Design Principles
物件導向設計原則

Shin-Jie Lee (李信杰)
Associate Professor
Department of CSIE
National Cheng Kung University

Design Principles

- **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
- Acyclic Dependencies Principle (ADP)
- Don’t Repeat Yourself (DRY)
- Keep It Simple Stupid (KISS)

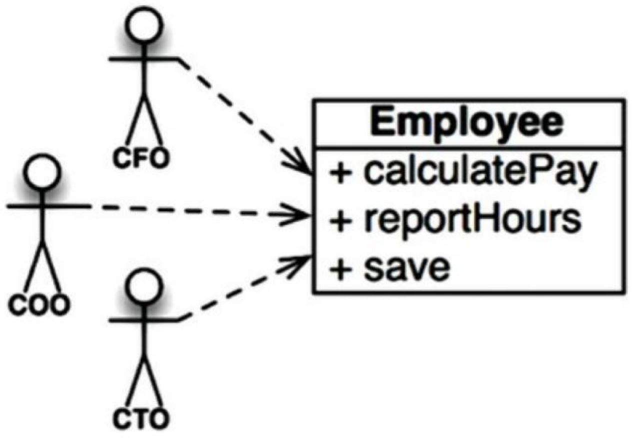
說不清的 Single Responsibility Principle?

兩種較具體的判定法

先來看看Robert C. Martin怎麼說

SRP: THE SINGLE RESPONSIBILITY PRINCIPLE[1]

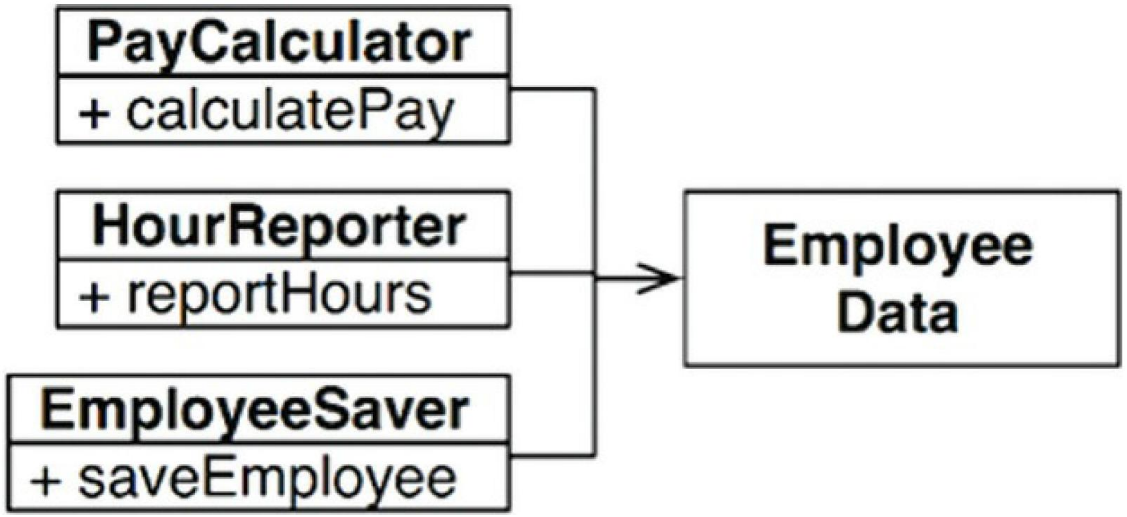
- Historically, the SRP has been described this way:
  - ***A module should have one, and only one, reason to change.***
- We can rephrase the principle to say this:
  - ***A module should be responsible to one, and only one, actor.***
- This class violates the SRP because those three methods are responsible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



“Clean Architecture: A Craftsman's Guide to Software Structure and Design,” Robert C. Martin, Prentice Hall, 2017.

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.



但是，判定SRP常是主觀的

- 我覺得有滿足SRP，但你不覺得有滿足SRP
- 爭辯的結果可能是先擱置不理，遇到痛了再討論
- 什麼是痛了? 當一個class久了變成**Large Class**時就有感覺了
- 變成Large Class後不得不面對拆解它，那有沒有「**更具體**」的SRP判斷與拆解Responsibility方式?

兩種更具體的SRP判定法

1. 結構內聚力判定法
   - ➢Method間的結構關係

2. 語意結合判定法
   - ➢Class Name與Method Name間的結合語意

結構內聚力判定法
Based on Hitz&Montazeri's LCOM4 Metrics

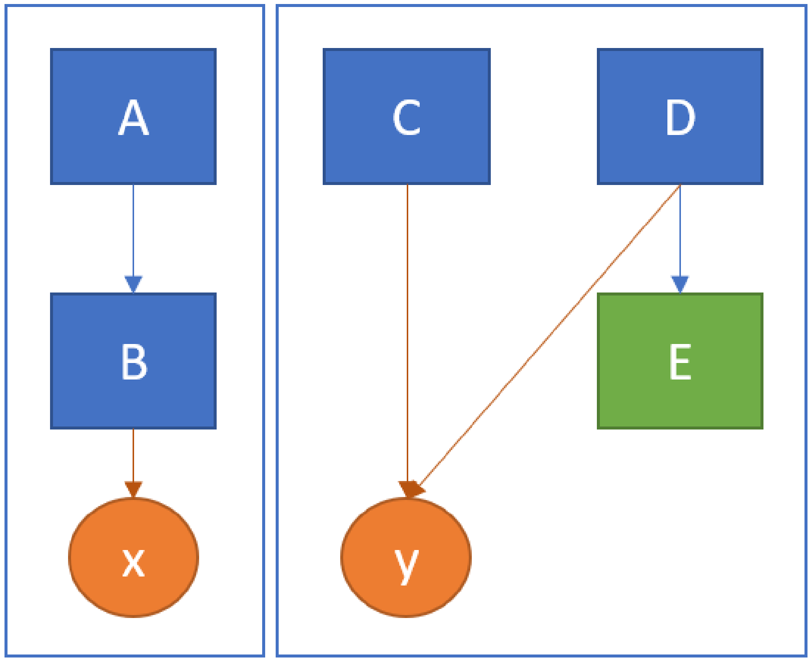
- 首先，檢視一個class中的methods相互依賴或共用屬性
  - 基於Hitz&Montazeri's LCOM4 (Lack of Cohesion in Methods)的內聚力度量觀念

> Methods A and B are related if:
> 1. they both access the same class-level variable, or
> 2. A calls B, or B calls A.

- 接著，將methods根據依賴分群，每一群可能代表一個responsibility

Example 1

- 例如一個class中有method A, B, C, D, E，關係如下，因此可參考是否判定為兩個responsibility，進而拆分為兩個Class



From: https://objectscriptquality.com/docs/metrics/lack-cohesion-methods-lcom4

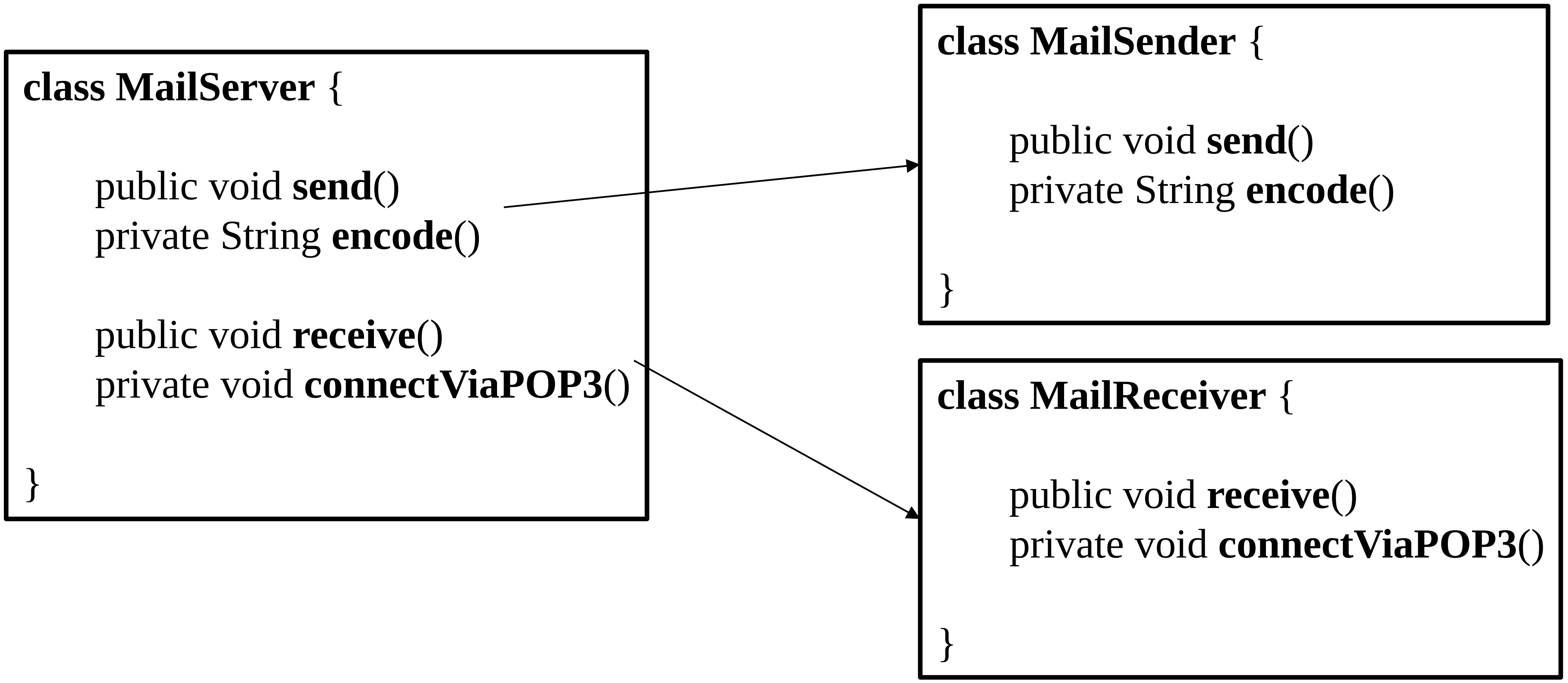
Example 2

```
class MailServer {
  public void send(String to, String content){
    encode(content);
    //…
  }
  private String encode(String content){
    // encode content;
  }
  public void receive(String account) {
    connectViaPOP3();
    // …
  }
  private void connectViaPOP3(){
    // connect to a server via POP3 protocol;
  }
}
```

send()依賴encode()，成為一群

receive()依賴connectViaPOP3()成為一群

可將兩群Method拆解成兩個Class (Refactoring by *Extract 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()

