Design Principles
&
Design Patterns

Design principles

- Symptoms of Rotting Design
  - Rigitity, Fragility, Immobility, Viscosity
- Changing Requirements & Dependency management
- Principles of Object Oriented Class Design
  - The Open Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

Patterns of Object Oriented Architecture

Design principles

- Symptoms of Rotting Design
  - Rigitity, Fragility, Immobility, Viscosity
- Changing Requirements & Dependency management
- Principles of Object Oriented Class Design
  - The Open Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

Symptoms of Rotting Design (1)

- Rigidity = software difficult to change
  - Every change → cascade of subsequent changes in dependent modules
    - 2 days of change ➔ multiweeks marathon of change in module after module
  - ➔ managers fear to allow engineers to fix non-critical problems
    - == design deficiency

Symptoms of Rotting Design (2)

- Fragility = a change → software to break in many places
  - Every fix ➔ introducing more problems than are solved
- Immobility = inability to reuse the software from other projects or other parts of the same project.
  - Reuse requires work and risk to separate the desirable parts

Symptoms of Rotting Design (3)

- Viscosity =
  - Viscosity of design = when engineers find more one way to make a change.
    - Some ways preserve the design.
    - If these ways are harder to employ. Then the viscosity ot design is high
  - Viscosity of environment = when the development environment is slow and inefficient.
    - Ex : Compile times are long → enginners make change that don't need large recompiles, regraless if the design is preserved

Design principles

- Symptoms of Rotting Design
  - Rigitity, Fragility, Immobility, Viscosity
- Changing Requirements & Dependency management
- Principles of Object Oriented Class Design
  - The Open Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

Changing Requirements & Dependency management

- Requirements will change in ways that the initial design did not antcipate
  - → thought the change works, it violates the orginal design
  - ➔ our designs must be resilient to such changes and protect them from rotting
- What kind of changes cause designs to rot?
  - Changes that introduce new and unplanned for dependencies → degrading of dependency architechure
  - ➔ creation of dependency firewalls that avoid the dependency propogation between modules

Design principles

- Symptoms of Rotting Design
  - Rigitity, Fragility, Immobility, Viscosity
- Changing Requirements & Dependency management
- Principles of Object Oriented Class Design
  - The Open Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

The Open Closed Principle (OCP)

« A module should be open for extension but closed for modification »

- Changing what the modules do, without changing the source code of modules
- OCP Goal = creation of modules extensibles without being changed
  - → add new features to existing code only by adding new code
  - So, Design modules should *never change*. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.
- *Abstraction is the key to OCP =* Polymorphism

OCP & Polymorphism

- مثال = مسألة معالجة الصور من السنة الثالثة (عملي لغات البرمجة) :

اكتب برنامج لمعالجة الصور من أنواع مختلفة (gif, bmp, jpg)

هذا البرنامج يسمح بمعرفة عرض و ارتفاع الصورة علماً بأن هذه المعالجة تعتمد على نوع الصورة.

نريد في هذا البرنامج معالجة الصور بأسلوب متماثل و إمكانية إضافة أنواع جديدة من الصور

Solution (1)

```
public class Image
{
        private String type;
        public Image(String type) { this.type = type;}
        …….
        public int getWidth() {
                if (this.type.equals("jpg"))
                {// procesing to determine the width of jpg image}
                else if (this.type.equals("gif"))
                {// procesing to determine the width of gif image}
                else if (this.type.equals("bmp"))
                {// procesing to determine the width of bmp image}
        }
        …..
}
public class ImageProcessor
{
        public int getImageWidth(Image img) {return img.getWidth(); }
        public int getImageHeight(Image img) {return img.getHight(); }
        …..
}
```

Solution (2)

```
public class JpgImage
{
        public int getWidth() {…..}
        ….
}
public class GifImage
{       …}

public class ImageProcessor
{
        public int getImageWidth(Object img) {
                if (img instanceof JpgImage)
                        {return ((JpgImage) img).getWidth();}
                else if (img instanceof GifImage)
                        {return ((GifImage) img).getWidth();}
                else if (img instanceof BitmapImage)
                        {return ((BitmapImage) img).getWidth();}
                else throw new NotAnImageException();
        }
        .........
}
```

