SOLID - Part 1

- SRP
- OCP

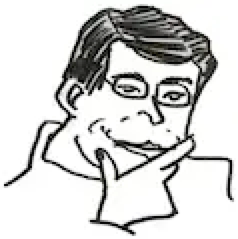
SOLID principles

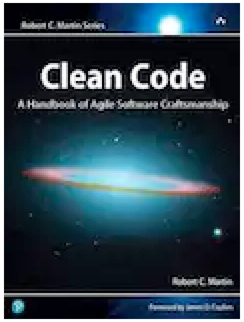
- SOLID principles are an acronym for five design principles.
- SOLID principles are practical guidelines to good design.

Who Created SOLID?

- SOLID principles were coined by a professional software engineer (Uncle Bob) to promote good software design principles.
- It is the accumulation of software engineering wisdom.

- He did not invent the SOLID principles.
- Each principle is known in the software engineering community.
- He published a book, Clean Code, to explain SOLID

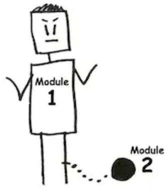




Why SOLID?

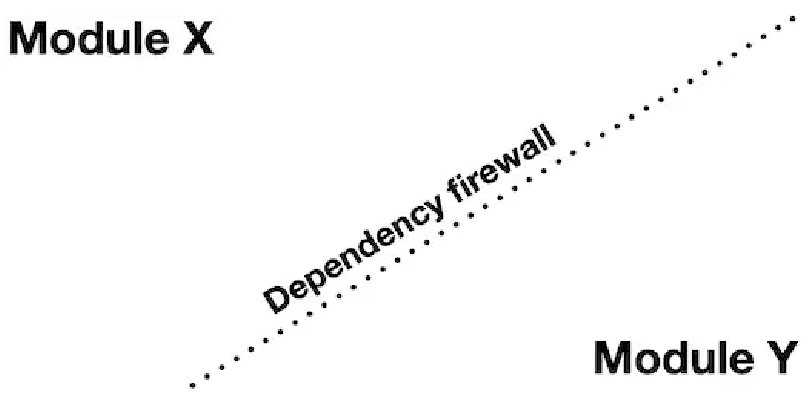
- To avoid rigid code (that is hard to refactor for changes).
- To avoid fragile code (that is easy to break by changes).

Rigid Code



Fragile Code

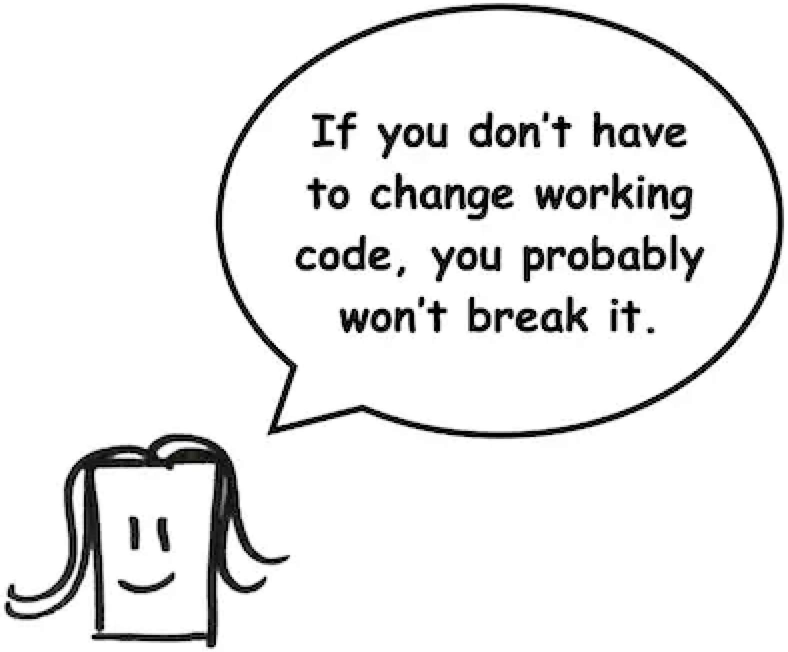
- To build a "dependency firewall" to make modules independent and communicate using well-defined interfaces (to prevent high coupling).



Change is Inevitable

- When we don't need to change anything in our design, our code is safe and won't break.
- However, it doesn't happen in the software engineering world.

- The SOLID principles are the guidelines for refactoring and the tool for managing changes effectively.



The Goal of SOLID

- The goal of the SOLID principles is to structure our software to allow us to **extend, change, and delete** parts of our code without changing other parts.
- Key Objective: Make software adaptable to change.

