SOLID - Part 2

- LSP
- ISP
- DSP

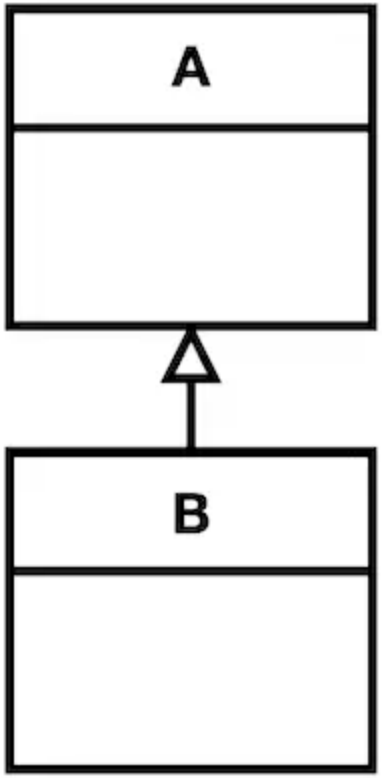
Liskov Substitution Principle (LSP)

Subclasses should not change the behavior of superclasses in unexpected ways.

LSP Definition

- If B is a subtype of A, objects of type A should be replaced with objects of type B without breaking anything.
- In other words, "no surprises if I replace a module" rule.

- We can safely substitute B (specific objects) with A (general objects).
- Key Point: Subclasses should be substitutable for their base classes.

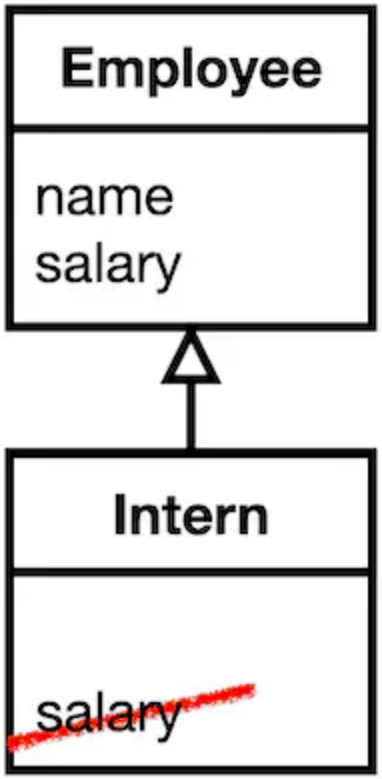


LSP in a real-world

- We have a light bulb socket.
- We are not going to be surprised whatever light bulb we replace for the socket.

The Intern Problem

- We have the Intern class, a new subclass of Employee.
- Interns do not have a salary, so the salary is irrelevant.



- In the constructor, we set the salary argument as None to express this business logic.

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

    def print_year_salary(self):
        print(f"{e.name} year salary ${e.salary * 12}")

class Intern(Employee):
    def __init__(self, name, salary):
        super().__init__(name, None)
```

Surprise!

- However, it causes an error.
- The print_year_salary() method in the Employee class cannot process None data.

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

    def print_year_salary(self):
        print(f"{e.name} year salary ${e.salary * 12}")
```

```
e = Intern("Chuck", 0)  # salary will be ignored
e.print_year_salary()
```

```
TypeError: unsupported operand type(s) for *:
'NoneType' and 'int'
```

Quick and Dirty Solution

- We can fix this issue by introducing the if/else statement.
- However, it violates OCP and LSP.

```
def print_year_salary(self):
    if type(self) is not Intern:
        print(f"{e.name} year salary ${e.salary * 12}")
```

