Design Patterns

أنيس حكمت أبوحميد

Email Github Stackoverflow Slideshare Linked-in

الفهرس

الفهرس

الفهرس

يا أخي، من اللطائف الجميلة والمعاني العظيمة في لغتنا هو معنى الإحسان! والإحسان -في أحد معانيه- أن تعبد الله كأنك تراه، فإن لم تكن تراه، فإنه يراك. وفي حق المخلوقين: بذل النفع الديني والدنيوي لهم، لذلك كن محسنًا دومًا!

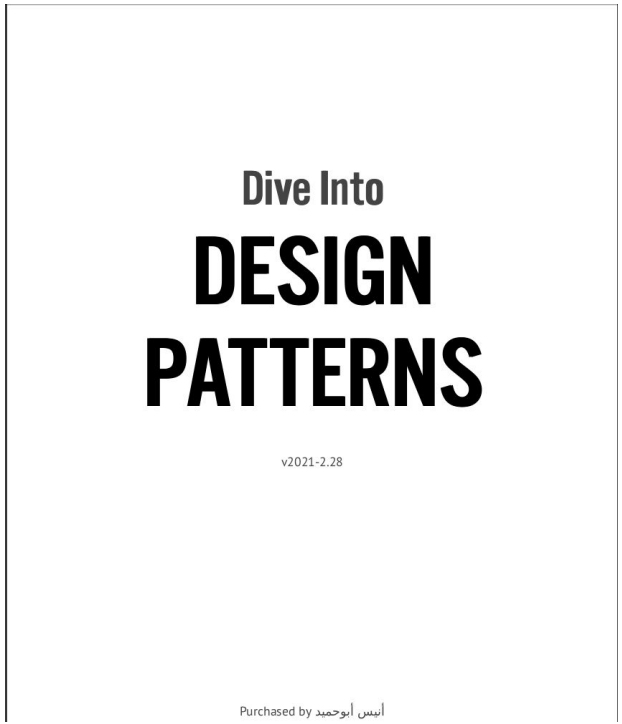
المقدمة

بسم الله الرحمن الرحيم

الحمد لله ربّ العالمين، يُحب من دعاه خفياً، ويُجيب من ناداه نجيّاً، ويزيدُ من كان منه حييّاً، ويكرم من كان له وفيّاً، ويهدي من كان صادق الوعد رضيّاً، الحمد لله ربّ العالمين.

يُقال، أن تنفيذ التعليمات البرمجية لحل مشكلة معينة هي أسهل خطوة في التطبيق!، فالأصل أن كتابة التعليمات البرمجية ما هي إلا تنفيذ وتطبيق لما يدور في رأسك بعد التفكير بآلية الحل المناسبة للمشكلة التي تصادفك!، لكن، ما هي الحلول الممكنة لأشهر المشاكل التي يمكن أن تصادف أي مبرمج؟!، وما هي مبادئ التصميم؟!، ما هي مفاهيم ال OOP الأساسية، هل هي صعبة حقا؟!، كيف يمكنني التفريق بين الحل الجيد والحل السيء؟!، كل هذه الأسئلة وأكثر، سنجد الإجابة عنها في هذه الشرائح بإذن الله تعالى، فاستعد!

ملاحظة 1



المرجع الأساسي لهذا الشرح كتاب Dive Into Design Patterns لمؤلفه Alexander Shvets

وقد قمت باقتباس واستخدام العديد من الصور والأمثلة منه، وقمت بالتقيد بما ذكر في الكتاب قدر ما أمكن، وذلك للوصول إلى أدق ما يمكن من الشرح، ولكن مع ذلك، فقد خرجت في كثير من الأحيان أو الأماكن عما ذكر وذلك لإثراء الشرح على نقطة معينة أراها مهمة أو إضافة العديد من التعاريف أو الشروحات الخارجية لإثراء المحتوى...كما تحتوي بعض التعريفات على صياغة تشرح طريقة العمل أكثر من الاهتمام بالمعنى الحرفي للتعريف الأصلي...

كما أنصح الجميع بشراء هذا الكتاب القيم...

ملاحظة 2

هناك العديد من الأمثلة المستخدمة في هذه الدروس، يمكنك الإطلاع عليها مباشرة من خلال رابط هذه الدورة على Github.

طريقة الشرح ستعتمد على مجموعة من الصور مرقمة بأرقام تسهل الشرح، وبجانبها نص مختصر صغير يشرح فكرة الصورة، ما عليك فعله هو محاولة فهم الصورة قبل الشروع بتنفيذ الفكرة، أو الإطلاع على المثال ومحاولة تطبيق فكرة شبيه في المثال...، هذا الشكل من الشرح ستجده عند البدء في شرح مواضيع ال Design Pattern.

تعتمد هذه الشرائح والأمثلة على لغة ال PHP بشكل عام في كل الأمثلة، لكن قد تجد بعض الشيفرات البرمجية المكتوبة بلغات أخرى مثل الجافا، والفكرة بسيطة...فالمطلوب هنا هو المفهوم، وتطبيق المفهوم لاحقا بأي لغة لن يكون عائقا...

ركائز ال OOP

قبل أن نبدأ بالحديث عن ال Design patterns، ننوه ونذكر بأن البرمجة الكائنية ترتكز على أربعة ركائز وهي:

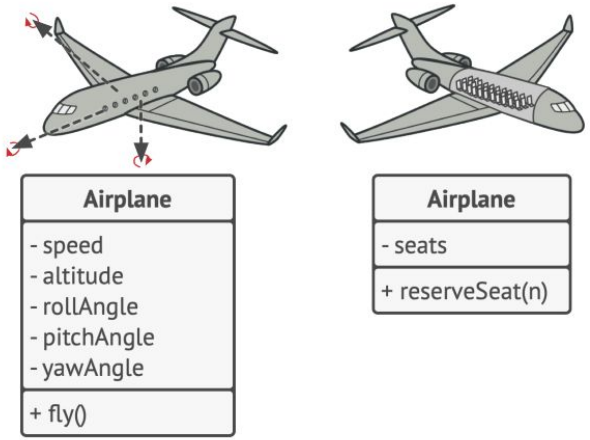
1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

هذه الركائز الأربعة تشكل البرمجة كائنية التوجه، وكلما كان الفهم لهذه الركائز أكثر عمقا في مخيلة المبرمج كانت الشيفرة البرمجية وفهمه لل Design patterns أفضل وأقوى...، وباعتبار أن البرمجة كائنية التوجه متطلب سابق سأقوم بالمرور على بعض العناوين وشرح هذه الركائز بشكل مختصر ومبسط...

Abstraction

ال Abstraction هو نموذج (model) من الحياة الواقعية لكائن ما (object) تقتصر وظيفته على سياق محدد ووظيفة معينة (specific context) بحيث يتعامل هذا ال Abstraction مع جميع التفاصيل والعناصر المتعلقة بالوظيفة المراد إنجازها بدقة عالية مع إهمال ما تبقى، وأشهر ما يستخدم هذا المفهوم من keywords ال Abstract وال Interface، باختصار هو وسيلة لعرض السمة الأساسية وإخفاء ما دون ذلك!

مثال:
في هذا المثال، ترى أن الطائرة تمثل النموذج الذي نرغب في تنفيذ بعض المهام لها، لكن لكل نموذج منها وظيفة محددة تختلف عن الأخرى، فنموذج الطائرة الأولى يهتم بالمقاعد (مثلا حجوزات المقاعد في الطائرة) بينما النموذج الثانية يهتم بحركة الطائرة واتجاهاتها...، وهنا يظهر جمال هذا المبدأ، فإننا أهملنا ما في الطائرة من خصائص كنظام الطيران والتحكم واهتممنا بال classes الخاصة بنا وبما تحتاجه فقط..
لنشاهد مثالا برمجيا في الشريحة التالية...



