01076105, 01075106

Object Oriented Programming

Object Oriented Programming Project

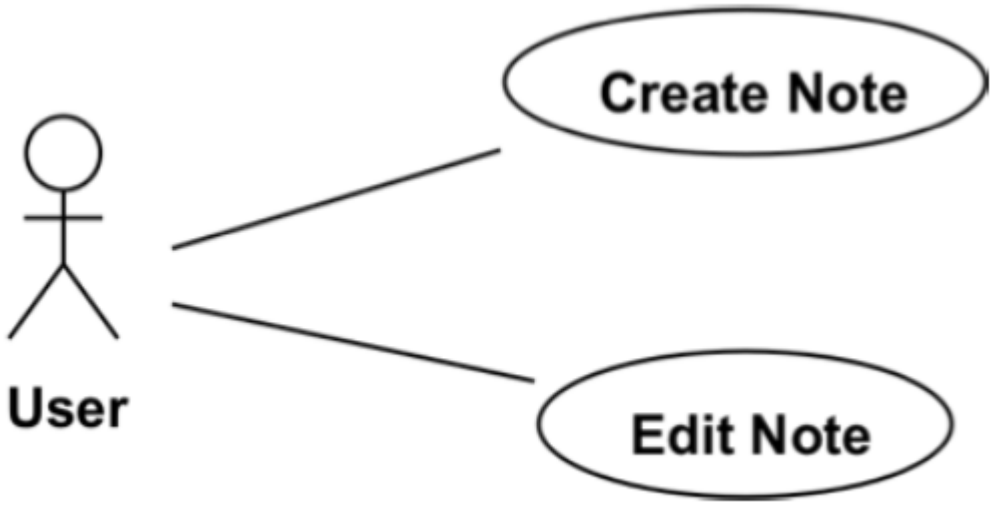
SOLID Principle

Case Study : Note-Taking App

- **ตัวอย่างการออกแบบ App Note-Taking**
- สามารถสร้าง และ แก้ไข Note ที่เป็นตัวอักษร
- สามารถแนบรูปภาพได้
- สามารถเก็บรูปภาพลายมือได้ (hand-drawn sketch)
- สามารถป้องกัน note ที่เป็นความลับ (sensitive) โดยใช้ password
- สามารถ sync ข้อมูลไปยัง cloud ได้ (เช่น iCloud, google drive)

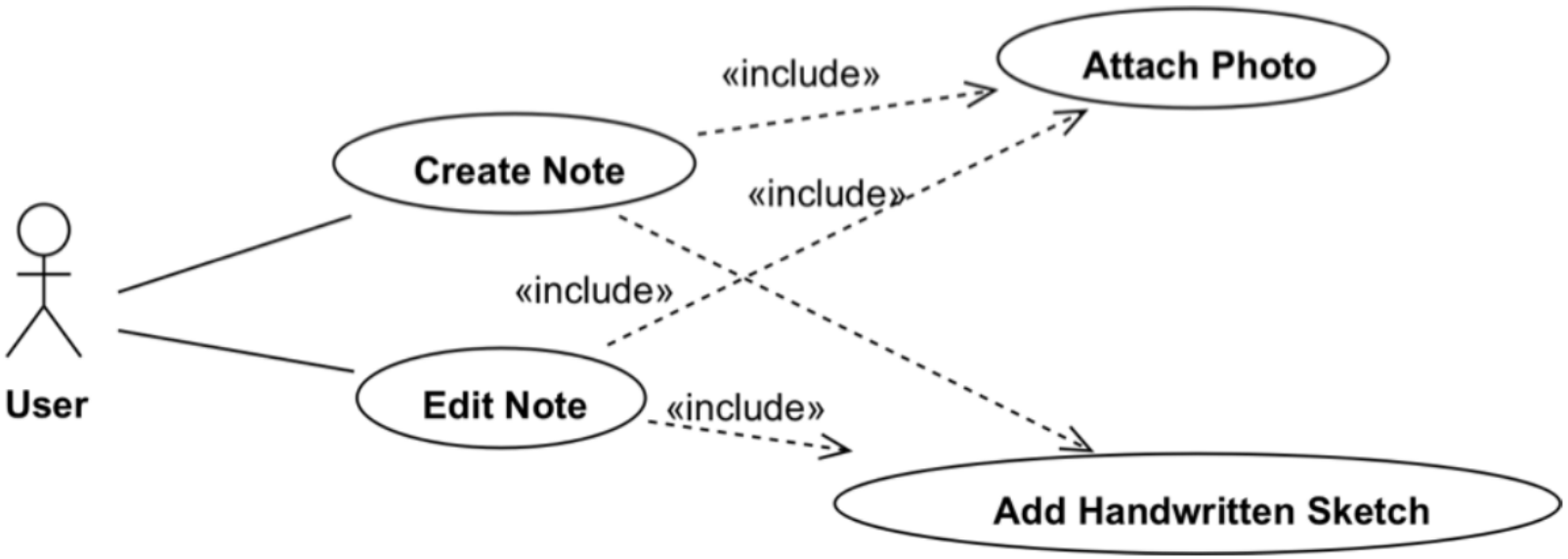
Case Study : Note-Taking App

- เราจะเริ่มทำ Use Case Diagram แบบ Step by Step โดยขั้นตอนแรกมีข้อกำหนดว่าสามารถสร้างและแก้ไข Note ก็สามารถเขียน Use Case ได้ดังนี้
- โดยระบบนี้จะมีเพียง Actor เดียว



