物件導向設計原則 (Design Principles)

李信杰 副教授
國立成功大學計算機與網路中心/資訊工程學系

(部分教材取自軟體工程聯盟)

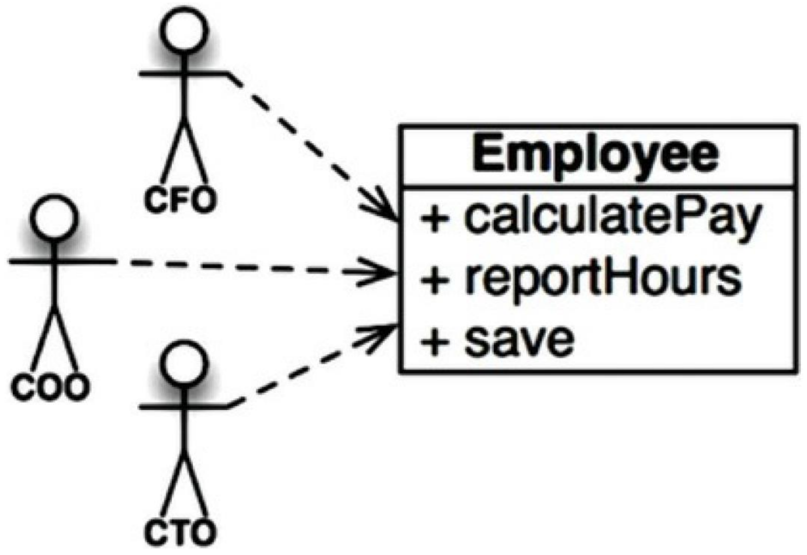
Design Principles

- SOLID
  - **S**RP: The Single Responsibility Principle
  - **O**CP: The Open-Closed Principle
  - **L**SP: The Liskov Substitution Principle
  - **I**SP: The Interface Segregation Principle
  - **D**IP: The Dependency Inversion Principle
- Encapsulate what varies
- Favor composition over inheritance
- Least Knowledge Principle

Single Responsibility Principle

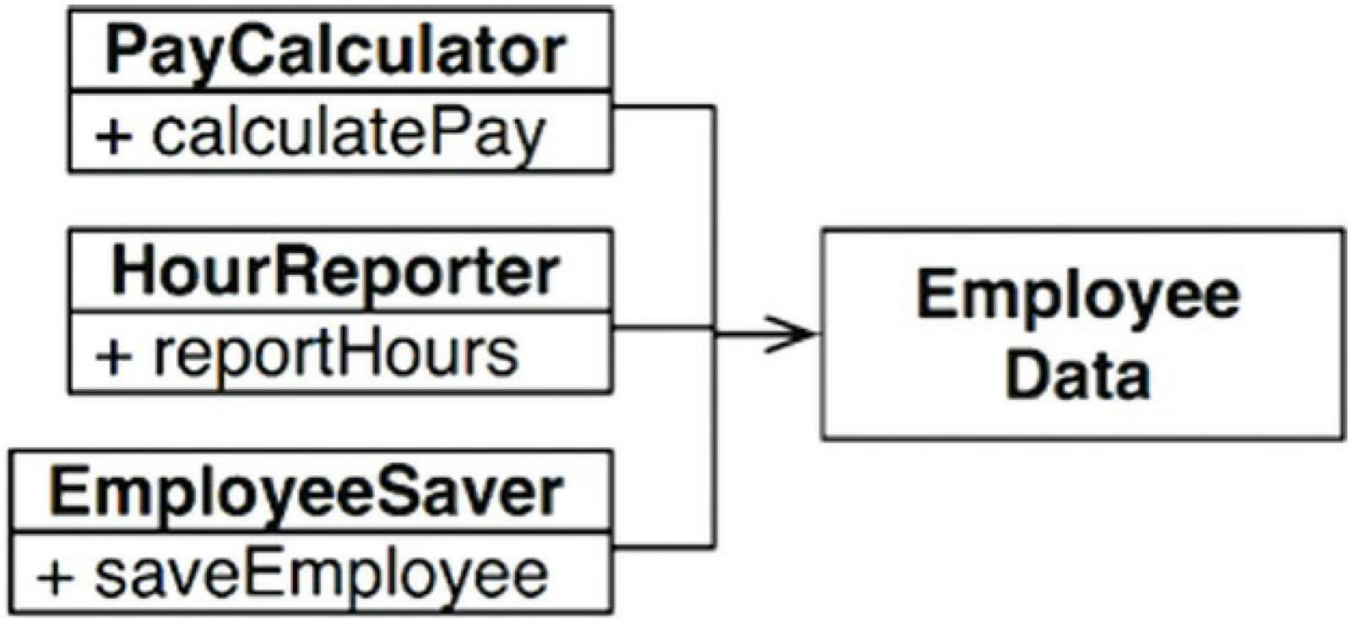
SRP: THE SINGLE RESPONSIBILITY PRINCIPLE$_1$

- Historically, the SRP has been described this way:
  - *A module should have one, and only one, reason to change*.
- We can rephrase the principle to say this:
  - *A module should be responsible to one, and only one, actor*.
- This class violates the SRP because those three methods are responsible to three very different actors.



SRP: THE SINGLE RESPONSIBILITY PRINCIPLE$_2$

- Perhaps the most obvious way to solve the problem is to separate the data from the functions.
- Each class holds only the source code necessary for its particular function.



範例：此Class因二種不同行為而需變更

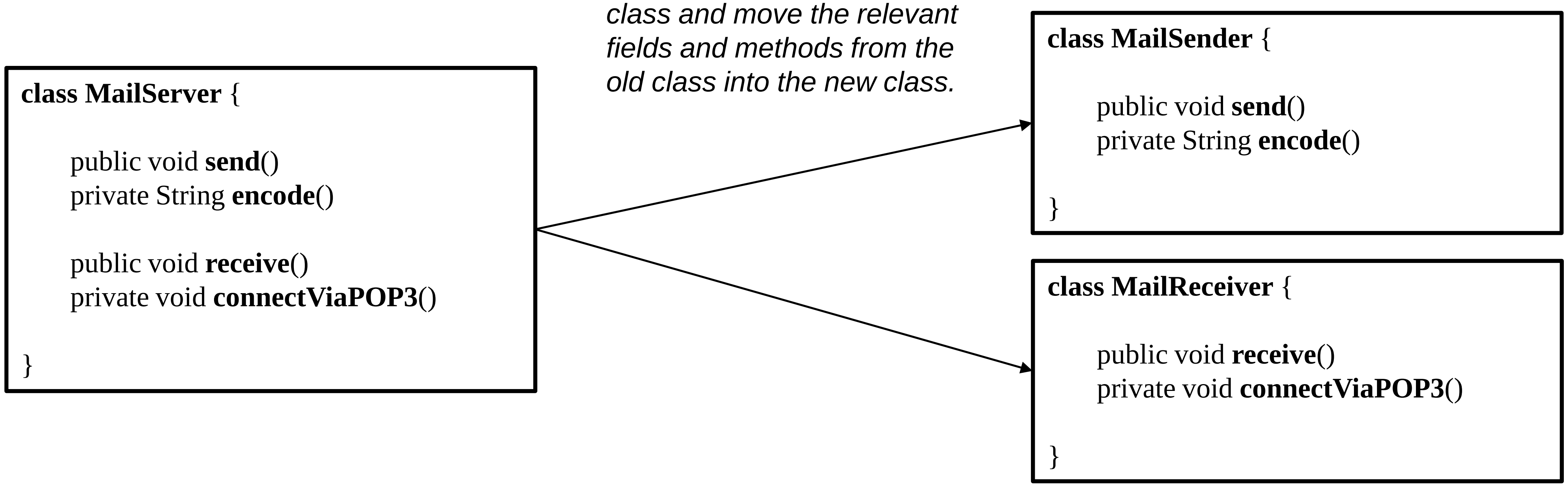
```
class MailServer {

        public void send(String from, String to, String content) {
                //…
                String encodedContent = encode(content, “UTF-8”);
        }
        private String encode(String content, String charset){//encode content; }

        public void receive(String account){
                connectViaPOP3();
                //…
        }
        private void connectViaPOP3(){// connect to a server via POP3 protocol; }

}
```

可預想這兩個Methods會因為「寄信」行為改變(如需加入Encryption)而需變更。

可預想這兩個Methods會因為「收信」行為改變(如需加入IMAP協定)而需變更。

Refactoring by *Extract Class*

***Extract Class**: Create a new class and move the relevant fields and methods from the old class into the new class.*



```
class MailServer {

     public void send()
     private String encode()

     public void receive()
     private void connectViaPOP3()

}
```

```
class MailSender {

     public void send()
     private String encode()

}
```

```
class MailReceiver {

     public void receive()
     private void connectViaPOP3()

}
```

Refactoring後遵循Single Responsibility Principle

- Single Responsibility Principle: 每個Class必須專注於提供整個系統中單一部分的功能，使得Class更Robust。每個Class必須僅因一個理由而有所修改。

- 在實務上，判定是否滿足此原則是主觀的。如果你瞇著眼仔細檢視程式碼，會發現一個Class常常存在因為多個理由而需修改，因此建議讓檢視是否同一個Class中的Methods相互依賴或共用屬性，若是，則內聚力較高。

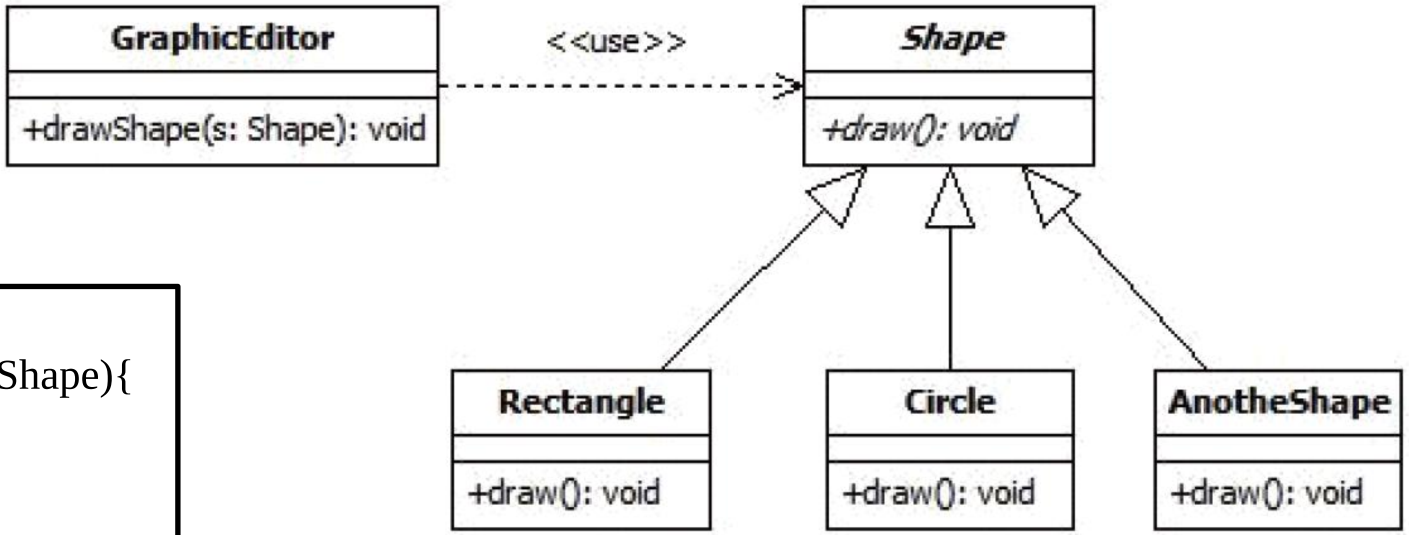
```
class MailSender {

    public void send()
    private String encode()

}
```

```
class MailReceiver {

    public void receive()
    private void connectViaPOP3()

}
```

