01076105, 01075106

Object Oriented Programming

Object Oriented Programming Project

OO Design Guides

Cohesion and Coupling

- cohesion และ coupling เป็นสิ่งสำคัญในการออกแบบและเขียนโปรแกรม โปรแกรมที่มี strong cohesion และ loose coupling ถือเป็นโปรแกรมคุณภาพสูงที่จะง่ายต่อการดูแลและ reuse ในอนาคต
- Cohesion คือระดับของความเกี่ยวข้องกันของ code ในฟังก์ชันเดียวกัน ในฟังก์ชันใดที่ code มีความเกี่ยวข้องกันมาก (หมายถึงเป็นเรื่องที่เกี่ยวข้องกัน หรือเป็นเรื่องเดียวกัน จนยากจะแยกออกจากกัน) ถือว่า strong cohesion ตัวอย่างของ strong cohesion เช่น sin() จะเห็นว่ามีหน้าที่เดียว และ แยกออกไม่ได้แล้ว
- Coupling คือ การที่แต่ละส่วนของ code เข้าไปเกี่ยวข้องกัน ถ้าเข้าไปเกี่ยวข้องกันมาก เรียกว่า high coupling ถ้าไปเกี่ยวข้องกันน้อยก็เรียกว่า low coupling เพราะถ้าส่วนหนึ่งของ code เปลี่ยน จะไม่ทำให้ต้องแก้ไขส่วนอื่น

Cohesion and Coupling

- ลองดู Code ต่อไปนี้
- คลาส VihecleRegistry ทำหน้าที่สร้างเลขทะเบียนยานพาหนะ

```
import string
import random

class VehicleRegistry:

    def generate_vehicle_id(self, length):
        return ''.join(random.choices(string.ascii_uppercase, k=length))

    def generate_vehicle_license(self, id):
        return f"{id[:2]}-{''.join(random.choices(string.digits, k=2))}- \
        {''.join(random.choices(string.ascii_uppercase, k=2))}"
```

Cohesion and Coupling

```
class Application:

    def register_vehicle(self, brand: string):
        registry = VehicleRegistry()
        vehicle_id = registry.generate_vehicle_id(12)
        license_plate = registry.generate_vehicle_license(vehicle_id)

        catalogue_price = 0
        if brand == "Tesla Model 3":
            catalogue_price = 60000
        elif brand == "Volkswagen ID3":
            catalogue_price = 35000
        elif brand == "BMW 5":
            catalogue_price = 45000

        tax_percentage = 0.05
        if brand == "Tesla Model 3" or brand == "Volkswagen ID3":
            tax_percentage = 0.02
        payable_tax = tax_percentage * catalogue_price

        print("Registration complete. Vehicle information:")
        print(f"Brand: {brand}")
        print(f"Id: {vehicle_id}")
        print(f"License plate: {license_plate}")
        print(f"Payable tax: {payable_tax}")
```

Cohesion and Coupling

- จาก code จะเห็นได้ว่าคลาส Application ทำหลายอย่างมาก สร้าง id, สร้าง license กำหนดราคา กำหนดอัตราภาษี กำหนดจำนวนภาษี และ พิมพ์ code ลักษณะนี้จะเรียกว่า low cohesion เพราะทำหน้าที่หลายอย่าง
- ในขณะเดียวกัน code นี้ ถือว่ามี high coupling เช่น ในคลาส VehicleRegistry มีฟังก์ชัน generate_vehicle_id และ generate_vehicle_license และมีการเรียกใช้ในคลาส Application ซึ่งหมายความว่า หากมีการเปลี่ยนแปลงในคลาส VehicleRegistry ก็อาจจะส่งผลกระทบกับคลาส Application ไปด้วย
- สมมติว่าจะเพิ่มยี่ห้อเข้าไป ก็ทำให้ต้องแก้ไขคลาส Application หรือ หากเปลี่ยนอัตราภาษีของรถไฟฟ้า ก็ต้องแก้ไขคลาส

Cohesion and Coupling

- จะแยก VehicleRegistry ออกเป็น 2 คลาส คือ VehicleInfo รับผิดชอบในการเก็บข้อมูลรถ และ คำนวณภาษี

```python
class VehicleInfo:

    def __init__(self, brand, electric, catalogue_price):
        self.brand = brand
        self.electric = electric
        self.catalogue_price = catalogue_price

    def compute_tax(self):
        tax_percentage = 0.05
        if self.electric:
            tax_percentage = 0.02
        return tax_percentage * self.catalogue_price

    def print(self):
        print(f"Brand: {self.brand}")
        print(f"Payable tax: {self.compute_tax()}")
```

