(name, salary)

    def test(self):
        print("{} is testing".format(self.name))

class Developer(Employee):

    def __init__(self, name: str, salary: str):
        super().__init__(name, salary)

    def develop(self):
        print("{} is developing".format(self.name))


class Company:

    def __init__(self, name: str):
        self.name = name

    def work(self, employee):
        if isinstance(employee, Developer):
            employee.develop()
        elif isinstance(employee, Tester):
            employee.test()
        else:
            raise Exception("Unknown employee")
```

If we need to add new employee ? Then we need to add elif in work and so on.

GOOD

```
from abc import ABC, abstractmethod


class Employee(ABC):
    def __init__(self, name: str, salary: str):
        self.name = name
        self.salary = salary

    @abstractmethod
    def work(self):
        pass


class Tester(Employee):

    def __init__(self, name: str, salary: str):
        super().__init__(name, salary)

    def test(self):
        print("{} is testing".format(self.name))

    def work(self):
        self.test()

class Developer(Employee):

    def __init__(self, name: str, salary: str):
        super().__init__(name, salary)

    def develop(self):
        print("{} is developing".format(self.name))

    def work(self):
        self.develop()

class Company:

    def __init__(self, name: str):
        self.name = name

    def work(self, employee: Employee):
        employee.work()



carbon = Company("Carbon")
developer = Developer("Nusret", "1000000")
tester = Tester("Someone", "1000000")
carbon.work(developer) # Will print Nusret is developing
carbon.work(tester) # Will print Someone is testing
```

This update closes the class to modifications. Now you can add new employees to your class design without the need to modify `Company`. In every case, you'll have to implement the required interface, which also makes your classes polymorphic.

LSP

> *Subtypes must be substitutable for their base types.*

```
class SuperClass:
  def check(name: str)-> str:
    return name
class SubClass(SuperClass):
  def check(name: dict) -> dict:
    return name
class AnotherSubclass(SuperClass):
    def check(name: str, surname: str) -> tuple:
      return name, surname
```

In python we can do smth like this and our program even can be executed. However, there is **pylint** and **mypy** who can help us catch this situations.

BAD

```
from abc import ABC, abstractmethod


class Member(ABC):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @abstractmethod
    def save_database(self):
        pass

    @abstractmethod
    def pay(self):
        pass


class Teacher(Member):
    def __init__(self, name, age, teacher_id):
        super().__init__(name, age)
        self.teacher_id = teacher_id

    def save_database(self):
        print("Saving teacher data to database")
    def pay(self):
        print("Paying")


class Manager(Member):
    def __init__(self, name, age, manager_id):
        super().__init__(name, age)
        self.manager_id = manager_id

    def save_database(self):
        print("Saving manager data to database")

    def pay(self):
        print("Paying")


class Student(Member):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def save_database(self):
        print("Saving student data to database")

    def pay(self):
        raise NotImplementedError("It is free for students!")


members: List[Member] = []
members.apped(Student('nusret',23,"12345"))
members.apped(Teacher('Teacher_nusret',23,"12345"))
for member in members:
  member.pay()
```

If a `Member` has to pay, we can clearly say that a `Student`cannot be a `Member`. To solve this problem, we can remove the `pay()` method from `Member` and create a new abstract class `Payer`.

GOOD

```
from abc import ABC, abstractmethod

class Payer(ABC):
    @abstractmethod
    def pay(self):
        pass
```

```python
class Member(ABC):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @abstractmethod
    def save_database(self):
        pass


class Teacher(Member, Payer):
    def __init__(self, name, age, teacher_id):
        super().__init__(name, age)
        self.teacher_id = teacher_id

    def save_database(self):
        print("Saving teacher data to database")

    def pay(self):
        print("Paying")


class Manager(Member, Payer):
    def __init__(self, name, age, manager_id):
        super().__init__(name, age)
        self.manager_id = manager_id

    def save_database(self):
        print("Saving manager data to database")

    def pay(self):
        print("Paying")


class Student(Member):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def save_database(self):
        print("Saving student data to database")


payers: List[Payer] = [Teacher("John", 30, "123"), Manager("Mary", 25, "456")]
for payer in payers:
    payer.pay()
```

ISP

> *Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies.*

BAD

```
from abc import ABC, abstractmethod

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}...")
```

This implementation violates the ISP because it forces `OldPrinter` to expose an interface that the class doesn’t implement or need. To fix this issue, you should separate the interfaces into smaller and more specific classes.

GOOD

```
from abc import ABC, abstractmethod

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}...")
```

DIP

> *Abstractions should not depend upon details. Details should depend upon abstractions.*

BAD

```
from sqlalchemy import create_engine, select
from sqlalchemy.orm import Session

class FrontEnd:
    def __init__(self, back_end):
        self.back_end = back_end
    def display_data(self):
        data = self.back_end.get_data_from_database()
        print("Display data:", data)

class BackEnd:
  engine = create_engine("postgresql+psycopg2://scott:tiger@localhost/")
    def get_data_from_database(self):
        with Session(engine) as session:
            statement = select(User).all()
            return session.execute(statement)
```

Suppose we need to add new resource, e.g. REST API. So, we need to add a new method to BackEnd class. However, that will also require you to modify `FrontEnd`, which should be closed to modification, according to the open-closed principle.

So, our FrontEnd class depends on concrete implementation rather than abstraction. To fix it, we can pass BackEnd as an attribute

GOOD

```
from abc import ABC, abstractmethod
from sqlalchemy import create_engine, select
from sqlalchemy.orm import Session
import requests

class FrontEnd:
    def __init__(self, data_source):
        self.data_source = data_source

    def display_data(self):
        data = self.data_source.get_data()
        print("Display data:", data)

class DataSource(ABC):
    @abstractmethod
    def get_data(self):
        pass

class Database(DataSource):
    engine = create_engine("postgresql+psycopg2://scott:tiger@localhost/")
    def get_data_from_database(self):
        with Session(engine) as session:
            statement = select(User).all()
            return session.execute(statement)
    def get_data(self):
        return self.get_data_from_database()

class API(DataSource):
    site_url = "https://some_site.com/rest/v1/resources"
    def get_data_from_api(self):
        data = requests.get(site_url)
        return data.json()
    def get_data(self):
        return self.get_data_from_api()
```