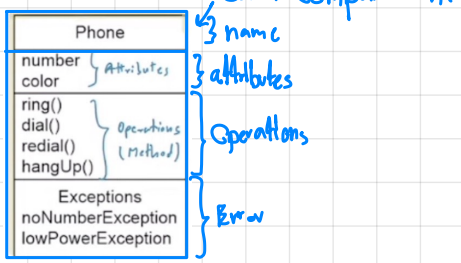
Intro and Class Diagram

Passakorn Puttama 6413050241

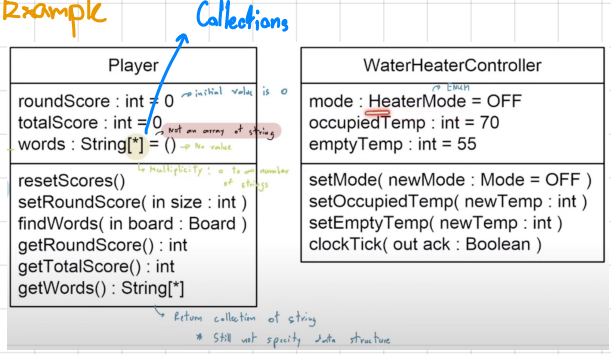
Modelling (การสร้าง code ก่อนสร้าง class diagram)

- Easier for managing and maintenance : helps "software dev" manage/handle software complexity better
- reduce time to see all of source code
- Modelling is about inc., exc. certain details.
  - include : ใส่ detail เข้ามา
  - exclude : เอา detail ออกไป

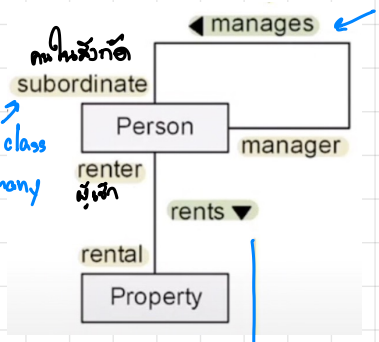
UML Symbols



Example



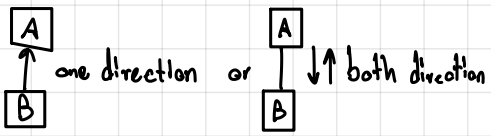
Association Lines



- A person a renter rents a property that is a rental
- A property that is a rental is rented by a person who is renter

Generalization

- is a relation between classes.
- *Not an association

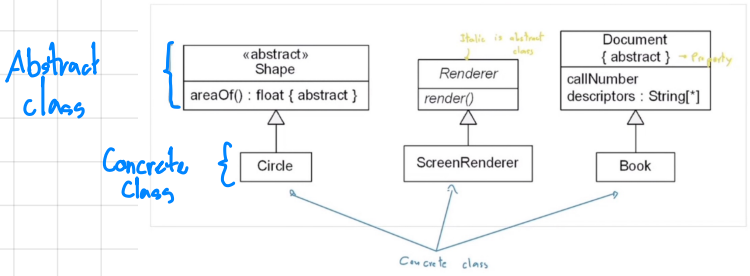


Abstract Operations and classes

- abstract operation : is an operation that without body;
- concrete operation : has a body (must implement)

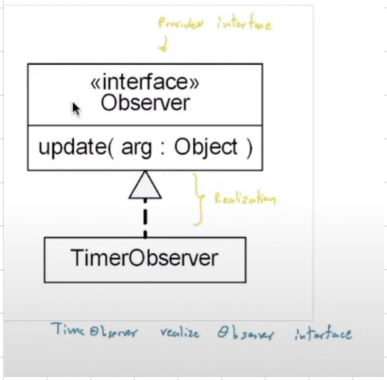
- abstract class : a class that cannot be instantiated (สร้าง Object or instance ไม่ได้) but can have concrete oper. or abstract oper in the class.
- concrete class : can be instantiated.