Open-Close Principle
（開放關閉原則）

Open-Close Principle (開放關閉原則)

- Open for extension, but closed for modification
  - 一個模組必須有彈性的開放往後的擴充，並且避免因為修改而影響到使用此模組的程式碼。

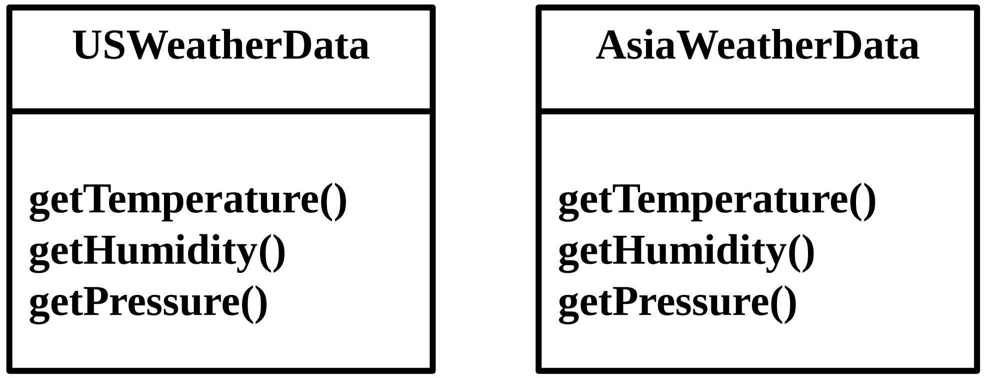


```
class GraphicEditor {
   void drawShape(s: Shape){
       s.draw();
   }
}
```

```
abstract class Shape {
   abstract public void draw();
}

class Rectangle extends Shape {
   public void draw() {…}
}

class Circle extends Shape {
   public void draw() {…}
}

class AnotherShape extends Shape
{
   public void draw() {…}
}
```

範例一：需求描述$_1$

- 天氣播報系統
  - 天氣資料包含了特定區域(如美國或亞洲)的溫度、濕度和氣壓。

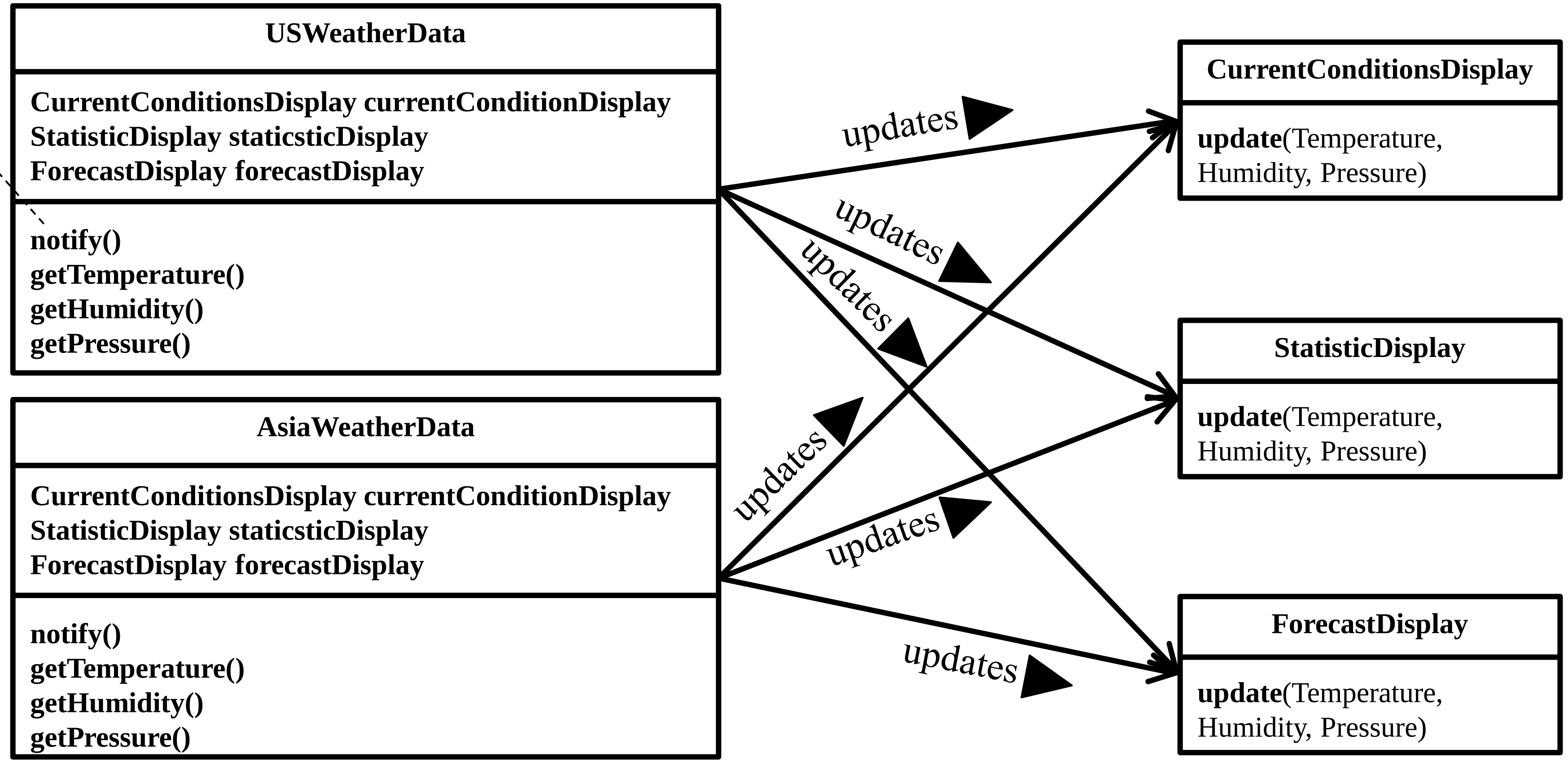


需求描述$_2$

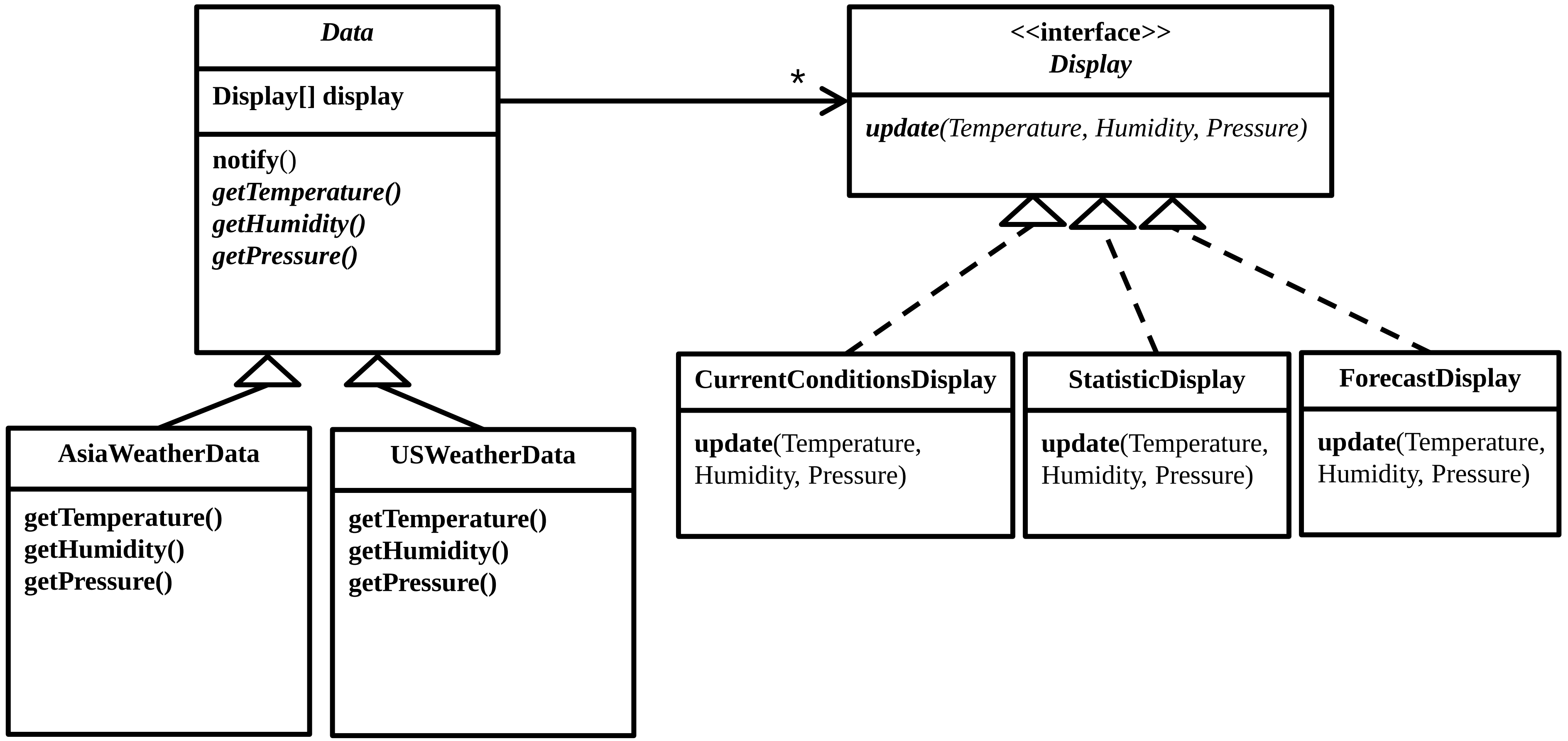
- 此系統提供三種氣象播報顯示: current conditions, weather statistics and a simple forecast，且當天氣資料更新時，所有顯示將會即時更新。

```
{  currentConditionsDisplay.update(temp,humidity, pressure);
   statisticsDisplay.update(temp, humidity, pressure);
   forecastDisplay.update(temp, humidity, pressure);      }
```