Adding smelly code

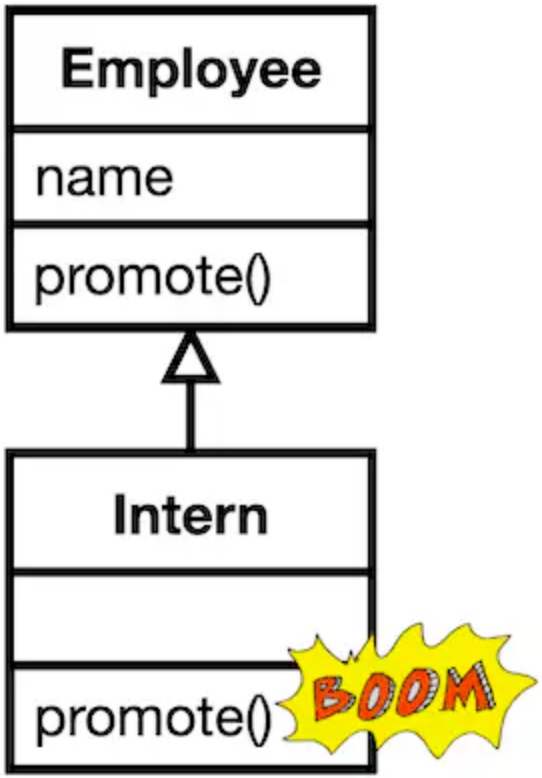
- Interns cannot be promoted, so we have to override the Intern's promote method to raise an error.

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

    def promote(self):
        print("Promote employee")

class Intern(Employee):
    def promote(self):
        raise NotImplementedError("Interns cannot be promoted")

def promote_employee(e):
    e.promote()  # crashes when e is an intern
```

The cause: LSP violation

- It all happens because we cannot replace the Intern object with the Employee object.
- The Intern class violates LSP to surprise users of the class.

- **LSP Violation**: Subclasses should not introduce unexpected behavior.



Another Violation of LSP: ID Problem

- We introduce an ID to manage employees.
- The ID is int type and should be more than 0, but Interns do not have ID.

Quick and Dirty Solution (again)

- We can give large ID number.
- But this won't work if we have many employees (it's a time bomb).

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

    def is_employee_id_valid(self):
        return type(self.employee_id) is int and self.employee_id > 0

class Intern(Employee):
    def __init__(self, employee_id, name):
        super().__init__(employee_id, name)

e = Intern(345, "Chuck")
print(e.is_employee_id_valid())
```

```
True
```

Quick and Dirty Solution (again)

- To solve this issue, we may decide to prepend 'I' for the interns' ID number.

Vera (144)
Dave (231)
Chuck (I345)

```
class Intern(Employee):
    def __init__(self, employee_id, name):
        super().__init__(f"I{employee_id}", name)
```

LSP and OCP violation

- This is not a good solution as it still violates LSP and OCP.

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

    def is_employee_id_valid(self):
        return type(self.employee_id) is int and self.employee_id > 0

class Intern(Employee):
    def __init__(self, employee_id, name):
        super().__init__(f"I{employee_id}", name)