Example



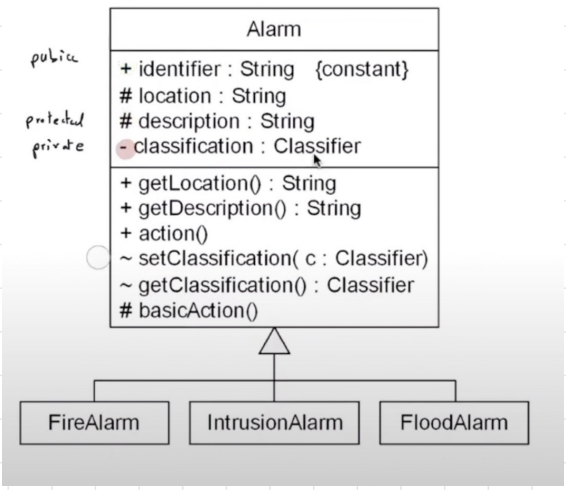
Interface (same as abstract but...)

- all operation must be abstract operations.
- and each attribute must be constant value.



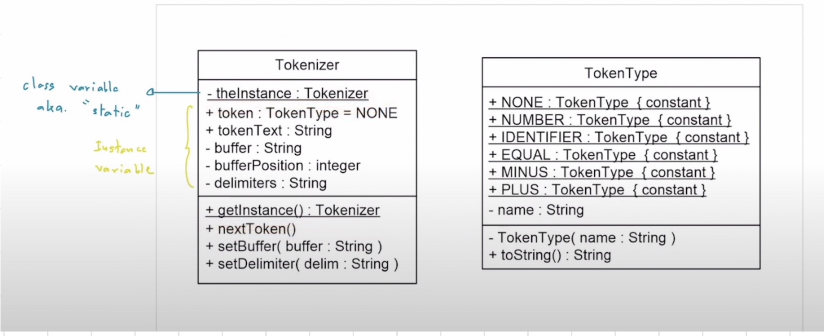
Visibility

+ Public
~ Package
# Protected
- Private

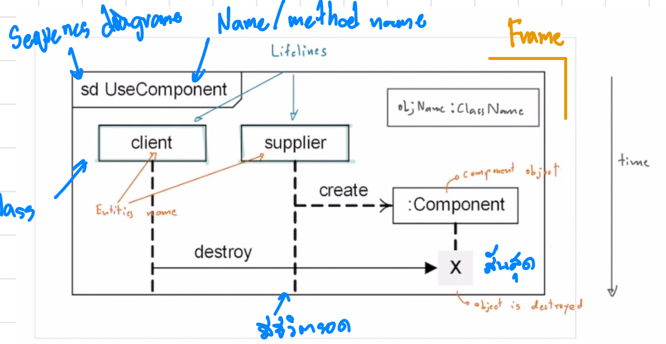


Instance variables and Operations

- instance variable: attribute whose value is stored by each instance of a class. (ค่าของมันจะต่างกันต่อ Object เป็น blueprint)
- class var: attribute whose value is stored only once and shared by all instances. (ค่าจะเป็นค่าเดียวกันกับ class; Object ไหนที่สร้างจาก class นี้จะเหมือนกันหมด)
- instance operation: called through an instance (เรียกผ่าน Object)
- static • class operation: May be called through class.

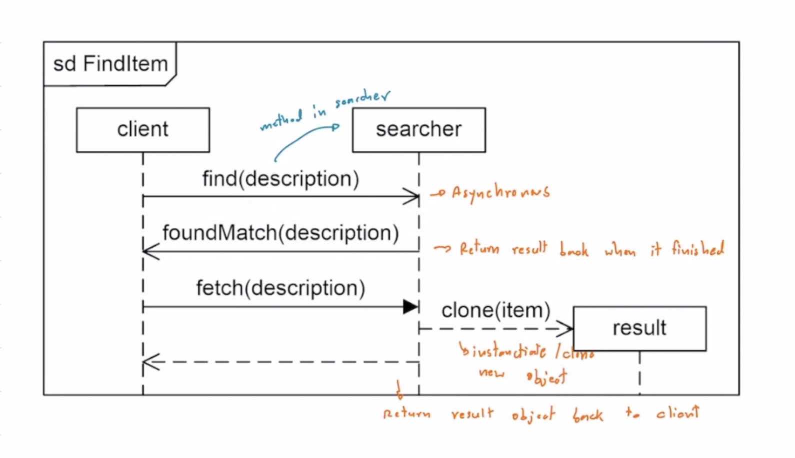


Sequence Diagram



Arrows

- Synchronous : รอจนกว่า return กลับมา
  →
- Asynchronous: ไม่รอ
  →
- Synchronous message: รอหรือไม่รับค่า return กลับมา
  - - - - ->