Abstraction

```php
<?php
abstract class AbstractClass
{
    // Force Extending class to define this method
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);

    // Common method
    public function printOut() {
        print $this->getValue() . "\n";
    }
}

class ConcreteClass1 extends AbstractClass
{
    protected function getValue() {
        return "ConcreteClass1";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass1";
    }
}

class ConcreteClass2 extends AbstractClass
{
    public function getValue() {
        return "ConcreteClass2";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass2";
    }
}

$class1 = new ConcreteClass1;
$class1->printOut();
echo $class1->prefixValue('FOO_') ."\n";

$class2 = new ConcreteClass2;
$class2->printOut();
echo $class2->prefixValue('FOO_') ."\n";
?>
```

Open Example

Encapsulation

ال Encapsulation تعني قدرة ال object على إخفاء بعض الأجزاء الخاصة به (state and behavior) عن ال objects الأخرى، وكشف فقط واجهة محدودة الخيارات لبقية ال objects.
وهذا يعني أن ال Encapsulation تقوم بجعل شيء معين private مع القدرة إلى الوصول إليه من خلال ال methods الموجودة داخل هذا ال class فقط، أو من خلال subclass في حال استخدام ال protected بدلا من ال private.

تعد ال interface/abstract في معظم لغات البرمجة إحدى التطبيقات الفعلية لل Encapsulation وال Abstraction، وفي لغات البرمجة الحديثة، معظمها يعتمد ال keyword لتمثيل ال interface، وال interface هي إحدى تطبيقات ال Encapsulation...، وبهذا فإن العالم الخارجي يمكنه الوصول إلى ما هو موجود داخل ال interface وكتابة الشيفرة التي تناسبه دون القدرة على تغيير ال interface أو ما تخفيه من flow...

لو رجعنا إلى مثال الطائرة، ورغبنا في بناء مكان للتحكم بحركة وطريقة تنقل الطائرة، فيمكن بناء interface، هذه ال interface قد تحتوي بداخلها fly, origin and destination، وهذه يمكن مشاركتها مع طائرة الركاب، والهيلوكبتر، وكل طائرة منهم تستخدم نفس ال interface لكن لكل واحدة منهم التنفيذ والتطبيق الخاص بها...

من الأمثلة البسيطة التي نستخدمها بشكل متكرر على مفهوم ال Encapsulation ال setter وال getter...(الشريحة التالية)

ملاحظة: البيانات المخزنة داخل ال objects يشار لها بال state، ويشار لل methods ب behavior.

Encapsulation

```java
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

Inheritance

الوراثة واحدة من أهم المفاهيم في عالم البرمجة، فمن خلالها يمكن تقليص حجم الشيفرة البرمجية ومنع أي تكرار أو نسخ غير لازم!، لأن الوراثة هي القدرة على أخذ جميع ال behavior وال state الموجودة في ال super class والمسموح لل sub class من الوصول إليها، وعادة ما تستخدم الوراثة للاستفادة مما هو موجود في ال super class مع وجود بعض ال method الإضافية الخاصة بال subclass، وعادة ما تسمح لغات البرمجة بعملية وراثة واحدة مع implement لأكثر من interface (one extend, multiple interfaces) -عادة وليست كلها-، وأحد أهم الفروق بين الوراثة وال interface هي أن ال interface بوجودها يلزم كتابة جميع ال method الموجودة بداخلها، بينما الوراثة لا داعي لذلك، ويمكن إعادة استخدام ال method مباشرة دون الحاجة لإعادة كتابتها بال sub class...

مثال: (الشريحة التالية)

Inheritance

```php
<?php

class Foo
{
    public function printItem($string)
    {
        echo 'Foo: ' . $string . PHP_EOL;
    }

    public function printPHP()
    {
        echo 'PHP is great.' . PHP_EOL;
    }
}

class Bar extends Foo
{
    public function printItem($string)
    {
        echo 'Bar: ' . $string . PHP_EOL;
    }

    public function customSubClass($string)
    {
        echo 'SubClassPrint: ' . $string . PHP_EOL;
    }
}

$foo = new Foo();
$bar = new Bar();
$foo->printItem('baz'); // Output: 'Foo: baz'
$foo->printPHP();       // Output: 'PHP is great'
$bar->printItem('baz'); // Output: 'Bar: baz'
$bar->printPHP();       // Output: 'PHP is great'
$bar->customSubClass("Subclass Inherit"); // Output: 'SubClassPrint: Subclass Inherit'
```

Polymorphism

ال Polymorphism هو مفهوم في ال OOP يشير إلى استخدام ال methods الموجودة بال super class في ال sub class لتنفيذ مهام مختلفة باستخدام نفس الإسم الموروث، ومع أن هذا المفهوم نستخدمه كثيرا أثناء البرمجة، إلا أننا نخاف عندما نسمع بكلمة Polymorphism!، لكن لو علمت أن هذا المفهوم هو ما تقوم بفعله مرارا وتكرارا لسهل عليك الأمر...، ويمكن القول بطريقة أخرى أن ال method الموجودة ب classes مختلفة والتي تقوم بنفس الوظيفة المراد إنجازها يجب أن يكون لها نفس الإسم، فمثلا إذا كان لدي class يمثل جميع الحيوانات (animal)، وفيه دالة لصوت الحيوان (sound)، ولدي class لكل من ال cat وال dog وكلاهما يرثون ال animal فإن دالة ال sound ستورث لكل من ال cat وال dog، بنفس الإسم، وهنا يمكن كتابة "ميعاو" كصوت لل cat و "عو عو" كصوت لل dog...ودون أن يتأثر أي منهم بالآخر، وتشير بعض التعريفات إلى أن Polymorphism يعني القدرة على معرفة إلى من تعود الدالة بالأساس وما يجب تنفيذه باختلاف المنفذ للدالة.

شاهد المثال في الشريحة التالية...

Polymorphism

```java
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

فلْتعلمْ أن صاحب الحق لا يهمه ولا يرده لوم الآخرين، ولا يهتم لرضاهم أو سخطهم، بل كل همه أن يبقى صابرًا على الحق متوكلًا على اللّٰه!

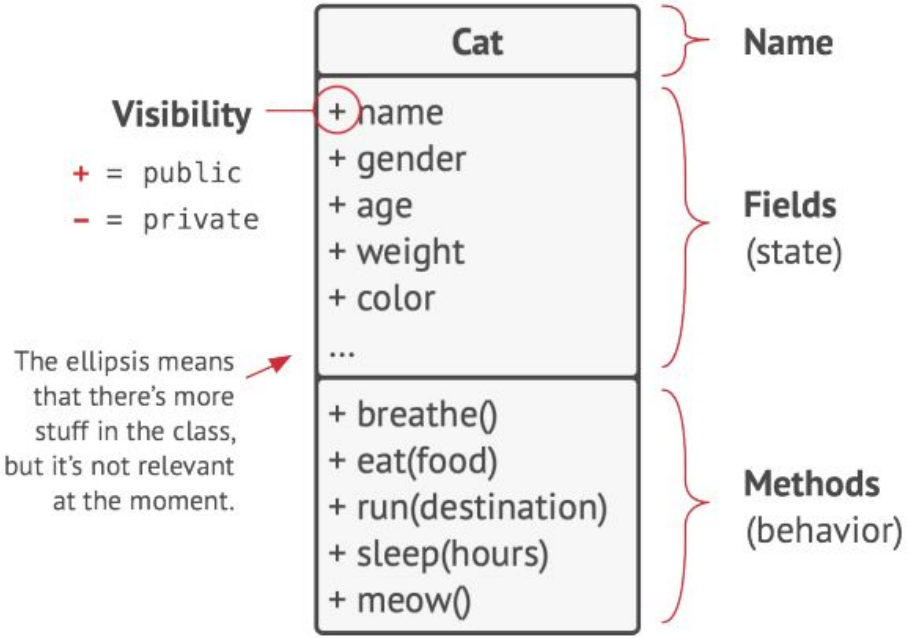
UML

ال UML هي اختصار ل Unified Modeling Language، وهو عبارة عن مخطط أو نموذج مرئي يهدف إلى تحسين وتطوير وتسريع الفهم لهيكلية النظام بطريقة توفر تصور لتصميم النظام ومساره والعلاقة بين مكوناته.