e = Intern(345, "Chuck")
print(e.is_employee_id_valid())
```

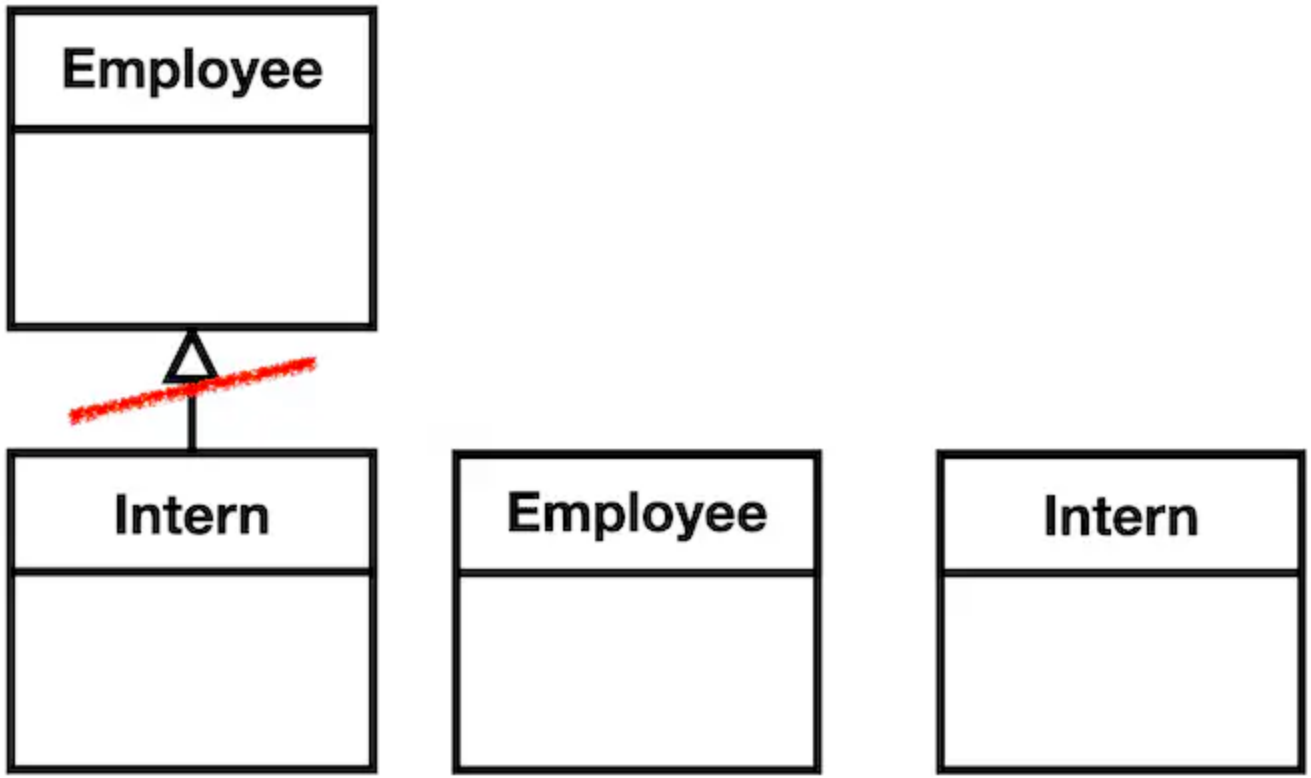
```
False
```

- And even worse, it complicates the code.
- **Warning**: Quick and dirty solutions that violate LSP will lead to worse problems.

Refactoring to LSP: The Right Solution

- We can observe that they are different objects to cause all the problems.
- The solution is disconnecting the relationship between Employee and Intern, following LSP.

- We can't replace Employee objects with Intern objects, as they are different.



LSP Summary

- **The Principle**: Subclasses must be substitutable for their base classes.
- **Warning Signs**:
  - Empty or exception-throwing overridden methods
  - Type checking in client code
  - Special cases for certain subclasses

- **Solution**: If objects can't be substituted, they shouldn't inherit.