Design Pattern

Cohesion (การเกาะกันเป็นก้อน): ของที่อยู่ใน class ว่าเกี่ยวเนื่องกันมากแค่ไหน ถ้าเกี่ยวกันน้อยคือ low cohesion ถ้ามากคือ high cohesion (เราต้องการ high cohesion)

Example

Low → staff
- checkEmail()
- sendEmail()
- emailValidate()
- PrintLetter()

High → Staff
- salary
- emailAdd.
- setSalary(newSalary)
- getSalary()
- setEmailAddr(newE)
- getEmailAddr():

Coupling

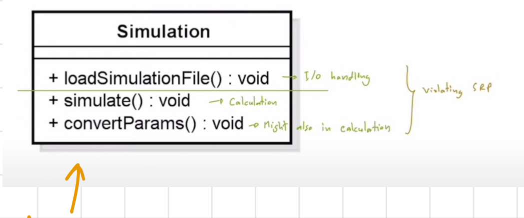
ความสัมพันธ์ class ถ้าแก้ class หนึ่งไปต้องแก้อีก class มาก เกิดไปคือ high coupling (ซึ่งไม่ดี) เราต้องการ low

* Good software design has high cohesion and low coupling. } ซึ่งเป็นการจัดโค้ด

SOLID Principle

SRP - Single Responsibility Principle

- A class should have one, and only one, reason to change (class ควรมีหน้าที่เดียว ซึ่งการมี 1 เหตุผล ที่จะต้องเปลี่ยน ซึ่งก็คือ เปลี่ยนหน้าที่ของ class)
- Always ask what the responsibility of the class/component (or microservice)* If the answer includes "and", then it's likely that SRP is violated.



หน้าที่ของมันมี 2 อย่าง คือคำนวณ loadSimulate and simulate ซึ่งไม่ได้ควร

Real-world examples:

- Java Persistence API (JPA): Defining a standardized way to manage data persisted in a relational db by using the object-relational mapping concept.
- JPA Entity Manager: Manging (i.e. persisting, update, removing, reading) the entities that are associated.

OCP - Open/Closed Principle

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

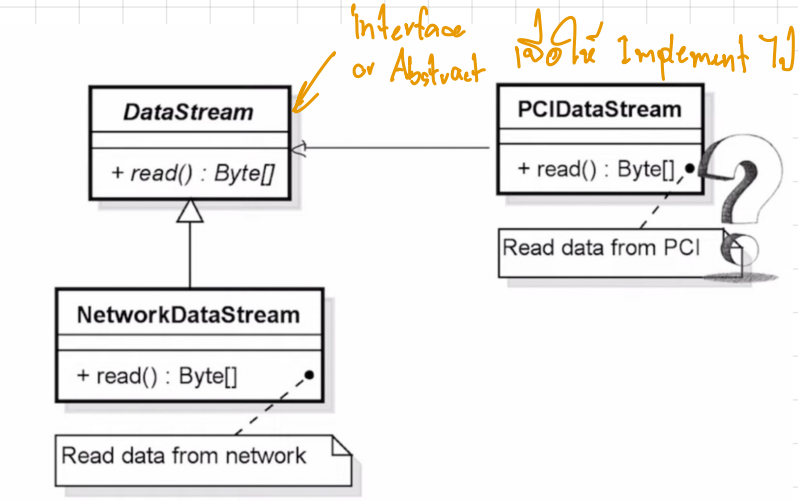
ไม่ว่าจะเป็น class หรือ Module ควรจะรู้สึกว่าอยากเพิ่มได้ แต่ไม่รู้สึกว่าต้องแก้ไข (เพื่อ function เก่าไม่ให้เกิดการแก้ไข)

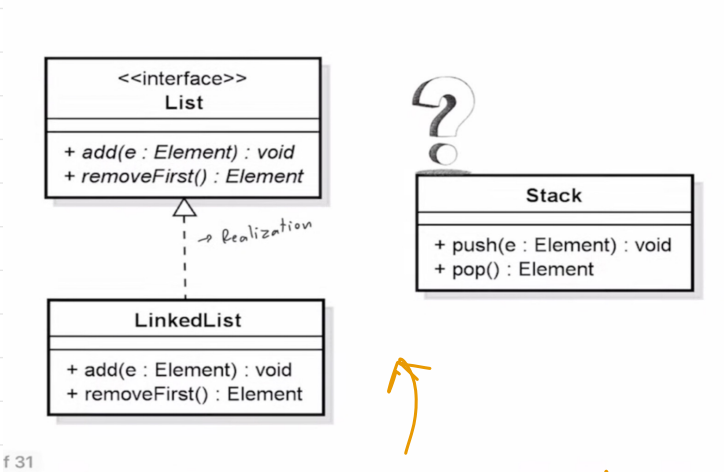
- Can't update the code but you can add new code
- If need to add functionality
  - don't modify the class
  - create a new class wrapped the calling class.
- Through inheritance
- Through composition : composition, Aggregation, Assodation.