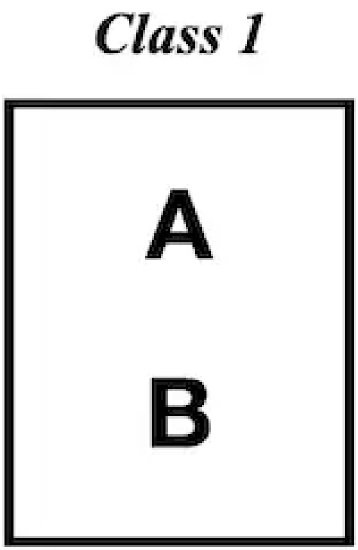
Single Responsibility Principle (SRP)

Things should have only one reason to change.

The Problem: Mixed Responsibilities

- We have the `class 1` that does two things, A and B.
- In other words, class 1 has two responsibilities.

- When we change A in class 1, we can impact and possibly break B because they are in the same class.



The class name

- Let's rename the class 1 based on what they do.
- The new class name is "ClassThatDoesAandB."

- We can see the responsibility is mixed, not cleanly separated.
- Key Insight: If you need "And" in your class name, it's probably violating SRP!

Employee Class

- Analyze the Employee class.
- Rename the class based on what it does.

```python
class Employee:
    xml_filename = "emp.xml"
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
    def raise_salary(self, factor):
        return self.salary * factor
    def save_as_xml(self):
        with open("emp.xml", "w") as file:
            file.write(
                f"<xml><n>{self.name}</n><salary>{self.salary}</salary></xml>")
```

Issues: two responsibilities

- The class has two responsibilities.
- One is a business logic to raise a salary.

- The other is storage logic, which saves the information in an XML file.

```
class Employee:
    xml_filename = "emp.xml"

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

    def raise_salary(self, factor):
        return self.salary * factor

    def save_as_xml(self):
        with open(self.xml_filename, "w") as file:
            file.write(f"<xml><name>{self.name}</name><salary>{self.salary}</salary></xml>")
```

Storage logic

Business logic

Too much information

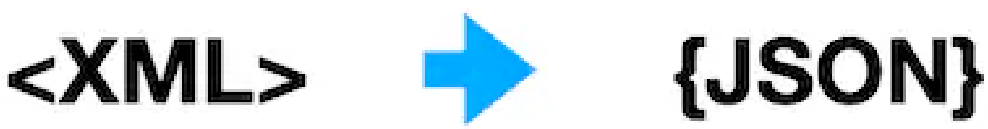
- Also, the Employee class has too much information.
- Why does it need to know the file name?

- **Problem**: Classes should only know what they need for their responsibility.

```python
class Employee:
    xml_filename = "emp.xml"

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

    def raise_salary(self, factor):
        return self.salary * factor
```

Change Requirements

- This makes the code rigid and fragile.
- What if we need to change the XML to JSON?



- We also need to change the employee class.
- It is unnecessary because the Employee class should not be impacted by the change.
- It is because it violates the SRP principle.

Class Names

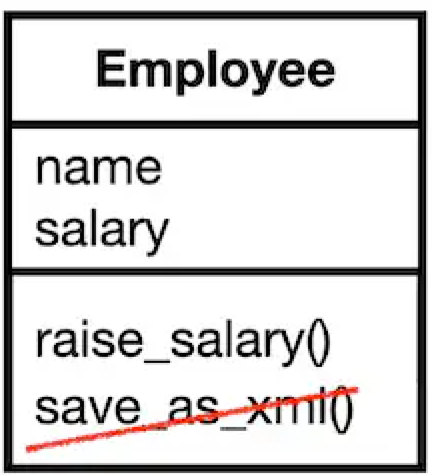
- We can sense smelly code when we make the class name based on its responsibilities.
- The class name should be "EmployeeAndStorage"
  - It manages employee and store information.

- Then, we can see that this class violates SRP.
- **Technique**: Use descriptive naming to identify SRP violations.