Cohesion and Coupling

- ส่วนคลาส Vehicle ทำหน้าที่เก็บข้อมูลรถยนต์และทะเบียน

```
class Vehicle:

    def __init__(self, id, license_plate, info):
        self.id = id
        self.license_plate = license_plate
        self.info = info

    def print(self):
        print(f"Id: {self.id}")
        print(f"License plate: {self.license_plate}")
        self.info.print()
```

Cohesion and Coupling

- จากนั้นจึงสร้างคลาสของ VehicleRegistry ขึ้นมา

```
class VehicleRegistry:

    def __init__(self):
        self.vehicle_info = { }
        self.add_vehicle_info("Tesla Model 3", True, 60000)
        self.add_vehicle_info("Volkswagen ID3", True, 35000)
        self.add_vehicle_info("BMW 5", False, 45000)
        self.add_vehicle_info("Tesla Model Y", True, 75000)

    def add_vehicle_info(self, brand, electric, catalogue_price):
        self.vehicle_info[brand] = VehicleInfo(brand, electric, catalogue_price)

    def generate_vehicle_id(self, length):
        return ''.join(random.choices(string.ascii_uppercase, k=length))

    def generate_vehicle_license(self, id):
        return f"{id[:2]}-{''.join(random.choices(string.digits, k=2))}-
{''.join(random.choices(string.ascii_uppercase, k=2))}"

    def create_vehicle(self, brand):
        id = self.generate_vehicle_id(12)
        license_plate = self.generate_vehicle_license(id)
        return Vehicle(id, license_plate, self.vehicle_info[brand])
```

Cohesion and Coupling

- ส่วนคลาส Application จะเหลืองานตามรูป ซึ่งจะเห็นว่าแต่ละคลาสรับผิดชอบงาน 1 อย่าง code เป็นอิสระต่อกันมากขึ้น การเพิ่มยี่ห้อรถไม่ต้องแก้โค้ด

```
class Application:

    def register_vehicle(self, brand: string):
        # create a registry instance
        registry = VehicleRegistry()

        vehicle = registry.create_vehicle(brand)

        # print out the vehicle information
        vehicle.print()

app = Application()
app.register_vehicle("Volkswagen ID3")
```

Dependency Injection

- Dependency injection เป็นเทคนิคอย่างหนึ่งในการลด coupling
- ลองพิจารณา code ต่อไปนี้
- จะเห็นว่ามีการผูกบริการ email เข้ากับคลาสอย่างมาก
- ทำให้เมื่อมีการเปลี่ยนผู้ให้บริการ email จะต้องมาแก้ไขโปรแกรมในคลาส

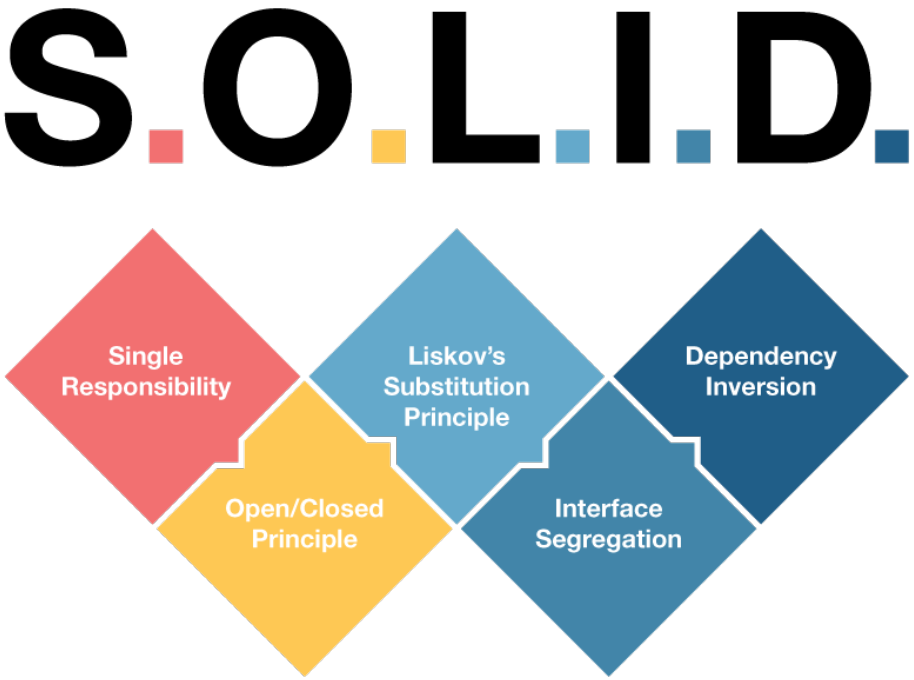
```
class Calculator:
    def __init__(self):
        pass

    def calculate(self, x, y):
        result = x + y
        email_service = SendGridService()
        email_service.send_email('example@abc.com', 'Calc. Result', f'The result is: {result}')
```