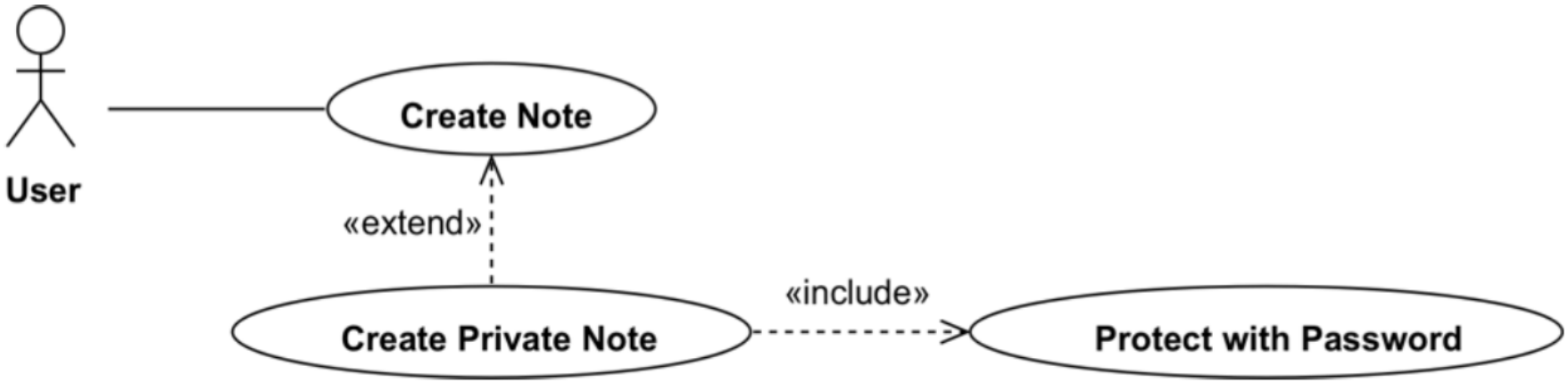
Case Study : Note-Taking App

- จากข้อกำหนดว่าสามารถแนบรูปถ่าย และ สามารถเก็บลายมือเขียนได้ เราจึงเพิ่มอีก 2 Use Case เข้าไป ซึ่งเนื่องจากเป็นการทำงานแบบเรียกใช้ความสามารถเพิ่มเติม จึงใช้เป็น <<include>> (จะใช้ก็ได้ ไม่ใช้ก็ได้)



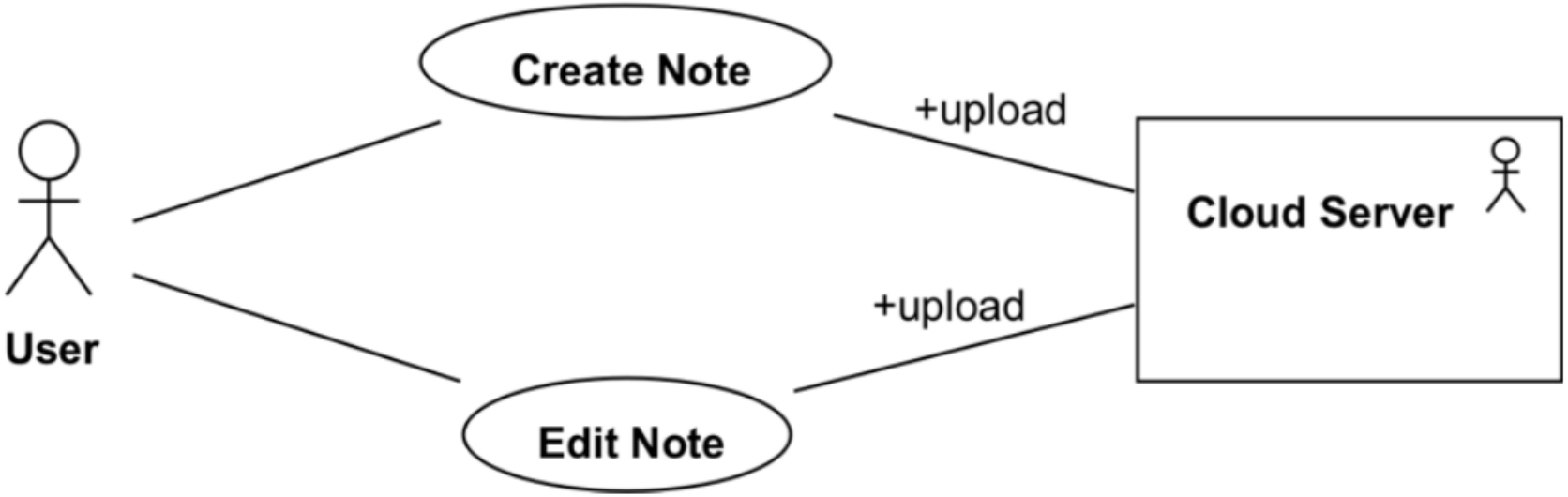
Case Study : Note-Taking App

- จากข้อกำหนดเรื่องสามารถป้องกัน Note ที่ Sensitive โดยใช้ Password จะสร้าง Use Case ใหม่ ชื่อ Create Private Note ตามรูป
- เนื่องจากการสร้าง Note ประเภทนี้ เป็นสร้าง Note อีกประเภทหนึ่ง คือ เป็น private note ที่มีการเข้ารหัส มีลักษณะของการ**ขยายความสามารถ** จึงใช้ <<extend>> และมีการเรียกใช้การป้องกันแบบ password ในแบบเรียกใช้ จึงใช้เป็น <<include)>>



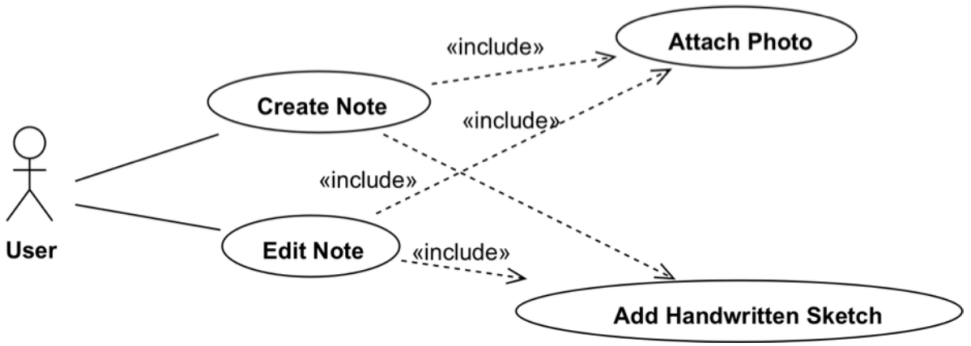
Case Study : Note-Taking App

- ข้อกำหนดสุดท้าย คือ สามารถ sync ข้อมูลไปยัง cloud ได้ เราจะมองว่า cloud เป็น Actor อีกตัว ที่เป็น External System ดังนั้นจะเขียนแบบนี้



Case Study : Note-Taking App

- ต่อไปจะสร้าง Class Diagram จะเริ่มจาก บันทึกแบบข้อความ จะได้ Class เริ่มแรกเป็น