Refactoring: Separating Responsibilities

- Refactor the Employee class to be responsible only for business logic.

```
class Employee:
    xml_filename = "emp.xml"

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

    def raise_salary(self, factor):
        return self.salary * factor

    def save_as_xml(self):
        with open(self.xml_filename, "w") as file:
            file.write(f"<xml><name>{self.name}</
name><salary>{self.salary}</salary></xml>")
```



Creating Separate Classes

- We create a new class to be responsible only for storage logic.

storage.py

```
class EmployeeStorage:
    json_filename = "emp.json"

    def save_as_json(self, employee):
        with open(self.json_filename, "w") as file:
            file.write(f"name: {employee.name}, salary: {employee.salary}")
```

employee.py

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

    def raise_salary(self, factor):
        return self.salary * factor
```

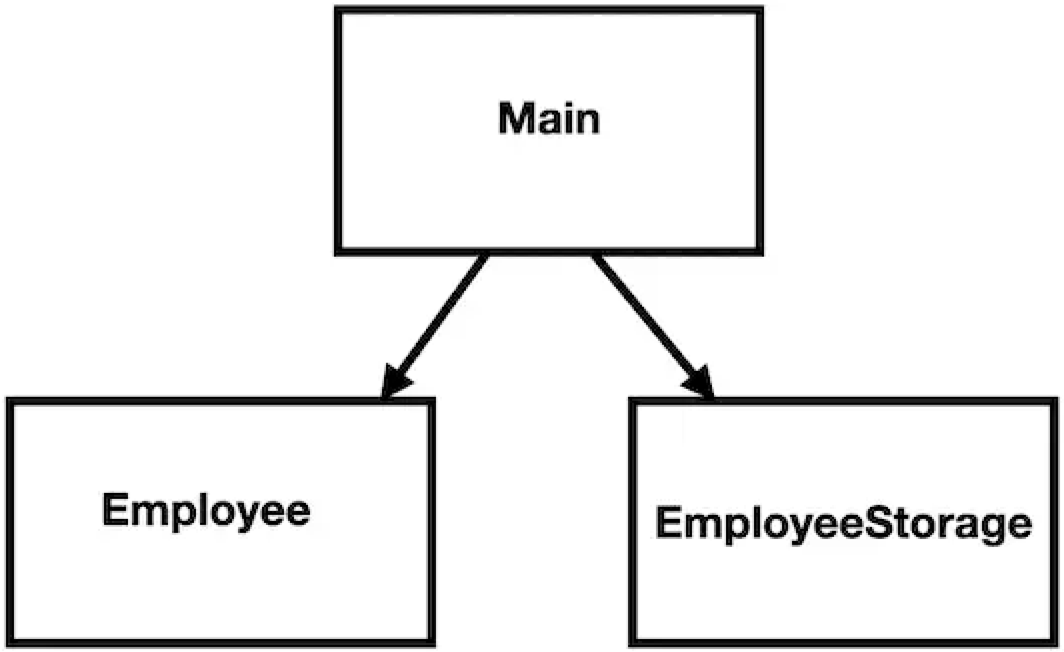
main.py

```
from employee import Employee
from storage import EmployeeStorage

e = Employee("Vera", 2000)
s = EmployeeStorage()
s.save_as_json(e)
```

Reduced Coupling Through SRP

- When we draw a UML diagram, the Employee and EmployeeStorage classes are separated and not impacted by changes in each other.

- In other words, the coupling is removed.



Applying SRP

- The SRP gives each module (class) only one responsibility.

Responsible for employee data and business logic:

```python
class EmployeeStorage:
    json_filename = "emp.json"

    def save_as_json(self, employee):
        with open(self.json_filename, "w") as file:
            file.write(f"name: {employee.name}, salary: {employee.salary}")
```

Responsible for employee storage:

```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def raise_salary(self, factor):
        return self.salary * factor
```

SRP is Easy to Break

- Let's use the Python json library in the EmployeeStorage class.
- However, an Error will occur.

```
import jsonlibrary

class EmployeeStorage:
    json_filename = "emp.json"

    def save_as_json(self, employee):
        jsonlibrary.save(self.json_filename, employee)
```

```
Object of type Employee has no serializable attributes.
Decorate attributes with @jsonserializable
```

- It is because to use the Employee class, we must:
  - (1) import the json library
  - (2) add a @jsonserializable decorator to use the JSON features.

Coupling again!

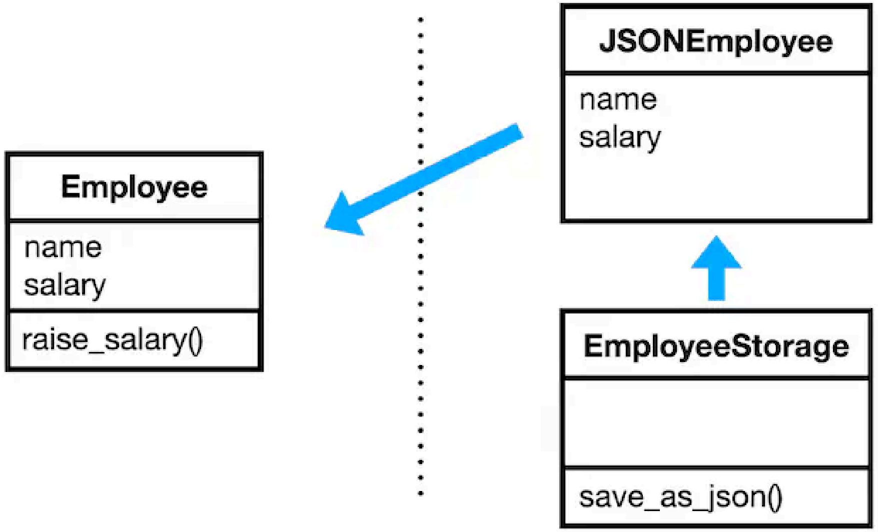
- The Employee class should know about the json library with the decorator.
- Any changes in the json library will impact the Employee class.

- We have to add coupling, and the SRP is broken.
- How can we solve this problem?