Dependency Injection

- แต่หากมีการแก้ไขโปรแกรม เป็นลักษณะนี้ เมื่อมีการเปลี่ยนแปลงผู้ให้บริการ email ก็เพียงแต่เปลี่ยนพารามิเตอร์ที่ส่งให้คลาสเท่านั้น
- เทคนิคแบบนี้ เรียกว่า dependency injection ซึ่งจะทำให้ coupling ระหว่างแต่ละส่วนของโปรแกรมลดลง

```
class Calculator:
    def __init__(self, email_service):
        self.email_service = email_service

    def calculate(self, x, y):
        result = x + y
        self.email_service.send_email('example@abc.com', 'Calc. Result', f'The result is: {result}')
```

SOLID Principle



SOLID Principle

- เป็นหลักการที่เสนอโดย Robert Martin เพื่อให้การออกแบบซอฟต์แวร์สามารถเข้าใจได้ง่ายขึ้น มีความยืดหยุ่น และบำรุงรักษาได้ง่าย โดยมีหลักการ 5 ข้อ
  - Single Responsibility Principle
  - Open-Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

SOLID Principle

- Single Responsibility Principle
  - หลักการข้อนี้ คือ แต่ละส่วนประกอบของซอฟต์แวร์ ควรรับผิดชอบเพียงหน้าที่เดียว ดังนั้น คลาส 1 คลาส ก็ควรรับผิดชอบหน้าที่เดียวด้วย (รวมถึง function หรือ method ด้วย)
  - ส่วนประกอบของ Class ควรมีความเกี่ยวข้อง (cohesion) กันให้มากที่สุด
  - นอกจากนั้น ส่วนประกอบควรขึ้นต่อกัน (coupling) ให้น้อยที่สุดอีกด้วย
  - แนวทางการตรวจสอบการออกแบบคลาส จะใช้หลักการดังนี้

    A class should have one and only one reason to change.

  - ทั้งนี้ก็เพื่อให้ ความจำเป็นต้องแก้ไข คลาส เกิดขึ้นได้น้อยที่สุด

SOLID Principle

- Single Responsibility Principle

  ยกตัวอย่าง

```python
class Order:

    def __init__(self):
        self.items = []
        self.quantities = []
        self.prices = []
        self.status = "open"

    def add_item(self, name, quantity, price):
        self.items.append(name)
        self.quantities.append(quantity)
        self.prices.append(price)
```

SOLID Principle

- Single Responsibility Principle

```
    def total_price(self):
        total = 0
        for i in range(len(self.prices)):
            total += self.quantities[i] * self.prices[i]
        return total

    def pay(self, payment_type, security_code):
        if payment_type == "debit":
            print(f"Debit type : security code: {security_code}")
            self.status = "paid"
        elif payment_type == "credit":
            print(f"Credit type : security code: {security_code}")
            self.status = "paid"
        else:
            raise Exception(f"Unknown payment type: {payment_type}")
```

SOLID Principle

- Single Responsibility Principle
  - จะเห็นว่าคลาส Order รับผิดชอบงานหลายอย่าง ซึ่งขัดกับหลักของ Single Responsibility
  - มีการ add order
  - มีการคำนวณราคารวมของ order
  - มีการเก็บเงินใน order
  - การเพิ่ม order และ คำนวณราคาของ order ถือได้ว่ามีความเกี่ยวข้องกัน
  - แต่การเก็บเงินมีความไม่เกี่ยวข้องกันพอสมควร ดังนั้นควรแยกคลาส