لاستخدام ال UML مزايا عديدة، فمن خلاله يمكن شرح مفهوم ما بدون الحاجة لكتابة شيفرة برمجية ودون الحاجة لكتابة مطولة وبطريقة يفهمها الجميع، وهذا يجعل من قدرة ال UML على التجاوب مع التغييرات وتقديم نماذج متعددة أمرا أسهل.

وأظن أنك متعجب، لماذا تطرقنا لهذا الموضوع؟ والجواب بكل بساطة، أنك ستشاهد الكثير من النماذج التي تعتمد على ال UML في الشروحات القادمة، لذلك كن مستعدا، لكن لا تقلق، فالموضوع سهل، وسأقوم بتلخيص أبرز المفاهيم في الشرائح التالية والتي سنستخدمها في هذا الشرح.

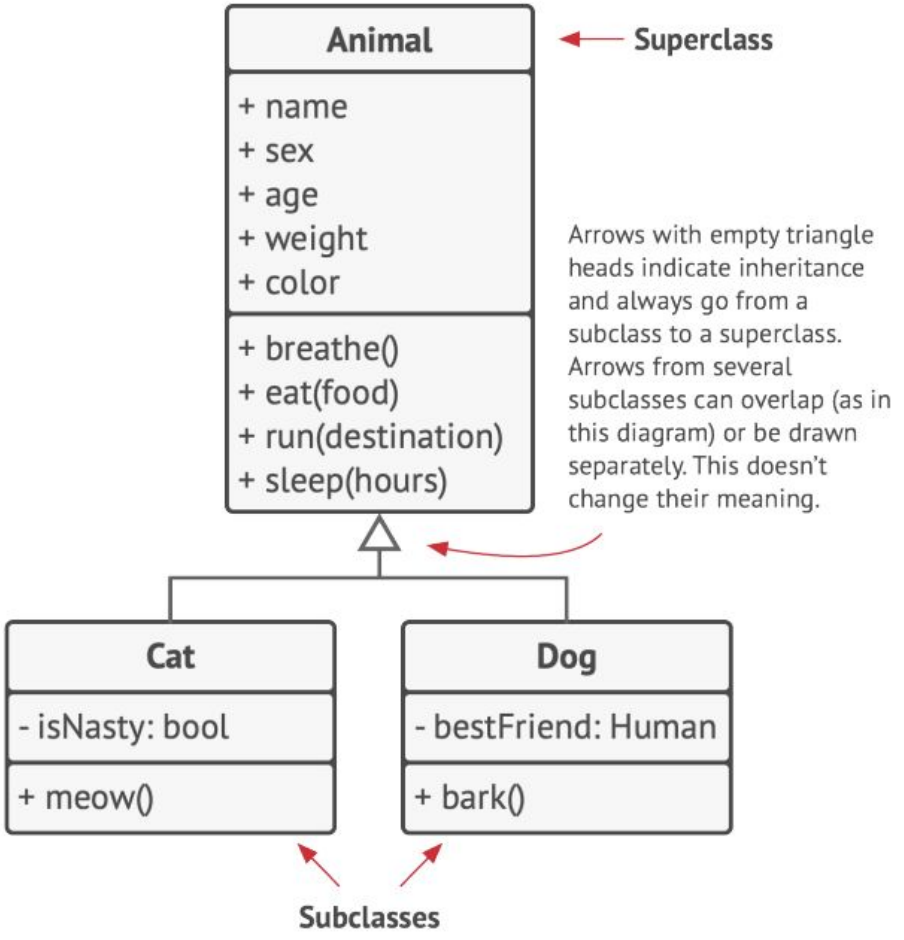
UML



تمثل هذه الصورة ال UML Class Diagram، وكما ترى، فهو يُقسم إلى 3 أجزاء، الأول هو اسم ال class، وفي هذه الصورة هو cat، والجزء الثاني هي ال state والثالث ال behavior، وتشير ال + إلى ال public وال - إلى ال private وال ال # إلى ال protected كما تشير النقاط أسفل ال state إلى وجود state أخرى لكن ليس من المهم ذكرها الآن...

مثال

Circle
- x-coord - y-coord # radius
+ findArea() + findCircumference() + scale()

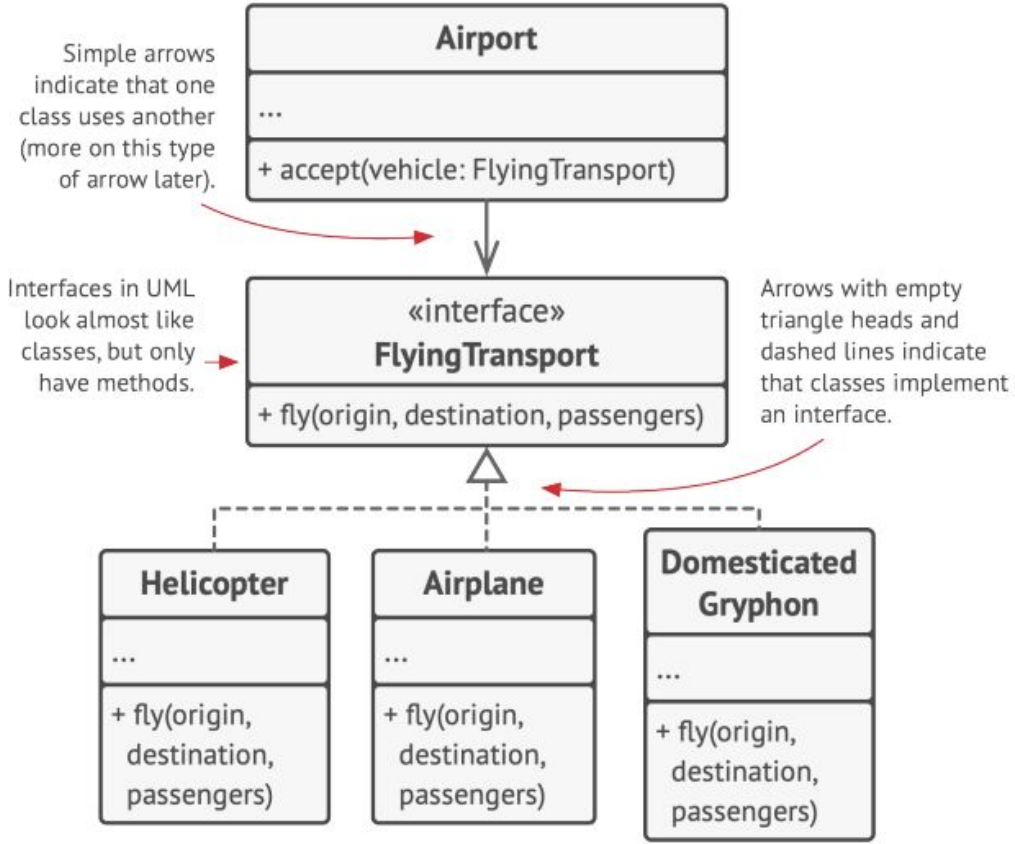


UML diagram of a class hierarchy. All classes in this diagram are part of the `Animal` class hierarchy.