- **Remember**: Inheritance should model "is-a" relationships that preserve behavior.

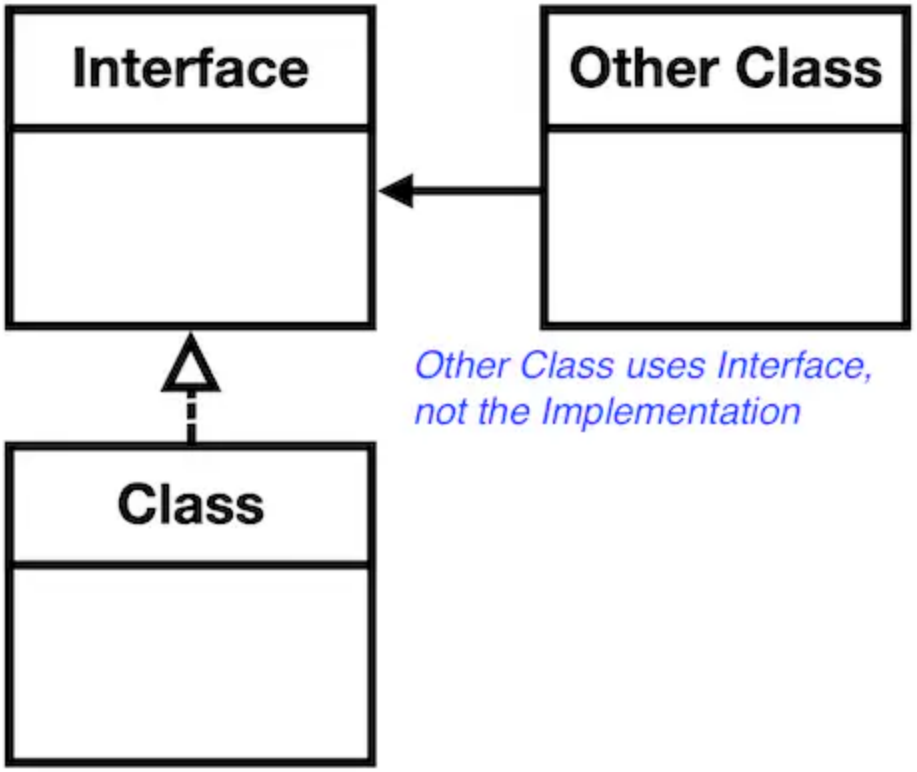
Interface Segregation Principle (ISP)

no client should be forced to depend on methods it does not use

ISP Fundamentals

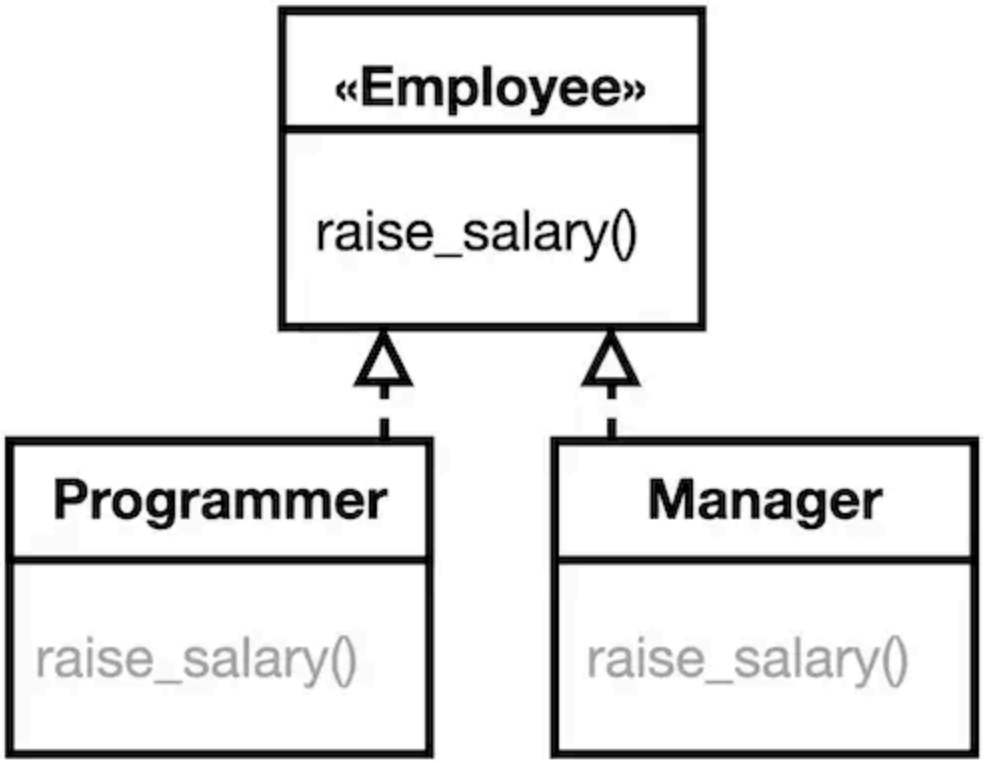
- ISP is all about keeping interfaces cohesive.
- Interfaces have no implementation body, they export only function names.

- In OOP, we program on interfaces, not concrete classes.



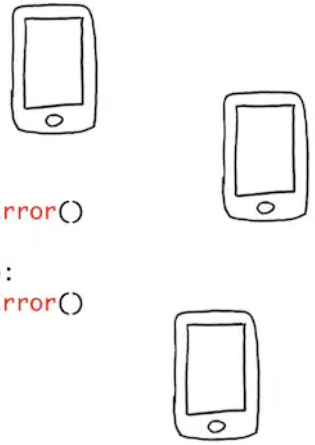
UML

- In UML, we use "<< ... >>" for interfaces.
- When a concrete class implements the interface, we use a dotted line.



Example of LSP Violation

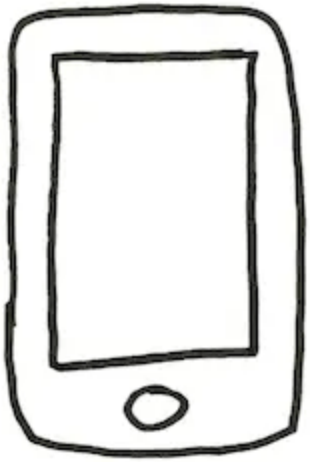
- We create the Phone class thinking all the phones can call and swipe.