}
```

高內聚與低耦合 (High cohesion, loose coupling)

- 結構內聚力判定法的精神就是將一個class內不互相依賴的method群拆解出去

- 進而維持高內聚與低耦合 (high cohesion, loose coupling)的原則

- 當然，有時一個class內可能無法完美的分群，會需要再搭配重構，但至少此方法提供了一個比較具體的single/multiple responsibility判定參考

語意結合判定法

From “Head First Software Development”

- ❑ Class name代表一個物件的語意，method name代表此物件的行為語意
- ❑ 可藉由兩個name的結合語句來判定是否具合理語意

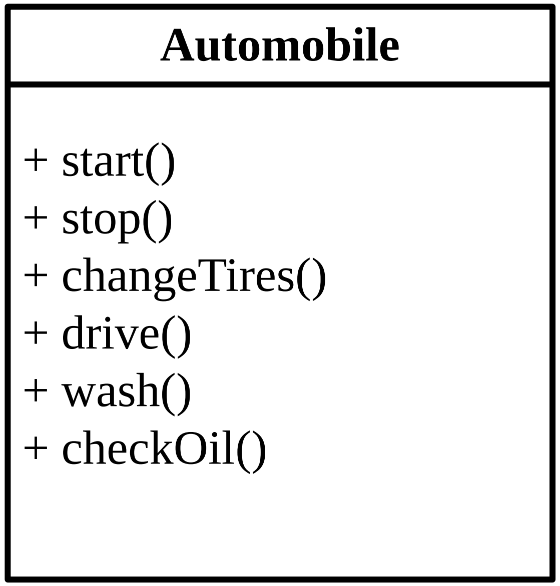
SRP Analysis for <class name> ____________

	Follows SRP	Violates SRP
The <class name> ____ <method name> ____ itself.	☐	☐
The ____________ ____________ itself.	☐	☐
The ____________ ____________ itself.	☐	☐
The ____________ ____________ itself.	☐	☐
The ____________ ____________ itself.	☐	☐
⋮		

From: Head First Software Development. Dan Pilone, Russ Miles. (2007)

Example

- 假設我們有一個class



Example

- ❑ 首先，依每個method填入下表，構成語句:
  - ➢ *The Automobile (主詞) starts (動詞) itself.*
- ❑ 接著，判定此句子是否具合理語意，若合理則留下，若不合理則考慮將此method移出此class

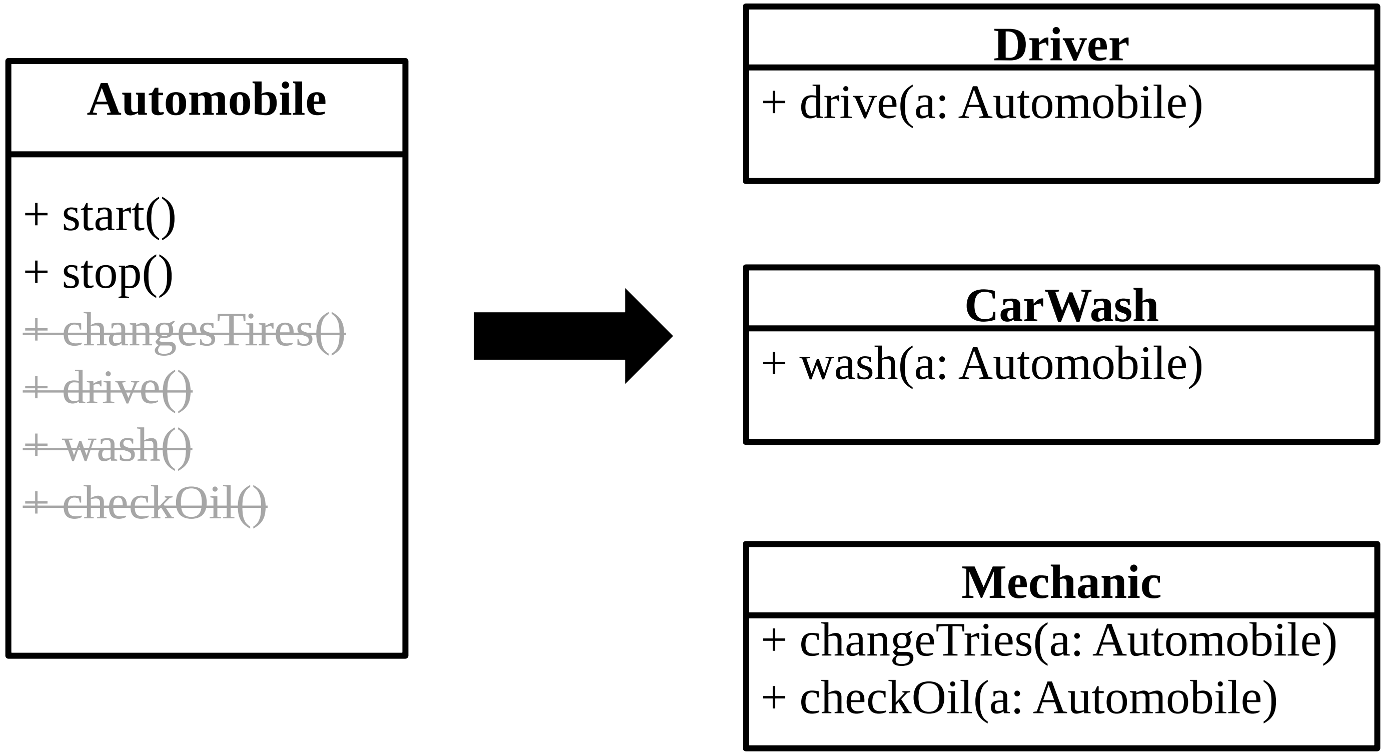
SRP Analysis for Automobile

				Follows SRP	Violates SRP
The	Automobile	starts	itself.	☑	☐
The	Automobile	stops	itself.	☑	☐
The	Automobile	changes tires	itself.	☐	☑
The	Automobile	drives	itself.	☐	☑
The	Automobile	washes	itself.	☐	☑
The	Automobile	checks	itself.	☐	☑

⋮

Example

- 拆解後的Automobile class (僅剩2個method)在語意上符合Single Responsibility Principle



總結：兩者判定法的限制

- 當一個class內method間結構關係複雜時，結構內聚力判定法可能較困難
- 當一個class name語意太general時(如XXXManager/Controller)，會讓所有method name都可與class name語意結合，造成語意結合判定失效

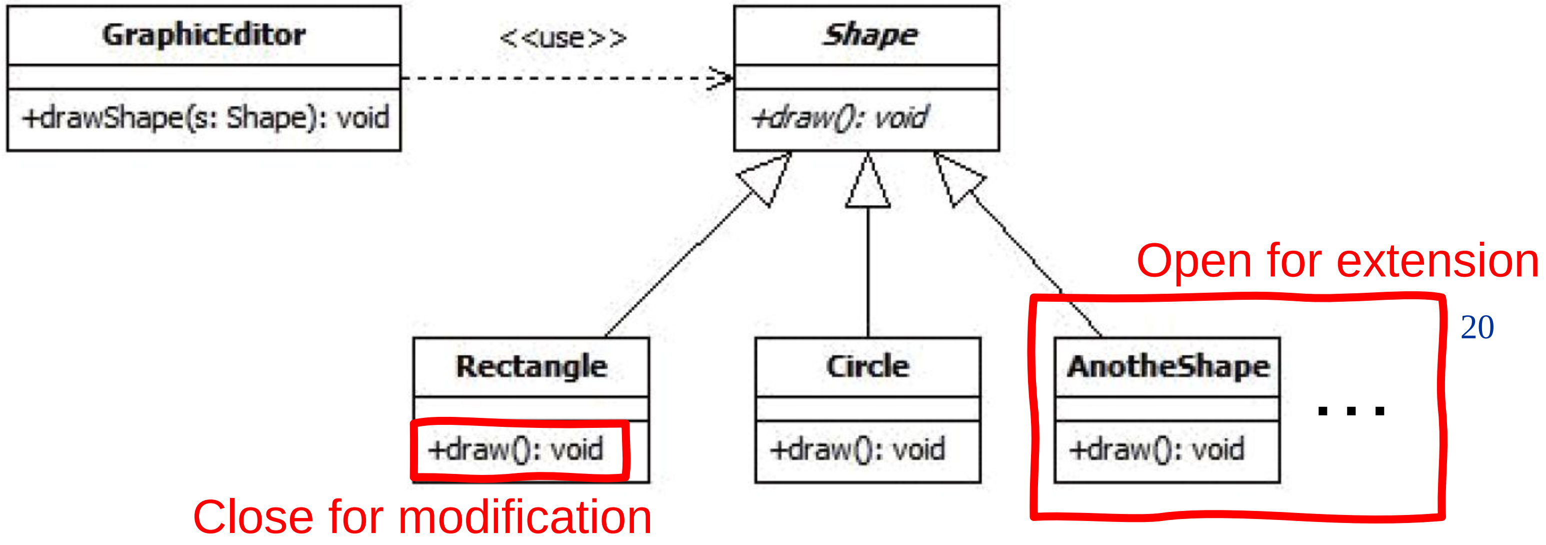
- 因此兩者可以互補參考使用

Open-Close Principle (開放關閉原則)

Open-Close Principle (開放關閉原則)

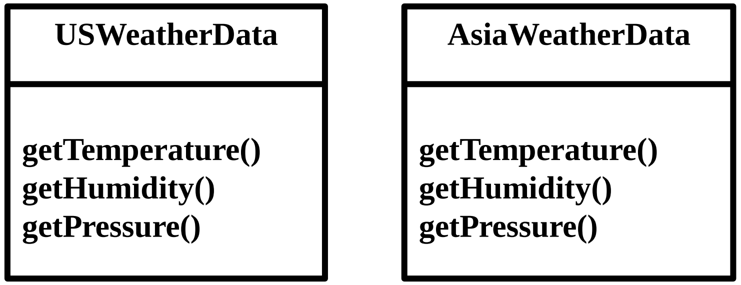
- Open for extension, but closed for modification
  - 一個模組必須有彈性的開放往後的擴充，並且避免因為修改而影響到使用此模組的程式碼。

- 一個具備Open-Close Principle的結構：



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。

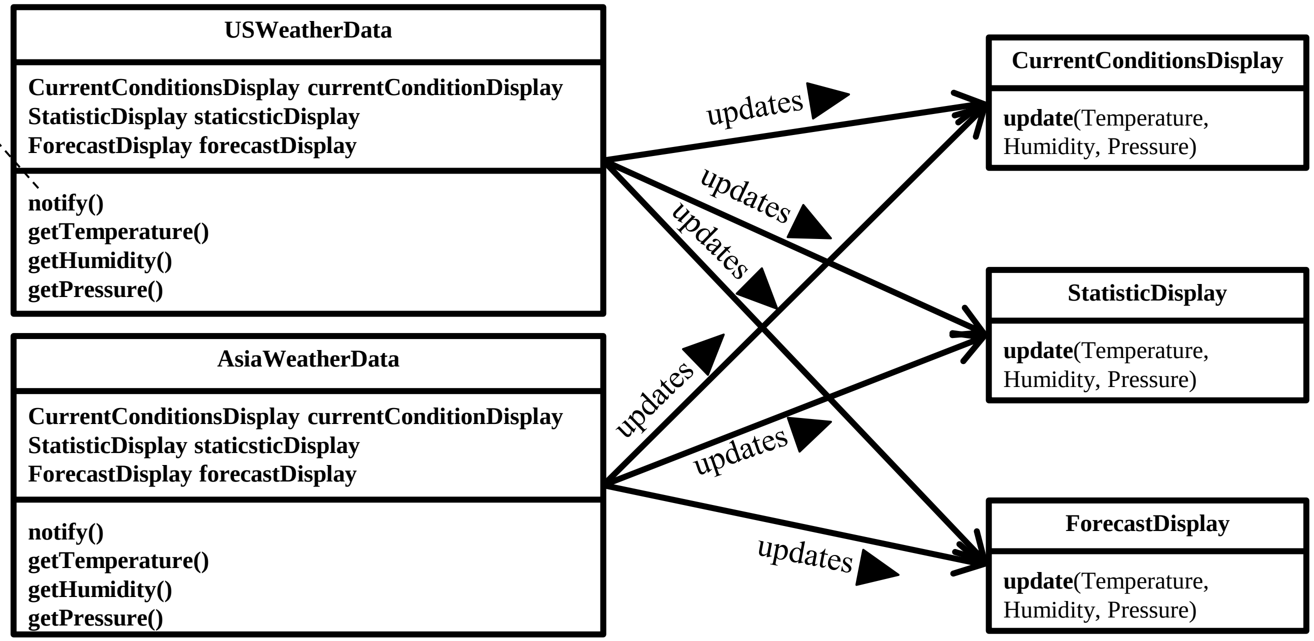


需求描述$_2$

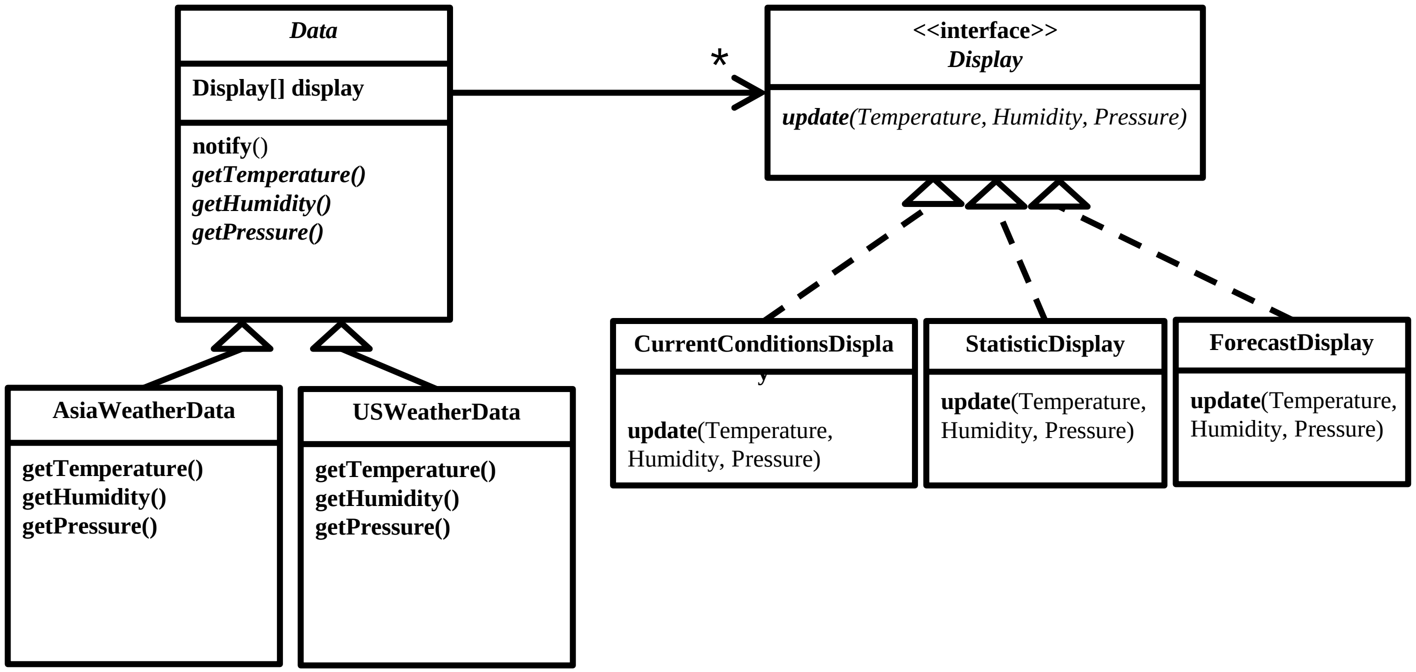
- 此系統提供三種氣象播報顯示: current conditions, weather statistics and a simple forecast，且當天氣資料更新時，所有顯示將會即時更新。