UML

هذا ال diagram يمثل التسلسل الهرمي لل class، وفيه يظهر ال SuperClass وجميع ال SubClass الخاضعين له، وما ينطبق على ما ذكرناه سابقا ينطبق هنا، لكن بإضافة مهمة وهي السهم، هذا السهم ذو الرأس الفارغ يشير لوجود علاقة inheritance، وفي هذا المثال فإن ال cat وال dog عبارة عن classes تنتمي إلى مجموعة الحيوانات، وهذا كلام منطقي، وجراء هذا الانتماء فإن هناك مجموعة من الخصائص المشتركة والتي سيرثها كلاهما كالإسم والعمر والجنس...إلى آخره

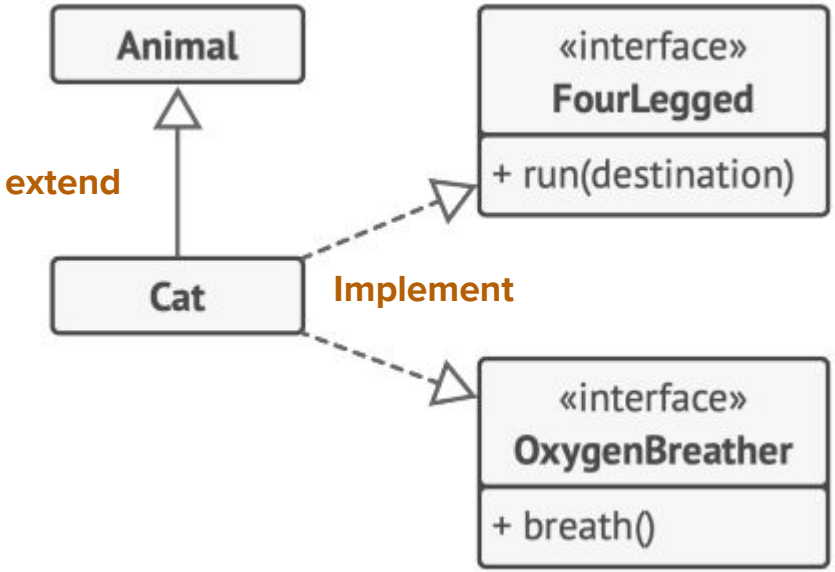
UML



هذا ال Diagram يمثل مجموعة من ال classes التي تقوم بعمل implement ل interface.
ال interface بال UML يتم بنائها كما يتم بناء ال class، لكن باختلاف بسيط وهو أن ال interface لا تحتوي State، فقط behavior، كما يشار للعلاقة بين ال class التي قامت بعمل implement لل interface بسهم متقطع برأس مثلث فارغ.
أما السهم الموجود بين ال Airport وال Flying فهذا يشير إلى وجود علاقة Association والتي تشكل relation تجعل من ال objects قادرة على التفاعل فيما بينها وبالتالي استخدام كل منها للآخر...

UML

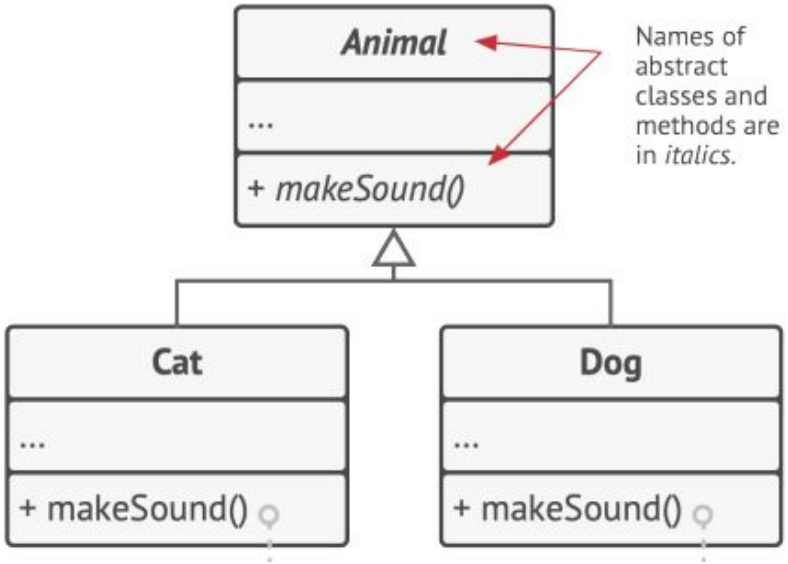
مثال على ما سبق:

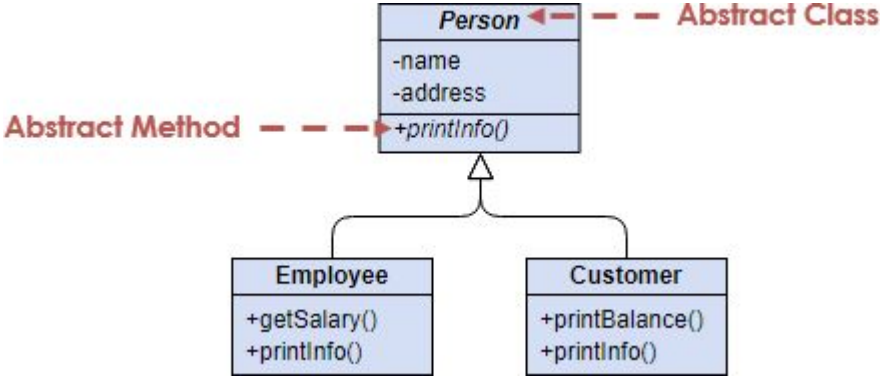


تلاحظ في هذا المثال أن ال Animal يمثل Superclass ورثه ال Cat، وال Cat قامت بعمل implement ل two interfaces.

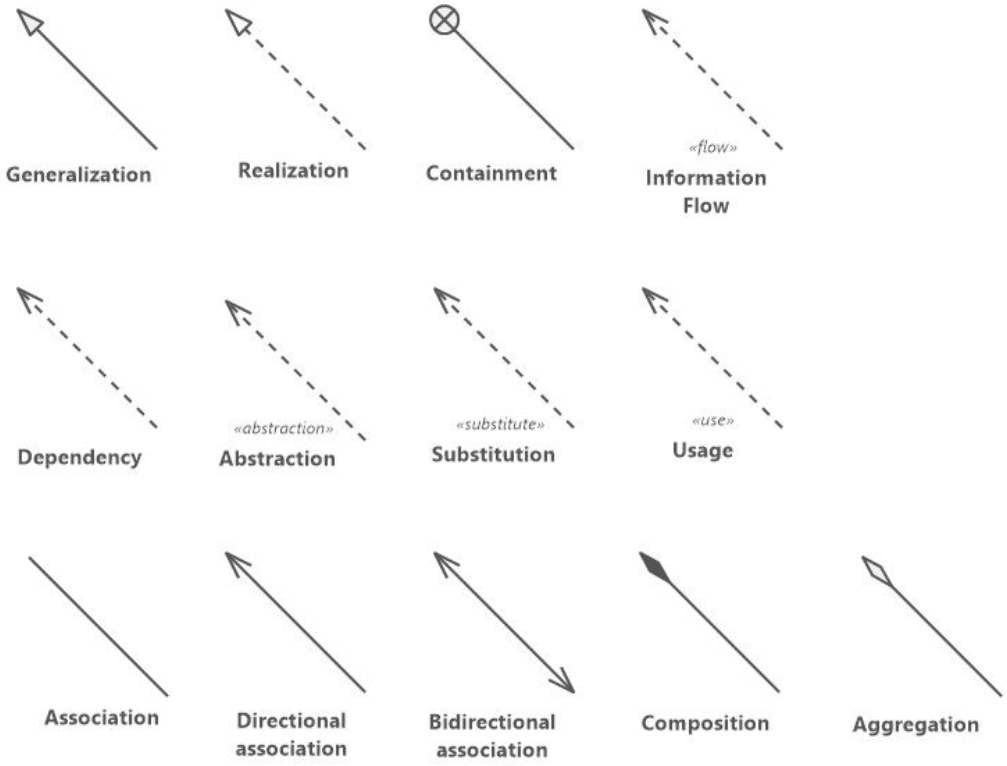
UML

هذا ال diagram يوضح وجود abstract class، وال abstract class وال method التي يحويها بداخله يتم الإشارة إليها من خلال الخط المائل (italic).