問題: 當有新的Display需新增時，則需修改此程式碼。

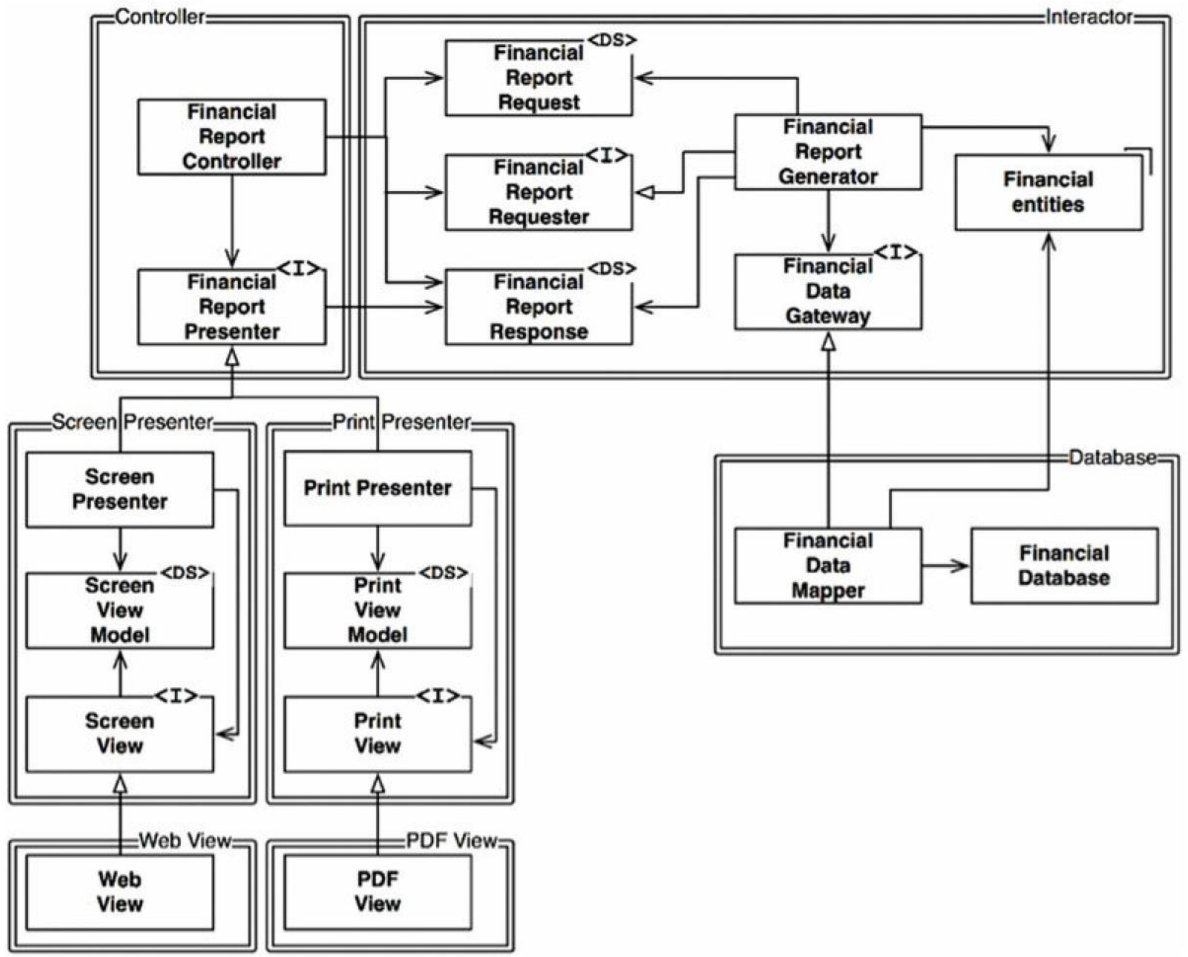

USWeatherData
CurrentConditionsDisplay currentConditionDisplay
StatisticDisplay staticsticDisplay
ForecastDisplay forecastDisplay
notify()
getTemperature()
getHumidity()
getPressure()
AsiaWeatherData
CurrentConditionsDisplay currentConditionDisplay
StatisticDisplay staticsticDisplay
ForecastDisplay forecastDisplay
notify()
getTemperature()
getHumidity()
getPressure()
updates
updates
updates
updates
updates
updates
CurrentConditionsDisplay
update(Temperature, Humidity, Pressure)
StatisticDisplay
update(Temperature, Humidity, Pressure)
ForecastDisplay
update(Temperature, Humidity, Pressure)

應用Open-Closed Principle



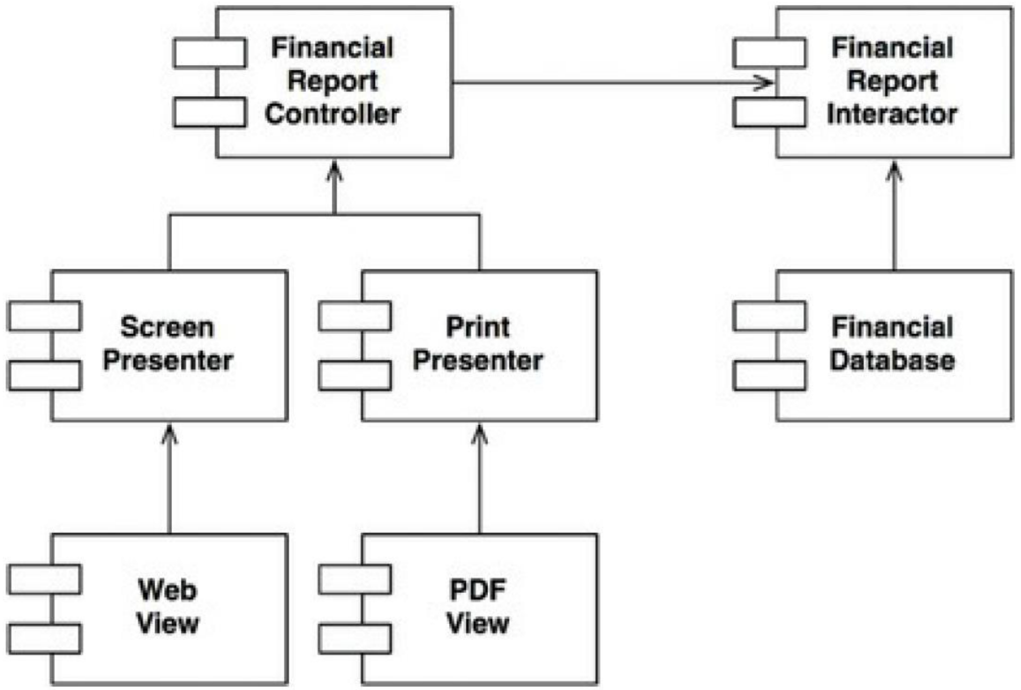
OCP: THE OPEN-CLOSED PRINCIPLE

- Partitioning the processes into classes, and separating those classes into components



OCP: THE OPEN-CLOSED PRINCIPLE

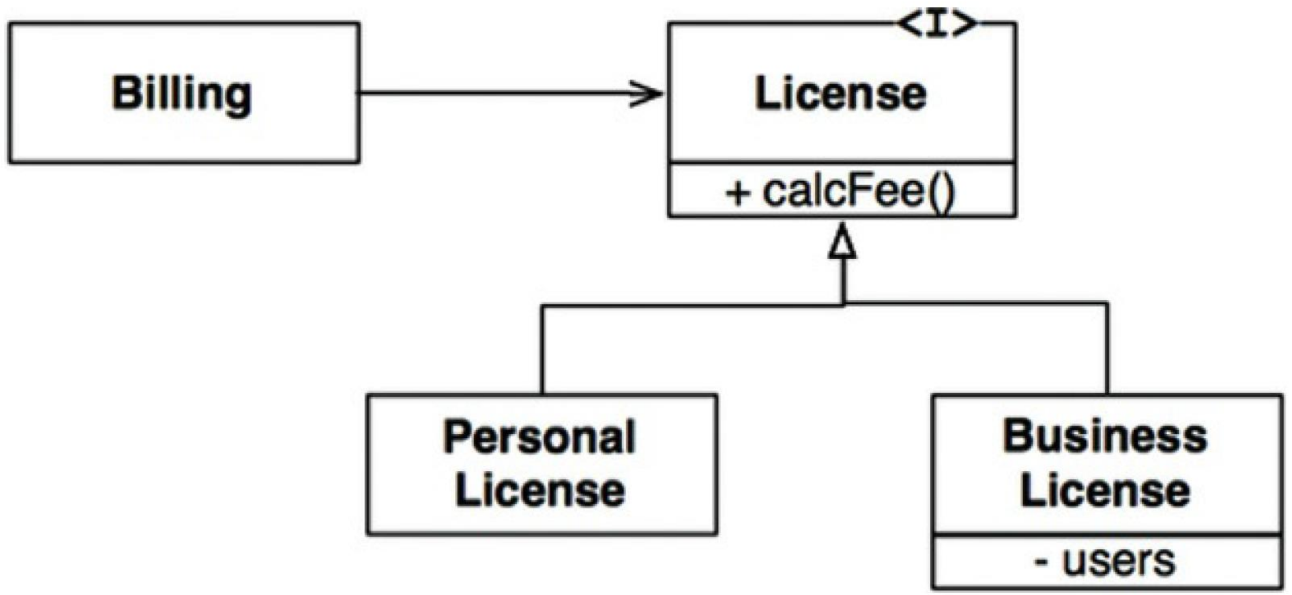
- All component relationships are unidirectional.
  - *If component A should be protected from changes in component B, then component B should depend on component A.*
- This is how the OCP works at the architectural level.
  - Higher-level components in that hierarchy are protected from the changes made to lower-level components.



Liskov Substitution Principle

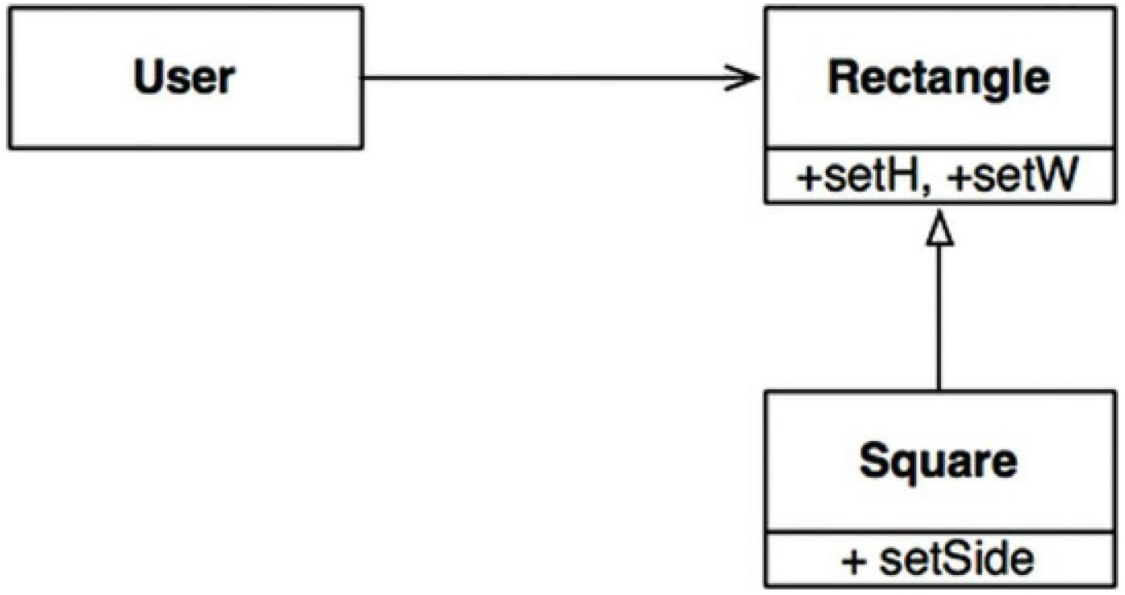
LSP: THE LISKOV SUBSTITUTION PRINCIPLE

- In 1988, Barbara Liskov wrote the following as a way of defining subtypes.
  - *If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T,*
  - *the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*