```
{  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。

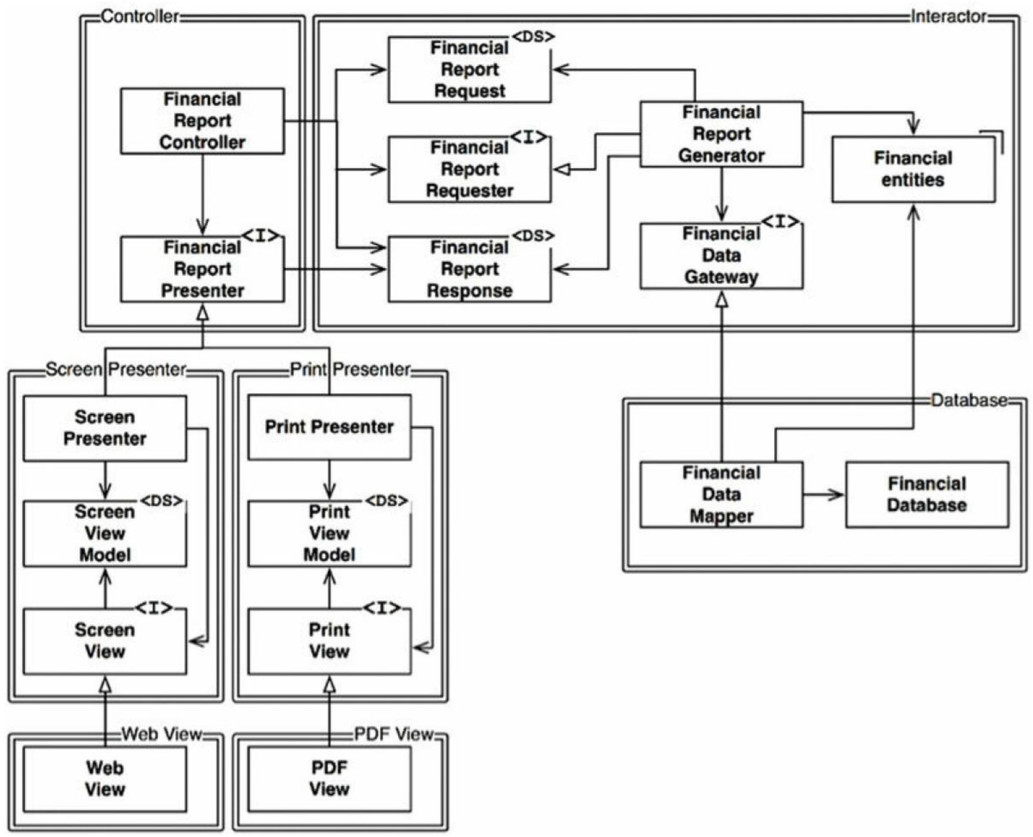

USWeatherData
CurrentConditionsDisplay currentConditionDisplay
StatisticDisplay staticsticDisplay
ForecastDisplay forecastDisplay
notify()
getTemperature()
getHumidity()
getPressure()
AsiaWeatherData
CurrentConditionsDisplay currentConditionDisplay
StatisticDisplay staticsticDisplay
ForecastDisplay forecastDisplay
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



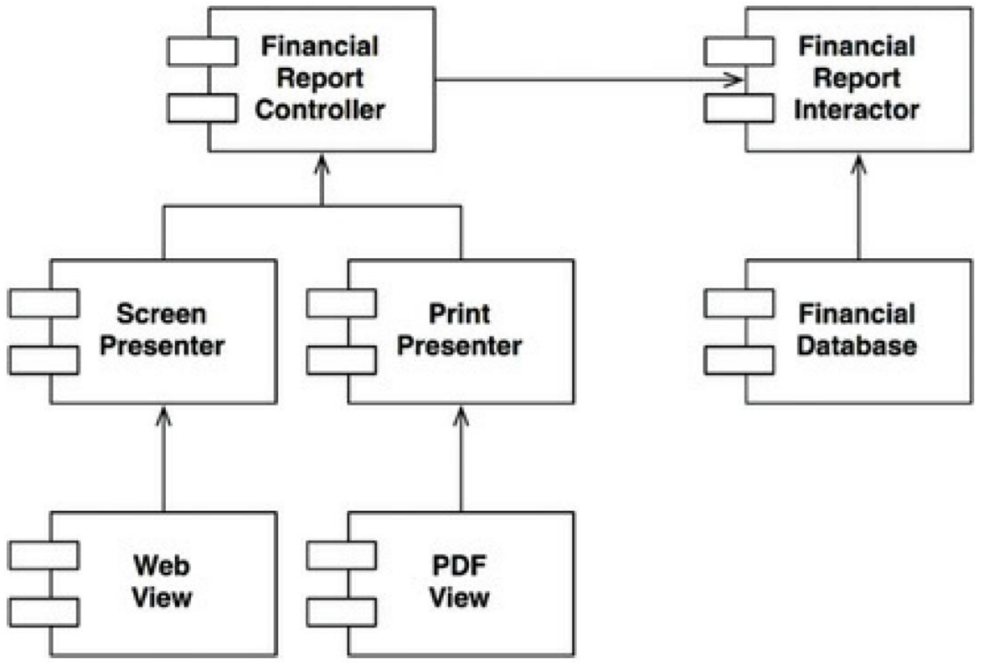
Open-Close Principle at the Architecture Level$_1$

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



Open-Close Principle at the Architecture Level$_2$

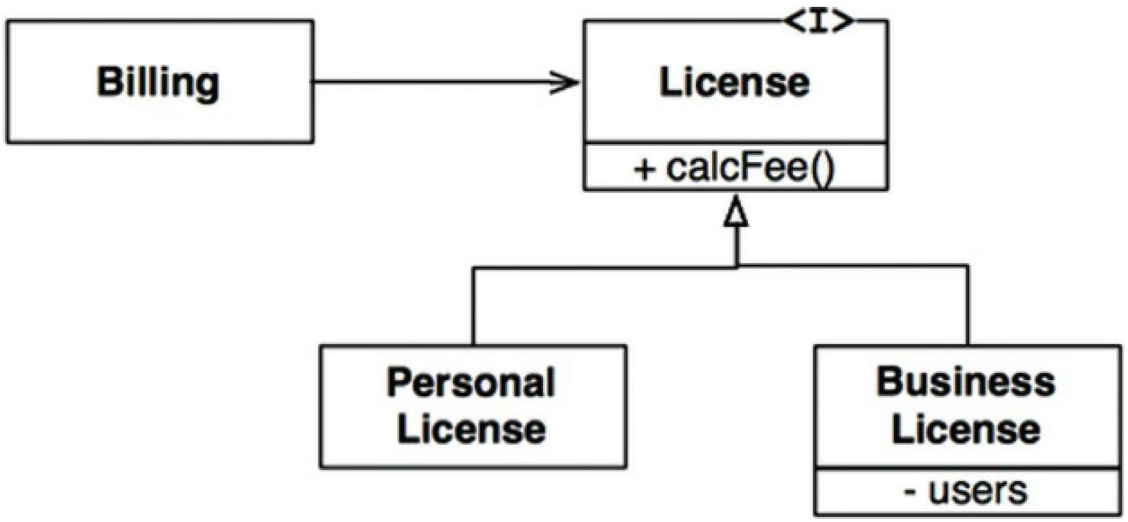
- All component relationships are **unidirectional**.
- Higher-level components in that hierarchy are protected from the changes made to lower-level components.
- This is how the OCP works at the architectural level.



Liskov Substitution Principle

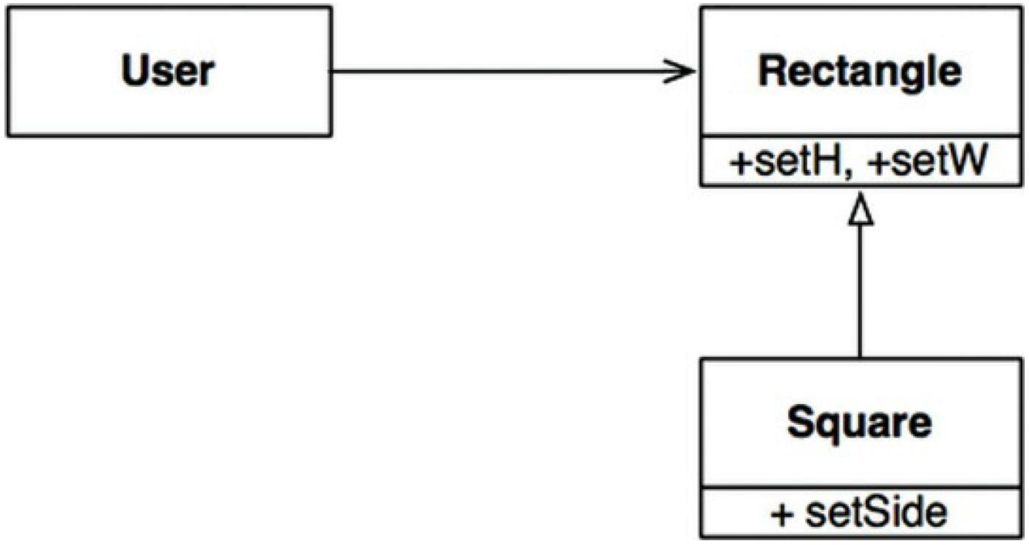
LSP: The Liskov Substitution Principle

- 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

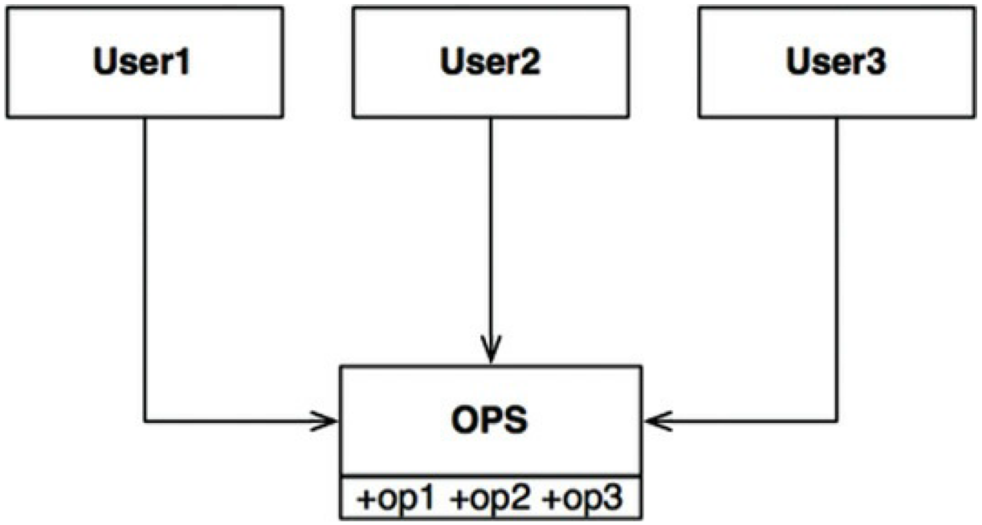
- The canonical example of a violation of the LSP is the square/rectangle problem