```
class Phone:
    def call(self, number):
        raise NotImplementedError()

    def swipe_to_unlock(self):
        raise NotImplementedError()
```

iPhone

- We have the IPhone class that can call and swipe to unlock, so we implement the interface.



```
class IPhone(Phone):
    def call(self, number):
        print(f"Calling Number: {number} from iPhone.")

    def swipe_to_unlock(self):
        print("iPhone is unlocked.")
```

Nokia

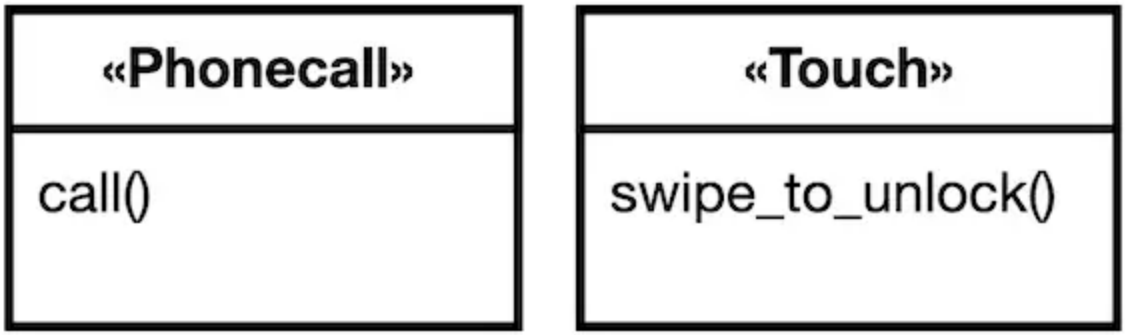
- We create the Nokia2720 class.
- But we don't know how to implement the swipe_to_lock method for Nokia phones.

- This will raise an error when the method is invoked because it violates ISP.

```
class Nokia2720(Phone):
    def call(self, number):
        print(f"Calling Number: {number} from Nokia 2720.")

    def swipe_to_unlock(self):
        raise NotImplementedError("Nokia 2720 has no touch screen.")
```

Refactoring: Separate Interfaces

- It violates ISP as the Nokia class is forced to depend on the method it does not use.
- The solution is to separate the Phone interface into two interfaces.