DataStream

+ readNetwork()
+ readPCI()
+ read...

← ไม่ดีเพราะถ้าอยากเพิ่มอะไร ก็จำเป็นต้องแก้ไข





f 31

If OCP by inheritance → use realization but will violate the inheritance relationship because Stack is not kind of List. (different methods) ↘ Violate LSP

* Better way → OCP by composition

LSP - Liskov Substitution Principle

Derived classes must be substitutable for their base classes. (subclass ต้องใช้งานแทน super class ได้)

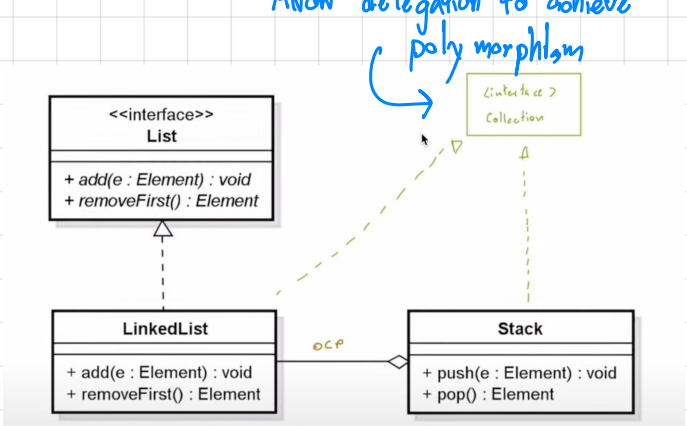
- An overridden method of a subclass must accept the same input parameter values as the method of the super class

```java
// LSP
class Rectangle {
    protected int width, height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getArea() {
        return this.width * this.height;
    }
}

class Square extends Rectangle {
    public Square(int width) {
        super(width, width);
    }

    public int getArea() {
        return this.width * this.height;
    }
}
```

```java
public static void main(String[] args) {

    // LSP
    Rectangle rec = new Rectangle( width: 10,  height: 2);
    printRectangleArea(rec);

    Square squ = new Square( width: 5);
    printRectangleArea(squ);

    // DIP
      Circle circle = new Circle(10);
      Triangle triangle = new Triangle(5, 10);
      printShapeArea(circle);
      printShapeArea(triangle);
}

public static void printRectangleArea(Rectangle rectangle) {
    System.out.println("Rectangle area is " + rectangle.getArea());
}
```

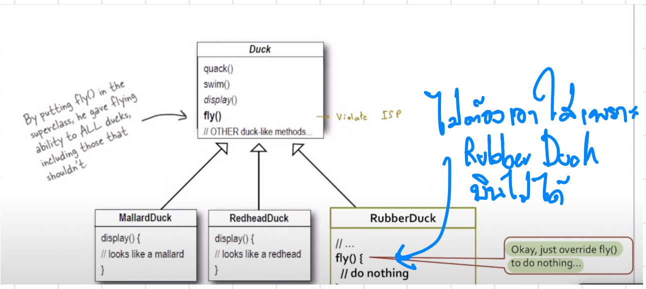
สามารถใช้ได้ เพราะ = save extend มาจาก Rectangle

Allow delegation to achieve poly morphism



ISP - Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use.
(อะไรที่ไม่มีเหตุผลจะใช้ก็ไม่ต้องมายัดใน Interface)



DIP - Dependency Inversion Principle

High-level modules should not depend upon low-level module.
Both should depend upon abstractions.
(Each module should program to an interface.)
คนที่จะมาใช้ class ในระดับ low level ทำควรจะเรียกใช้ Interface or abstract