LSP: THE LISKOV SUBSTITUTION PRINCIPLE

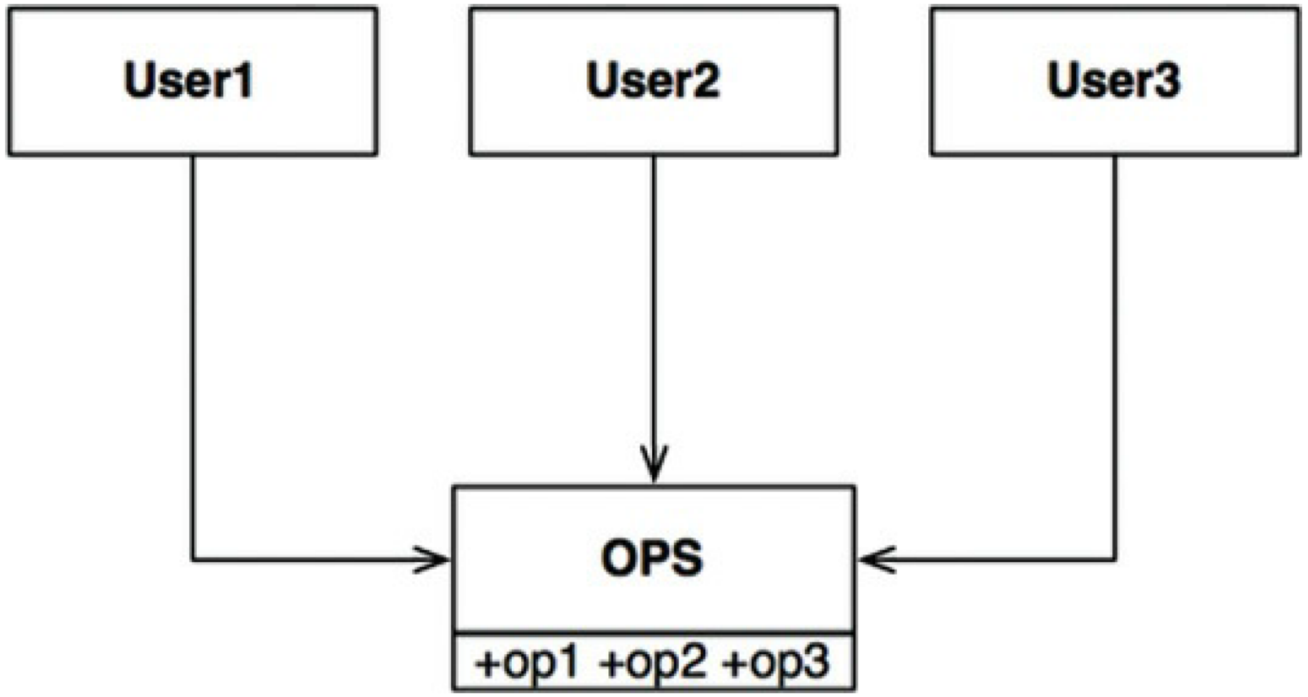
- The canonical example of a violation of the LSP is the square/rectangle problem
  - The only way to defend against this kind of LSP violation is to add mechanisms to the `User` (such as an `if` statement) that detects whether the `Rectangle` is, in fact, a `Square`



Interface Segregation Principle

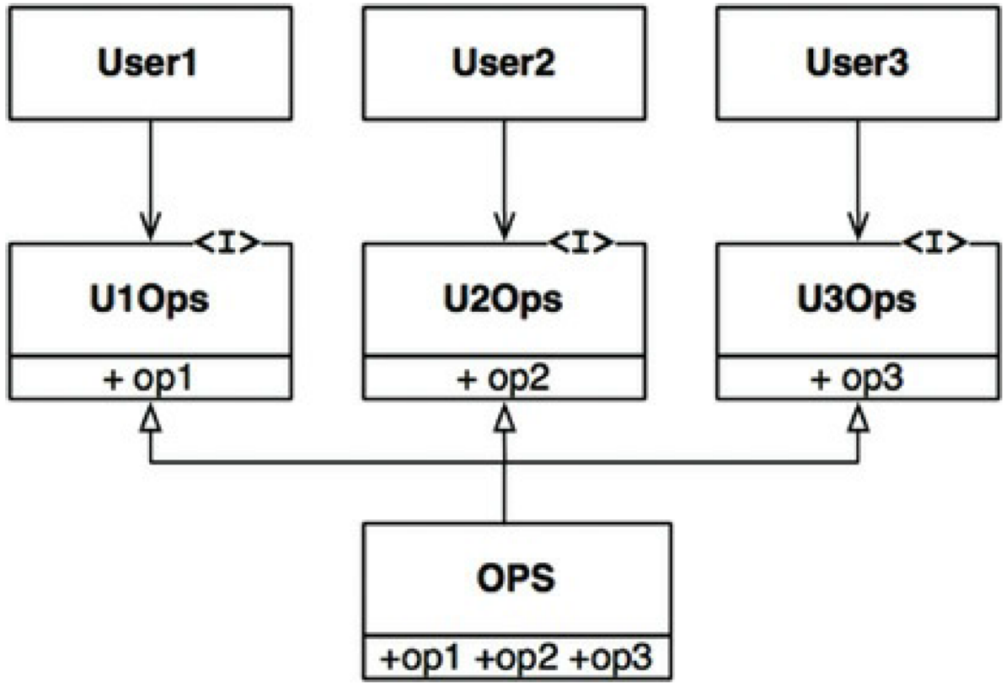
ISP: THE INTERFACE SEGREGATION PRINCIPLE[1]

- Let's assume that `User1` uses only `op1`, `User2` uses only `op2`, and `User3` uses only `op3`.
  - A change to the source code of `op2` in `OPS` will force `User1` to be recompiled and redeployed, even though nothing that it cared about has actually changed.



ISP: THE INTERFACE SEGREGATION PRINCIPLE$_2$

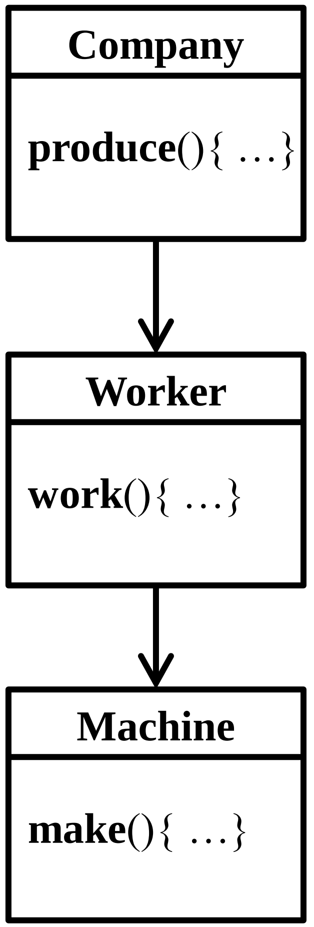
- This problem can be resolved by segregating the operations into interfaces
- Thus a change to `OPS` that `User1` does not care about will not cause `User1` to be recompiled and redeployed.



Dependency Inversion Principle
(依賴反向原則)

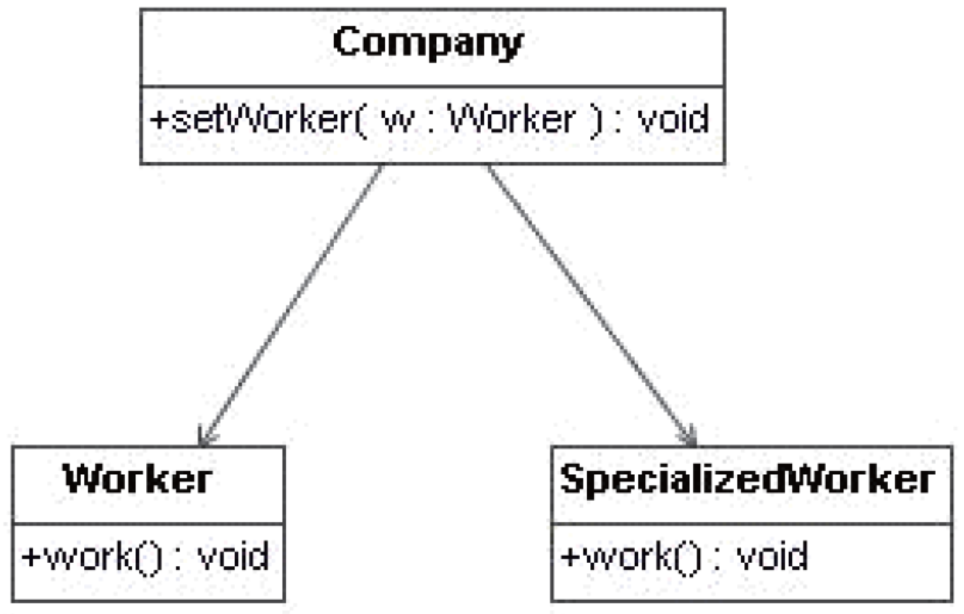
Dependency Inversion Principle (依賴反向原則)

- 軟體設計的程序開始於簡單高層次的概念（Conceptual），慢慢的增加細節和特性，使得越來越複雜
  - 從高層次的模組開始，再設計低層詳細的模組。
- Dependency Inversion Principle (依賴反向原則)
  - 高階模組不應該依賴低階模組，兩者必須依賴抽象（即抽象層）。



違反Dependency Inversion Principle

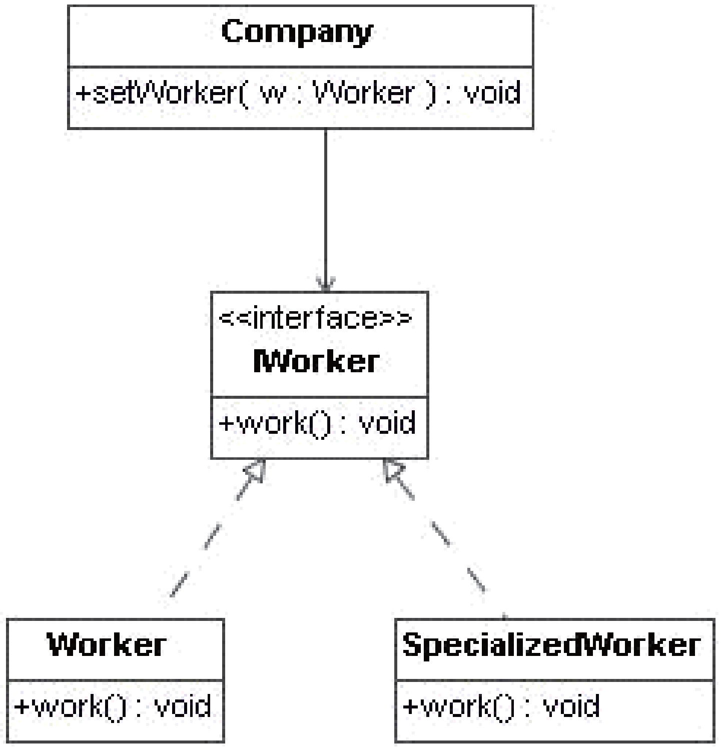
- 如果公司因業務需要必須增聘具有專長的員工（也許稱之為SpecializedWorker 模組），則必須修改複雜的公司模組（即高層模組）的程式碼，這種修改將影響公司的模組結構，這樣就違反DIP 原則，



```
class Worker {
            public void work () {…..}
}
class SpecializedWorker {
            public void work () { …..}
}
class Company {
            Worker worker;
           public void setWorker (Worker w) {worker = w;}
            public void produce() {worker.work ();}
}
```