Interface Segregation Principle

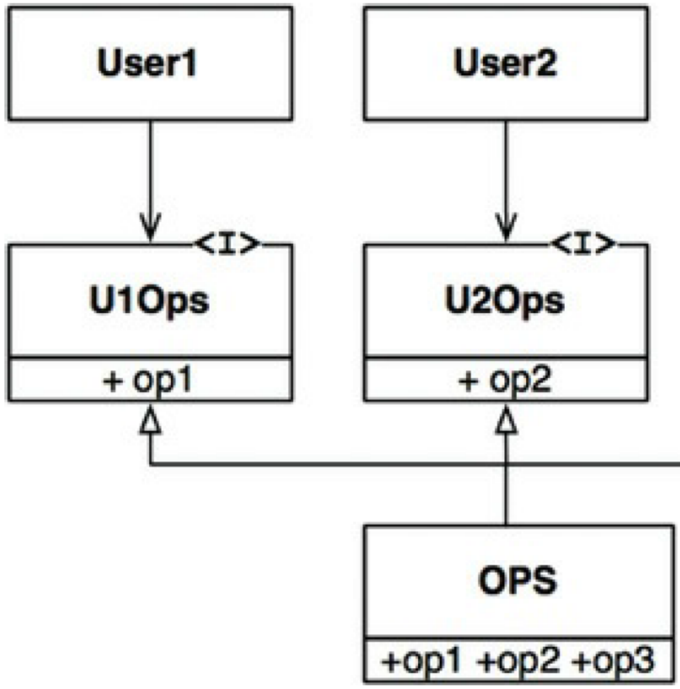
ISP: The Interface Segregation Principle[1]

- 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$_2$

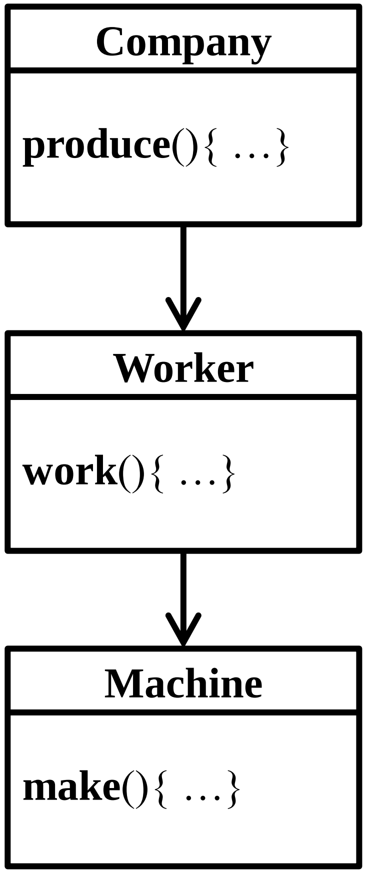
- This problem can be resolved by segregating the operations into interfaces
- Thus a change to `OPS` that `User1` does not care about will not cause `User1` to be recompiled and redeployed.



Dependency Inversion Principle (依賴反向原則)

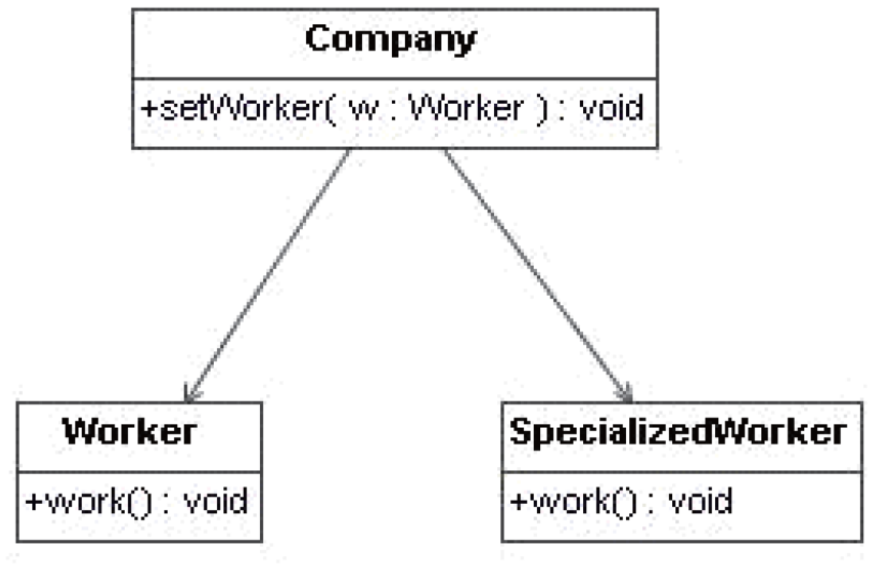
Dependency Inversion Principle (依賴反向原則)

- 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

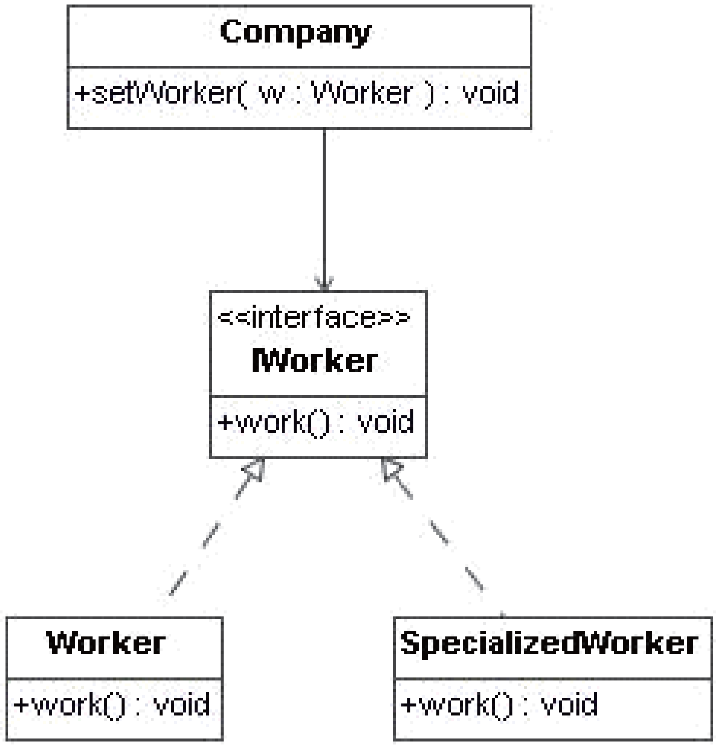
- 如果公司因業務需要必須增聘具有專長的員工（也許稱之為SpecializedWorker 模組），則必須修改複雜的公司模組（即高層模組）的程式碼，這種修改將影響公司的模組結構，這樣就違反DIP 原則



```
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

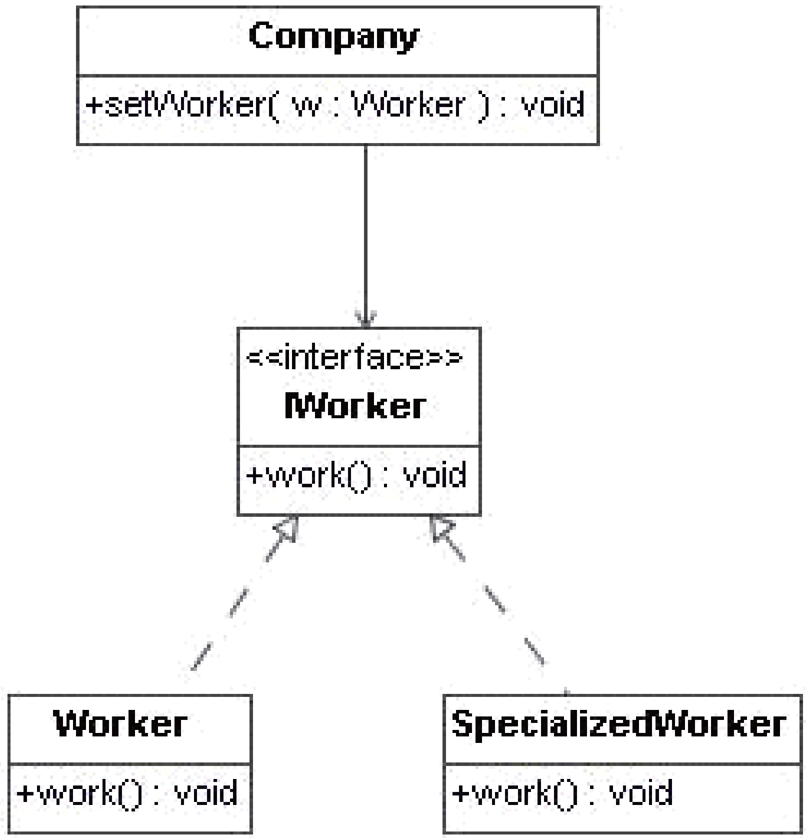
- 可以在公司模組與員工模組之間介入一種「抽象層」（Abstract Layer），公司模組與員工模組皆依存於這種抽象層，如此不論如何增聘員工都不致於影響公司模組本身，這種抽象層一般都是一種介面，亦即IWorker 類別



```
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。



Tips

- 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.

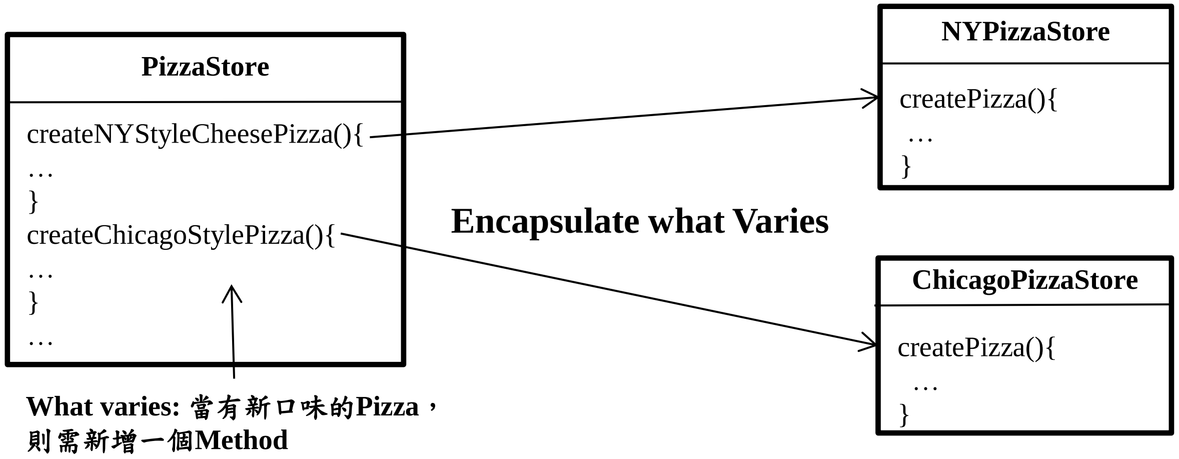
Tips

- 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.