```
class Phonecall:
    def call(self, number):
        raise NotImplementedError()
```

```
class Touch:
    def swipe_to_unlock(self):
        raise NotImplementedError()
```

Implementation

- The IPhone class can implement both the Phonecall and Touch interfaces.
- The Nokia class that does not have Touch functionality implements only the Phonecall interface.

```
class IPhone(PhoneCall, Touch):
    def call(self, number):
        print(f"Calling {number} from iPhone")
    def swipe_to_unlock(self):
        print("iPhone unlocked")
```

```
class Nokia2720(PhoneCall):
    def call(self, number):
        print(f"Calling {number} from Nokia")
```

Interface in Python

- Compared to Java, Python does not support interfaces.
- But, we can use `abc` package for Python interfaces.
- As a convention, we prepend 'I' for interfaces, and we use `@abstractmethod` decorator to imply the interface method.

```
from abc import ABCMeta, abstractmethod

class IProduct(metaclass=ABCMeta):
    "A Hypothetical Class Interface (Product)"
    @abstractmethod
    def create_object():
        "An abstract interface method"

class Product(IProduct):
    def create_object(self):
        return "Object"
```

PhoneCall and Touch Interface

- We can implement the Python interface using the @abstractmethod.

```
from abc import ABCMeta, abstractmethod

class PhoneCall(metaclass=ABCMeta):
    @abstractmethod
    def call(self, number):""

class Touch(metaclass=ABCMeta):
    @abstractmethod
    def swipe_to_unlock(self):""

# Alternative approach
class PhoneCall():
    def call(self, number): pass

class Touch():
    def touch(self): pass
```

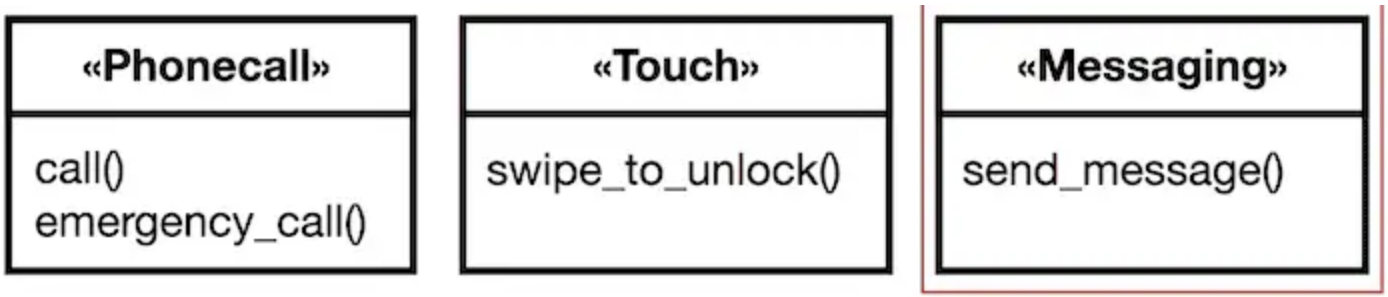
Complete Implementation