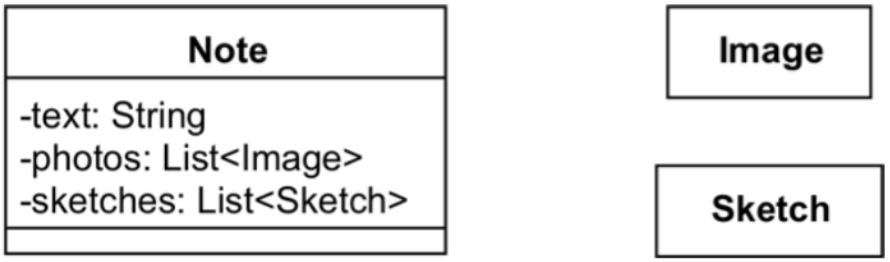
Note
-text: String

- จากนั้นเมื่อเพิ่ม ภาพและลายมือเขียน เข้าไป ก็จะได้เป็น Class แบบนี้

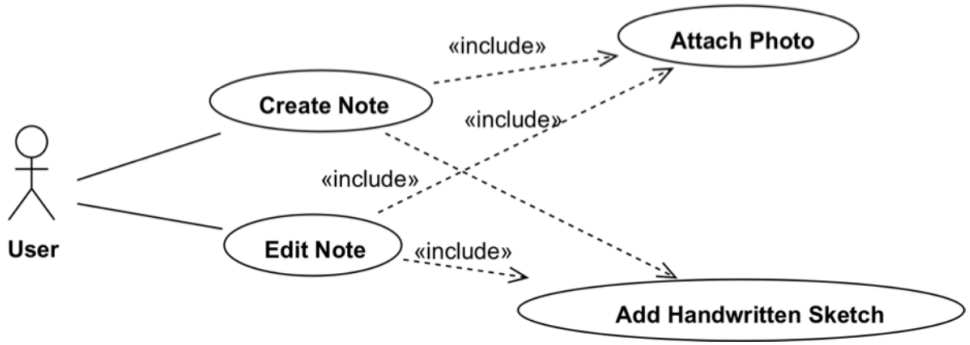
Note
-text: String -photos: List<Image> -sketches: List<Sketch>

Case Study : Note-Taking App

- การที่ Class Note มี photo และ sketch แสดงว่าต้องมีความสัมพันธ์กับอีก 2 Class คือ Image และ Sketch ตามรูป

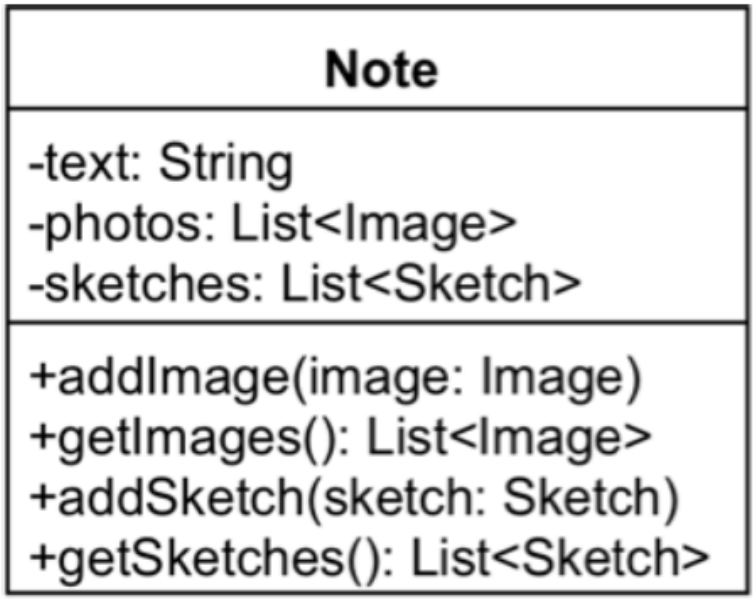


- และจาก Use Case ที่เป็น <<include>> แสดงว่าเพิ่มรูปและ Sketch ภายหลังได้



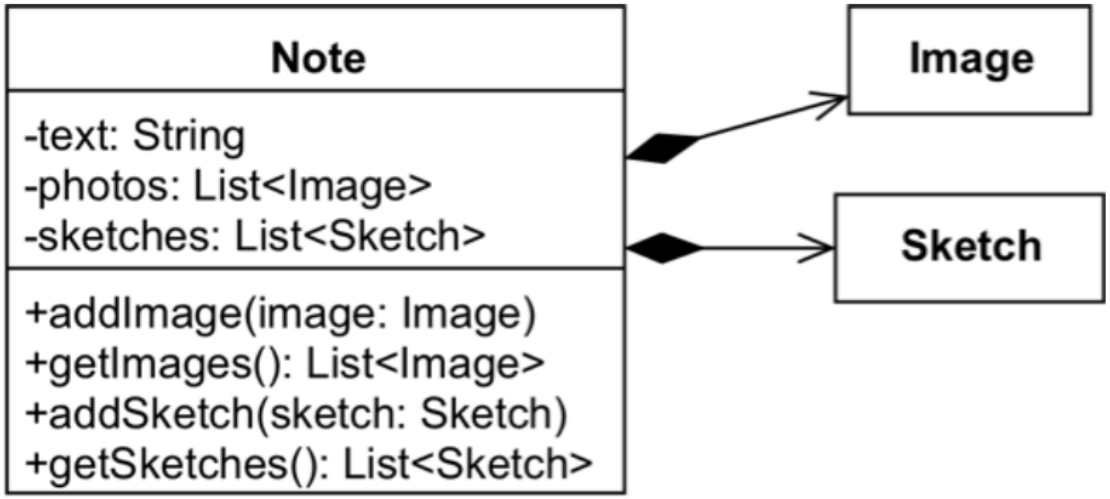
Case Study : Note-Taking App

- ดังนั้นใน Class จะต้องมี Method สำหรับ เพิ่มรูปเข้าไป ดังนั้น Class จะมีโครงสร้างดังนี้
  - *addImage สำหรับเพิ่มรูป*
  - *getImage สำหรับดึงรูปทั้งหมดของ Note ฉบับนั้น*
  - *addSketch สำหรับเพิ่ม Sketch*
  - *getSketch สำหรับดึง Sketch ทั้งหมดของ Note ฉบับนั้น*