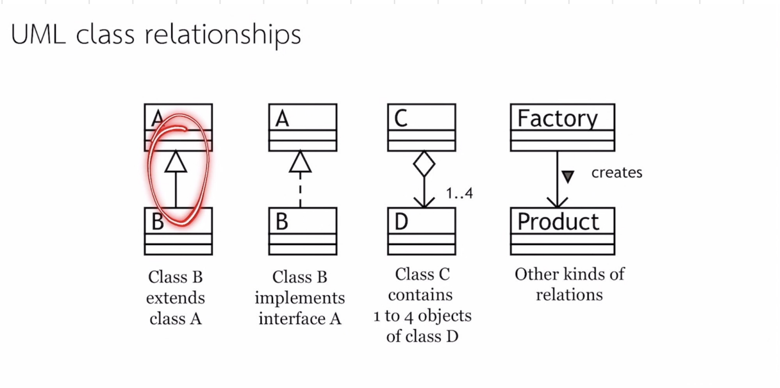
Example ↙ Include calAreamethod

interface
Shape ← Circle
↖ Triangle

(must implement when subclass was extended)

เวลาเรียกใช้ใน Low level ไม่ใช้หลายรี ✓

```java
    // DIP
    Circle circle = new Circle( radius: 10);
    Triangle triangle = new Triangle( base: 5,  height: 10);
    printShapeArea(circle);
    printShapeArea(triangle);
}

  public static void printRectangleArea(Rectangle rectangle) {
     System.out.println("Rectangle area is " + rectangle.getArea());
  }

public static void printShapeArea(Shape shape) {
    System.out.println("Area of the shape is " + shape.calculateArea());
}
```



อันที่วงคือ abstract

The Hollywood Principle (similar to DIP)

Don't call us, we will call you
คนที่ใหญ่กว่าจะเป็นคนสั่งให้คนข้างล่างทำงาน

GRASP - Information Expert

- One should assign a responsibility to the class which has the information necessary to fulfill that responsibility.

GRASP - Creator

- decide which class should be responsible for creating a new instance of a class.

GRASP - Pure Fabrication

- class that does not represent a concept in the problem domain (business domain)

GRASP - Controller (คล้ายกับ MVC)

- controller จะเป็นด่านหน้าเวลามี request เข้ามา ซึ่งจะตัดสินใจว่าจะส่งให้ใครไปทำอะไร
- Encapsulating system operations.
- Providing a layer between the UI and the domain model.

Implement Design Pattern

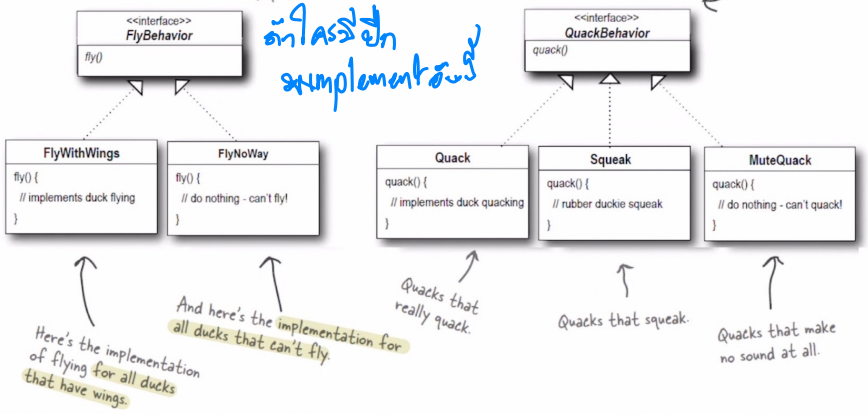
Design Pattern : a common solution to recurring (เป็นปัญหา) problem in design

Architectures: model software structure at the highest possible level. (มองเห็นภาพรวมของ system ทั้งหมด)
Patterns: small-scale or local architectures for architectural components.
Frameworks: partially completed software systems.
(template หรือ เอามา apply ได้เลย)

"Program to an interface, not an implementation."

FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly method.

Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.



Here's the implementation of flying for all ducks that have wings.

And here's the implementation for all ducks that can't fly.

Quacks that really quack.

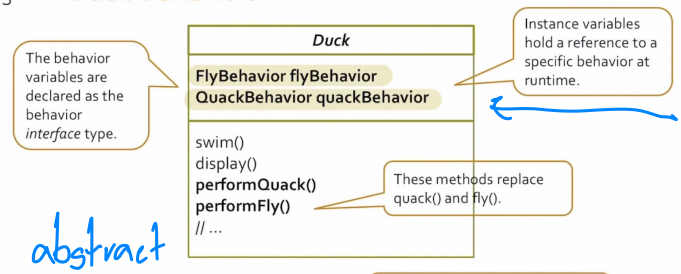
Quacks that squeak.