```python
import jsonlibrary

class Employee:
    def __init__(self, name, salary):
        @jsonlibrary.jsonserializable
        self.name = name

        @jsonlibrary.jsonserializable
        self.salary = salary

    def raise_salary(self, factor):
        return self.salary * factor
```

Solution: Decoupling

- We make a class responsible for dealing with JSON libraries.



SRP Violations Are Common

- As a software engineer, you will see SRP violations as a code smell.
- You know how to deal with them.

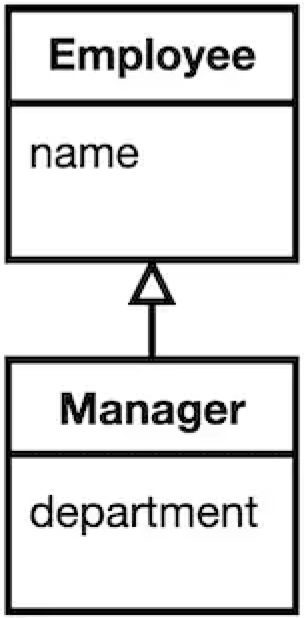
- However, SRP is easy to break, so we should keep an eye on it.
- **Remember**: One class, one responsibility, one reason to change.

Open-Closed Principle (OCP)

Open for extension but closed for modification

The Problem: Type Checking

- We have the Manager class that extends the Employee class.



- We need to know if the object is to print the correct message.
- We make the print_employee(e) method.

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

class Manager(Employee):
    def __init__(self, name, department):
        super().__init__(name)
        self.department = department

def print_employee(e):
    if type(e) is Employee:
        print(f"{e.name} is an employee")
    elif type(e) is Manager:
        print(f"{e.name} leads department {e.department}")
```

OCP Violation

- It uses an if/else statement for type checking.
- This code smells of an if/else statement unnecessarily.
- We know that this is easily breakable due to changes.

- This is an OCP violation.
- **Problem**: New employee type requires modifying the existing code.

```
if type(e) is Employee:
    ...
elif type(e) is Manager:
    ...
elif type(e) is Programmer:
    ...
```

Another Example

- Python supports the `isinstance` method for checking type.
- This is another example with the if/else statement using the method.

- **Pattern**: If you see the isinstance method or type checking for each object, it may violate OCP.

Report module

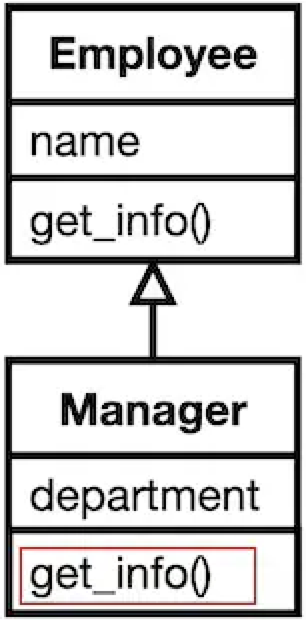
```python
from employees import Employee
from employees import Manager

def print_employee(e):
    if type(e) is Employee:
        # print regular employee
    elif type(e) is Manager:
        # print manager
```

Database module

```python
from employees import Employee
from employees import Manager

def save_employee(e):
    if type(e) is Employee:
        # save regular employee
    elif type(e) is Manager:
        # save manager
```

Refactoring: Polymorphism

- The solution to this problem is to use "Polymorphism."