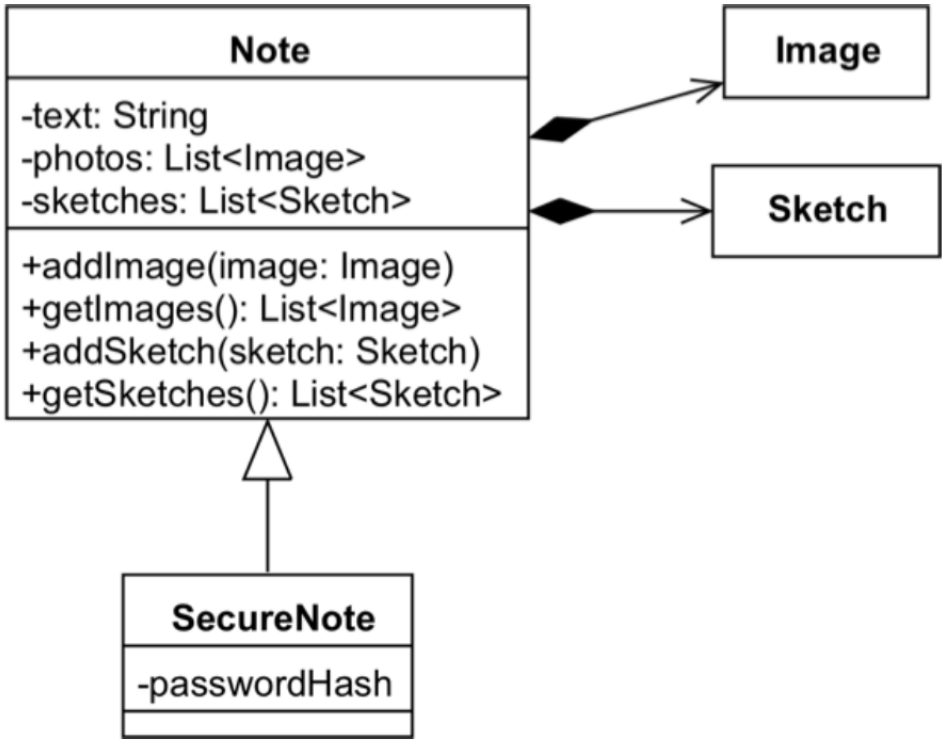
Case Study : Note-Taking App

- ต่อไปจะพิจารณาประเด็นเรื่องความสัมพันธ์ระหว่าง Class
- จากที่ Image และ Sketch เป็น Instance ของ Class ดังนั้นจึงอาจเป็นความสัมพันธ์แบบ Association หรือ Aggregation หรือ Composition ก็ได้
- แต่เนื่องจากในระบบนี้ รูปไม่สามารถอยู่อย่างโดดเดี่ยวได้ ดังนั้นความสัมพันธ์จึงเป็น Composition



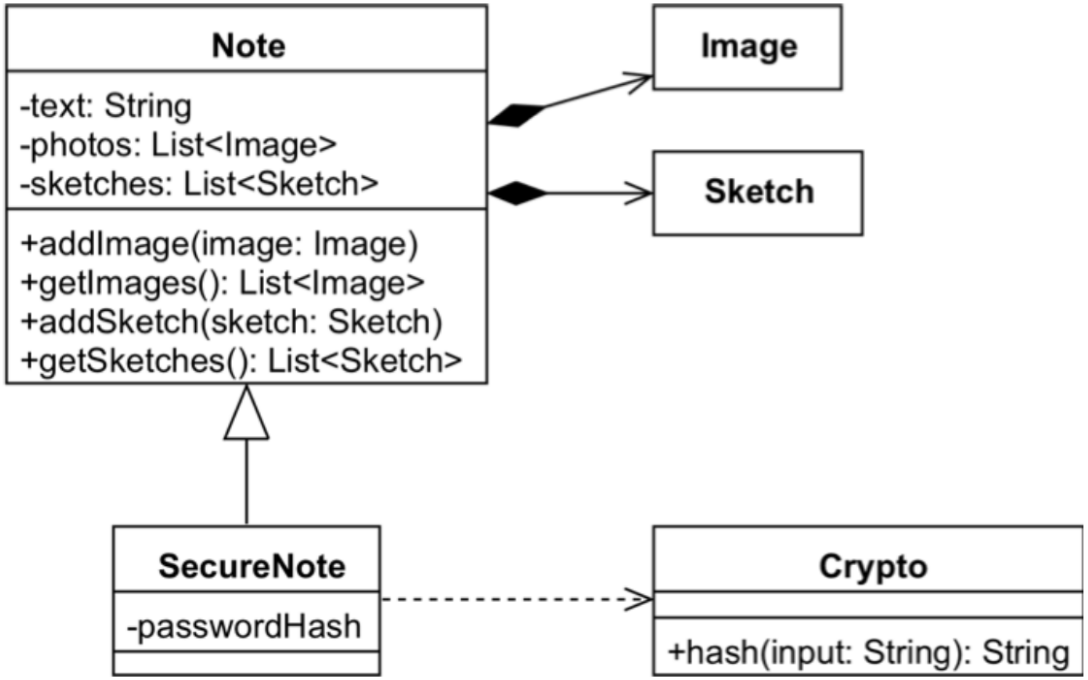
Case Study : Note-Taking App

- ข้อกำหนดเพิ่มเติม คือ สามารถป้องกัน note ที่เป็นความลับ ซึ่งพิจารณาว่าเป็น Note เหมือนกัน แต่มีการเข้ารหัส จึงเป็น Class ลูกของ Note โดยมี attribute เพิ่ม คือ รหัสผ่านที่จะใช้ในการอ่าน note