Quacks that make no sound at all.

11 of 23

11

Integrating the Duck Behaviors




```
public class Duck {

   QuackBehavior quackBehavior;

   public void performQuack() {
      quackBehavior.quack();
   }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object **delegates that behavior** to the object referenced by quackBehavior.

12

Integrating the Duck Behaviors [Cont'd]

```
public interface FlyBehavior {
   public void fly();
}

public class FlyWithWings implements FlyBehavior {
   public void fly() {
      System.out.println("I'm flying!!");
   }
}

public class FlyNoWay implements FlyBehavior {
   public void fly() {
      System.out.println("I can't fly...");
   }
}
```

tegrating the Duck Behaviors [Cont'd]

Instance variables inherited from Duck.

A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

```
public class MallardDuck extends Duck {

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings();
   }

   public void display() {
      System.out.println("I'm a real Mallard duck!");
   }

}
```

Get the concrete behavior

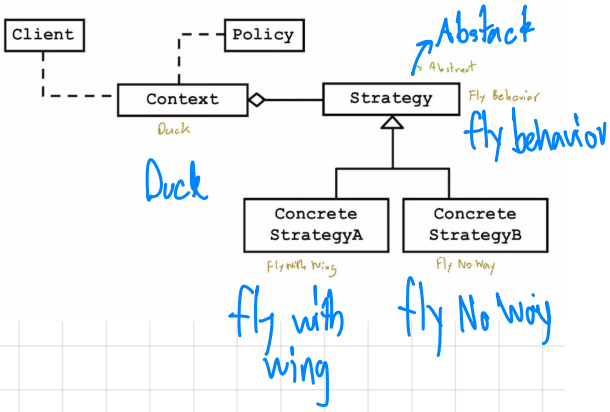
Note that we have just programed to an implementation!
Later on, we will have more patterns to help us fix it.

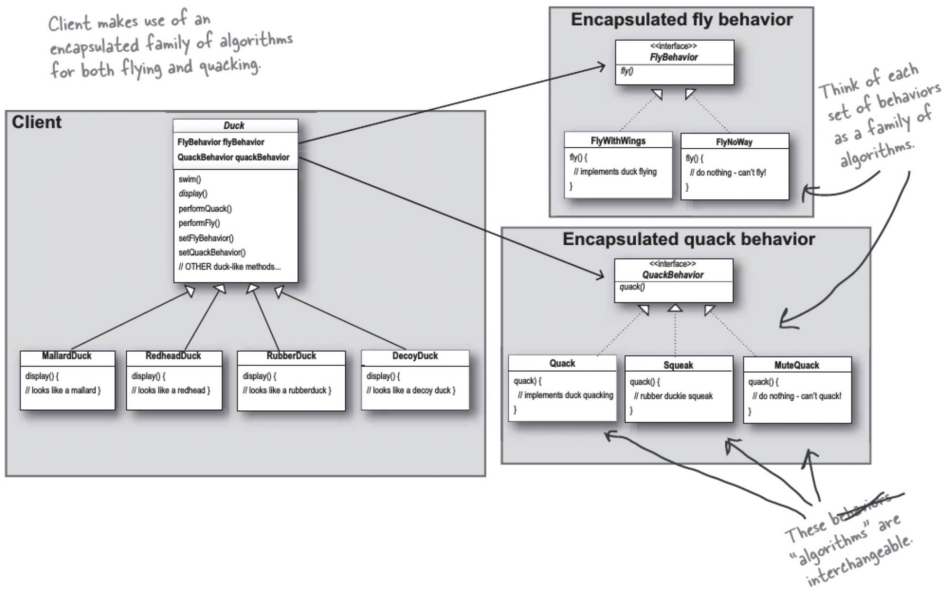
```
public class MiniDuckSimulator {