符合Dependency Inversion Principle

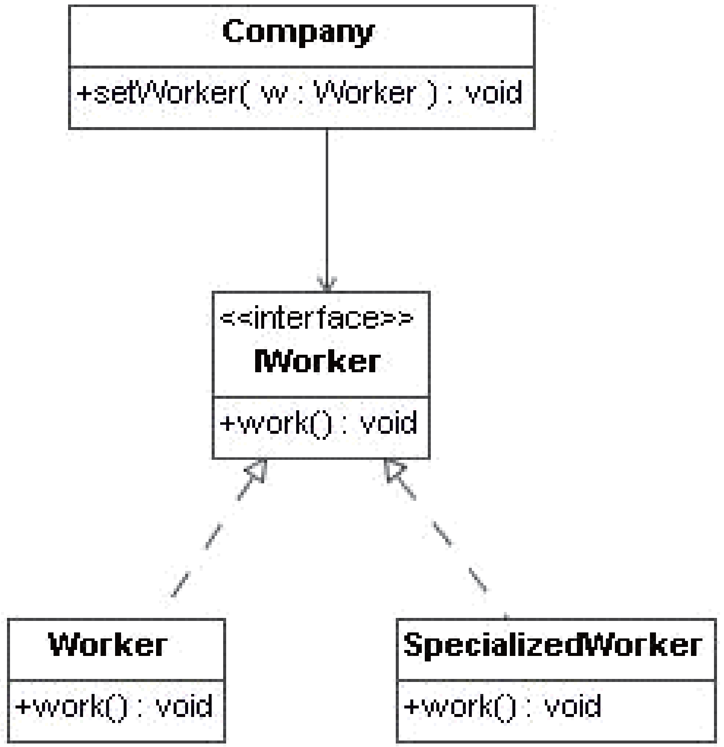
- 可以在公司模組與員工模組之間介入一種「抽象層」（Abstract Layer），公司模組與員工模組皆依存於這種抽象層，如此不論如何增聘員工都不致於影響公司模組本身，這種抽象層一般都是一種介面，亦即IWorker 類別



```
class Company {
            IWorker worker;
publich void setWorker (IWorker w) {
            worker = w;
}
public void produce () {
            worker.work (); }
}
```

Dependency Inversion Principle優點

- 高層的抽象層含宏觀和重要商務邏輯，低層的實作層含實作相關演算法與次要商業邏輯，DIP 可讓實作改變時，商業邏輯無須變動。



DIP: THE DEPENDENCY INVERSION PRINCIPLE[1]

- Clearly, treating this idea as a rule is unrealistic. For example, the `String` class in Java is concrete.
- By comparison, the `String` class is very stable. Changes to that class are very rare and tightly controlled.
- We tend to ignore the stable background of operating system and platform facilities when it comes to DIP. We tolerate those concrete dependencies because we know we can rely on them not to change.

DIP: THE DEPENDENCY INVERSION PRINCIPLE$_2$ - STABLE ABSTRACTIONS

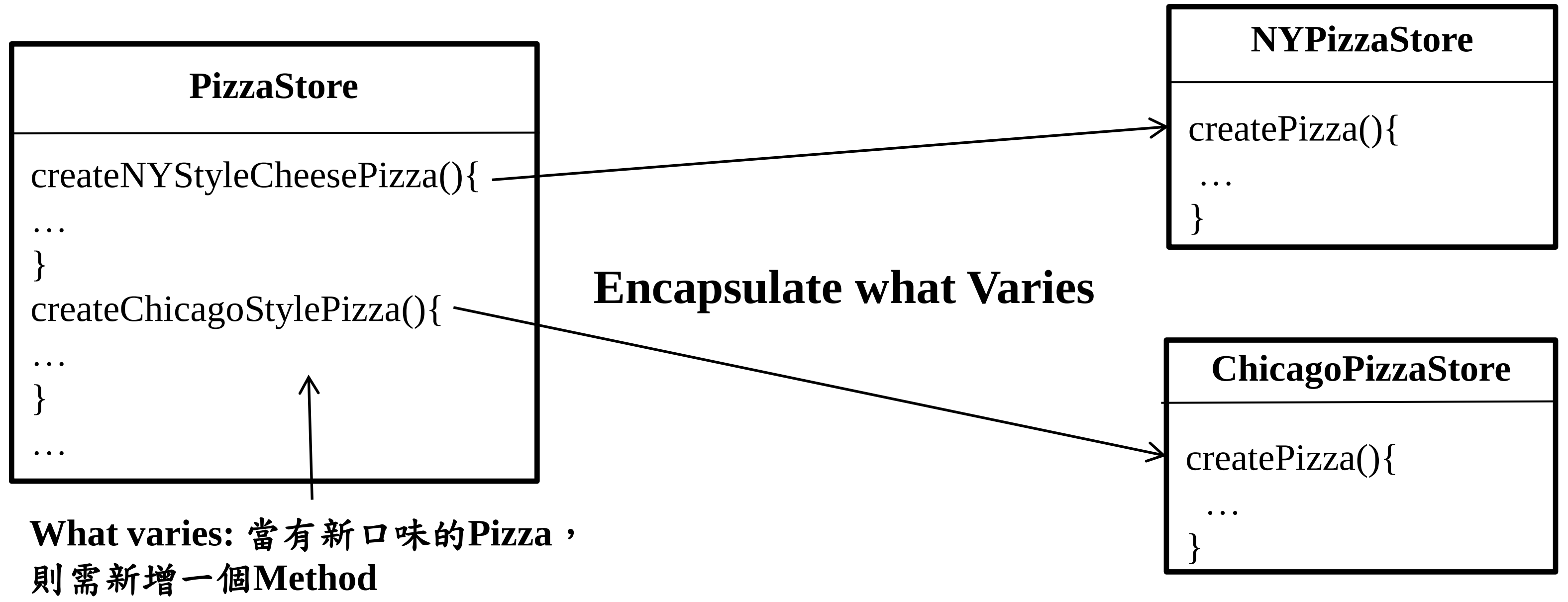
- Good software designers and architects work hard to reduce the volatility of interfaces. They try to find ways to add functionality to implementations without making changes to the interfaces.

- **Don’t refer to volatile concrete classes.**
- **Don’t derive from volatile concrete classes**
- **Don’t override concrete functions**
- **Never mention the name of anything concrete and volatile**

Encapsulate what varies (封裝改變)

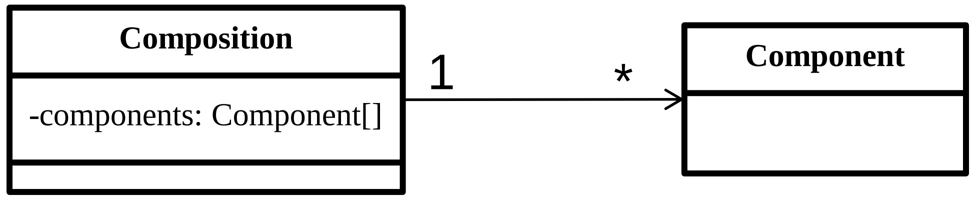
Encapsulate what varies (封裝改變)

- Encapsulate what varies (封裝改變)
  - 將易改變之程式碼部份封裝起來，以後若需修改或擴充這些部份時，能避免不影響到其他不易改變的部份。
  - 換言之，將潛在可能改變的部份隱藏在一個介面(Interface)之後，並成為一個實作(Implementation)，爾後當此實作部份改變時，參考到此介面的其他程式碼部份將不需更改。



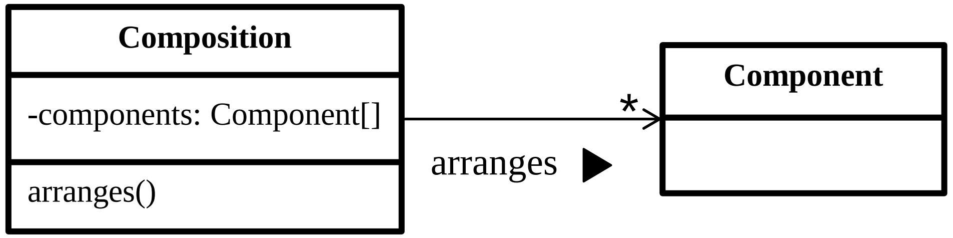
範例一(文件編輯器)：需求描述$_1$

- 一個Composition物件包含一群Component物件，一個Component物件代表著一份文件(Document)中的一段文字(Text)件或一個視覺化(Graphic)元件。