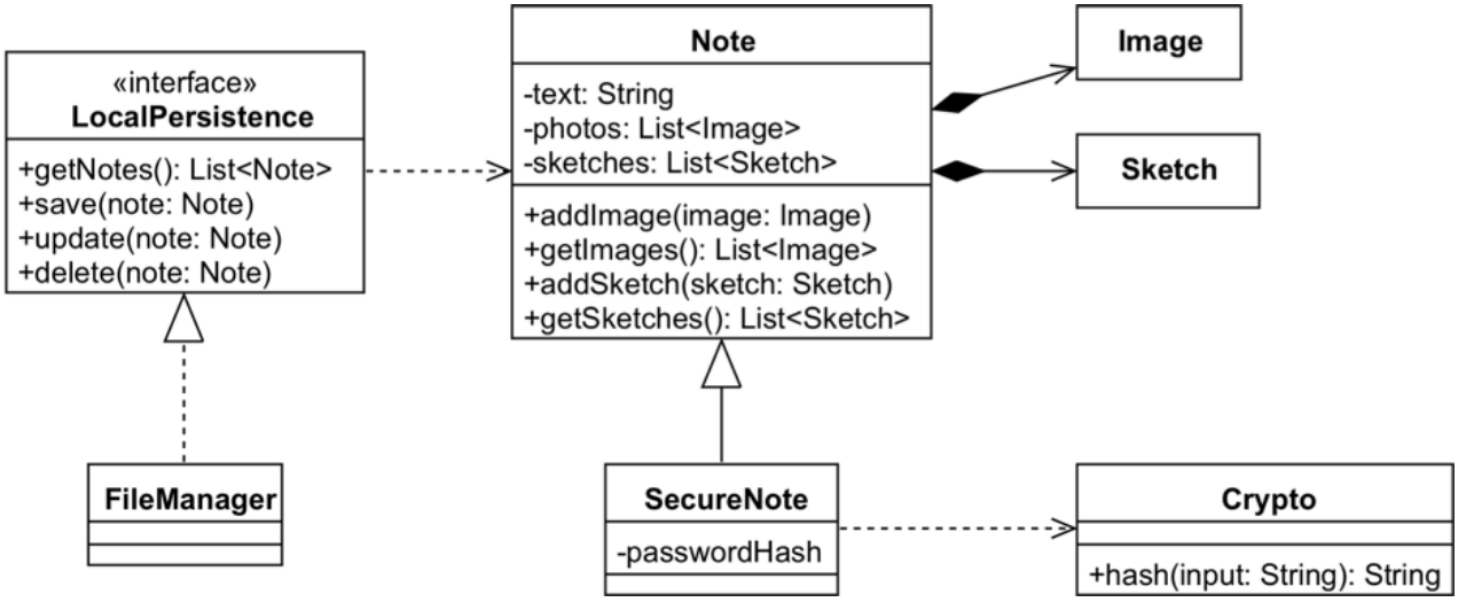
Case Study : Note-Taking App

- ในการเข้ารหัสนั้น เนื่องจากหลักการสำคัญหนึ่งของการออกแบบคลาส คือ คลาสควรรับผิดชอบเรื่องเดียว ดังนั้นการเข้ารหัสควรเป็นหน้าที่ของอีกคลาส เนื่องจากไม่ได้มีการสร้าง instance ของอีกคลาส เป็นการเรียกใช้เฉยๆ จึงเป็นแบบ Dependency

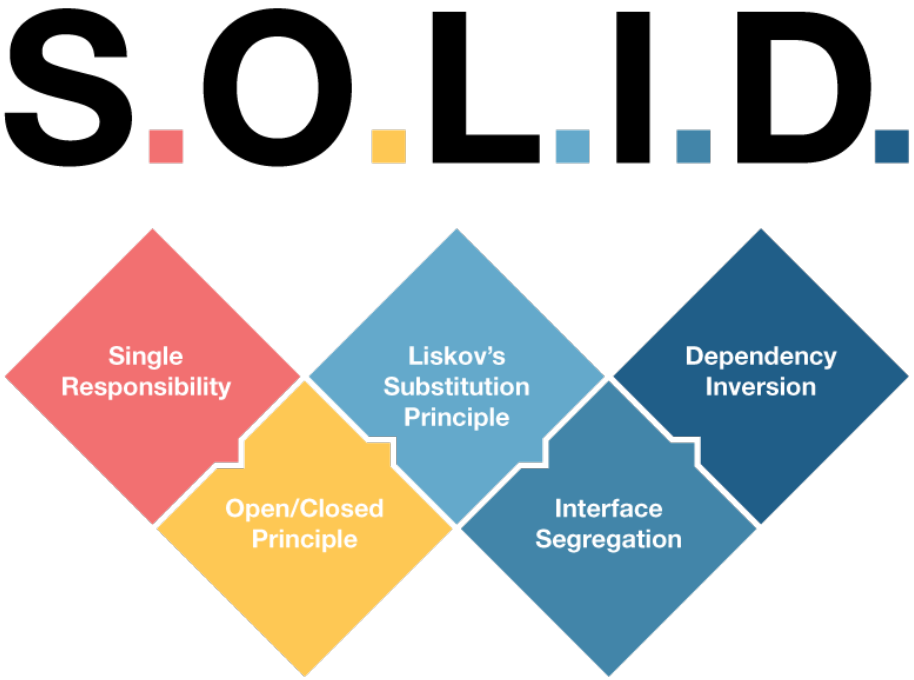


Case Study : Note-Taking App

- ใน OOP จะมีการดำเนินการอย่างหนึ่งเรียกว่า Interface ใช้สำหรับกำหนด spec ของคลาสที่จะถูกใช้งาน คล้ายกับ abstract classes แต่มีข้อกำหนดมากกว่า โดย Interface จะเป็นคลาสต้นแบบเพื่อให้คนที่นำไปใช้งานต้อง Implement ตามที่กำหนด (ใน python ไม่มี interface)



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
  - ในกรณีของ code ตัวอย่าง ควรจะนำ 2 method ล่าง แยกไปสร้างเป็นอีก Class

```python
class Square:

    def __init__(self, side):
        self.__side = side

    def calculate_area(self):
        return self.__side * 2

    def calculate_perimeter(self):
        return self.__side * 4

    def draw(self): # render image on display
        pass

    def rotate(self):
        pass
```

SOLID Principle

- Single Responsibility Principle
  - นอกจากนั้น ส่วนประกอบควรขึ้นต่อกัน (coupling) ให้น้อยที่สุดอีกด้วย
  - จาก Code ส่วนของ method save ซึ่งนำข้อมูลนักศึกษา save ลง database แต่ใน code กำหนดว่า save ลง database ที่ชื่อ MySQL ซึ่งกรณีนี้ถือว่าเป็น tight coupling คือ ขึ้นต่อกันมากเกินไป เพราะเมื่อมีการเปลี่ยนฐานข้อมูลจะต้องแก้ code
  - กรณีนี้ควรแยกส่วนที่เชื่อมต่อกับฐานข้อมูลออกเป็นอีก class

```python
class Student:
    def __init__(self, student_id, name):
        self.__student_id = student_id
        self.__name = name

    def save(self):
        pass    # save student to mysql DB

    def get_student(self, student_id):
        return __student_id

    def get_name(self, name):
        return __name
```

SOLID Principle

- Single Responsibility Principle
- แนวทางการตรวจสอบการออกแบบคลาส จะใช้หลักการดังนี้

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

- จากคลาสใน slide ที่แล้ว คลาสจะมีการแก้ไข เมื่อ 1) เปลี่ยนวิธีการกำหนดรหัสนักศึกษาหรือชื่อนักศึกษา 2) เปลี่ยนฐานข้อมูล
- แต่หากเราแยกส่วนของฐานข้อมูลออกไป ก็จะเหลือเหตุผลเดียว ซึ่งจะเป็นไปตามหลักการ
- ทั้งนี้ก็เพื่อให้ ความจำเป็นต้องแก้ไข คลาส เกิดขึ้นได้น้อยที่สุด