   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }

}
```

```
Quack
I'm flying!!
```

Strategy pattern : defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

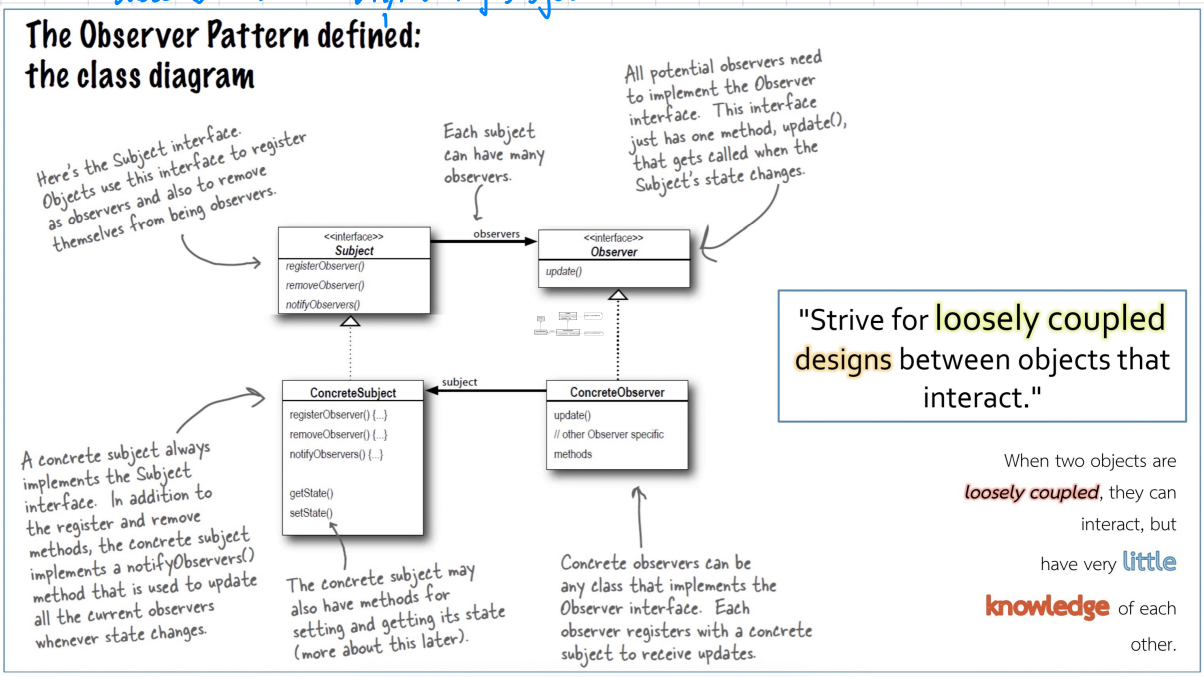
(เราสร้างรูปแบบการทำงานหรือ algor. เยอะๆ แล้วสามารถเปลี่ยนรูปแบบสลับกันได้)

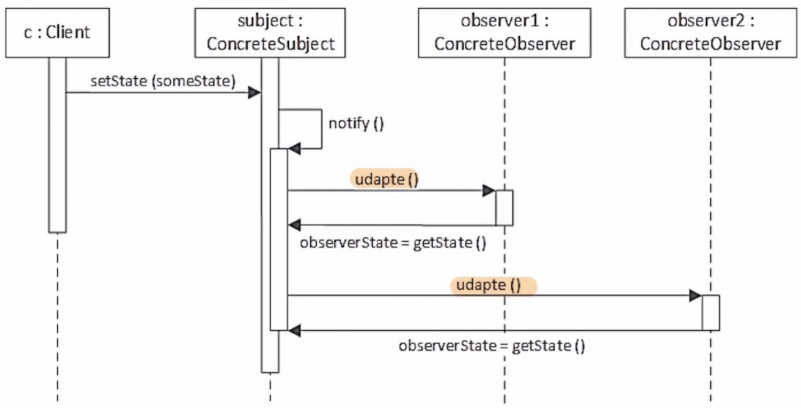




Observer

- สร้างการแจ้งเตือนให้กับ Object ต่างๆ
  - Subject ส่งข้อมูล
  - Publisher แจ้งเตือนเหตุการณ์
  - Subscriber รับข้อมูลที่ส่งมาจาก subject



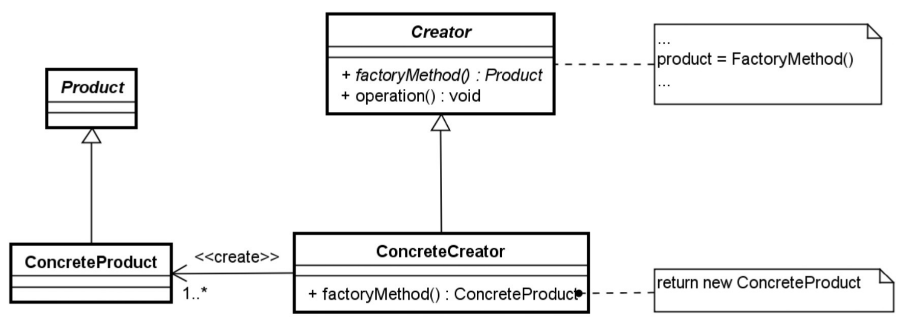