```
1  class IPhone(PhoneCall, Touch):
2      def call(self, number):
3          print(f"Calling Number: {number} from iPhone.")
4      def swipe_to_unlock(self): print("iPhone is unlocked.")
5
6  class Nokia2720(PhoneCall):
7      def call(self, number):
8          print(f"Calling Number: {number} from Nokia 2720.")
9
10 i = IPhone()
11 n = Nokia2720()
12
13 i.call('1234')
14 i.swipe_to_unlock()
15 n.call('1234')
```

ISP Applied: Extending Functionality

- What if we need a new emergency call feature for all phones?
- And what if we need to implement the message feature?

- We can add a new interface.



ISP Summary

- **The Principle**: Clients should not be forced to depend on interfaces they do not use.
- **Benefits**:
  - Classes only implement methods they actually need
  - Smaller, more focused interfaces
  - Easier to extend and maintain

- **Key Technique**: Separate large interfaces into smaller, cohesive ones.
- **Remember**: Many small interfaces are better than one large interface.

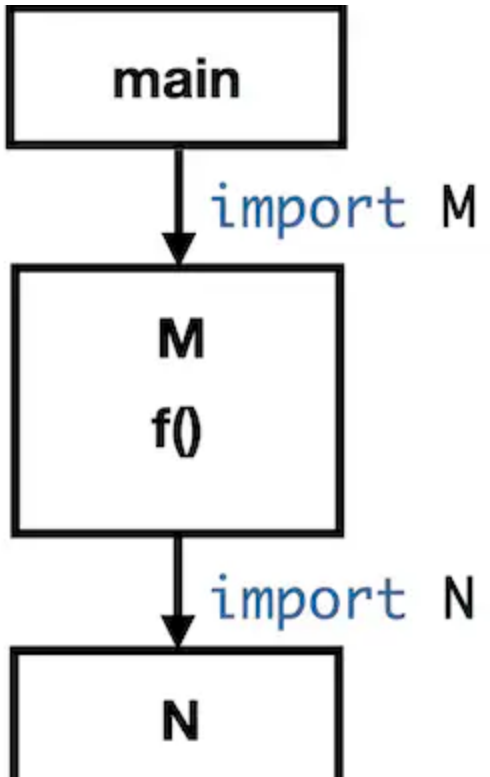
Dependency Inversion Principle (DIP)

High level modules should not depend on low level modules. Instead, they should depend on abstractions

Dependency Structure

<ul">

- The main method uses M.f(), which uses N.g().
- Let's say N is the low-level module called by the high-level module M.



- For example, M module is Drive, and N module is "Tesla".
    - What if M needs to drive "Toyota"?
- **Problem**: High-level modules depend on low-level modules.
    - High-level means more abstract object.
    - Low-level means more concrete object.

Code Smell: Concrete Dependencies

- The problem is that concrete classes are volatile, as they can be modified anytime by anyone.
- **Issue**: Depending on concrete implementations creates tight coupling.

Example: Reporting and Printer

- We have the CashRegisterPrinter class that is used by the Reporting.

```
class CashRegisterPrinter:
    def print_receipt(self, receipt_text):
        print("Print receipt_text to CashRegisterPrinter")
```

```
from printers import CashRegisterPrinter

class Reporting():
    def print_receipt(self, receipt_text):
        printer = CashRegisterPrinter()
        printer.print_receipt(receipt_text)
```

Change

- What if the Reporting() needs to use the LaserPrinter class?
- We create the class and change the main program.

```
class CashRegisterPrinter:
    def print_receipt(self, receipt_text):
        print("Print receipt_text to CashRegisterPrinter")

class LaserPrinter:
    def print_receipt(self, receipt_text):
        print("Print receipt_text to LaserPrinter")