Encapsulate what varies (封裝改變)

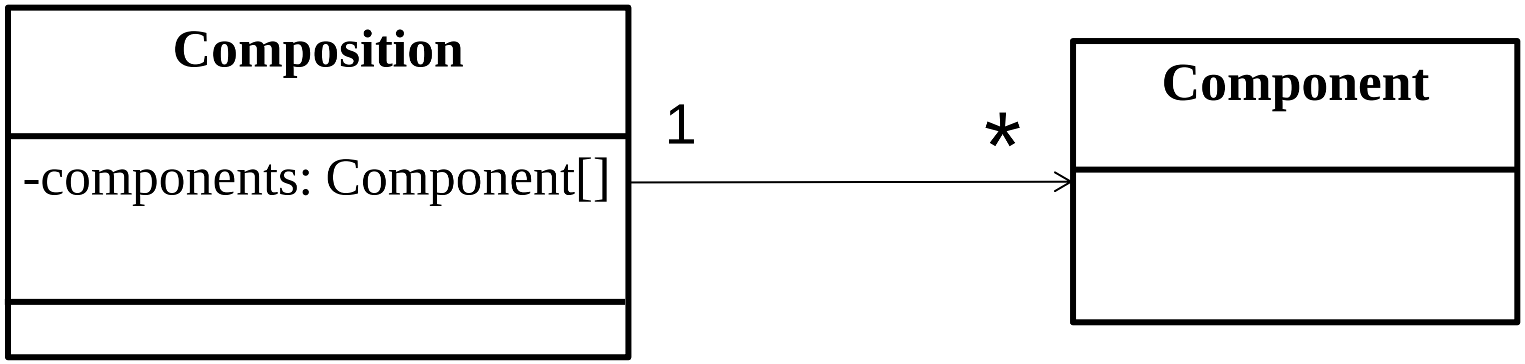
Encapsulate what varies (封裝改變)