UML



ملخص الأسهم والمعاني التي تشير إليها:
ويمكنك الإطلاع على هذا الرابط لمزيد من التفاصيل مع مثال عملي.

سنتطرق في الشرائح التالي لاستعراض سريع لكل من هذه ال relations:

1. Dependency
2. Association
3. Aggregation
4. Composition

UML

1. Dependency: ويقصد بها ال relation التي تربط بين شيئين بحيث يعتمد أحدهما على الآخر، وبسبب هذه الاعتمادية فحصول تغيير على الأول قد يؤثر على الثاني الذي يعتمد عليه، وهذا يمكن أن يتسبب بخلل في أي وظيفة عند أي تعديل...، ومن الأمثلة على ذلك Customer Class و Order Class، ال order class سيحتوي في داخله reference لل customer class حتى يتم ربط ال order بصاحبه، فلو تغير مثلا ال type أو ال Method المستخدمة في ال customer فإن ال order سيتأثر ويمكن أن يفقد قدرته على ربط ال order بصاحبه، لذلك، يكون التعديل على هذا النوع من ال relation بحرص شديد، وعلى صاحب التعديل أن يعلم مقدار التأثير الحاصل لكل class والذي يعتمد على ال class الحالي...

UML

2. Association: وهي العلاقة التي توضح أي من ال object يستخدم أو يتفاعل مع object آخر، وبشكل عام يمكن القول بأننا نستخدمها لتمثيل العلاقة بين ال classes من خلال تمثيل الإرتباط بين ال objects وكيف يصل ال object لل class الآخر من خلال هذه العلاقة...، ومن الأمثلة على ذلك الطلاب والمعلمين، فكل طالب يمكن أن يرتبط بعدة معلمين، وكل معلم يمكن أن يرتبط بين عدة طلاب...الطالب في علاقته هذه مع المعلم تسمى Association...باختصار (علاقة بين classes بحيث يستفيد كل واحد منهم من الآخر أو أحدهما من الآخر، ولكل واحد منهم ال Life-cycle الخاصة به)

والآن، لنشاهد المثال في الشريحة التالية...

UML

إليك الرابط الخاص بالمثال ^^

1

لاحظ الدالة printContent، هذه الدالة تستقبل ال Object الخاص ب Class معين، هذه ال Classes تشترك ب abstract، فبكل بساطة يمكننا استخدام print بكل سهولة...، هذا الشكل من العلاقات هو Association، لأن حذف هذه ال Method أو عدم استخدامها لن يعمل broken لل Website Content Class...

```php
class Website {// 2nees.com
    private string $content;

    public function __construct(string $content)
    {
        $this->content = $content;
    }

    public function printContent(WebsiteContent $websiteContent): void {
        $websiteContent->setText($this->content);
        $websiteContent->print();
    }
}
```

2

```php
abstract class WebsiteContent {// 2nees.com
    protected string $text;

    public function setText($text): void {
        $this->text = $text;
    }

    abstract function print(): void;
}

class ListContent extends WebsiteContent {
    function print(): void
    {
        echo "<ul><li>{$this->text}</li></ul>" . PHP_EOL;
    }
}

class ParagraphContent extends WebsiteContent {
    function print(): void
    {
        echo "<p>{$this->text}</p>" . PHP_EOL;
    }
}
```

3

لاحظ هنا أن حذف ال website لم يتسبب في مشكلة لل List or Parag، وكل منهما استطاع متابعة عمله بشكل طبيعي...

```php
$listContent = new ListContent();
$paragraphContent = new ParagraphContent();

$website = new Website("2nees.com");
$website->printContent($listContent);
$website->printContent($paragraphContent);
unset($website);// Flow still work after remove website
$listContent->setText("List will not effect if we
remove website, I'm Association With Website!");
$listContent->print();
$paragraphContent->setText("Paragraph will not effect
if we remove website, I'm Association With Website!");
$paragraphContent->print();
```

4

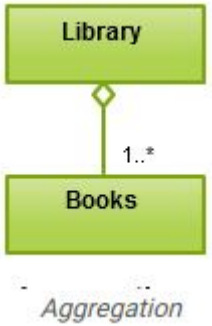
```
<ul><li>2nees.com</li></ul>
<p>2nees.com</p>
<ul><li>List will not effect if we remove website, I'm Association With Website!</li></ul>
<p>Paragraph will not effect if we remove website, I'm Association With Website!</p>
```

UML

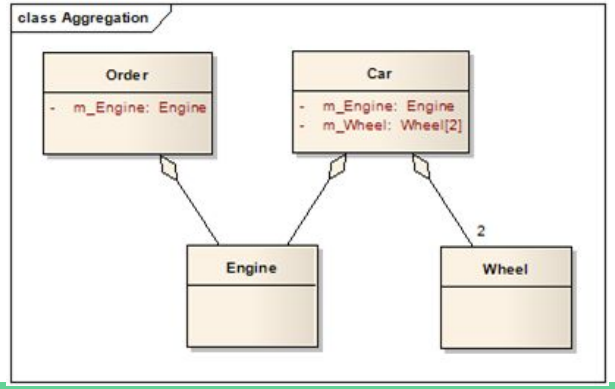
3. Aggregation: يمثل ال aggregation علاقة association لكنها أكثر تحديدا (special type of association)، وأهم ما يميز هذه العلاقة أنها توضح التمثيل الخاص بال classes على شكل مجموعة تنتمي إلى class واحد أساسي، مثلا، ال course وال course offering، فال course offering ينتمي لل course فالعلاقة بينهم aggregation، وكذلك مثلا المكتبة تحتوي مجموعة من الكتب، لذلك فالمكتبة ستحتوي collection من ال classes التي تنتمي لها مثل books class...

مثال:

مثال 1:
Library هو ال ال class الأساسي، لذلك يوضع المعين في اتجاهه



ملاحظة: يمكن أن يتغير وصف هذه العلاقة لتصبح مثلا composition بناءا على المعطيات الخاصة بالمشروع...



مثال 2:
لاحظ هنا أن المحرك والعجلات مرتبطة بال class الأساسي وهو السيارة، لذلك فهي aggregation، ويمكن أن يتبع الكلاس في علاقته أكثر من class كما في المحرك

UML

مثال عملي على ال Aggregation:

تخيل أن لدينا كلية، هذه الكلية فيها مجموعة من الطلاب والمعلمين، الكلية هي ال Owner لل Students وال Teachers، وبنفس الوقت، فإن حذف الكلية لا يعني موت الطلاب أو المعلمين، ولا يعني أن الطلاب سيبقون دون كلية...، بل يمكن إذا أغلقت الكلية أن ينقل الطالب لكلية أخرى...