from printers import CashRegisterPrinter
from printers import LaserPrinter

class Reporting():
    def print_receipt(self, receipt_text):
        printer = CashRegisterPrinter()
        printer = LaserPrinter()
        printer.print_receipt(receipt_text)
```

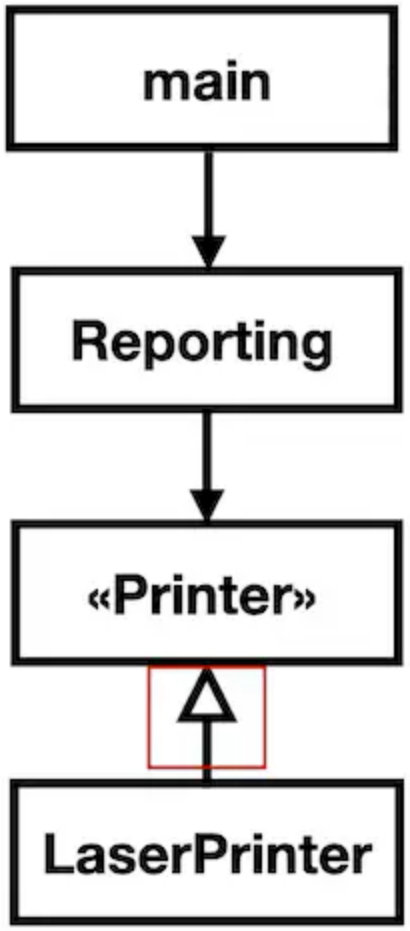
The Impact

- We have this unnecessary change because the Reporting class depends upon low-level concrete classes.

Refactoring: Inverting Dependencies

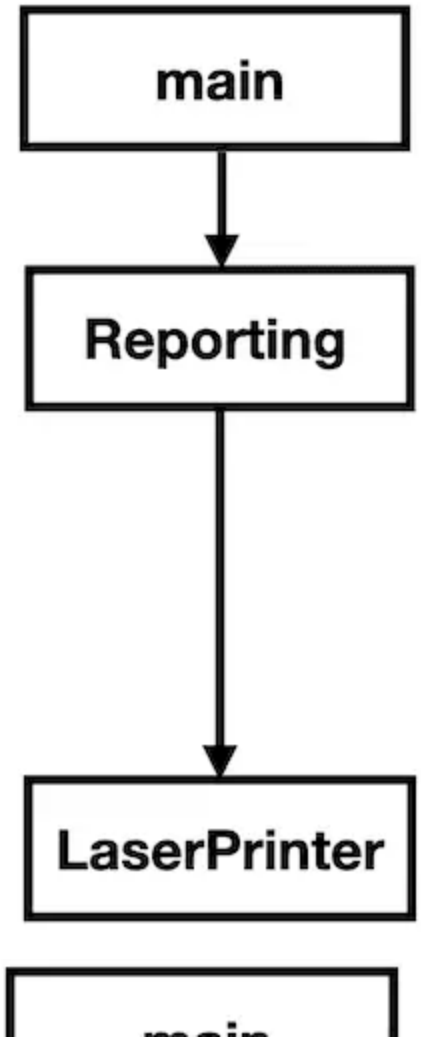
- We can invert the dependency and break the coupling when the Reporting class depends on high-level abstractions (interfaces).

- When we have a Printer interface and each concrete class implements it, any change cannot impact the Reporting class.



DI Visualized

- See the dependency is inverted.



DI Code

- For DI, the Reporting class contains the high-level Printer object as its member.

```
class Reporting():
    def __init__(self, printer):
        self.printer = printer

    def print_receipt(self, receipt_text):
        self.printer.print_receipt(receipt_text)
```

- And the main function injects the concrete printer object to the Reporting.
- The concrete LaserPrinter depends on Printer, so the dependency is inverted.

```
from reporting import Reporting
from printers import LaserPrinter

p = LaserPrinter()
r = Reporting(p)
r.print_receipt("TOTAL: $45,00")
```

Managing Changes

- Instead of LaserPrinter, we need to use CashRegisterPrintr.
- In this case, we can inject the CashRegisterPrintr object into the Reporting.
- Dependency inversion is implemented with dependency injection.

- No other changes are necessary because of the application of DIP.

```
from reporting import Reporting
from printers import LaserPrinter
from printers import CashRegisterPrinter

p = LaserPrinter()
p = CashRegisterPrinter()
r = Reporting(p)
r.print_receipt("TOTAL: $45,00")
```

DIP Summary

- **The Principle**:
  - High-level modules should not depend on low-level modules
  - Both should depend on abstractions
  - Abstractions should not depend on details
- **Key Technique**: Dependency Injection

- **Benefits**:
  - Loose coupling between modules
  - Easy to swap implementations
  - More testable code
- **Remember**: Depend on abstractions, not concretions.