SOLID Principle

```python
class Item:
    def __init__(self,items,quantities,prices):
        self.items = items
        self.quantities = quantities
        self.prices = prices

class Order:

    def __init__(self):
        self.item_list = []
        self.status = "open"

    def add_item(self, name, quantity, price):
        self.item_list.append(Item(name, quantity, price))

    def total_price(self):
        total = 0
        for i in self.item_list:
            total += i.quantities * i.prices
        return total
```

SOLID Principle

- Single Responsibility Principle
  - แยกคลาส จะทำให้การเพิ่มประเภทของการชำระเงินง่ายขึ้นด้วย

```python
class PaymentProcessor:
    def pay_debit(self, order, security_code):
        print("Processing debit payment type")
        print(f"Verifying security code: {security_code}")
        order.status = "paid"

    def pay_credit(self, order, security_code):
        print("Processing credit payment type")
        print(f"Verifying security code: {security_code}")
        order.status = "paid"
```

SOLID Principle

- Single Responsibility Principle
- ในส่วนของ method ก็มีข้อแนะนำเช่นกัน ให้รับผิดชอบเพียงหน้าที่เดียว และไม่ควรมีความยาวเกินกว่า 15 บรรทัด (ทั้งนี้ตามความเหมาะสม) และพารามิเตอร์ก็ให้มีประมาณ 3 ตัว
- ในกรณีของ if else ก็ให้มีเงื่อนไขเดียว
- การคำนวณก็ให้คำนวณออกมาในเรื่องเดียว
- โดยสรุป คือ software ควรให้เรียบง่าย ลดความซับซ้อน เพิ่มการ reuse ดูแลรักษาง่าย ข้อผิดพลาดน้อย

SOLID Principle

- Open-Closed Principle
- หลักการข้อนี้มีอยู่ว่า ส่วนประกอบของ Software ควรจะ Close สำหรับการแก้ไข แต่ Open สำหรับการเพิ่มเติม
- หมายความว่าหลังจากที่ Software เขียนเสร็จแล้ว ไม่ควรมีการแก้ไขใดๆ อีก กรณีของคลาส คือ ไม่ไปแตะต้องคลาสนั้นอีก กรณีที่มีการเพิ่มเติม ก็ควรใช้วิธีการ Inheritance มากกว่าจะไปแก้ไขที่คลาสเดิม
- จะยกตัวอย่างเดิม คลาส PaymentProcessor สมมติว่าต้องการจะเพิ่มการชำระเงินแบบใหม่เข้าไป จะต้องแก้ไขคลาสนี้ ซึ่งขัดกับหลักการ Open-Close principle

SOLID Principle

- ดังนั้นเราจะใช้การ inherit จะเห็นว่าหากมีการชำระเงินประเภทอื่นก็สร้างเพิ่ม

```python
class PaymentProcessor(ABC):

    @abstractmethod
    def pay(self, order, security_code):
        pass

class DebitPaymentProcessor(PaymentProcessor):
    def pay(self, order, security_code):
        print("Processing debit payment type")
        print(f"Verifying security code: {security_code}")
        order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):
    def pay(self, order, security_code):
        print("Processing credit payment type")
        print(f"Verifying security code: {security_code}")
        order.status = "paid"
```

SOLID Principle

- สมมติว่าสร้างการจ่ายแบบ PromptPay

```
class PromptPaymentProcessor(PaymentProcessor):
    def pay(self, order, security_code):
        print("Processing PromptPay payment type")
        print(f"Using telephone no.: {security_code}")
        order.status = "paid"

order = Order()
order.add_item("Keyboard", 1, 50)
order.add_item("SSD", 1, 150)
order.add_item("USB cable", 2, 5)

print(order.total_price())
processor = PromptPaymentProcessor()
processor.pay(order, "081-234-5678")
```

SOLID Principle

- Liskov Substitution Principle
- ที่ได้ชื่อนี้ เนื่องจากเสนอโดย Barbara Liskov

  “We present a way of defining the subtype relation that ensures that subtype objects preserve behavioral properties of their supertypes.” Liskov.