1

تعتبر ال CollageA إحدى الكليات التي ينتمي لها مجموعة من الطلاب، لكن ذهاب هذه الكلية لا يعني أن الطلاب سيختفون أو سيخرجون من الجامعة،
فبكل بساطة يمكن أن ينتمي الطلاب والمعلمون لكلية أخرى...، كما أن الطلاب والمعلمين سيحافظون على ال life cycle الخاصة بهم..

```php
class CollageA {// 2nees.com
    private string $name;
    private array $teachers;
    private array $students;
    public function __construct(string $name, array $teachers, array $students)
    {
        $this->name = $name;
        $this->teachers = $teachers;
        $this->students = $students;
    }
    public function printCollageDetails(): void
    {
        $teachersList = [];
        $studentsList = [];
        foreach ($this->teachers as $teacher){
            $teachersList[] = $teacher->getName();
        }
        foreach ($this->students as $student){
            $studentsList[] = $student->getName();
        }
        $printableTeachers = implode(", ", $teachersList);
        $printableStudents = implode(", ", $studentsList);
        echo "
            Collage Name: {$this->name}
            Teachers: {$printableTeachers}
            Students: {$printableStudents}" . PHP_EOL;
    }
}
```

UML

هذه مجموعة ال Classes التي تنتمي لل Owner **2**

```php
abstract class Users {// 2nees.com
    private string $name;
    public function __construct(string $name)
    {
        $this->name = $name;
    }
    public function getName(): string
    {
        return $this->name;
    }
}

class Teacher extends Users {
    // Do any thing...
}

class Student extends Users {
    // Do any thing...
}
```

في هذه الجزئية تلاحظ أننا لو حذنا ال Owner فإن المعلمين والطلاب لن يتأثروا...، وبنفس الوقت، فإن ال CollageA هي التي تمثل بيانات المعلمين والطلاب، فبحذفها اختفى هذا التمثيل وتحتاج إلى Owner آخر... **3**

```php
// Client - 2nees.com
$tech1 = new Teacher("Hikmat");
$tech2 = new Teacher("Ahmad");

$std1 = new Student("Anees");
$std2 = new Student("Taher");
$std3 = new Student("Saed");

$collage = new CollageA("Information Technology",
[$tech1, $tech2], [$std1, $std2, $std3]);
$collage->printCollageDetails();
unset($collage);
echo $tech1->getName() . PHP_EOL;
echo $std1->getName();
```

```
Collage Name: Information Technology
Teachers: Hikmat, Ahmad
Students: Anees, Taher, Saed
Hikmat
Anees
```

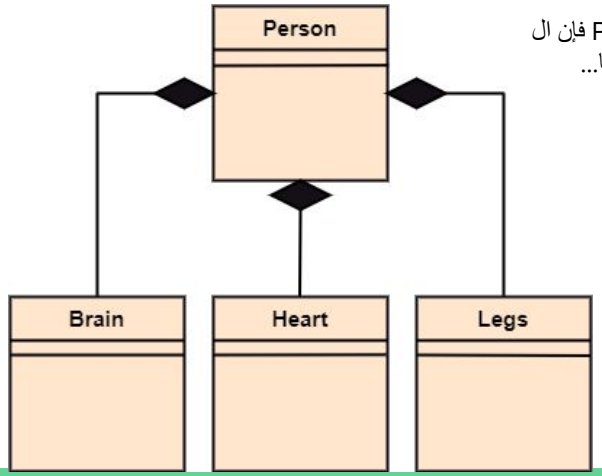
4

إليك الرابط الخاص بالمثال ^^

UML

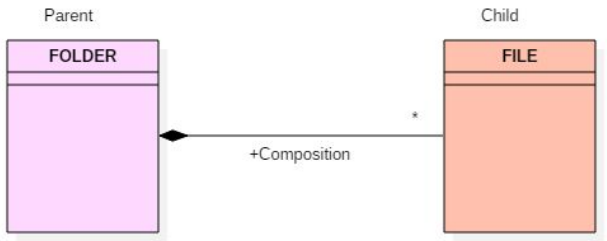
4. Composition: ال Composition هو specific kind of aggregation، والفرق الجوهري هو أن ال Composition يرتبط وجوده بال class الأساسي الذي ينتمي له، ويكون ضمن ال life cycle الخاصة به، فإن تم حذف ال class الأساسي فإن أي علاقة من نوع composition ستدمر، ومن الأمثلة على ذلك، مبنى فيه مجموعة من الغرف، مثلا حمام ومطبخ، إذا تم هدم هذا المبنى، فبكل تأكيد الحمام والمطبخ سيتم تدميرهم معه، وهذا ما يسمى بال composition...

مثال:



مثال 1: هنا إذا تم تدمير ال Person class فإن ال classes الخاصة بالأعضاء لا جدوى منها...

مثال 2: وجود ال folder شرط أساسي لوجود ال file،
فإذا تم حذف ال folder فإن ال file سيذهب أيضا...



UML

إليك الرابط الخاص بالمثال ^^

مثال:

1

يمثل هذا ال Class ال Owner لمجموعة ال OBject، والذي يجب أن يتواجد حتى تكون لل Object المترابطة معه لها قيمة، مثلا إذا ذهب الإنسان فالعقل والقلب سيذهبان معه، لأن وجودها أصبح بدون معنى، ولن يستفيد منهما Classes أخرى...

```php
class Person {// 2nees.com
    private Heart $heart;
    private Brain $brain;

    public function __construct(Heart $heart, Brain $brain)
    {
        $this->heart = $heart;
        $this->brain = $brain;
    }

    public function printPersonDetails(): void {
        echo "
            Heart is {$this->heart->getStatus()} And
            Brain is {$this->brain->getStatus()}
        ";
    }
}
```

2

هذه ال Classes هي جزء من ال Person، فذهاب ال Person يعني عدم أهميتها

```php
abstract class Members {// 2nees.com
    private string $status;

    public function __construct(string $status)
    {
        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
    }
}

class Heart extends Members {
    // Do any thing...
}

class Brain extends Members {
    // Do any thing...
}
```

3

```php
// Client - 2nees.com
$heart = new Heart("Strong");
$brain = new Brain("Smart");

$person = new Person($heart, $brain);
$person->printPersonDetails();
```

```
Heart is Strong And
Brain is Smart
```

UML

ملاحظة مهمة: كما لاحظت في الأمثلة السابقة، فجميعها متشابه وقريب من بعضه البعض، وهذا صحيح، لأننا قلنا أن كل نوع ما هو إلى نوع مخصص للذي سبقه، وبناءا على هذا يتضح لك أن هذه العلاقات وتمثيلها هي مفاهيم تؤخذ بعين الاعتبار عند بناء المشروع ومعرفة العلاقات بينها وبين ما سيتم بنائه، فبمجرد الاطلاع على ال UML مثلا الخاص بال Composition فإن هذا سيعني لك أن مجموعة ال Classes المرتبطة به سيتم حذفها إذا تم حذف ال Owner، لأنه لا جدوى منها!، وعند مشاهدتك لل Aggregation ستقول، أن التمثيل لو تم حذفه فليست مشكلة عندي، سنجد تمثيلا آخر لل class المرتبطة بي، أما ال Association فإنك ستقول لا بأس، كل واحد منهم له شأنه الخاص...، ويمكن الاستفادة مما عند الآخر دون أن يكون هو Owner لي...

بناءا على هذا الفهم، عندما نتطرق لبعض التطبيقات العملية وتطبيق ال Composition بدلا من الوراثة، سيكتمل المنظور لديك...لن نستبق الأحداث الآن، فما زلنا في أول الطريق...

فلْتعلمْ أن واحدة من أعظم العقوبات التي قد تقع للإنسان هي قسوة القلب، وقسوة القلب معناها كل قلب غليظ لا تجدي فيه المواعظ، ولا تنفعه الآيات والنذر، فلا يُرغبهم تشويق، ولا يزعجهم تخويف، فاحذر أن تكون منهم!