Solution (3) == OCP & Polymorphism

```
public interface Image
{
        public int getWidth();
        ….
}
public class GifImage implements Image
{
         public int getWidth() { //procesing to determine the width of gif image}
}
public class JpgImage implements Image
{
         public int getWidth() {//procesing to determine the width of jpg image}
}
-----------------------------------------------------------------------------------------------------------------
public class ImageProcessor
{
        public int getImageWidth(Image img)
        {
                return img.getWidth();
        }
}
```

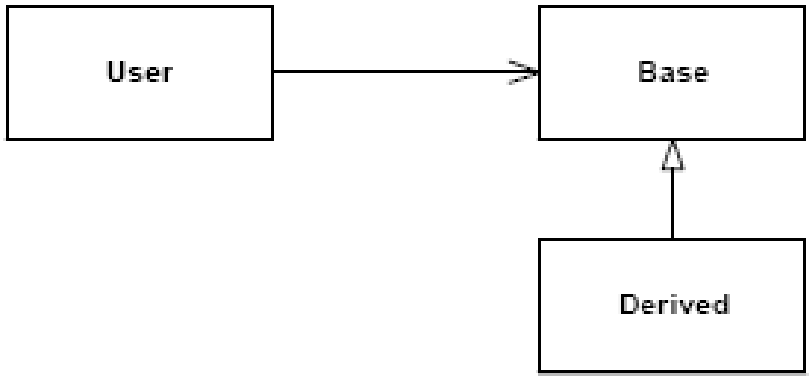
Design principles

- Symptoms of Rotting Design
  - Rigitity, Fragility, Immobility, Viscosity
- Changing Requirements & Dependency management
- Principles of Object Oriented Class Design
  - The Open Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

The Liskov Substitution Principle (LSP)

« Subclasses should be substituable for their base classes »

- FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.



The Liskov Substitution Principle (LSP)

- Example 1 : violation of LSP

```
void DrawShape(Shape s) {
        if (s instanceof Square)
                DrawSquare((Square)s);
        else if (s instanceof Circle)
                DrawCircle((Circle)s);
}
```

- *This code is not extensible because it must know about all sub-classes of Shape.*

The Liskov Substitution Principle (LSP)

- Example 2 : violation of LSP
  - If we pass for f() object of Square
  - Then, this square will be corrupted

```
class Rectangle {
        private double itsWidth;
        private double itsHeight;
        void SetWidth(double w) {itsWidth=w;}
        SetHeight(double h) {itsHeight=w;}
        GetHeight() const {return itsHeight;}
        GetWidth() const {return itsWidth;}
}
```

```
class Square extends Rectangle {
        Void SetWidth(double w) {
                Super.SetWidth(w); super.SetHeight(w);
        }
        Void SetHeight(double h) {
                Super.SetWidth(h); super.SetHeight(w);
        }
}
```

```
void g(Rectangle& r) {
  r.SetWidth(5)
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight())
== 20);
                                    }
```

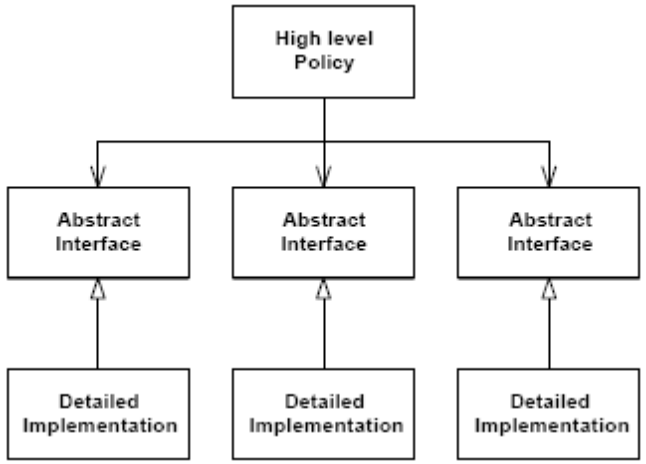
Design principles

- Symptoms of Rotting Design
  - Rigitity, Fragility, Immobility, Viscosity
- Changing Requirements & Dependency management
- Principles of Object Oriented Class Design
  - The Open Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

The Dependency Inversion Principle (DIP)

«Depend upon Abstractions. Do not depend upon concretions.»