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 : exercise

- จากโจทย์นี้ ควรแก้ไข อย่างไร

```
class Employee:

    def save(self): # save to MySQL
        pass

    def calculate_tax(self): # full_time, contract
        pass

    def get_employee_no(self):
        pass

    def set_employee_no(self):
        pass

    def get_employee_name(self):
        pass

    def set_employee_name(self):
        pass
```

SOLID Principle

- Open-Closed Principle
- หลักการข้อนี้มีอยู่ว่า ส่วนประกอบของ Software ควรจะ Close สำหรับการแก้ไข แต่ Open สำหรับการเพิ่มเติม
- หมายความว่าหลังจากที่ Software เขียนเสร็จแล้ว ไม่ควรมีการแก้ไขใดๆ อีก กรณีของ Class คือ ไม่ไปแตะต้องคลาสนั้นอีก กรณีที่มีการเพิ่มเติม ก็ควรใช้วิธีการ Inheritance มากกว่าจะไปแก้ไขที่ Class เดิม

SOLID Principle

- Open-Closed Principle
- จากโปรแกรมจะเห็นว่า การเปลี่ยนความเร็ว จะขึ้นกับคำสั่ง เพราะ function จะทำบางอย่างที่ต่างไป ขึ้นกับว่าใน do_something จะส่งอะไรมา

```
def car_behavior(total_speed:float, do_something:str, speed:flaot):
    if do_something == "accelarate":
        total_speed += speed
    elif do_something == "brake":
        total_speed += -speed * 0.2
    print("{}! current speed:".format(do_something),total_speed)
    return true

print(car_behavior(100, "brake", 30))
```

SOLID Principle

- Open-Closed Principle เพื่อให้เป็น OCP เราจะแก้ไขเป็นดังนี้

```python
from abc import ABC, abstractmethod
class CarDoes(ABC):
    @abstractmethod
    def doit(self):
        pass

class CarBr(CarDoes):
    def doit(self,total_speed,speed):
        total_speed += -speed * 0.2
        print("brake! current speed:",total_speed)
        return total_speed

class CarAcc(CarDoes):
    def doit(self,total_speed,speed):
        total_speed += speed
        print("accelerate! current speed:", total_speed)
        return total_speed

def car_behavior(cardoes:CarDoes, total_speed, speed):
    cardoes.doit(total_speed, speed)
    return true
```

SOLID Principle

- Open-Closed Principle
- จาก slide ที่ผ่านมา แม้ว่าเราจะมีคำสั่งอื่นๆ เพิ่มเติม เราก็ใช้วิธีการเพิ่ม Class ที่ Inherit มาจาก CarDoes โดยจะไม่มีการแก้ไขคลาสอื่นๆ
- ดังนั้นในการเขียนโปรแกรม กรณีที่อาจมีเงื่อนไขใหม่ๆ ในโปรแกรม (ที่ตอนนี้ยังไม่มี ให้สร้างในลักษณะของ Interface เพื่อให้สามารถขยายเพิ่มเติมในอนาคตได้
- การเพิ่ม feature ก็จะทำได้ง่าย การทดสอบก็จะง่ายด้วย
- เป็นการทำ decoupling ซึ่งเป็นการทำตาม SRP ไปในตัว

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 ตัวอย่าง มีคลาส รถยนต์ (Car) กับรถแข่ง (RacingCar) ซึ่งเป็น subclass
- ในรถยนต์มี cabin (ห้องโดยสาร) แต่ในรถแข่งไม่มี จะเรียก cockpit แทน
- ดังนั้นในคลาส RacingCar จึงไม่ทำ method get_cabin_width
- ทำให้ขัดกับหลัก LSP เพราะทำให้ความสามารถของ subclass ลดลง (ใช้ get_cabin_width ไม่ได้)

```python
class Car:
    def get_cabin_width(self):
        return cabin_width

class RacingCar(Car):
    def get_cabin_width(self):
        pass # unimplemented

    def get_cockpit_width(self):
        return cockpit_width
```

SOLID Principle : LSP

- วิธีการตรวจสอบง่ายๆ ของปัญหา LSP คือ ให้สมมติว่าถ้าจับ Object ในทุกลำดับชั้นมาไว้ใน list เดียวกัน เช่น สมมติเขียนว่า

```
car1 = Car()
car2 = RacingCar()

lst = [car1,car2]
for i in lst:
    print(i.get_cabin_width())
```

- โปรแกรมจะต้องทำงานได้ แต่โปรแกรมนี้จะทำงานไม่ได้ เพราะในคลาส Racing Car ไม่มี method get_cabin_width()
- กรณีนี้แนวทางการแก้ไข คือ สร้าง Super Class ขึ้นใหม่ เช่น Vehicle()

SOLID Principle : LSP

```python
class Rectangle:
    def __init__(self, width: float = 0, height: float = 0):
        self._width = width
        self._height = height

    @property
    def width(self) -> float:
        return self._width

    @width.setter
    def width(self, value: float) -> None:
        self._width = value

    @property
    def height(self) -> float:
        return self._height

    @height.setter
    def height(self, value: float) -> None:
        self._height = value
```

```python
class Square(Rectangle):
    def get_width(self) -> float:
        return self._width

    def set_width(self, value: float) -> None:
        self._width = value
        self._height = value

    def get_height(self) -> float:
        return self._height

    def set_height(self, value: float) -> None:
        self._width = value
        self._height = value

    width = property(get_width, set_width)
    height = property(get_height, set_height)
```

SOLID Principle : LSP

```python
class Rectangle:
    def __init__(self, width: float, height: float):
        self._width = width
        self._height = height

    @property
    def width(self) -> float:
        return self._width

    @width.setter
    def width(self, value: float):
        self._width = value

    @property
    def height(self) -> float:
        return self._height

    @height.setter
    def height(self, value: float):
        self._height = value

    @property
    def area(self) -> float:
        return self.width * self.height
```

```python
class Square(Rectangle):
    def __init__(self, side: float):
        self.__side = side
        super(Square, self).__init__(side, side)

    @property
    def width(self) -> float:
        return self.__side

    @width.setter
    def width(self, value: float):
        self.__side = value

    @property
    def height(self) -> float:
        return self.__side

    @height.setter
    def height(self, value: float) -> None:
        self.__side = value

r = Rectangle(width=10, height=5)
print(r.area)
# 50
s = Square(side=5)
print(s.area)
```