- 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。



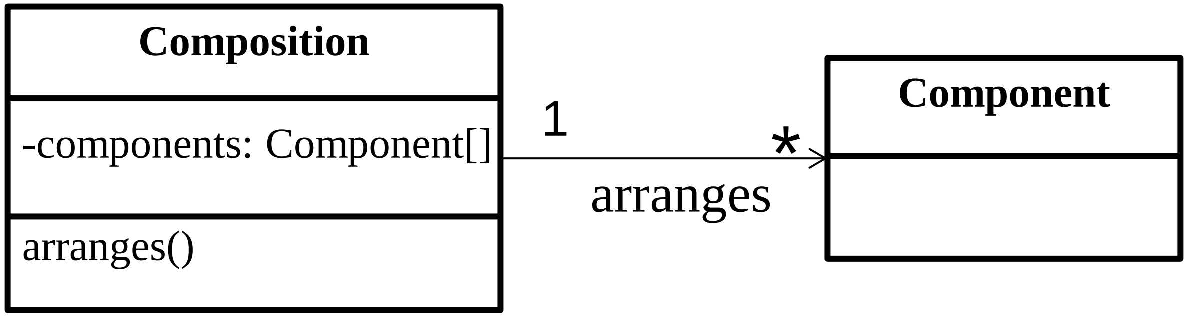
範例一：需求描述$_1$

- 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。



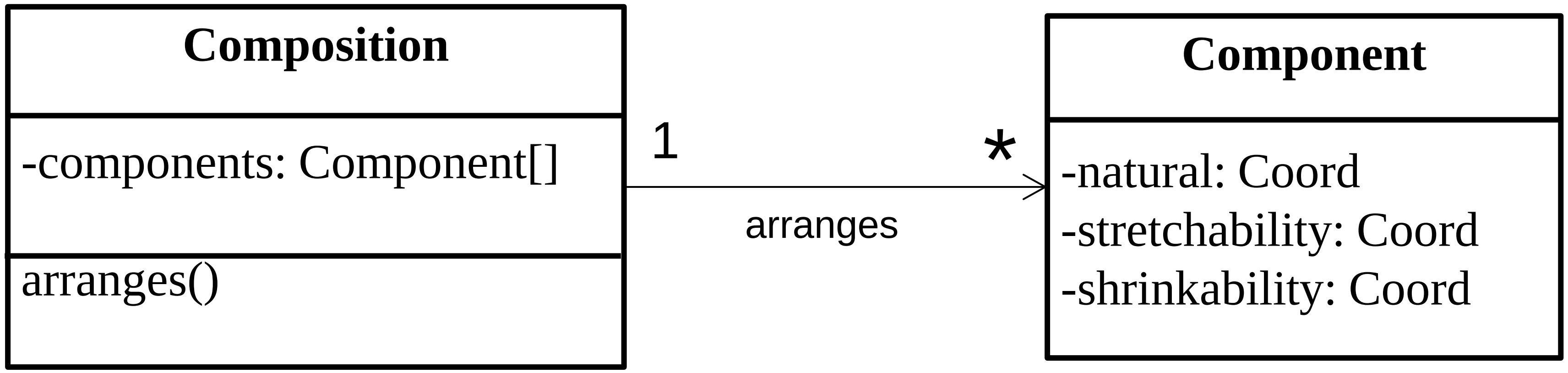
需求描述$_2$

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



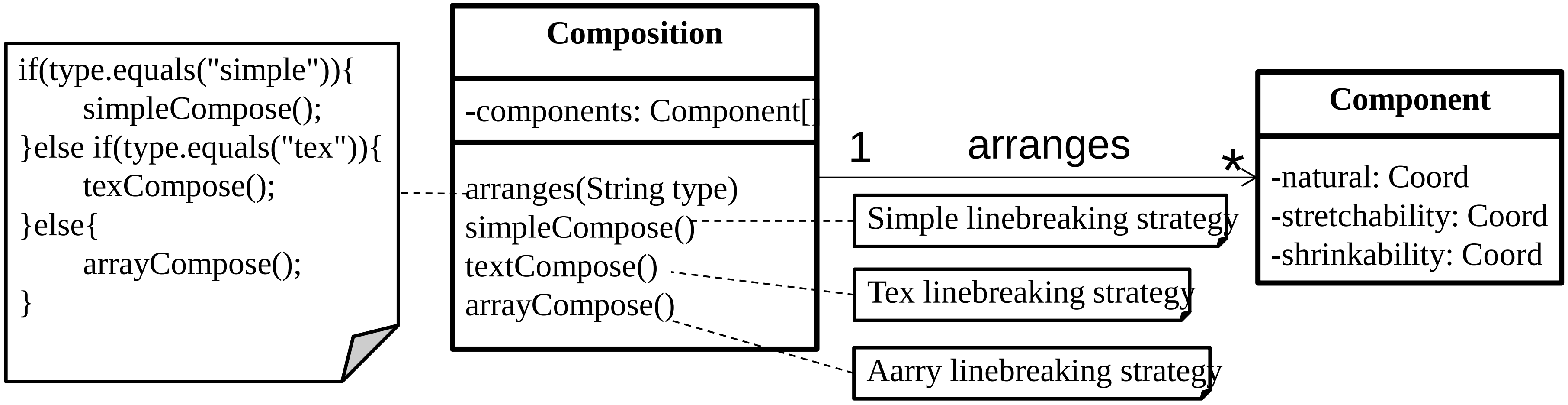
需求描述$_3$

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



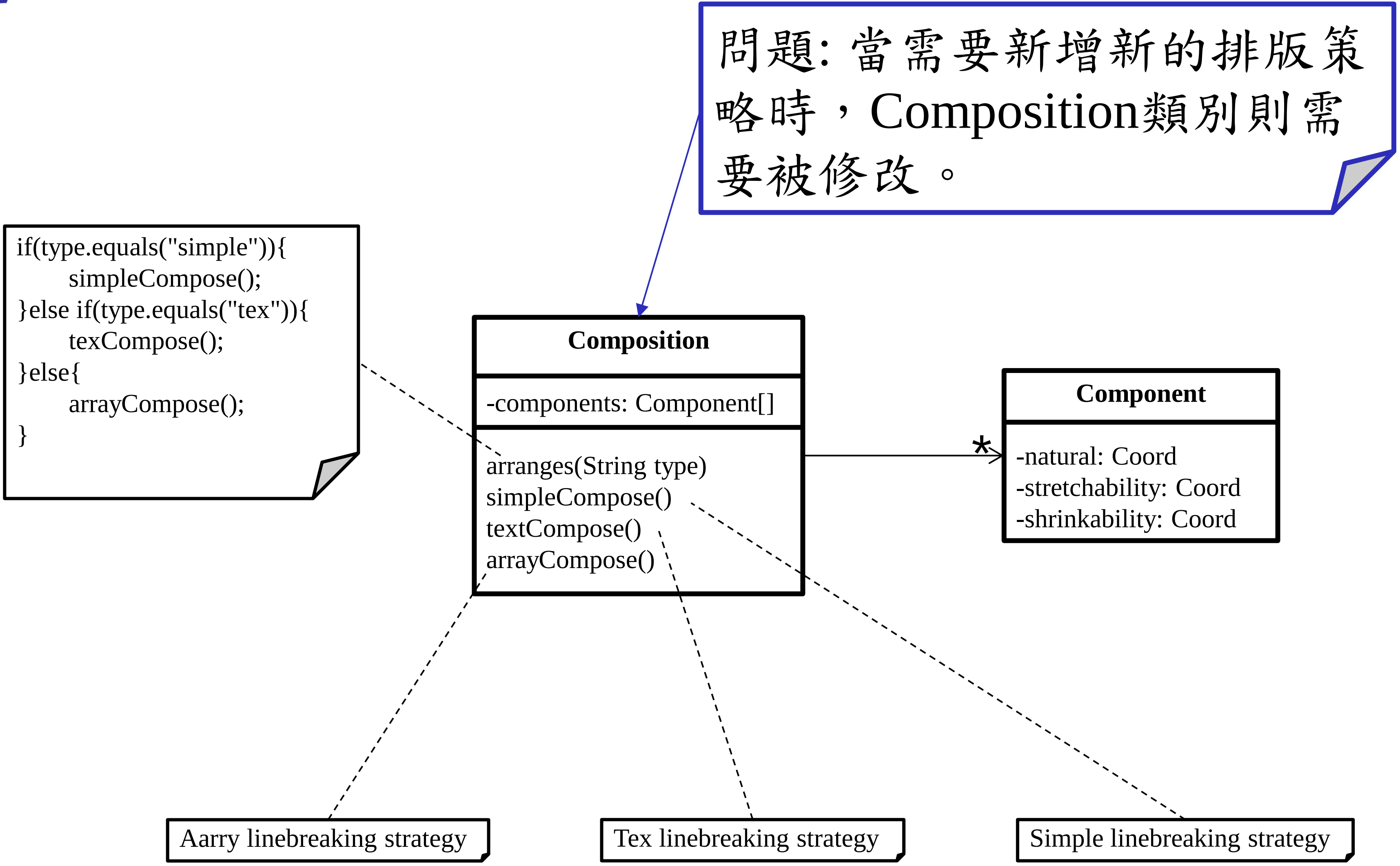
需求描述$_4$

- 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。



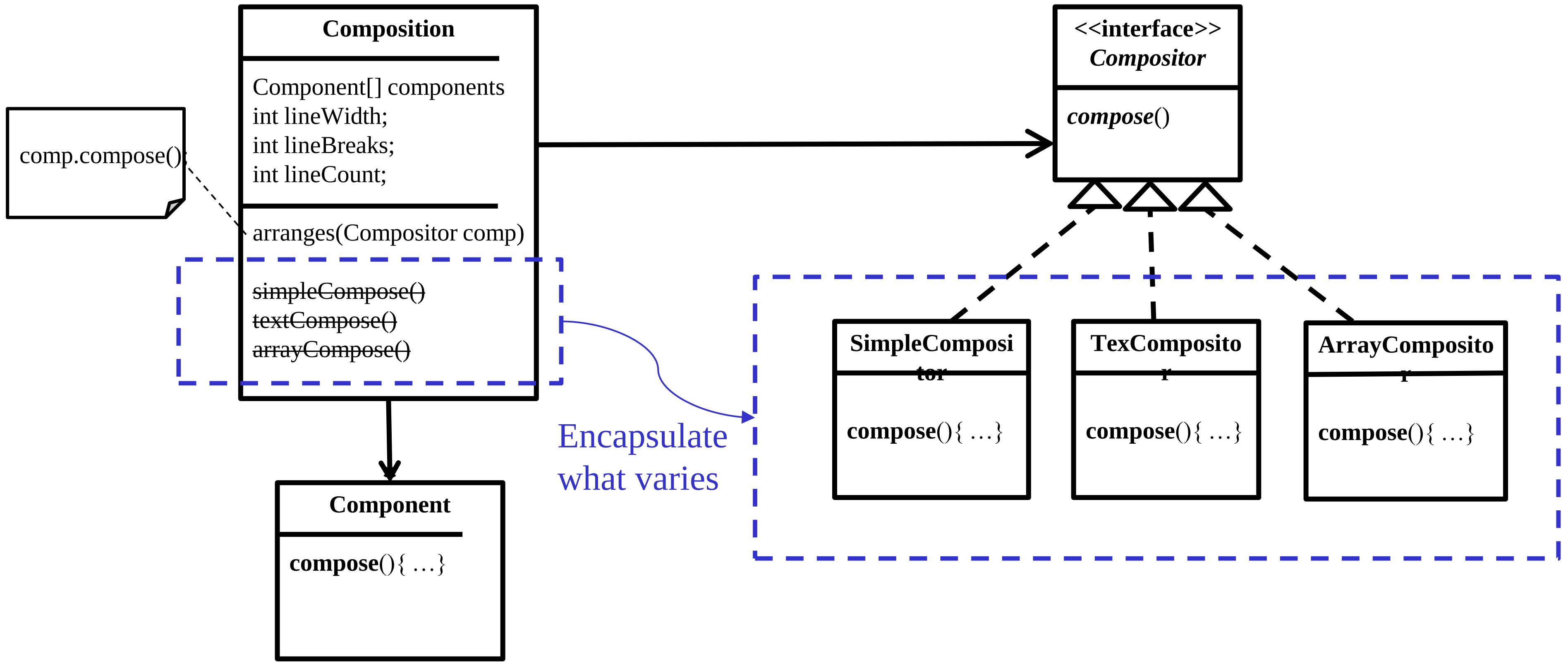
```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

初步設計之問題



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

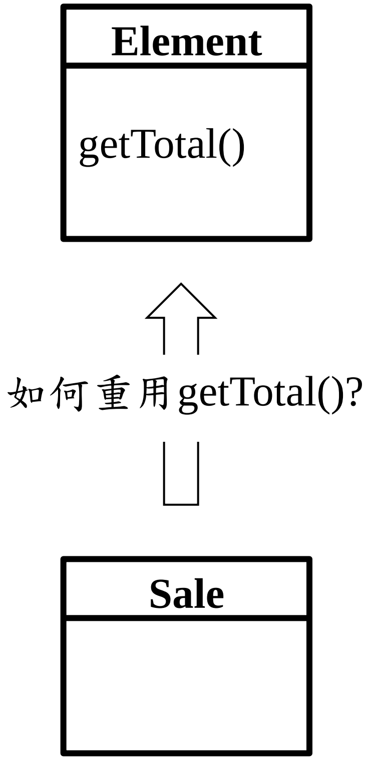
應用Encapsulate what varies 設計原則



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

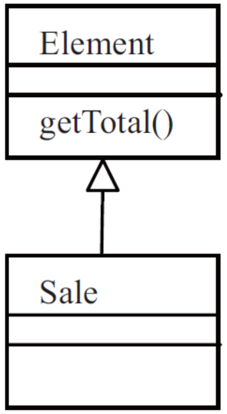
Favor composition over inheritance (善用合成取代繼承)

- 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

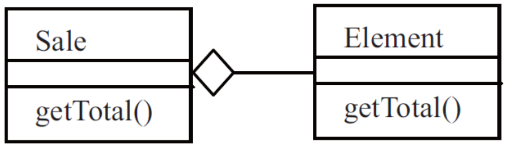
- 優點
  - 透過簡便的擴充(Extend)繼承，就可以實做新的功能。
- 缺點
  - 父類別修改會影響到子類別
    - 因為子類別繼承父類別的屬性和方法，父類別的屬性和方法一旦修改，將可能會影響到所有繼承它的子類別。
  - 無法在執行時期改變所需物件
    - 從父類別實做繼承，無法在執行時期設定不同物件以呼叫不同的Method功能。

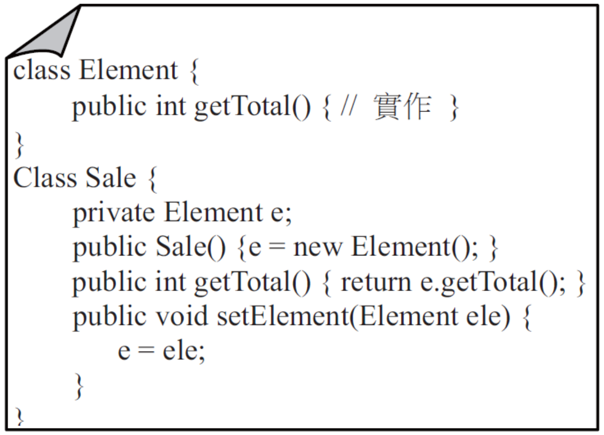


```
class Element {
        public int getTotal() { // 實作 }
}
Class Sale extends Element { }
```

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



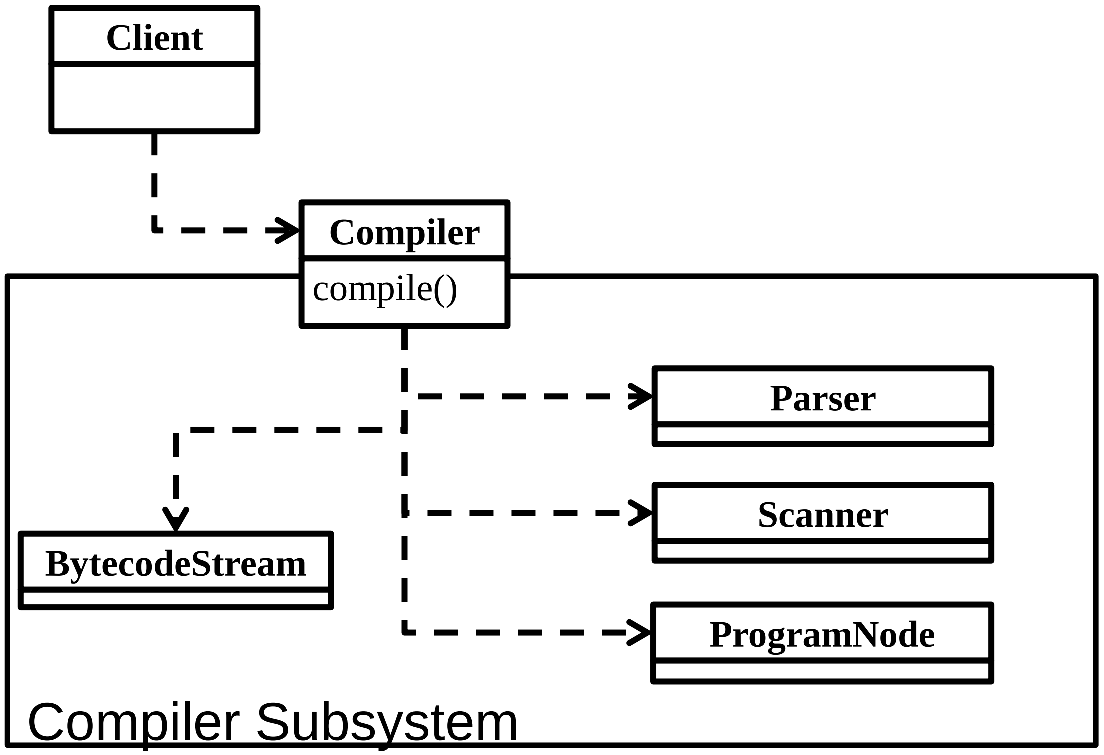