- OCP = goal of the OO Architecture
- DIP = mechanism or strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes.
  - **Depending upon Abstractions :**
    - Why = because the concrete things change a lot. So, the abstractions must be the hinge points = where the design can be extended.



The Dependency Inversion Principle (DIP)

- DIP
  - **Depending upon Abstractions**
  - **Mitigating Forces =** DIP prevent you from depending on volatile modules
    - DIP make assumption that anything cocrete is volatile
    - Exception : *string.h* is very concrete but not volatile
      - but depending on string.h is bad when the requirements are to change to UNICODE characters.
      - >>> So, No volatility != abstract interfaces
  - **Object Creation** = where the designes create instances (where depending on concrete classes)

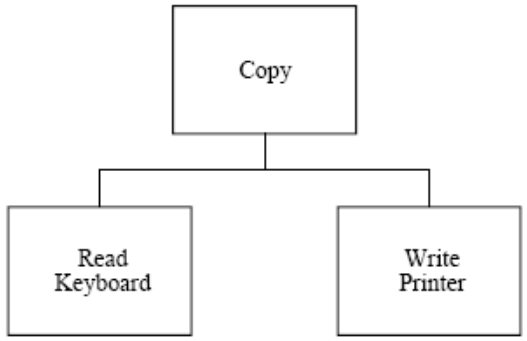
The Dependency Inversion Principle (DIP)

- Example : Copy Program
  - The copy program depends on WritePrinter >>> So, We can not use it in new context
  - copy module is not reuseable because it depends on lower level modules

```
void Copy() {
        int c;
        while ((c = ReadKeyboard()) != EOF)
                WritePrinter(c);
}
```

- Problem : new devices >> we have to modify the copy module

```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev) {
        int c;
        while ((c = ReadKeyboard()) != EOF)
                if (dev == printer)
                        WritePrinter(c);
                else
                        WriteDisk(c);
}
```



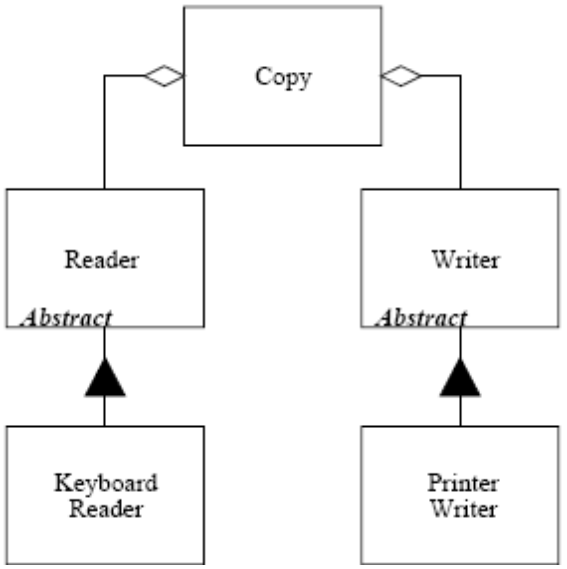
The Dependency Inversion Principle (DIP)

- Example : Copy Program Solution
  - Copy module is reuseable >>> because it does not depend on devices
  - Copy module depends on **abstractions** of devices

```
class Reader {
        public int Read();
}
```