- ความหมายของหลักการนี้ คือ “subclass ต้องสามารถแทนที่ base class ของตัวมันได้”
- เพราะ subclass ควรจะเพิ่มเติมความสามารถจาก base class ของมัน แต่ต้องไม่ทำให้ความสามารถของ class นั้นลดลง

SOLID Principle

- Liskov Substitution Principle
- จาก code ที่ผ่านมาจะเห็นว่าในคลาส PromptPaymentProcessor มีการใช้หมายเลขโทรศัพท์ ไม่ใช่ security code ซึ่งทำให้ขัดกับหลักการข้อนี้ เนื่องจากคลาสที่ inherit มา เข้าไปแก้ไขจาก security code เป็นอย่างอื่น
- แนวทางการแก้ไข คือ ใน abstract base class จะนำเอา security code ออกให้เหลือเพียง order

SOLID Principle

- Liskov Substitution Principle

```python
class PaymentProcessor(ABC):

    @abstractmethod
    def pay(self, order):
        pass

class PromptPaymentProcessor(PaymentProcessor):

    def __init__(self, tel_no):
        self.tel_no = tel_no

    def pay(self, order):
        print("Processing promptpay payment type")
        print(f"Using telephone no: {self.tel_no}")
        order.status = "paid"
```

SOLID Principle

- Liskov Substitution Principle
- มาสมมติตัวอย่างกันอีกสัก 1 ตัวอย่าง
- สมมติว่า เราทำงานในบริษัท OK-ALL ซึ่งเป็นร้านสะดวกซื้อ
- สมมติว่ารับของคนอื่นมาขาย จะอยู่ใน class Product โดยมีส่วนลด 20% (คือ รับมา 80% ของราคาขาย)
- บริษัทในเครือ ก็ผลิตสินค้าของตัวเองด้วย โดยจะ Inherit มาจาก class Product ตั้งชื่อว่า InHouseProduct โดยจะมีส่วนลดมากกว่า คือ 1.5 เท่า = 30%

SOLID Principle

- Liskov Substitution Principle
- สมมติว่าเขียน class Product แบบนี้ คำถาม คือ จะเขียน class InhouseProduct อย่างไร จึงจะไม่ขัดกับ LSP

```
class Product:
    discount = 20

    def get_discount(self):
        return Product.discount
```

SOLID Principle

- จาก LSP สิ่งที่ต้องทำ คือ ต้องทำให้ method get_discount สามารถทำงานได้เช่นเดียวกับใน Super Class ดังนั้นต้องเขียนเป็น

```
class InHouseProduct(Product):
    def get_discount(self):
        return self.apply_extra_discount()

    def apply_extra_discount(self):
        return self.discount * 1.5

product1 = Product()
product2 = InHouseProduct()

lst = [product1, product2]
for i in lst:
    print(i.get_discount())
```

SOLID Principle

- Interface Segregation Principle (Segregation = ทำให้แยกออกจากกัน)
- หลักการข้อนี้ นำเสนอโดย “Robert Martin”
- หลักการนี้กล่าวว่า “การแยก Interface ออกตามการใช้งานนั้น ดีกว่าการสร้าง general interface เพื่อใช้งานร่วมกันหลายๆ คลาส”
- เพราะบาง class ไม่ควรจะโดนบังคับให้ implement method ที่ไม่ได้ใช้งาน
- อธิบาย คือ กรณีที่มี object ที่คล้ายๆ กัน ให้แยกแยะว่าแต่ละ object ต้องมี method อะไรบ้าง และจะแบ่งคลาสอย่างไร เพื่อไม่ให้เกิดการ implement method ที่ไม่ได้ใช้งาน

SOLID Principle

- เรามาลองสมมติ Class ของลานจอดรถแห่งหนึ่ง มีดังนี้

```python
class ParkingLot:
    def park_car(self):          # Decrease empty spot count by 1
        raise NotImplementedError

    def unpark_car(self):        # Increase empty spots by 1
        raise NotImplementedError

    def get_capacity(self):      # Returns car capacity
        raise NotImplementedError

    def calculate_fee(self, car): # Returns the price based on number of hours
        raise NotImplementedError

    def do_payment(self, car):
        raise NotImplementedError
```

SOLID Principle

- สมมติว่า มีการสร้างลานจอดรถอีกแห่ง แต่ไม่เก็บเงิน หากสร้าง Class ดังนี้

```
class FreeParkingLot(ParkingLot):
    def park_car(self):          # Decrease empty spot count by 1
        print('parking')

    def unpark_car(self):        # Increase empty spots by 1
        print('unparking')

    def get_capacity(self):      # Returns car capacity
        print('get_capacity')

    def calculate_fee(self, car): # Returns the price based on number of hours
        return 0

    def do_payment(self, car):
        raise Exception("Parking lot is free")
```