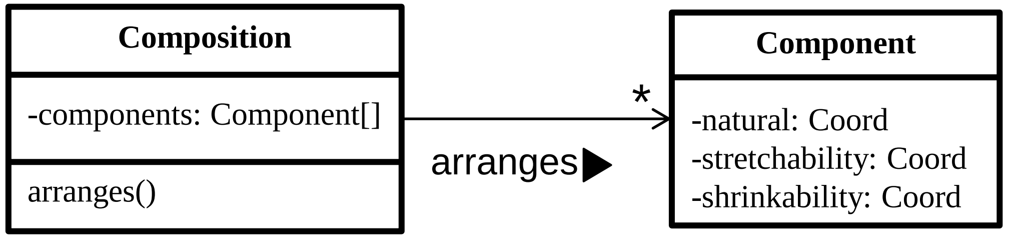
需求描述$_2$

- Composition物件利用一個linebreaking策略來排版此一群Component物件。



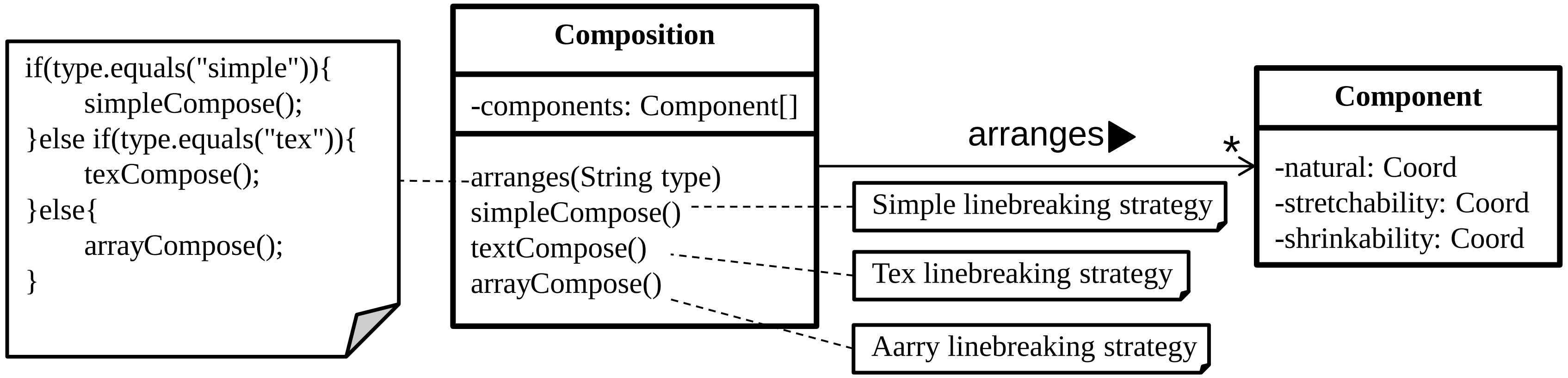
需求描述3

- 每個Component擁有三種屬性：Natural Size、Stretchability和Shrinkability。

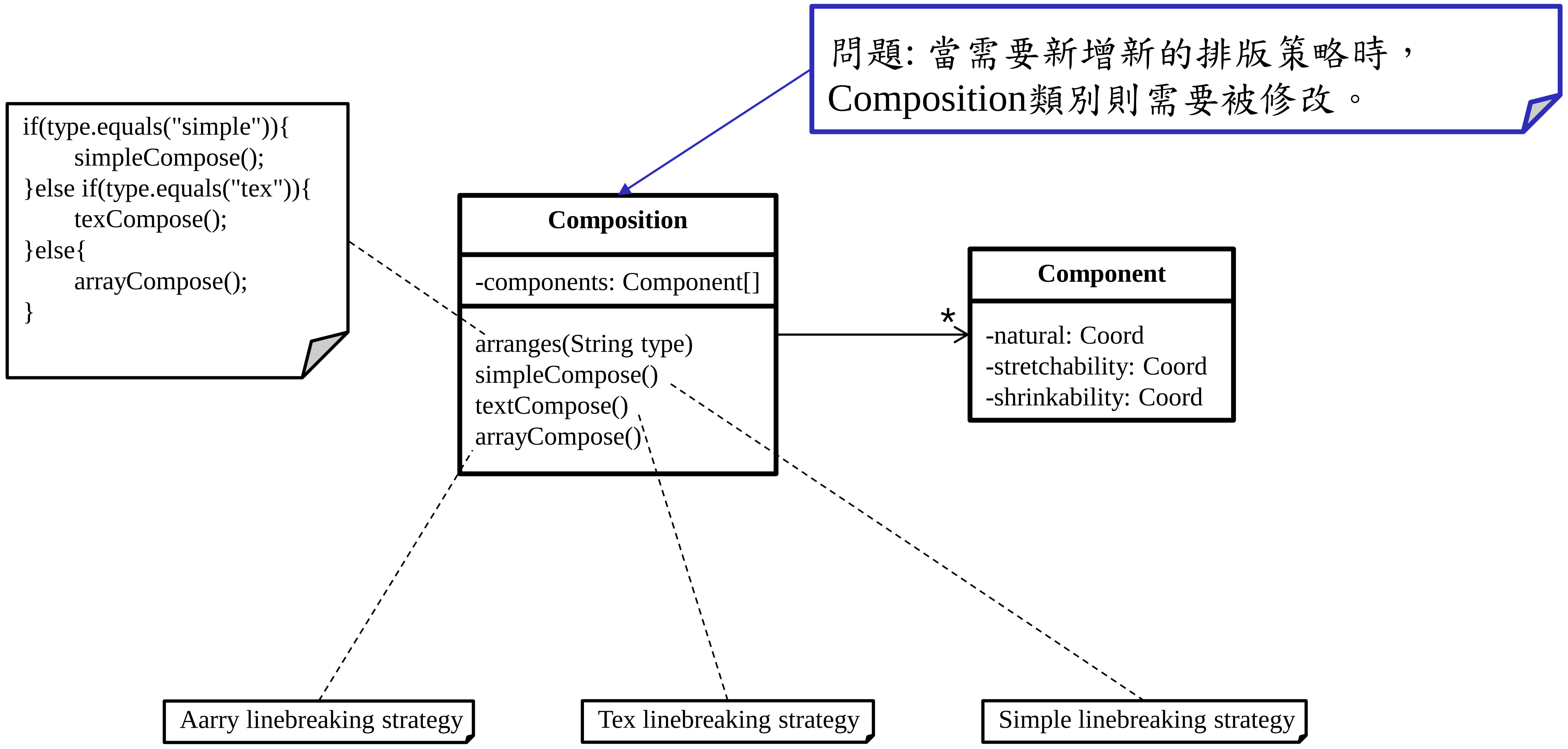


需求描述$_4$

- 當需要新增新的排版方式時(如以下三種不同策略)：
  - **Simple Composition:** 每一列皆插入一個斷行符號。
  - **Tex Composition:** 每一段落皆插入一個斷行符號。
  - **Array Composition:** 每一行包含固定數量的元件。

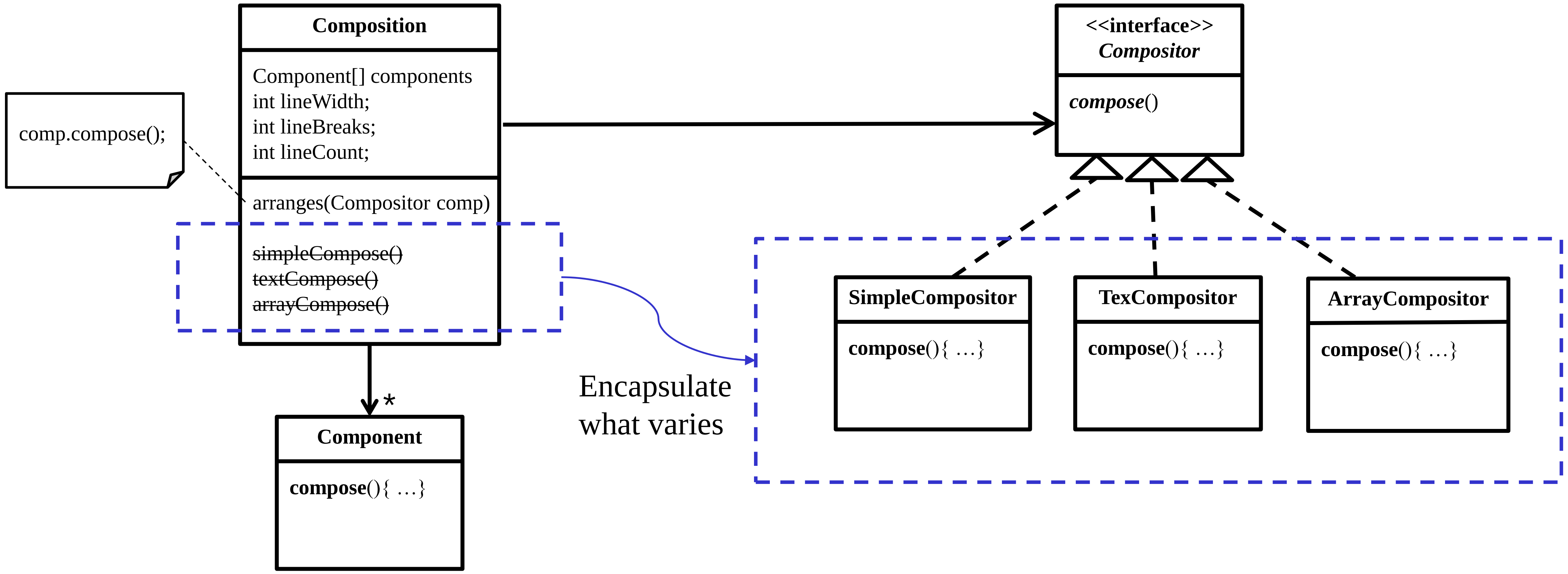

```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

初步設計之問題



```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

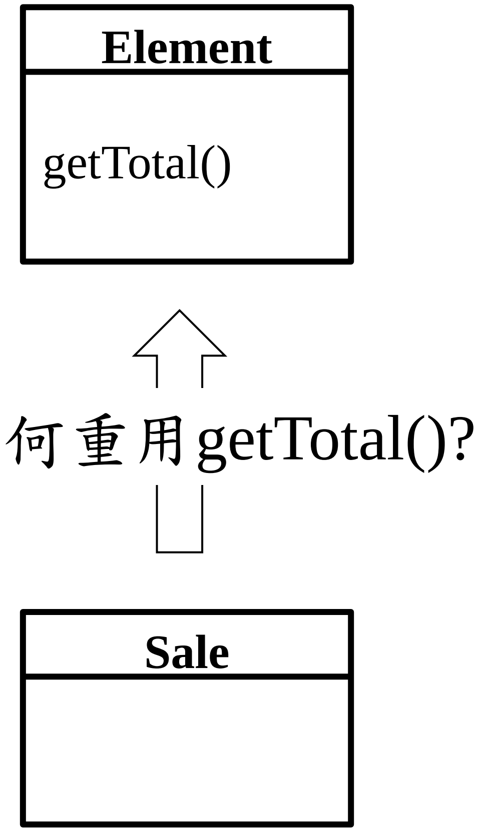
應用Encapsulate what varies設計原則



Favor composition over inheritance (善用合成取代繼承)

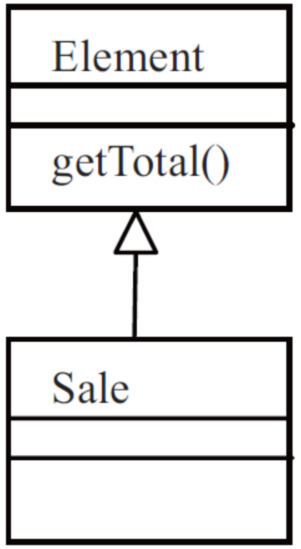
Favor composition over inheritance (善用合成取代繼承)

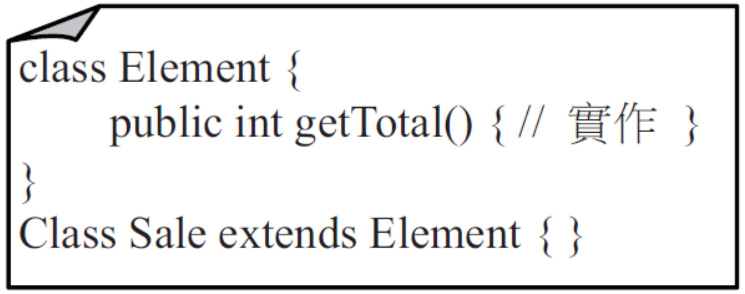
- 程式碼重用(Reuse)
  - 物件類別可藉由Composition來達到多型(Polymorphism)與程式碼重用(Code Reuse)之效果，而非一定得使用繼承。
- 不要一味的使用繼承，只是為了達到程式碼的重用。只有當兩者真的有 IS-A 的關係時才使用繼承。
- 有別於繼承，Composition可在Runtime時更有彈性地動態新增或移除功能