```
class Element {
        public int getTotal() { // 實作 }
}
Class Sale {
        private Element e;
        public Sale() {e = new Element(); }
        public int getTotal() { return e.getTotal(); }
        public void setElement(Element ele) {
            e = ele;
        }
}
```

Least Knowledge Principle (最小知識原則)

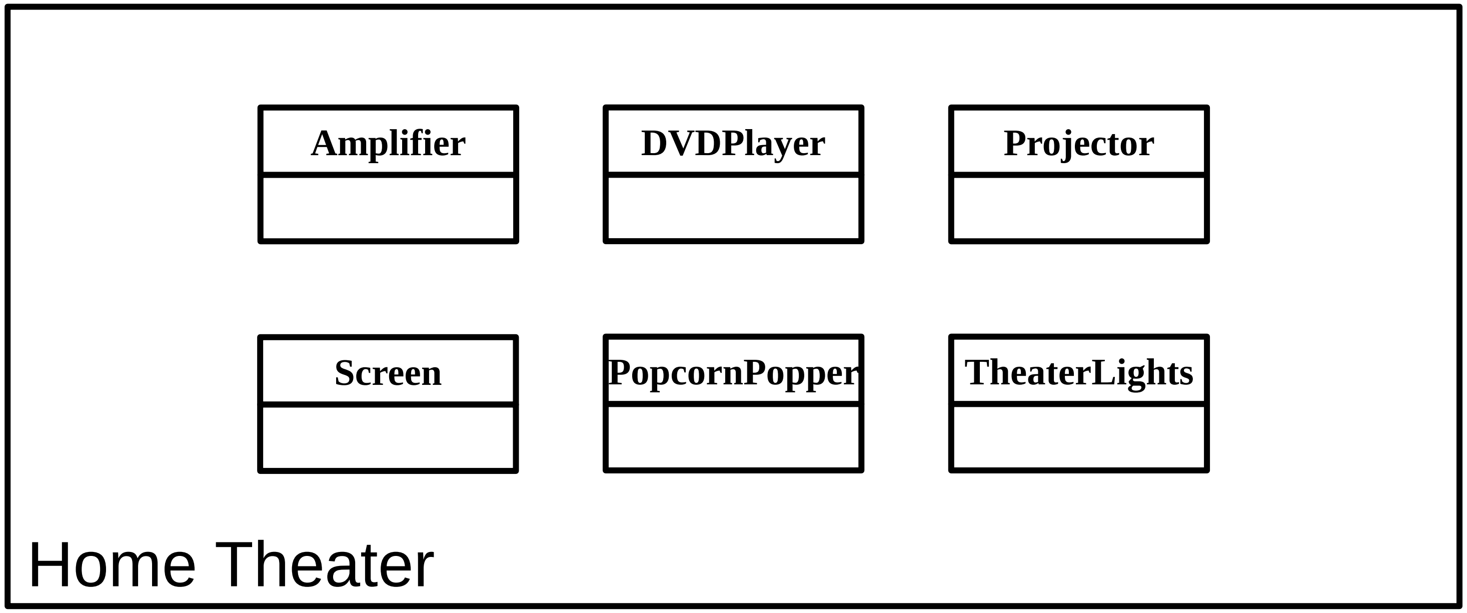
Least Knowledge Principle (最小知識原則)

- 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$_1$

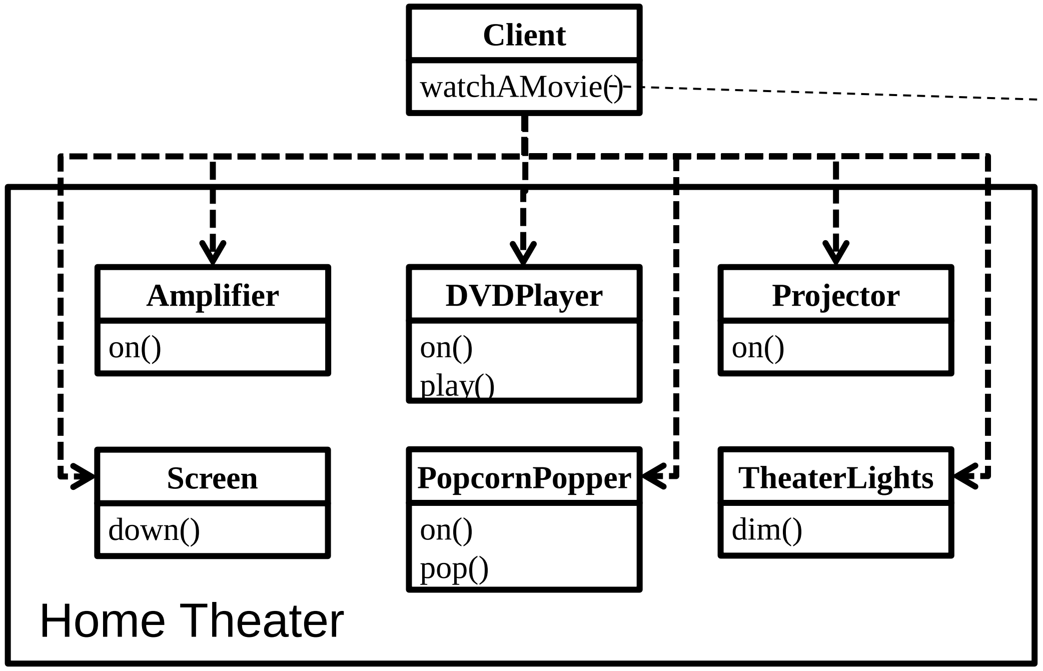
- 一個家庭劇院系統包含Amplifier、DVD Player、Projector、Screen、Popcorn Popper和Theater Lights.



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

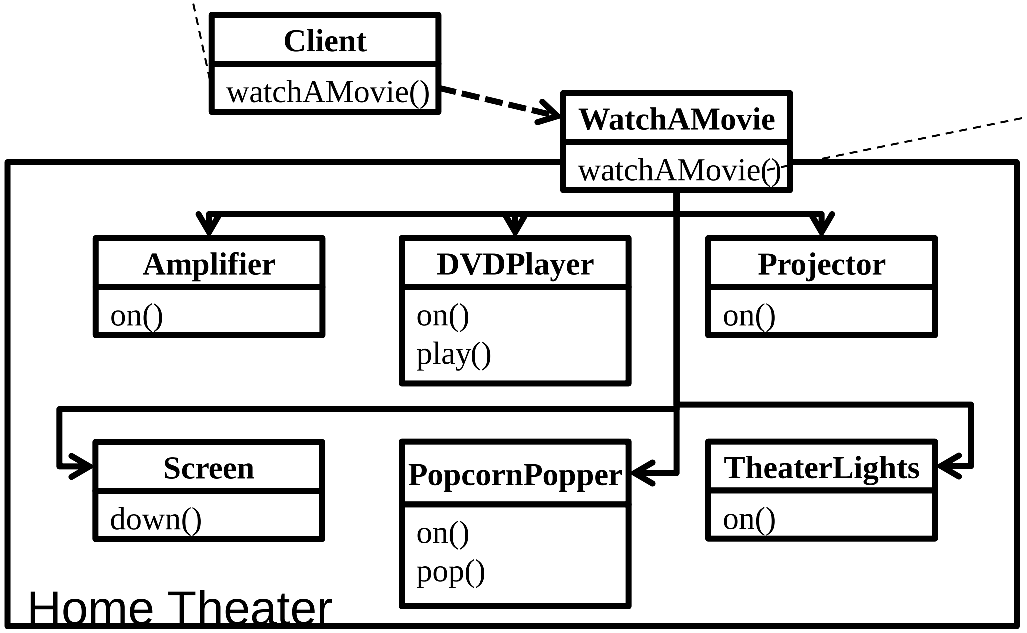


```
PopcornPopper popper = new PopcornPopper();
popper.on();
popper.pop();

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();
```



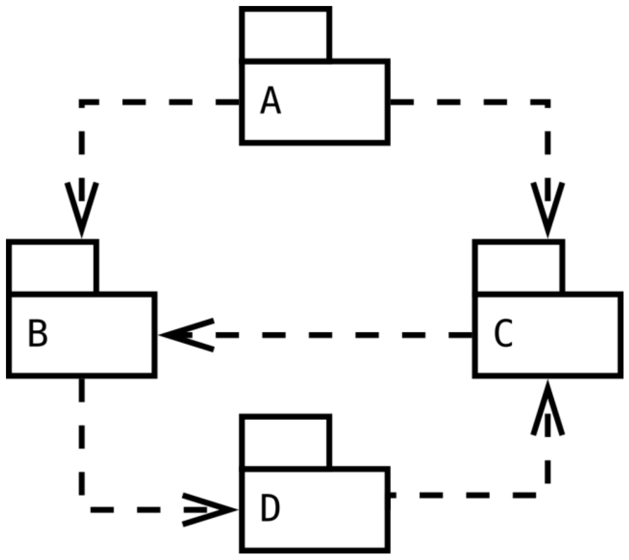
```
PopcornPopper popper = new PopcornPopper();
popper.on();
popper.pop();