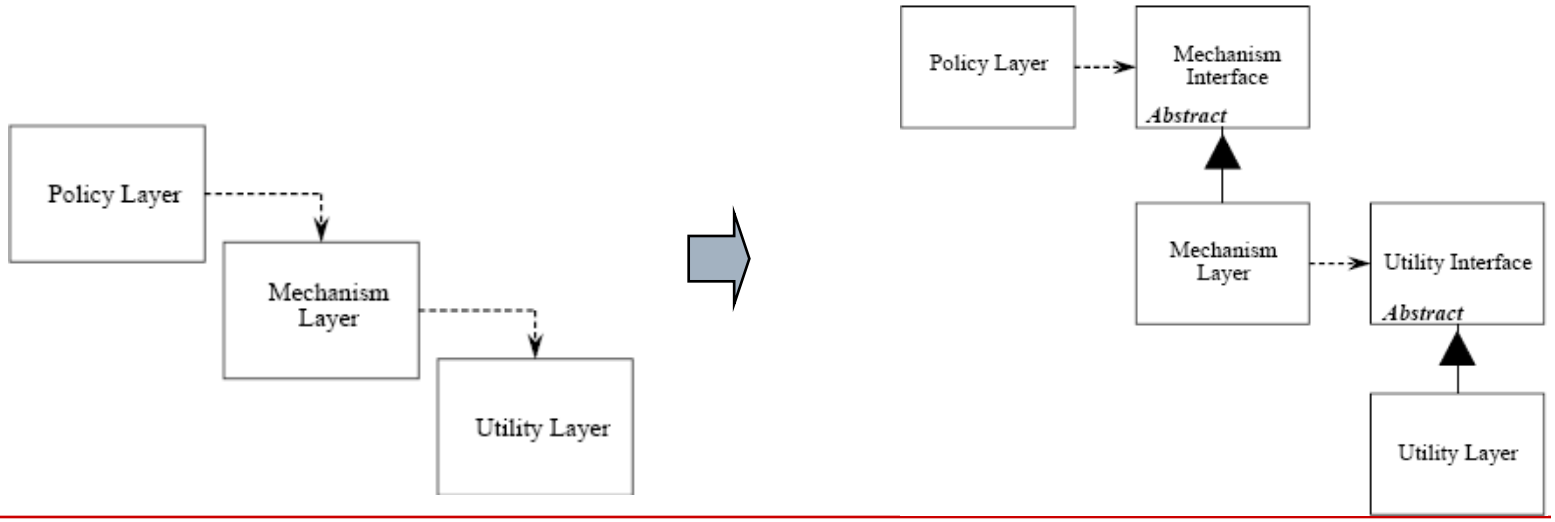
```
class Writer {
        public void Write(char);
}
```

```
void Copy(Reader r, Writer w) {
        int c;
        while((c=r.Read()) != EOF)
                w.Write(c);
}
```



The Dependency Inversion Principle (DIP)

- HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.
- ASTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.
- THE HIGHT LEVEL MODULES ARE THE REUSABLE PARTS
- the high level modules must contain the important policy decisions and business models of an application.