繼承(Inheritance)優缺點

- 優點
  - 透過簡便的擴充(Extend)繼承，就可以實做新的功能。
- 缺點
  - 父類別修改會影響到子類別
    - 因為子類別繼承父類別的屬性和方法，父類別的屬性和方法一旦修改，將可能會影響到所有繼承它的子類別。
  - 無法在執行時期改變所需物件
    - 從父類別實做繼承，無法在執行時期設定不同物件以呼叫不同的Method功能。

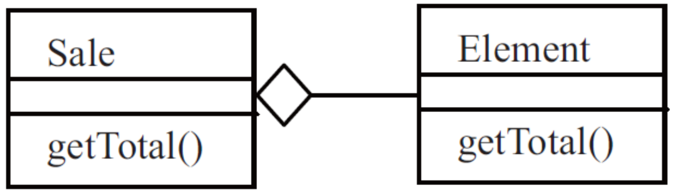


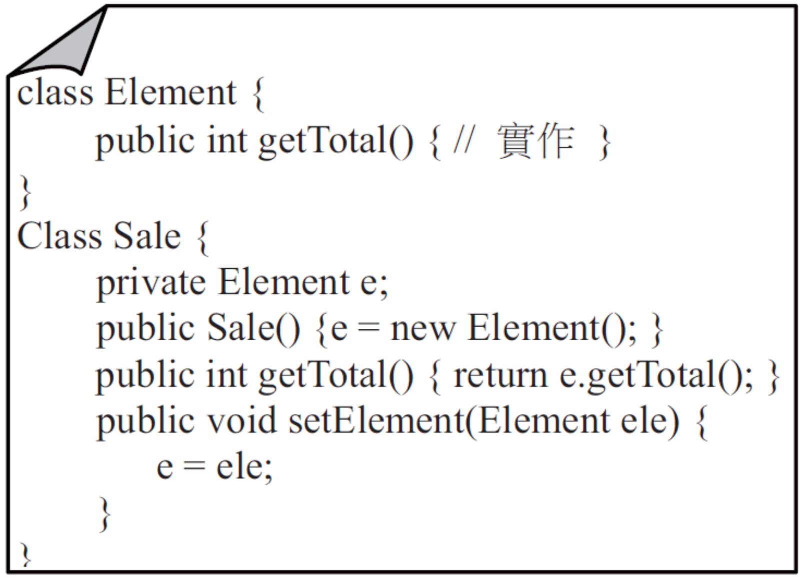


```
class Element {
        public int getTotal() { // 實作 }
}
Class Sale extends Element { }
```

合成(Composition)優缺點

- 優點
  - 封裝性良好
    - 物件透過介面（interfaces）存取被合成或被包含的物件
  - 執行時期動態組合新功能
    - 例如Sale 物件可以在執行時期透過setElement 動態設定不同的Element 物件。
- 缺點
  - 造出較多物件



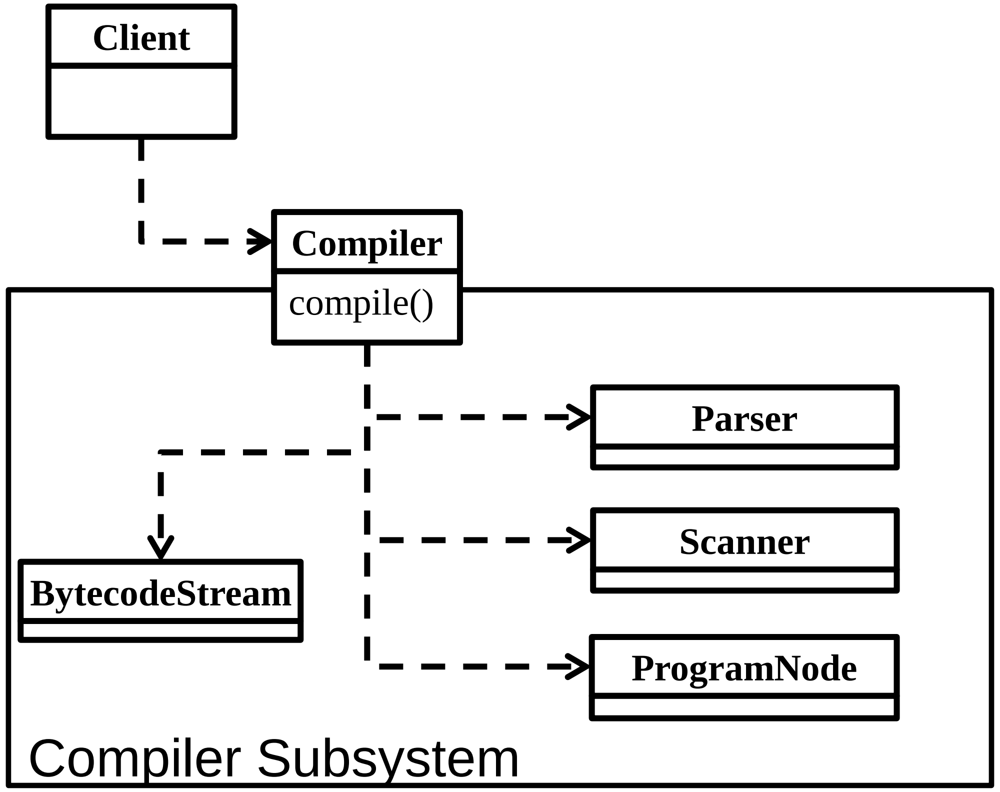


```
class Element {
        public int getTotal() { //  實作  }
}
Class Sale {
        private Element e;
        public Sale() {e = new Element(); }
        public int getTotal() { return e.getTotal(); }
        public void setElement(Element ele) {
              e = ele;
        }
}
```

Least Knowledge Principle
(最小知識原則)

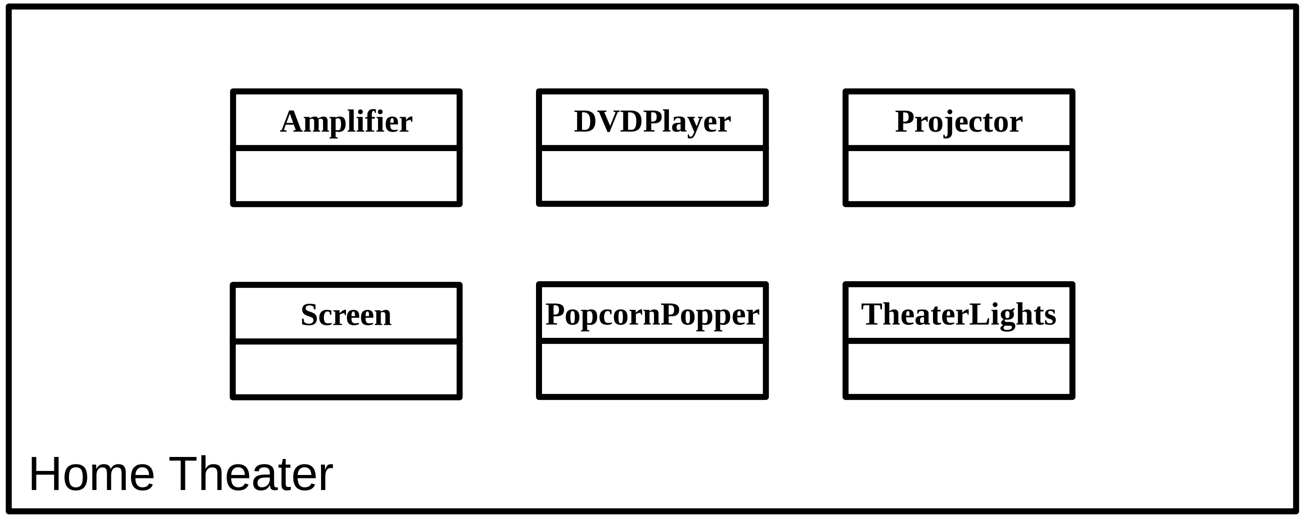
Least Knowledge Principle (最小知識原則)

- 設計系統時必須注意類別的數量，並且避免製造出太多類別之間的耦合關係。
  - 知道子系統中的元件越少越好



範例一：需求描述$_1$

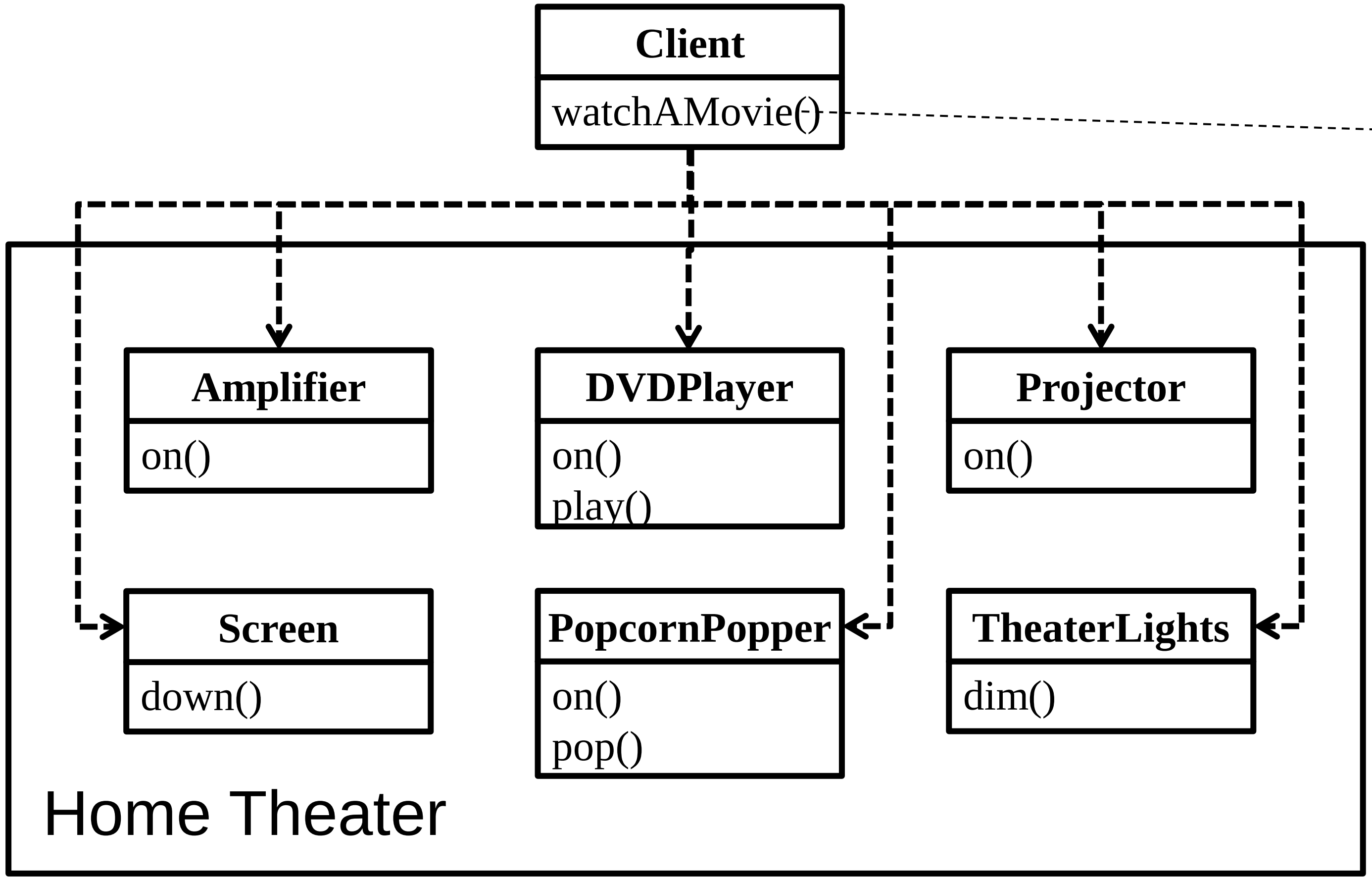
- 一個家庭劇院系統包含Amplifier、DVD Player、Projector、Screen、Popcorn Popper和Theater Lights.