SOLID Principle

- จะเห็นได้ว่าใน subclass มี method ที่ไม่ได้ใช้อยู่ 2 method คือ do_payment และ calculate_fee ซึ่งทำให้ขัดต่อหลักการ ISP เพราะหากมีการเรียกใช้ขึ้นมาก็จะเกิด Error
- สำหรับแนวทางการแก้ไข คือ ใน ParkingLot ให้ตัด method do_payment และ calculate_fee ออกไป
- และสร้าง subclass ขึ้นมา 2 class คือ PaidParkingLot และ FreeParkingLot โดยใน PaidParkingLot ให้มี method do_payment และ calculate_fee
- เพียงเท่านี้ก็จะไม่มีการบังคับให้ต้อง implement method ที่ไม่ได้ใช้งาน
- อันที่จริง ในตัวอย่างนี้ก็ขัดกับหลักการอื่นๆ ของ SOLID ที่ผ่านมาอีกด้วย เพราะมี cohesion ระหว่างกันน้อย

SOLID Principle

- ในคลาส PaymentProcessor สมมติว่าเราเพิ่มการ OTP ผ่าน SMS เข้าไป เพื่อให้มีความปลอดภัยในการใช้งานมากยิ่งขึ้น ดังนั้นจะแก้ไขคลาสเป็นดังนี้
- นอกเหนือจะต้อง implement method pay แล้วยังต้อง implement method auth_sms

```python
class PaymentProcessor(ABC):

    @abstractmethod
    def auth_sms(self, code):
        pass

    @abstractmethod
    def pay(self, order):
        pass
```

SOLID Principle

- แต่ปกติการใช้ credit card จะไม่ต้องใช้ SMS หากมีการบังคับก็จะต้อง implement แบบนี้ ทำให้ไม่เป็นไปตามหลัก เรื่อง Interface Segregation Principle

```python
class CreditPaymentProcessor(PaymentProcessor):

    def __init__(self, security_code):
        self.security_code = security_code

    def auth_sms(self, code):
        raise Exception("Credit card payments don't support SMS code authorization.")

    def pay(self, order):
        print("Processing credit payment type")
        print(f"Verifying security code: {self.security_code}")
        order.status = "paid"
```

SOLID Principle

- ในกรณีนี้ การแก้ไขคือ จะเพิ่ม subclass ของ PaymentProcessor แบบที่ใช้ SMS เพิ่มเข้ามา จากนั้นให้เครดิตการ์ดใช้ PaymentProcessor และการชำระเงินแบบอื่นๆ ให้ใช้ PaymentProcessorSMS

```
class PaymentProcessor(ABC):

    @abstractmethod
    def pay(self, order):
        pass

class PaymentProcessorSMS(PaymentProcessor):

    @abstractmethod
    def auth_sms(self, code):
        pass
```

SOLID Principle

- นอกจากนั้นยังสามารถใช้วิธี composition ในการแก้ปัญหานี้ได้ด้วย โดยการสร้างคลาส SMSAuthorizer

```python
class SMSAuthorizer:

    def __init__(self):
        self.authorized = False

    def verify_code(self, code):
        print(f"Verifying SMS code {code}")
        self.authorized = True

    def is_authorized(self) -> bool:
        return self.authorized
```

SOLID Principle

- จากนั้นคลาสที่ต้องการ authorize ผ่าน SMS ก็ให้ทำดังนี้

```python
class DebitPaymentProcessor(PaymentProcessor):

    def __init__(self, security_code, authorizer: SMSAuthorizer):
        self.security_code = security_code
        self.authorizer = authorizer

    def pay(self, order):
        if not self.authorizer.is_authorized():
            raise Exception("Not authorized")
        print("Processing debit payment type")
        print(f"Verifying security code: {self.security_code}")
        order.status = "paid"
```

SOLID Principle

- Dependency Inversion Principle
- เนื้อหาของหลักการข้อนี้ คือ

  “High level modules should not depend upon low level modules. Both should depend upon abstractions.”

  “Abstractions should not depend upon details, details should depend upon abstractions.”