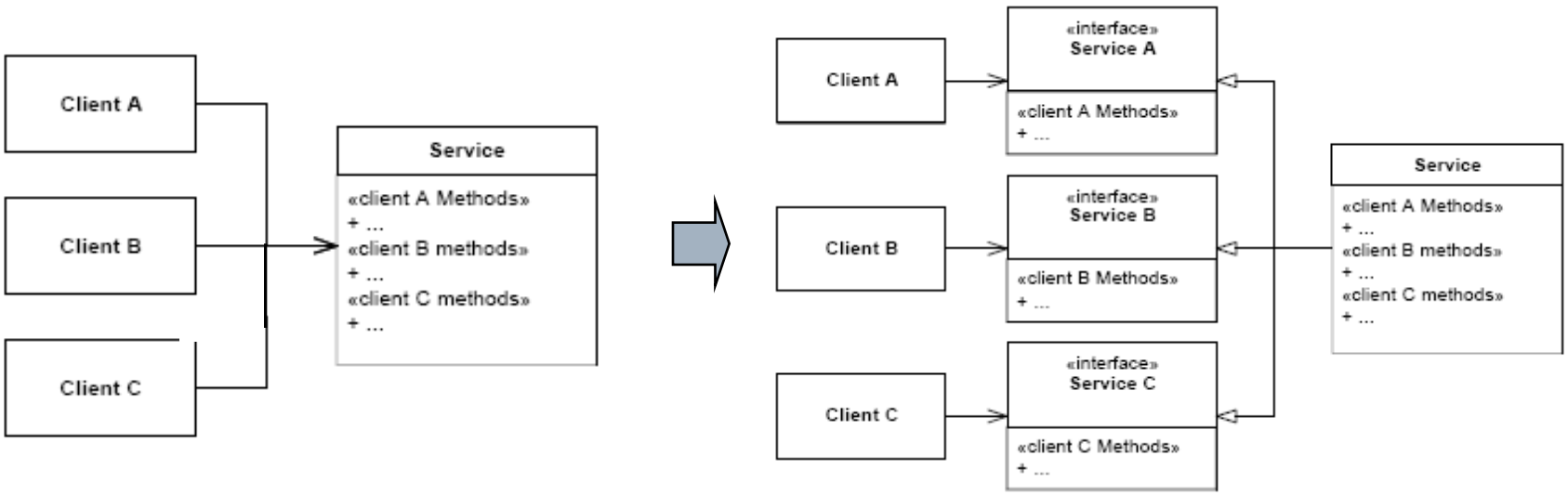
Design principles

- Symptoms of Rotting Design
  - Rigitity, Fragility, Immobility, Viscosity
- Changing Requirements & Dependency management
- Principles of Object Oriented Class Design
  - The Open Closed Principle (OCP)
  - The Liskov Substitution Principle (LSP)
  - The Dependency Inversion Principle (DIP)
  - The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

« Many client specific interfaces are better than one general purpose interface »

- Classes that have "fat" interfaces are classes whose interfaces are not cohesive
- **« ISP = CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE »**



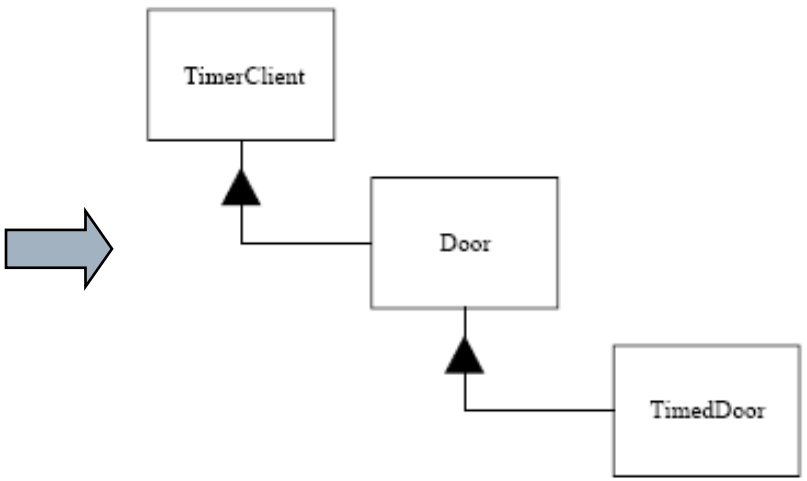
The Interface Segregation Principle (ISP)

- Example (the example):
  - **Pb 1** = Door depends on TimerClient (Not all Doors need timing and they don't use TimeOut())
  - **Pb 2 =** new interfaces >>> changes on the base classe >>> recompile for all clients

```
class Door {
        void Lock();
        void Unlock();
        boolean IsDoorOpen();
}
```

```
class Timer {
        void Regsiter(int timeout, TimerClient client);
}
```

```
class TimerClient {
        void TimeOut();
}
```



The Interface Segregation Principle (ISP)

- Example (the Problem = timeout for each user):
  - **We must change the TimerClient class & all his users**
  - **But, also the Door class and his users**

```
class TimerClient {
        void TimeOut(int timeOutId);
}
```

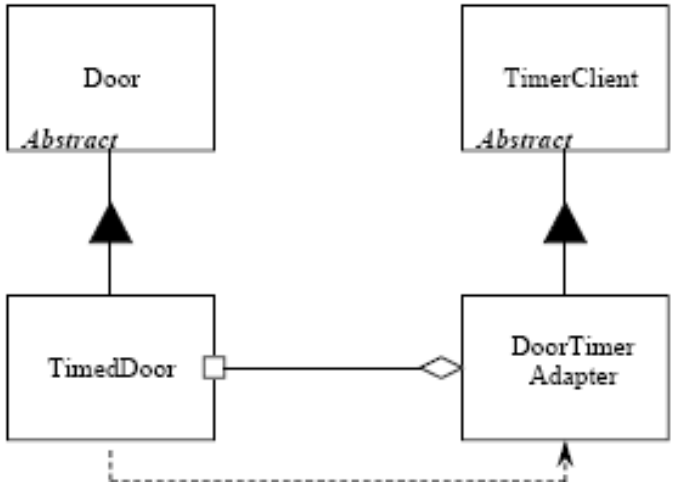
```
class Timer {
        void Regsiter(int timeout,
                      int timeOutId,
                      TimerClient client);
}
```