SOLID Principle : LSP

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

SOLID Principle : LSP

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

```python
class Product:
    discount = 20

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

SOLID Principle : LSP

- 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 class”
- เพราะบาง class ไม่ควรจะโดนบังคับให้ Implement method ที่ไม่ได้ใช้งาน
- อธิบาย คือ กรณีที่มี Object ที่คล้ายๆ กัน ให้แยกแยะว่าแต่ละ Object ต้องมี Method อะไรบ้าง และจะแบ่ง Class อย่างไร เพื่อไม่ให้เกิดการ Implement method ที่ไม่ได้ใช้งาน

SOLID Principle : ISP

- 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 : ISP

- สมมติว่า มีการสร้างลานจอดรถอีกแห่ง แต่ไม่เก็บเงิน หากสร้าง Class ดังนี้

```python
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 : ISP

- 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

- 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"

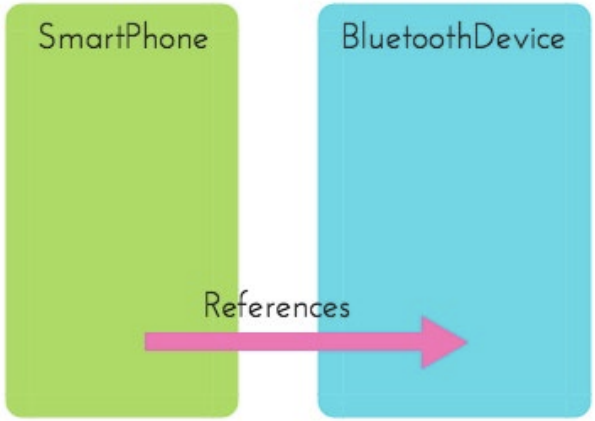
SOLID Principle : DIP

- 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 : DIP

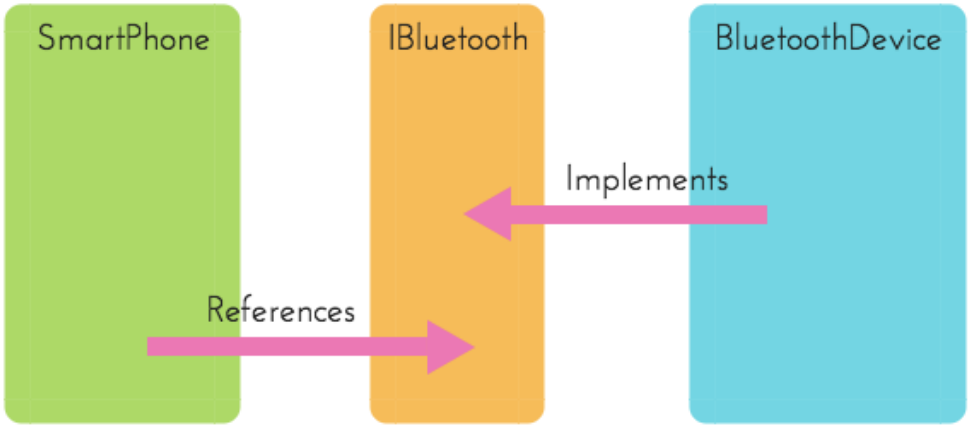
- สมมติมี Class ดังนี้

```python
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 : DIP

- คลาสตาม slide ก่อนหน้า ออกแบบได้ถูกหลัก Single Responsibility เพราะได้แยก Class ของ 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 : DIP

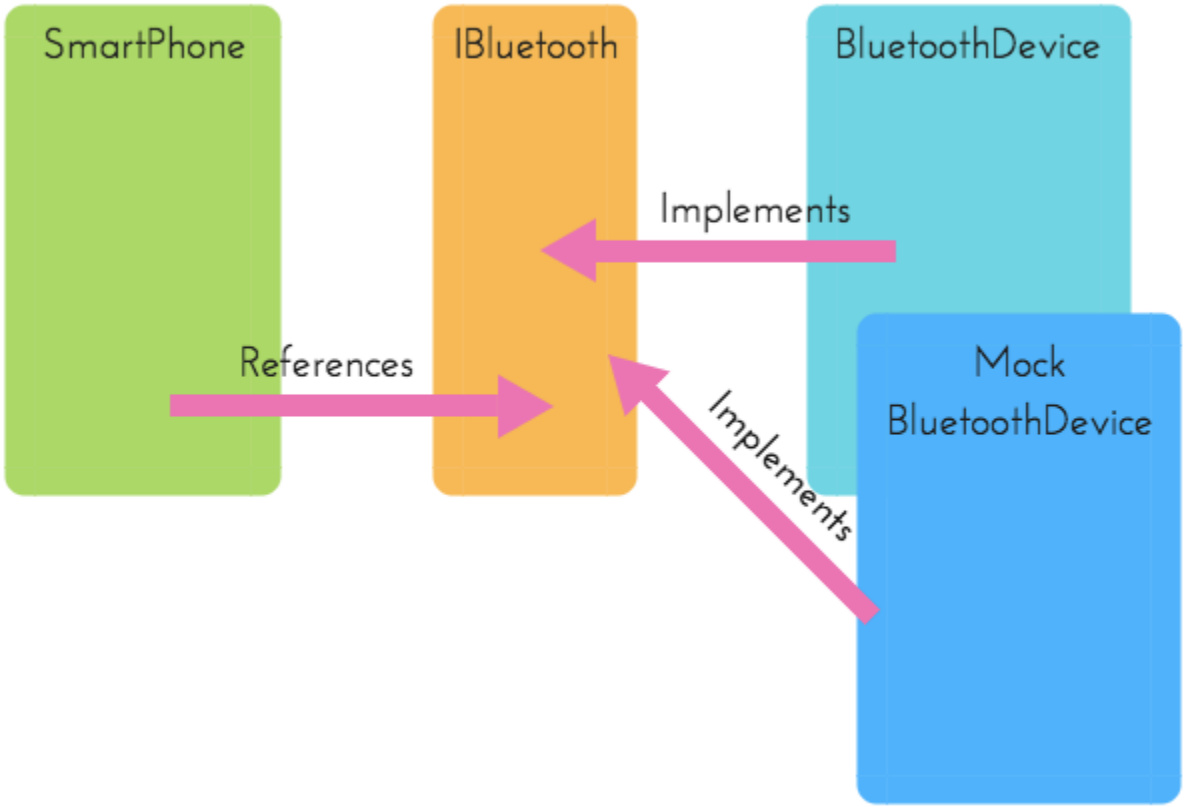
```python
from abc import ABC