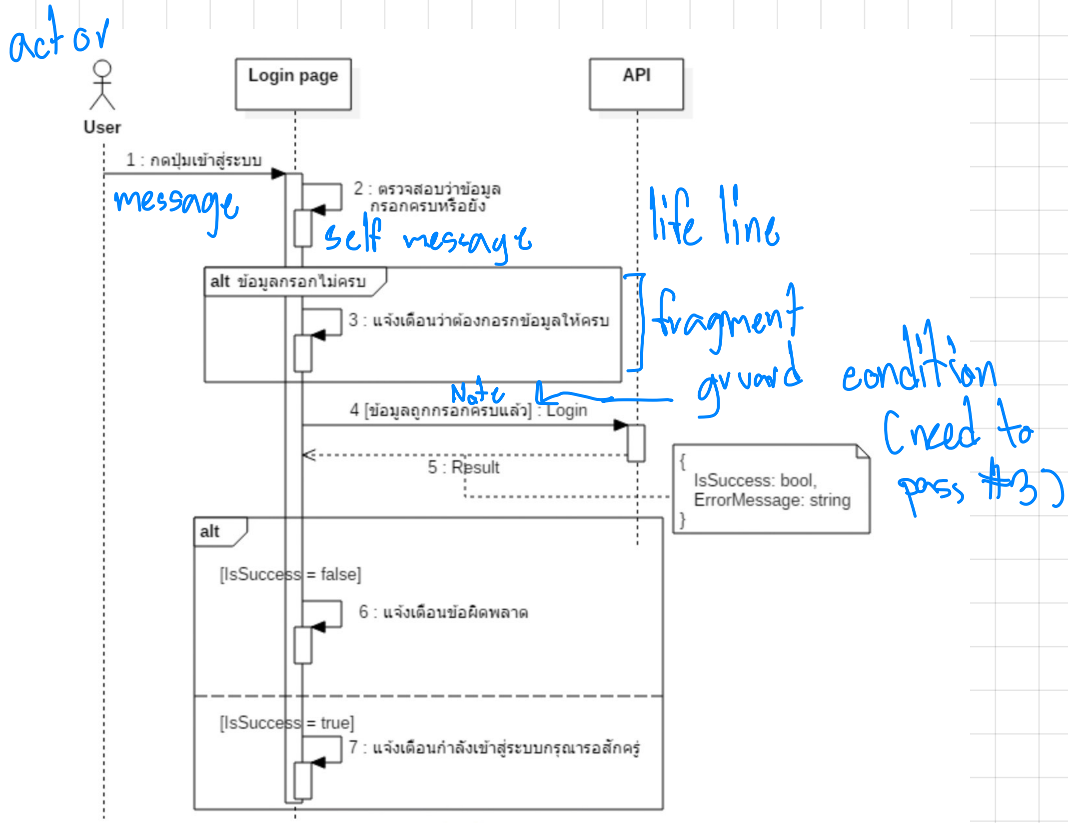
4 pillars of OOP

- Abstraction: เราไม่ต้องรู้ว่าข้างในทำงานยังไง แค่เรียกใช้
- Encapsulation: ป้องกัน Attribute ภายใน Object เพื่อไม่ให้ Access directly
- Inheritance: สามารถสืบทอดได้
- Polymorphism: ดัดแปลงได้ถ้า inherit มาได้ (compile time overloading and runtime overriding)

factory

- สร้าง Object โดยไม่ต้องกำหนดเจาะจงตัว class ที่จะนำมาสร้างล่วงหน้า คืออะไรโดยให้ subclass เป็นคนไปจัดการเอง





ถ้าไม่ชัดให้กดที่รูปเพื่อดูแบบเต็มจอได้นะ