The Interface Segregation Principle (ISP)

- Example (solution 1 = separation through **delegation**):

```
class TimedDoor extends Door {
        public void DoorTimeOut(int timeOutId);
}
```

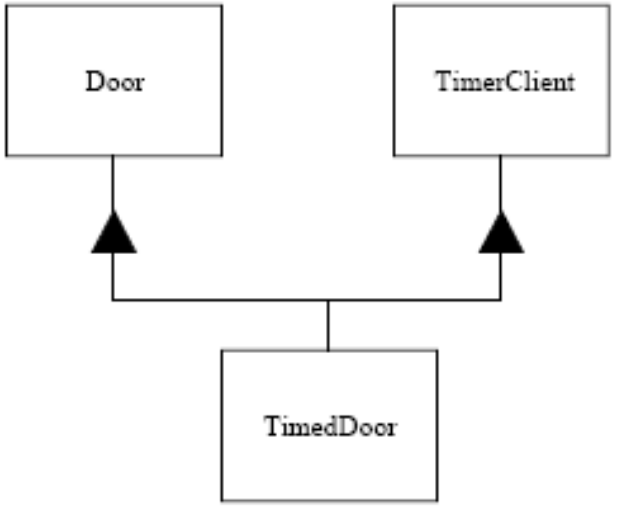
```
class DoorTimerAdapter extends TimerClient {
        private TimedDoor itsTimedDoor;

        DoorTimerAdapter(TimedDoor theDoor)
        {itsTimedDoor = theDoor;}

        void TimeOut(int timeOutId)
        {itsTimedDoor.DoorTimeOut(timeOutId);}
}
```



The Interface Segregation Principle (ISP)

- Example (solution 2 = separation through Multiple inheritance):

```
class TimedDoor extends Door, TimerClient
        {
    public void TimeOut(int timeOutId);
}
```



Patterns of Object Oriented Architecture

الـ Design Pattern هو :

- Name : نستخدمه كمفردات تصميمية للتواصل
- Problem : متى يمكن استخدام هذا الـ Pattern
  - مسائل محددة حيث بنية الأغراض و الصفوف تمثل تصميماً غير مرن
  - قائمة الشروط اللازمة لتطبيق هذا الـ pattern
- Solution : توصيف مجرد للمسألة التصميمية و كيفية ترتيب عناصر التصميم (صفوف و أغراض) و كيفية تعاونهم و مسؤولياتهم.
- Consequences : تناقش التنفيذ و لغات البرمجة و مرونة النظام و قابليته للتوسع و محموليته.

الـ Design Paterns تسمح بـ :

- إعادة استخدام التصميمات و المعماريات الناجحة (رأسملة الخبرة في التصميم).
- إيجاد التصميم الأفضل بسرعة (قائمة و دليل للحلول)
- تحسين من توثيق و صيانة الأنظمة من خلال تقديمها توصيف للصفوف و التفاعل بين أغراض النظام (مفردات مشتركة للتواصل).
- مستوى تجريد عالي يسمح ببناء برمجيات ذات جودة أعلى

Patterns of Object Oriented Architecture

تطبيق الـ Design Paterns عند التصميم =

- إيجاد الأغراض المناسبة و اختيار حجم الأغراض المناسب
- توصيف واجهات الأغراض
- توصيف تنفيذ الأغراض
- تحديد هرمية الصفوف
- بناء العلاقات الأساسية بين الصفوف
- إعادة الاستخدام بشكل أفضل :
  - Composition <> Inhertance
  - Delegation
- Compiled-Time <> Run-Time Structures
- تصميم قابل للتطور