- “ของที่เป็น High level module ไม่ควรไปผูกติดกับ Low level module และทั้งสองควรรู้จักกันในรูปแบบ abstraction เท่านั้น" กับ "Abstraction ไม่ควรรู้รายละเอียดการทำงาน แต่โค้ดที่ทำงานที่แท้จริงต้องทำตาม Abstraction ที่วางไว้"

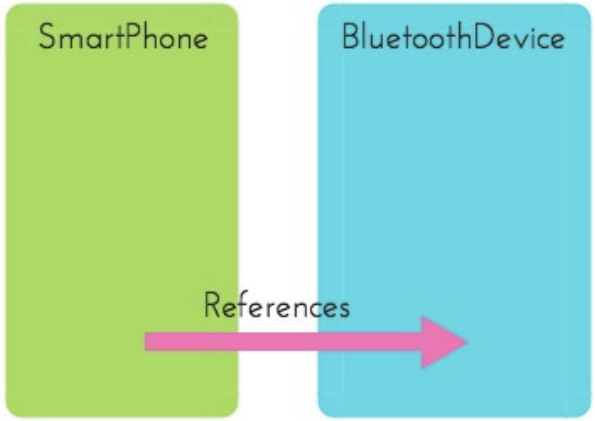
SOLID Principle

- ก่อนอื่นก็ต้องรู้จัก High Level Module กับ Low Level Module ก่อน
- **High level module** คือโค้ดที่รับผิดชอบดูแลภาพรวมของระบบ ซึ่งภายใน High Level Module จะไปเรียกใช้ Low level module ต่างๆ มาทำงานอีกที (สมมติว่าเป็นงานก่อสร้าง High Level Module คือคนคุมงานก่อสร้าง คนคุมงานจะไม่โบกปูนเอง แต่จะดูภาพรวมว่าต้องทำอะไรงานถึงจะเสร็จ)
- **Low level module** คือโค้ดที่มีหน้าที่ทำงานจริงๆ เช่น เขียนลงไฟล์ สมมติว่าเป็นงานก่อสร้าง ก็คือคนโบกปูน)
- จะเห็นว่าถ้าออกแบบระบบให้ดี ทั้ง High Level และ Low Level ไม่ควรจะต้องขึ้นต่อกัน เช่น คนคุมงาน ก็สามารถคุมงานคนไหนก็ได้ที่โบกปูนเป็น และ คนโบกปูน ก็สามารถทำงานไหนก็ได้ ที่ได้รับการสั่งงาน คือ จะมีการขึ้นต่อกันน้อย

SOLID Principle

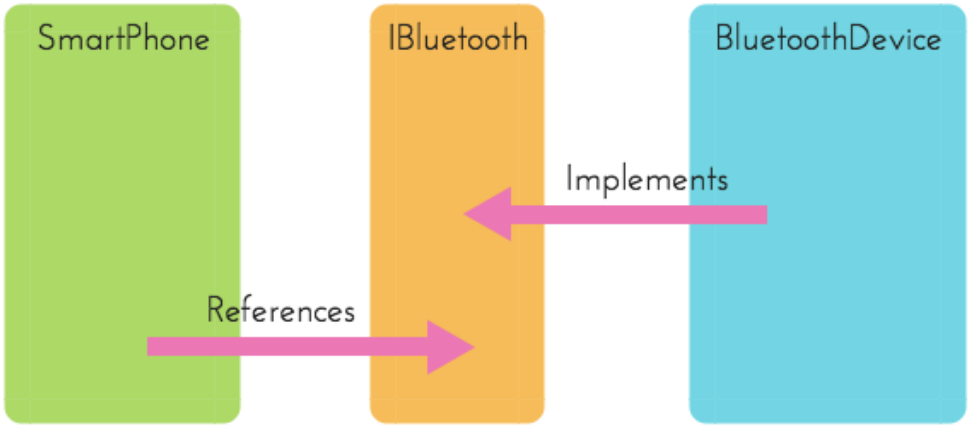
- สมมติมี Class ดังนี้

```
class BluetoothDevice:
    def connect(self):  # do low level network tasks
        pass
    def scan(self):
        pass

class SmartPhone:
    def __init__(self,tel_no):
        self.__tel_no = tel_no
        self.__bt = BluetoothDevice
        # create object BluetoothDevice in Class

    def connect_bluetooth(self):
        self.__bt.connect()
```



SOLID Principle

- คลาสตาม slide ก่อนหน้า ออกแบบได้ถูกหลัก Single Responsibility เพราะได้แยกคลาสของ SmartPhone และ BluetoothDevice ออกจากกันตามหน้าที่แล้ว
- แต่ก็ขัดกับหลัก DIP เพราะ High level module ไปผูกติดกับ Low level module หากมีการเปลี่ยนแปลงของ Low level module ก็จะกระทบกับ High level module
- ดังนั้นจะ redesign เป็นแบบนี้ ตัว SmartPhone จะเป็นอิสระจาก BluetoothDev.