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

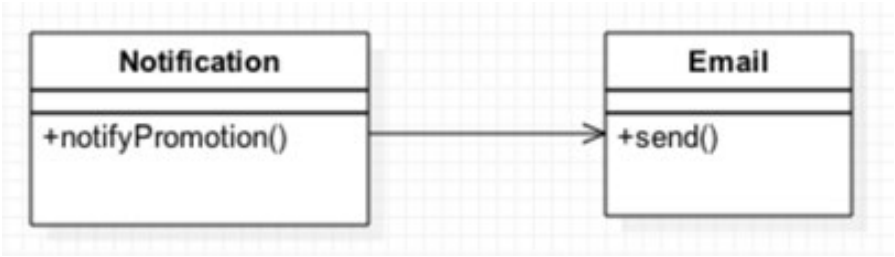
SOLID Principle : DIP

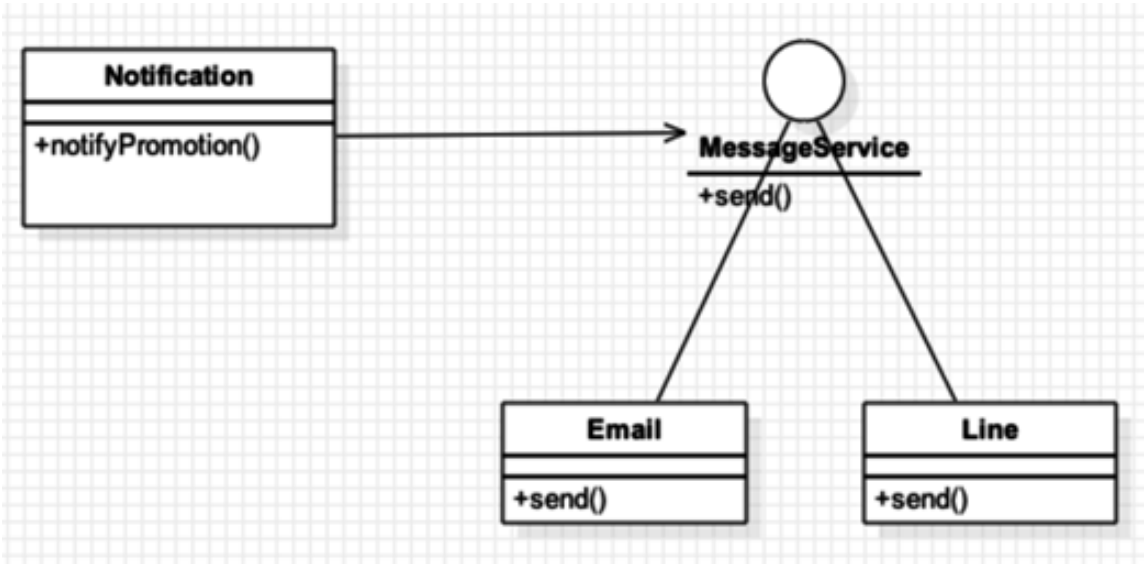
- นอกจากนั้นยังสามารถเชื่อมต่ออุปกรณ์ Bluetooth แบบใหม่ง่ายขึ้น (หรือ test)



SOLID Principle : DIP

- จะเห็นว่าเป้าหมายของ DIP คือต้องการขจัด code ที่ผูกมัดกันแบบแน่นๆ ออกไป ช่วยลดความซับซ้อนของ code และลดเวลาในการดูแลรักษา code อีกด้วย
- ยกอีกตัวอย่าง





SOLID Principle : DIP

- เพื่อให้ลดการยึดคิดระหว่างคลาส เราต้องมีการทำงานอีก 1 อย่างเรียกว่า Dependency Injection (DI) เพื่อลด coupling ระหว่าง Class โดยการสร้าง Dependency Object ให้กับ Object ที่ต้องใช้เพื่อไม่ต้องสร้าง Object ใน Object
- DI มีวิธีการย่อยๆ อยู่ 3 วิธี
  - Constructor Injection คือ การส่ง Object ผ่านทาง Constructor (ตัวอย่างใน Slide ที่ 40 ใช้วิธีนี้) ข้อเสีย คือ เปลี่ยน Object ไม่ได้
  - Setter Injection คือ การส่ง Object ผ่านทาง Setter ข้อดี คือ เปลี่ยน Object ได้ ข้อเสียคือต้องเรียก setter อีก 1 ครั้ง
  - Method Injection คือ การส่ง Object ผ่านไปที่ Method เฉพาะเมื่อมีการใช้งาน

Assignment #2

- ให้เลือกระบบจำนวน 1 ระบบ จะเป็นอะไรก็ได้ โดยมีเงื่อนไข ดังนี้
  - ต้องประกอบด้วย Class ไม่น้อยกว่า 5 Class
  - ต้องมี Class ที่ Inherit กัน อย่างน้อย 1 คู่
- ให้เขียน Use Case Diagram และ Use Case Description ของทุก Use Case
- ให้เขียน Class Diagram ของทุก Class
- ให้เขียน Sequence Diagram ของระบบงานหลัก ให้สมบูรณ์

Assignment #2

- ให้เขียนโปรแกรมที่ทำงานได้ตามระบบที่ออกแบบไว้ โดยโครงสร้างโปรแกรมต้องเป็นไปตาม Class Diagram และ Sequence Diagram
- โครงสร้างของ Class ต้องไม่ขัดหลัก SOLID
- การเขียนโปรแกรม ต้องเป็นไปตามหลักที่ได้เรียนไปก่อนหน้านี้ มีการใช้ getter, setter ตามความเหมาะสม มีการดัก exception โดยต้องไม่เกิน exception error ในโปรแกรม

Assignment #2

- การออกแบบโปรแกรม จะเป็นแบบ Client/Server คือ แยก Frontend กับ Backend ออกจากกัน โดย Frontend จะเป็น TkInter หรือจะใช้ Web ก็ได้
- ในการเรียนครั้งต่อไป อ. จะสอนการติดต่อระหว่าง Client/Server โดยจะมี 2 รูปแบบ คือ ใช้ Socket และผ่าน API
- คะแนน ส่วนทฤษฎี 20 % และ Lab 10 %
- ให้มีการรายงานความคืบหน้าในทุกวันศุกร์
- กำหนดให้มีการนำเสนอการทำงานทั้งหมดในวันที่ 6 พฤษภาคม

For your attention