Classfication of Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Classfication of Design Patterns

- Scope :
  - Class patterns : تتعامل مع العلاقات بين الصفوف و صفوفهم الأبناء (علاقات مبنية بالوراثة و بالتالي ثابتة عند زمن الترجمة)
  - Object Patterns : تتعامل مع العلاقات بين الأغراض (علاقات يمكن أن تغيير مع الزمن و بالتالي أكثر ديناميكية)
- Creational Patterns : تهتم بعملية إنشاء الأغراض
  - Class creational patterns : تؤجل إنشاء الأغراض إلى الصفوف الأبناء
  - Object creational patterns : توكل هذه العملية لغرض آخر
- Structural Patterns : تهتم بتجميع الصفوف و الأغراض لتشكيل بنى أوسع
  - Class structural patterns : تستخدم الوراثة لتجميع الصفوف
  - Object structural patterns : تستخدم الـ composition لتجميع الأغراض
- Behavioural Patterns: توصف طرق تفاعل الصفوف و الأغراض و توزيع المسؤولية فيما بينهم
  - Class behavioral patterns : تستخدم الوراثة لتوزيع المسؤولية
  - Object behavioral patterns : تجمع الأغراض و تصف كيفية تعاونهم لإنجاز مهمة

حل المسائل التصميمية باستخدام الـ Design patterns (١)

- Finding Appropriate Objects :
  - تقسيم النظام إلى أغراض يراعي العديد من العوامل : Encapsulation, Granularity, Dependency, Flexibility, Performance, evolution, reusability, ...
  - نموذج التحليل يحدد العديد من أغراض العالم الحقيقي
  - الـ Design patterns تساعد بتحديد أغراض ليس لها مقابل في العالم الحقيقي (ذات تجريد قليل الوضوح) مثل الـ Process, Algorithm .-.-.-. (Strategy Pattern)
- Determinig Object Granularity :
  - الأغراض في نظام يمكن أن تتغير بحجها و عددها .-.- (Facade & Flyweight Patterns)
- Specifying Object Interfaces :
  - في بعض الأنظمة نطلب من الصفوف امتلاك واجهات متماثلة أو نفرض محددات عليها .-.- (Decorator, Poxy)

حل المسائل التصميمية باستخدام الـ Design patterns (٢)

- Specifying Object Implementaion
  - Class Object <> Type Object (Interface)
  - Inheritance reuse <> Object composition reuse
  - Program to an interface, not an Implementation
    - الزبائن غير واعين للأنماط المحددة التي يستخدمونها
    - الزبائن غير واعين للصفوف الحقيقية التي تنفذ الواجهات
  - = Class inheritance
    - Breaks encapsulation == White-box reuse
    - Changes in parent classes go to subclasses
    - الاعتمادية بين التنفيذ قوية و بالتالي يمكن أن نحتاج لتعديل الصفوف الآباء
  - Object composition
    - no break to encapsulation == Black-box reuse
    - يتم تعريفه عند الـ Run-time (احترام لواجهات الأغراض الأخرى) + إمكانية الاستبدال أثناء الـ run-time بغرض من نفس النمط
    - الاعتمادية بين التنفيذ ضعيفة (لأننا نستخدم الواجهات لتجميع الأغراض)

حل المسائل التصميمية باستخدام الـ Design patterns (٣)

- Specifying Object Implementaion
  - Delegation <> Inheritance
    - Inheritance = تؤجل الطلبات إلى الصفوف الآباء + إمكانية الإشارة إلى الغرض المستقبل للرسالة من خلال this
    - Delegation = توكل العمليات إلى الأغراض delegate + إمكانية الإشارة إلى الغرض المستقبل للرسالة من خلال تمرير نفسه إلى الغرض delegate
- Desigining for change
  - الأساس = توقع المتطلبات الجديدة و التغيرات في المتطلبات الموجودة
  - أسباب إعادة تصميم نظام ما :
    - Creating an object by specifying a class explicitly
    - Dependence on hardware and software platform
    - Dependence on object representations or implementations
    - Algorithmic dependencies
    - Tight coupling
    - Extending functionality by subclassing