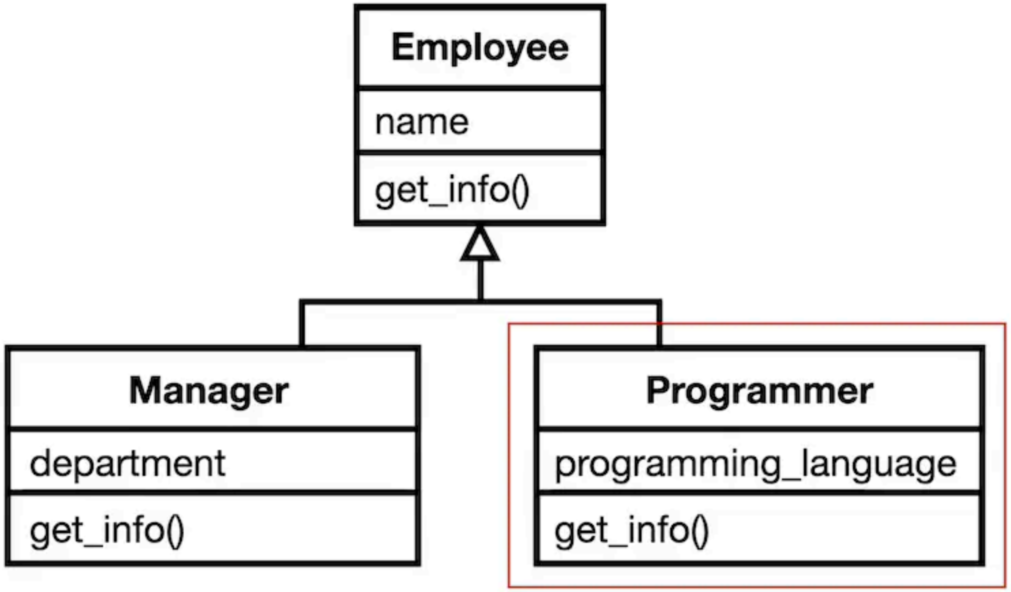


- Each class has the method get_info() with a different implementation.
- Python can invoke the correct get_info() method.

```python
class Employee:
    def __init__(self, name):
        self.name = name

    def get_info(self):
        return f"{self.name} is an employee"

class Manager(Employee):
    def __init__(self, name, department):
        super().__init__(name)
        self.department = department

    def get_info(self):
        return f"{self.name} leads department {self.department}"

def print_employee(e):
    print(e.get_info())
```

OCP Advantage 1: Easy Extension

- The Programmer class extends the Employee class.



- We override get_info() polymorphic method.

```python
class Programmer(Employee):
    def __init__(self, name, programming_language):
        super().__init__(name)
        self.programming_language = programming_language

    def get_info(self):
        return f"{self.name} programs in {self.programming_language}"
```

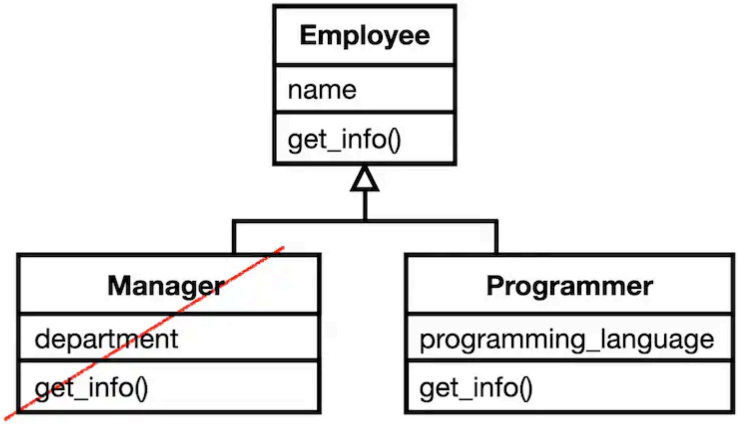
No if/else needed

- We don't need to add lines of code in the if statement.
- Instead, we invoke the get_info() method on the object.

```
def print_employee(e):
    if type(e) is Employee:
        print(f"{e.name} is an employee")
    elif type(e) is Manager:
        print(f"{e.name} leads department {e.department}")
    elif type(e) is Programmer:
        print(f"{e.name} programs in {e.programming_language}")
```

OCP Advantage 2: Easy Removal

- We can remove the Manager class without impacting any other code.



- We can safely delete the Manager class; there is no need to remove any other code.

```python
class Manager(Employee):
    def __init__(self, name, department):
        super().__init__(name)
        self.department = department

    def get_info(self):
        return f"{self.name} leads department {self.department}"
```

OCP Summary

- **Open for Extension**: You can add new functionality by adding new classes.
- **Closed for Modification**: You don't need to modify existing code.

- **Key Technique**: Use polymorphism instead of type checking.
- **Benefits**:
  - Easier to add new types
  - Easier to remove types
  - Less risk of breaking existing functionality