What's a Design Pattern؟

أول مرحلة في هذا الشرح، هو معرفة ما يعينه ال Design Pattern، وبما يختلف عن الخوارزميات؟، ويمكن القول بأن ال Design patterns هي الحل القياسي لمعظم المشاكل التي تتكرر عادة عند تصميم أي software، ويمكن تخصيص هذا التصميم ليتناسب مع ال software design حتى يتم حل المشاكل المتعلقة بالشيفرة البرمجية!

وعليك أن تعلم أن Design pattern لا يمكن البحث عنه ثم نسخه للمشروع وكأنه قطعة من شيفرة برمجية!، لأنه يمثل مفهوم عن كيفية تصميم الشيفرة البرمجية وحل مشاكل معينة بالتصميم الخاص بال software!، بل إن ما يمكن فعله هو فهم التفاصيل الخاصة بال Design pattern ومن ثم محاولة تطبيق المفاهيم المتعلقة به ضمن البرنامج الخاص بك ليتم حل المشكلة من خلال تطبيق هذا الفهم، وهذا يعني أن تطبيق هذا المفهوم قد يختلف من شيفرة برمجية لأخرى بما يتناسب مع طريقة تنفيذ البرنامج ودون الخروج عن المفهوم الخاص بها!

?Why Should I Learn Patterns

هناك العديد من المبرمجين لم يسمعوا أو يدركوا أي design pattern ومع ذلك فهم يمارسون عملهم في البرمجة بشكل اعتيادي، كما أن هناك الكثير من المبرمجين الذين يستخدمون ال design pattern ولا يدركون ذلك، هذه الفئات من المبرمجين وإن تابعت عملها ونجحت في ذلك، إلا أنها فقدت جزء مهما في القدرة على تطوير المهارات وحل المشاكل بالطريقة الأفضل باستخدام مفاهيم ال OOP، والتي عادة ما نجد هذه الأخطاء تتكرر بكثرة...

ومن هنا، يمكن القول أننا نتعلم ال design pattern ل:

1. لأن ال design pattern يمثل أداة تحتوي مجموعة من الحلول التي تم استخدامها والتأكد من فعاليتها العديد من المرات في حل أكثر المشاكل شيوعا أو تكرارا، كما أنها تعلمك طرقا جديدة لحل المشكلات بناءا على مفاهيم ال OOP، وبهذا ستتمكن من حل المشكلات بطريقة صحيحة وأكثر ذكاءا، وأفضل وتتبع القواعد بشكل أفضل.

?Why Should I Learn Patterns

2. من خلال معرفة ال design pattern يمكن لل team أو المبرمجين من التواصل مع بعضهم البعض لحل المشكلات فقط من خلال اقتراح ال design pattern المناسب دون الحاجة لشرح هذا ال design pattern، فلو فرضنا أننا في فريق عمل ضمن شركة واحدة تستخدم أكثر من لغة برمجة، وواجه واحد من فريق العمل مشكلة ما، فيمكن بكل بساطة أن أشير عليه باستخدام ال Singleton أو Decorator أو Observer ودون الحاجة لشرح هذا ال design pattern، وبهذا سيتمكن فريق العمل من اقتراح الحلول ومعالجتها من خلال المفاهيم بكل سهولة...

3. من خلال فهمك لل design pattern يمكنك التخطيط لبناء ال system الخاص بك قبل الشروع به واختيار ال pattern المناسب لك، وهذا سيجعل من عملية تصميم أي system أفضل، وأدق، والشيفرة البرمجية أفضل وتتبع معايير ال OOP بشكلها الصحيح.

ما هو الفرق بين الخوارزميات وال Design Pattern

الكثير من الناس يحصل لها لُبس بين ال Pattern وال Algorithm، والسبب في ذلك يعود لأن المفهومين يشيران إلى الطريقة الأفضل لحل المشاكل التي يمكن أن تواجهنا، لكن الفرق الأساسي هو أن ال Algorithm تقوم بتعريف مجموعة من ال sets of actions والتي من خلالها يمكن تحقيق الهدف المراد، بينما يمثل ال Pattern درجة أعلى فهو يمثل الوصف الخاص بالحل والذي يمكن تنفيذه بطريقتين مختلفتين في برنامجين مختلفين!

وبكل بساطة يمكن القول بأن الخوارزميات تمثل الخطوات اللازمة للوصول إلى الهدف، بينما ال Pattern يمثل ما ستراه من نتائج ومميزات، لكن الترتيب الدقيق للتنفيذ متروك لك.

مما يتألف ال pattern؟

يتم كتابة معظم ال patterns بشكل formal حتى يتمكن الناس من إعادة صياغتها بنماذج مختلفة، هذا النموذج يحتوي على مجموعة من الأقسام التي عادة ما تكون موجودة في ال description الخاص بأي pattern وهي:

- **Name:** لكل pattern اسم unique وقصير، بحيث يمثل هذا الإسم الوصف الخاص بال pattern والذي يمكن أن يوصل فكرة ال pattern.
- **Intent**: يمثل هذا المكون الغرض من وجود هذا ال pattern، ويتم وصف المشكلة وحلها بشكل مختصر.
- **Motivation**: يمثل هذا المكون الدافع الذي يقوم بشرح مشكلة نموذجية محددة تمثل فئة واسعة من المشاكل التي يتعامل معها ال pattern، باختصار، يمكن القول بأنه ال scenario الذي يوضح المشكلة.

مما يتألف ال pattern؟

- **Structure**: يمثل هذا المكون الهيكلية الخاصة بال classes والتي توضح كل جزء من ال pattern وكيف يمكن أن يرتبط هذا ال class مع ال pattern، باختصار يكون في هذه النقطة ال class diagram وال object diagram التي تخص بال pattern والتي يتم تطبيقها على ال Motivation.

- **Code example**: يمثل هذا المكون شيفرة برمجية يتم إضافتها كمثال على ال pattern، وعادة ما يتم كتابتها بإحدى لغات البرمجة المشهورة والأكثر انتشارا حتى تصل الفكرة الخاصة بال pattern إلى أكبر عدد ممكن بأسهل طريقة!

طبعا هناك catalogs تقوم بإظهار بعض المكونات الأخرى كال known uses وال related use...إلى آخره، لكننا نكتفي بالمكونات الأساسية والمتواجدة في معظم ال patterns.

مما يتألف ال pattern؟

بناءا على المكونات السابقة، تختلف ال design patterns في مستوى التعقيد الخاص بها، وهذا ينعكس على عدة مستويات فتشمل كمية التفاصيل الخاصة بال pattern، وقدرته في التعامل باختلاف ال scale والقدرة على تطبيقه أثناء البدء ببناء أي system.

لهذا فإن ال design pattern يمكن تقسيمها إلى:

1. Low-Level: وهي ال patterns التي صممت لخدمة لغة برمجية معينة، ويطلق على هذا النوع اسم "idioms"، وفي الواقع هي مجرد فكرة برمجية لحل مشكلة معينة في لغة معينة.
2. High-Level: وهي ال patterns التي صممت لخدمة أكثر من لغة برمجة، فهي تمثل ال architectural الذي يمكن من خلال فهمه تنفيذ ال pattern في أي لغة برمجية، وهذا النوع هو الأشهر طبعا...