需求描述$_2$

- 使用者看電影時包含下列步驟：
  1. 製作爆米花(Popcorn Popper)
  2. 調暗燈光(Lights)
  3. 將投影幕(Screen)放下
  4. 打開投影機(Projector)
  5. 打開音響(Amplifier)
  6. 打開DVD撥放器
  7. 播放DVD電影

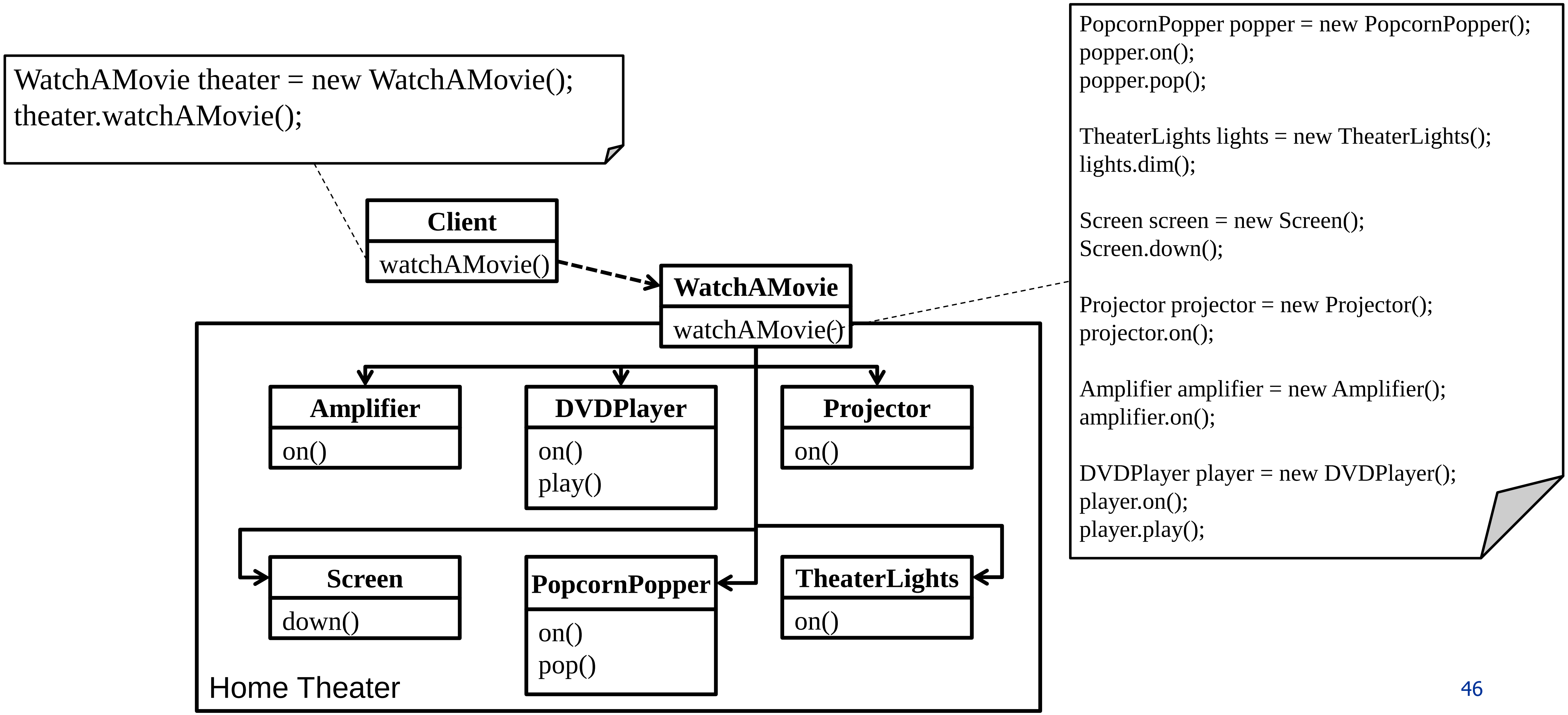
初步設計



```
PopcornPopper popper = new PopcornPopper();
popper.on();
popper.pop();

TheaterLights lights = new TheaterLights();
lights.dim();

Screen screen = new Screen();
Screen.down();

Projector projector = new Projector();
projector.on();

Amplifier amplifier = new Amplifier();
amplifier.on();

DVDPlayer player = new DVDPlayer();
player.on();
player.play();
```

問題: 當系統有更新時，如新增設備或設備功能修改時，Client的程式碼需要配合修改。

應用 Least Knowledge Principle

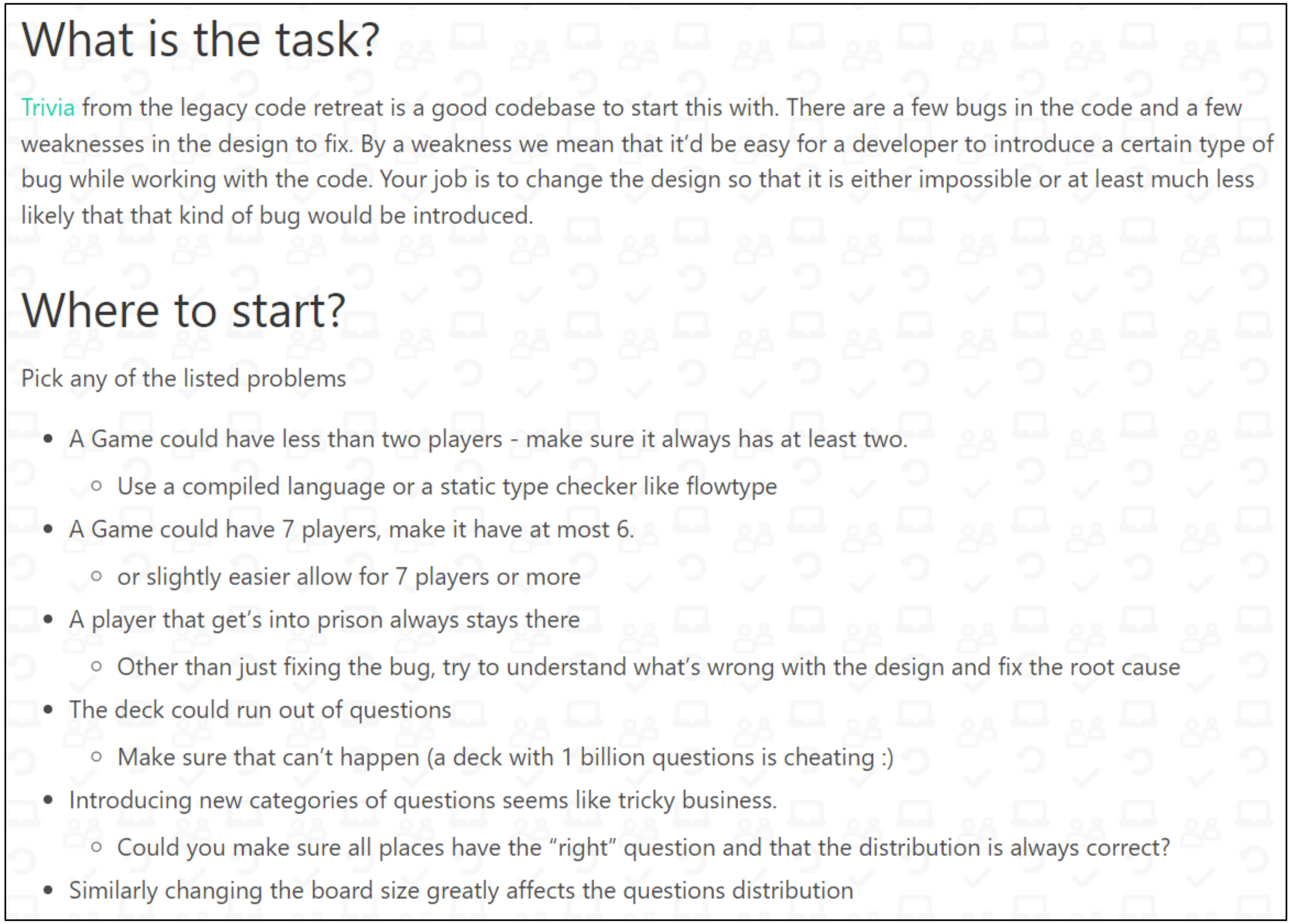


```
WatchAMovie theater = new WatchAMovie();
theater.watchAMovie();
```

```
PopcornPopper popper = new PopcornPopper();
popper.on();
popper.pop();

TheaterLights lights = new TheaterLights();
lights.dim();

Screen screen = new Screen();
Screen.down();

Projector projector = new Projector();
projector.on();

Amplifier amplifier = new Amplifier();
amplifier.on();

DVDPlayer player = new DVDPlayer();
player.on();
player.play();
```

Homework

1. 目標系統如右圖說明
2. 選擇一個程式語言實作版本 (https://github.com/caradojo/trivia)
3. 繪製出未重構前的Class Diagram
4. 重構此範例
5. 繪製出重構後的Class Diagram
6. 講解重構後的Class Diagram與程式碼並錄製為Video，將Video上傳至個人雲端空間或Youtube
7. 上傳以下資料至Moodle
   - 重構前Class Diagram
   - 重構後Class Diagram
   - 重構後程式碼
   - 講解Video連結 (可設為"知道連結者才能讀取")



What is the task?

Trivia from the legacy code retreat is a good codebase to start this with. There are a few bugs in the code and a few weaknesses in the design to fix. By a weakness we mean that it'd be easy for a developer to introduce a certain type of bug while working with the code. Your job is to change the design so that it is either impossible or at least much less likely that that kind of bug would be introduced.

Where to start?

Pick any of the listed problems

- A Game could have less than two players - make sure it always has at least two.
  - Use a compiled language or a static type checker like flowtype
- A Game could have 7 players, make it have at most 6.
  - or slightly easier allow for 7 players or more
- A player that get's into prison always stays there
  - Other than just fixing the bug, try to understand what's wrong with the design and fix the root cause
- The deck could run out of questions
  - Make sure that can't happen (a deck with 1 billion questions is cheating :)
- Introducing new categories of questions seems like tricky business.
  - Could you make sure all places have the "right" question and that the distribution is always correct?
- Similarly changing the board size greatly affects the questions distribution

https://kata-log.rocks/bugs-zero-kata