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();
```

Acyclic Dependencies Principle (ADP)

Acyclic Dependencies Principle (ADP)

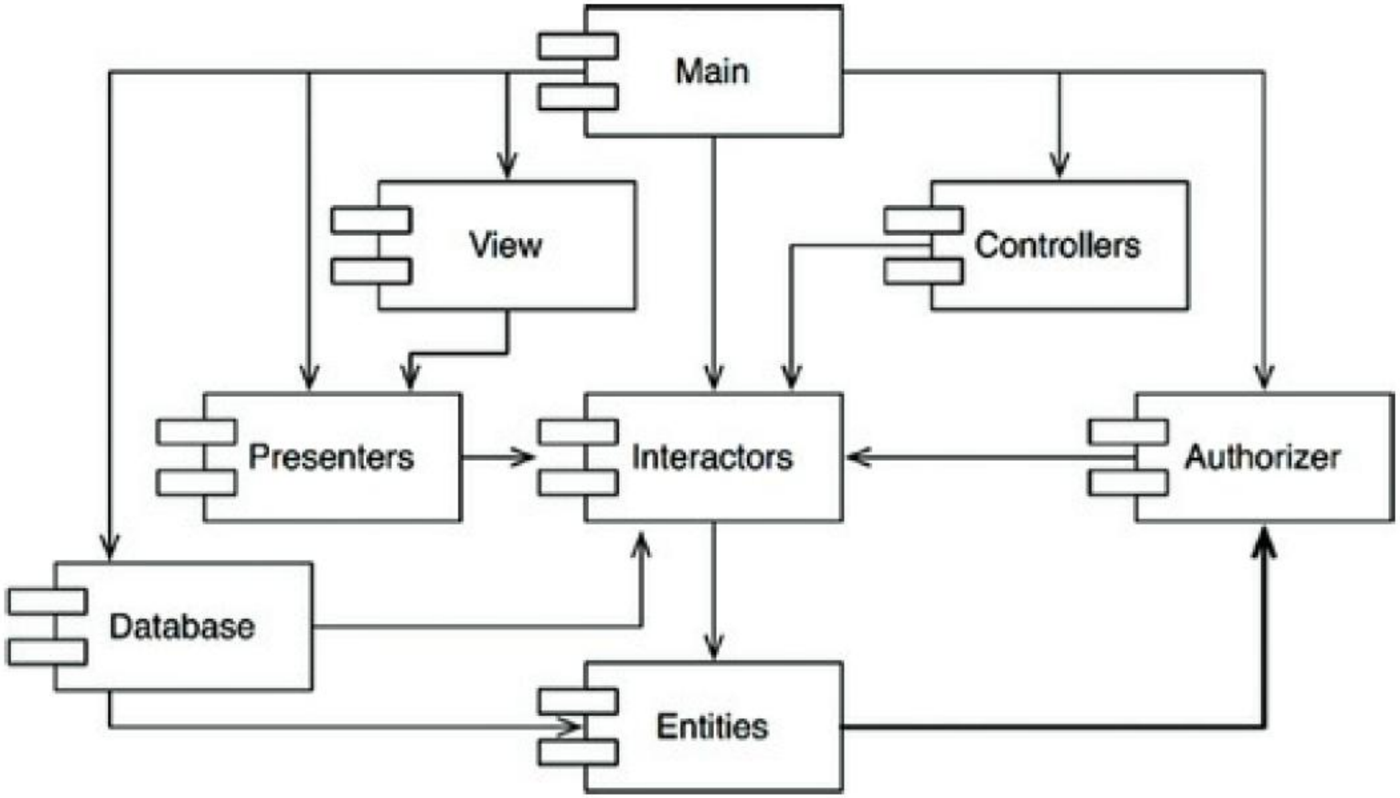
- "the dependency graph of packages or components should have no cycles" – Robert C. Martin

- ADP主要針對的是套件之間的關係，適用於架構級



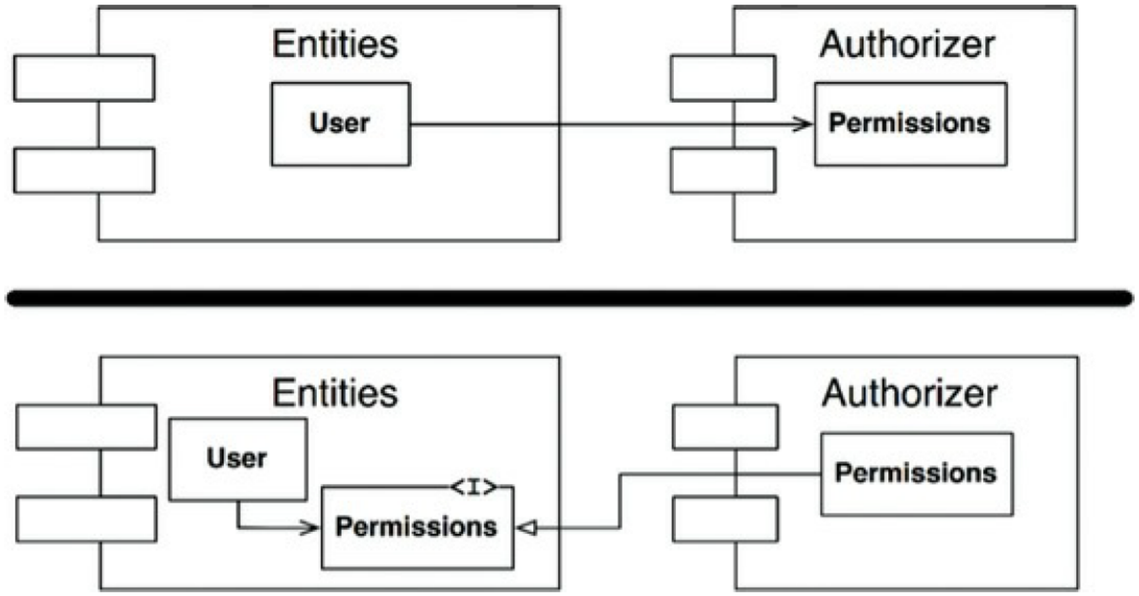
From: Wikipedia

Example


Main
View
Controllers
Presenters
Interactors
Authorizer
Database
Entities

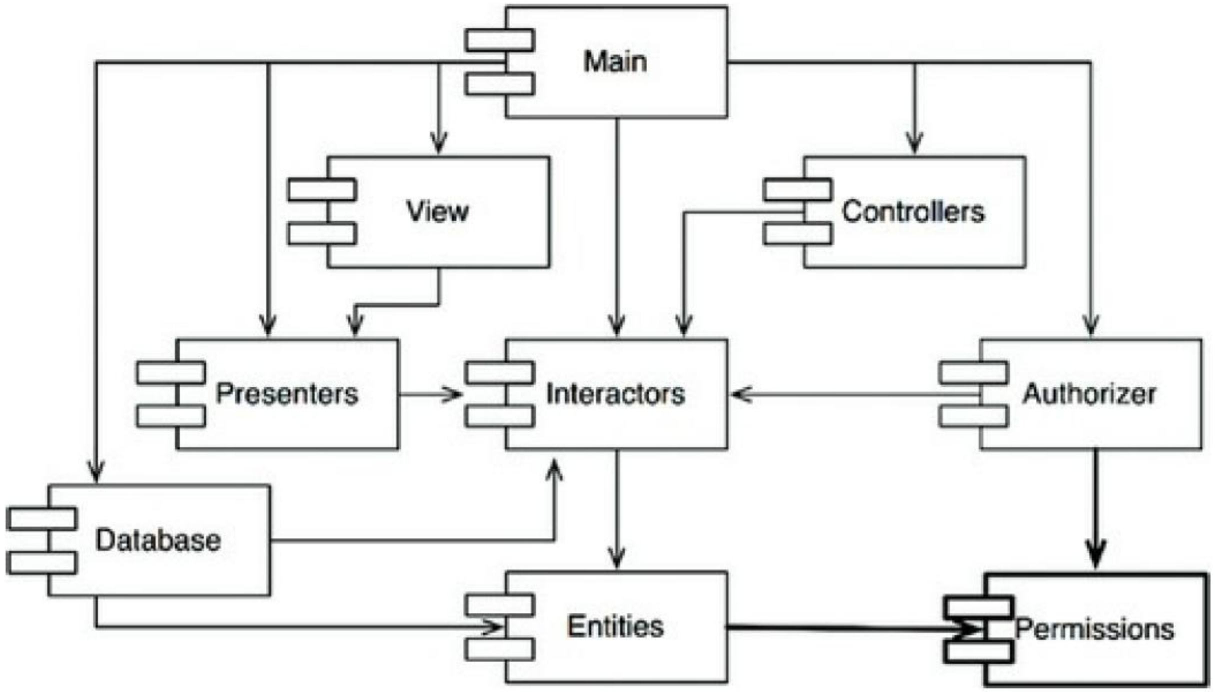
Cycle breaking strategies$_1$

- ❑ Dependency inversion principle



Cycle breaking strategies$_2$

- Create a new package, and move the common dependencies there.

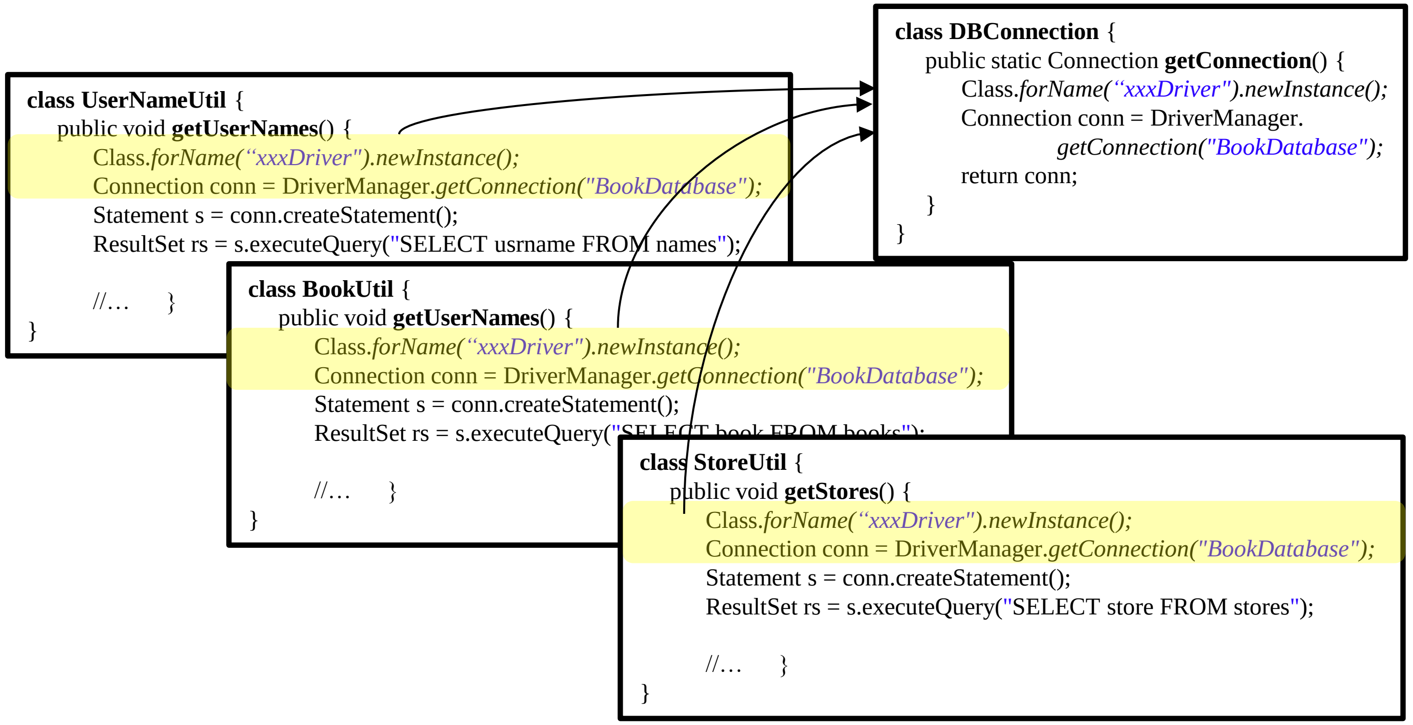


Don’t Repeat Yourself (DRY)

Don’t Repeat Yourself (DRY)

- 對於每個知識點，系統中都只有一個明確而權威的表示
- 單一事實源(Single Source of Truth)
- 適用於所有的軟體工作，包括文件、架構和設計、測試程式和原始程式

Avoid duplicate code by separation



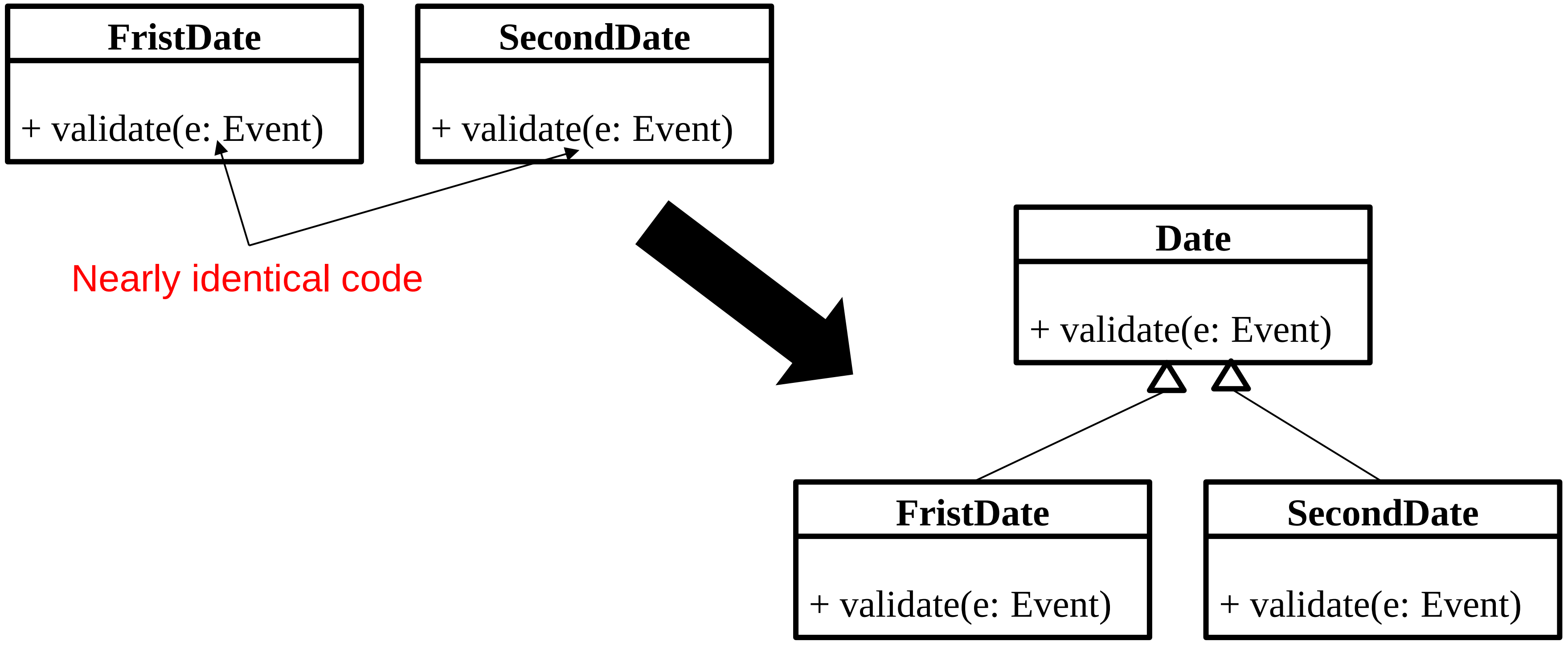
```
class UserNameUtil {
    public void getUserNames() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT usrname FROM names");

        //…      }
}
```

```
class BookUtil {
    public void getUserNames() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT book FROM books");

        //…      }
}
```

```
class StoreUtil {
    public void getStores() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT store FROM stores");

        //…      }
}
```

```
class DBConnection {
    public static Connection getConnection() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.
                getConnection("BookDatabase");
        return conn;
    }
}
```

Avoid duplicate code by abstraction



Keep It Simple Stupid (KISS)

Keep It Simple Stupid (KISS)

- simplicity should be a primary goal in design and development, advocating for straightforward and uncomplicated solutions over unnecessarily complex ones
- 簡潔是軟體系統設計的重要目標，應避免引入不必要的複雜性

Benefits of KISS principle

- **Ease of Understanding**: Simple solutions are easier to understand for both developers and end-users.
- **Reduced Errors**: Complex systems are more prone to mistakes due to their intricate nature.
- **Improved Maintenance**: Simple systems are easier to maintain over time.
- **Faster Development**: Developers can focus on implementing essential features without getting bogged down by unnecessary complexity
- **Enhanced Scalability**: When new requirements arise or user demands shift, simple systems can be modified or extended with less effort compared to complex architectures.

From: https://www.geeksforgeeks.org/kiss-principle-in-software-development/

Apply KISS Principle

- Identify Core Objectives
  - Identify the essential goals and requirements.
- Focus on Essentials
  - Prioritize essential features or components necessary to achieve objectives.
- Simplify Design and Workflow
  - Eliminate redundant steps or unnecessary complications.
- Prioritize Clarity and Understandability
  - Ensure that solutions are clear and easy to understand for all stakeholders.
  - Use simple and straightforward language in documentation and communication.

From: https://www.geeksforgeeks.org/kiss-principle-in-software-development/

Apply KISS Principle

- Iterate and Refine
  - Continuously review and refine solutions to simplify further.
- Use Simple Tools and Techniques
  - Avoid unnecessary complexity in tooling and technology choices.

From: https://www.geeksforgeeks.org/kiss-principle-in-software-development/

References

- Refactoring for Software Design Smells Managing Technical Debt.(設計重購) G. Suryanarayana, G. Samarthyam, T. Sharma. (2014)
- Head First Software Development. Dan Pilone, Russ Miles. (2007)
- “Clean Architecture: A Craftsman's Guide to Software Structure and Design,” Robert C. Martin, Prentice Hall, 2017.
- “Clean Code: A Handbook of Agile Software Craftsmanship,” Robert C. Martin, Prentice Hall, 2008.