SOLID Principle

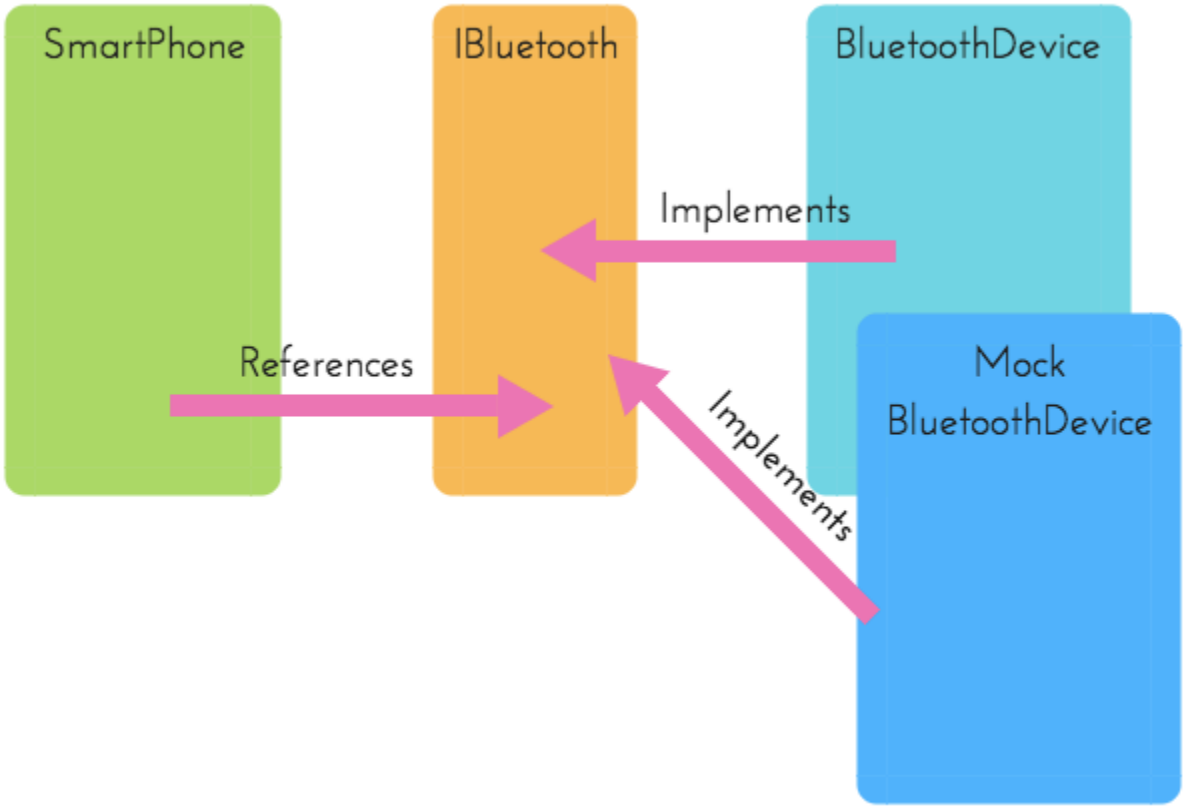
```python
from abc import ABC

class IBluetooth(ABC):
    def connect(self):  # do low level network tasks
        pass
    def scan(self):
        pass

class BluetoothDevice(IBluetooth):
    def connect(self):  # do low level network tasks
        print('bluetooth connect')
    def scan(self):
        print('bluetooth scan')
```

```python
class SmartPhone:
    def __init__(self,tel_no, bt):
        self.__tel_no = tel_no
        self.__bt = bt

    def connect_bluetooth(self):
        self.__bt.connect()

bt = BluetoothDevice()
phone = SmartPhone('0812345678', bt)
```

- จะเห็นว่ามีการสร้าง object ขึ้นก่อน แล้วส่งเป็น argument ให้กับ SmartPhone

SOLID Principle

- นอกจากนั้นยังสามารถเชื่อมต่ออุปกรณ์ Bluetooth แบบใหม่ง่ายขึ้น (หรือ test)



For your attention