مما يتألف ال pattern؟

بناءا على ما سبق، يمكن تصنيف ال patterns بناءا على الغرض (intent) الخاص بها، هذا التصنيف يقوم بوضع جميع ال pattern ضمن category أو group، وسنشير هنا إلى أشهر ثلاث مجموعات وهي:

1. **Creational patterns**: وهي مجموعة ال patterns التي تهتم بالآلية أو الطريقة التي يتم من خلالها إنشاء object بطريقة تجعل من إعادة استخدامه أمرا سهلا وأكثر مرونة.
2. **Structural patterns**: وهي مجموعة ال patterns التي تهتم بكيفية جمع ال classes وال objects داخل هياكل أكبر (Larger Structure) مع الحفاظ على الكفاءة والمرونة الخاصة بال structure.
3. **Behavioral patterns**: وهي مجموعة ال patterns التي تهتم بطريقة التواصل بين ال objects بشكل فعال وكيفية توزيع المسؤوليات بينهم.

فلْتعلمْ أن اللّٰه -سبحانه وتعالى- كما أرانا طريق الهداية وطهر قلوبنا بالإيمان، جعل طهارة الجسد شطر الإيمان، فالإسلام دين الطهارة الكاملة، فطهر رب العزة قلوبنا من الكفر، وطهر ألسنتنا من الفحش والكذب، وطهرنا من الأخلاق الرذيلة، وأمرنا بطهارة أجسادنا من كل نجس، ودلنا على كيفية ذلك، فالحمد لله!

Software Design Principles

مع تقدم السنوات والعمل على العديد من المشاريع يصل العديد من المطورين إلى خبرة تمكنهم من كتابة شيفرة برمجية بطريقة احترافية أو جيدة، كما يتمكن من فهم الأنظمة ومعرفة الجيد منها من السيء، وقبل أن يخوض في غمار هذا النظام!، كما أن هذا الخبير يصبح مهتما بشكل كبير بكيفية بناء ال software architecture بأفضل طريقة وذلك لأهمية ال designing software architecture على سير العمل وكتابة الشيفرات البرمجية والقدرة على استخدامها لاحقا وتقسيم العمل ومعرفة المحددات والتقنيات اللازمة والموارد والوقت المناسب لذلك...إلى آخره، كل هذا سببه الإهتمام بالوقت والمال، فكلما كان الوقت اللازم لإنجاز المشروع أقصر فهذا يعني توفير المال للتسويق، واستغلال الوقت قبل ظهور منافسين يحملون نفس الفكرة ومجاراتهم ومنافستهم في حال وجودهم!، ومن هذا المنطلق سنبدأ بمجموعة من المبادئ المهمة عند تصميم أي تطبيق، وسيتم تقسيمها لعدة أجزاء...

Software Design Principles

- **أولا: Features of Good Design:**

التصميم الجيد للشيفرة البرمجية الجيدة يحتوي بداخله جزئين مهمين، وهما:

1. **Code reuse**: مع بساطة هذا المبدأ إلا أنه من أهم الخصائص التي يجب أن تتواجد في أي شيفرة برمجية، ولها دور كبير في توفير الوقت اللازم لتنفيذ أي عملية بدلا من تكرار كتابة الشيفرة البرمجية مرارا وتكرارا! وهذا يعني توفير المال والجهد اللازم، وهذا يمكن أن يجزأ إلى إمكانية استخدام الشيفرة البرمجية خلال المشروع في أكثر من مكان، وإمكانية استخدام هذه الشيفرة البرمجية خارج هذا النظام، وبمقدار الإحترافية التي بني عليها النظام والشيفرة البرمجية فيمكن نقل هذه الشيفرة والتي تمثل عادة module أو class من software إلى آخر بسهولة أكبر...ويمكن تقسيم ال reuse إلى ثلاث درجات، الأولى low-level وهي التي تمثل عملية ال reuse لل class/lib/containers، والثانية Highest-level والتي تمثل ال frameworks بحيث تحتوي في ثناياها ال design pattern وال behavior الذي يحكم كيفية استخدام ال framework، ويقسم ال framework ال software إلى UNITs بحيث تسهل إدارة أي جزئية وإعادة استخدامها، وهو الأكثر راحة واحترافية، لكن مشكلته في التكلفة اللازمة لبنائه -في حال لم يتوفر- والجهد اللازم لذلك، والثالثة هي الدرجة الوسيطة بين low-level وال highest-level وهي استخدام ال design pattern في بناء الشيفرة البرمجية لكن بشكل أقل وموجز أكثر من ال framework والذي من خلاله يتيح طريقة التفاعل بين ال classes/objects وكيف تتفاعل مع بعضها البعض...لهذا فأفضل درجة هي ال frameworks....

Software Design Principles

2. **Extensibility**: عند التفكير ببناء أي software يجب الأخذ بعين الإعتبار القدرة على تطور هذا ال software ومرونته لتحديث التقنيات المستخدمة أو القيام ببعض التعديلات أو الإضافات أو التحسينات أو حذف مزايا أو استبدالها، كل هذا يسبب مشاكل عظيمة إن لم تكن بالحسبان، فإن لم تكن الشيفرة البرمجية تم تصميمها بطريقة صحيحة، فإن أي تعديل من هذا النوع سيؤدي بكارثة، لذلك تجد العديد من المبرمجين خصوصا بعد اكتساب الخبرة ينظرون مليا في إمكانية قدرة ال software على التجاوب مع التحديثات، مثلا لو قمت ببناء موقع إلكتروني فيه عمليات الدفع، فيجب أن تأخذ بعين الإعتبار أن ال user قد يطالبون بإضافة وسيلة دفع جديدة، أو إضافة تقرير معين، وإذا قمت بتصميم لعبة فيجب الأخذ بعين الإعتبار القدرة على تطور التصاميم، والقدرة على تطوير المحاكاة، والقدرة على تلبية متطلبات تشغيل تختلف من مستخدم لآخر، وهنا نصل إلى نقطة مهمة وهي أن ال software لو كان ال ver 1 ممتاز ونال إعجاب المستخدمين، إلا أنهم مع الوقت سيحتاجون وسيفكرون في مزايا أخرى تلبي احتياجاتهم خصوصا بعد تعاملهم مع النظام الخاص بك، ولهذا معظم المطورين الخبراء يصممون أنظمتهم بهذا الشكل...

Software Design Principles

- **ثانيا: Design Principles:**

كيف يمكنني بناء شيفرة برمجية مرنة وسهلة للفهم؟ وكيف يمكن تقييم ما قمت ببنائه؟ ما هي المبادئ التي يجب أن أتبعها لبناء architecture جيد، مرن، stable؟ هذه الأسئلة وغيرها من هذا النوع ليس لها إجابة محددة لأنها تختلف حسب نوع التطبيق المراد بناءه، لكن هناك مجموعة من المبادئ التي تمت كتابتها بشكل عام ويمكنك الإطلاع عليها ومن ثم إسقاطها على طبيعة التطبيق الخاص بك، وسنتحدث هنا عن ثلاث مبادئ:

1. Encapsulate What Varies
2. Program to an Interface, not an Implementation
3. Favor Composition Over Inheritance

Software Design Principles

1. Encapsulate What Varies: الهدف الأساسي من هذا المبدأ هو تقليل الآثار المترتبة على أي تعديل، وهذا المبدأ فكرته تقوم على فصل العناصر القابلة للتغير (العناصر التي ستتغير) عن العناصر التي ستبقى كما هي ولن يحصل لا أي تغيير عادة، وهذا الأمر يجعل من عملية التعديل والصيانة أمرا أسهل، لأن الوقت المستغرق سيكون أقل، والمكان محدد، والتأثير معروف فيمكن تجربته وحصر المشاكل المتوقعة بسببها... ومن الأمثلة التي توضح هذا المفهوم، السفينة، فالسفينة في البحر لها جزءان، جزء في البحر لا يظهر، وجزء تشاهده، الجزء الموجود في البحر لو تعرض للغم بحري، فإن النتيجة هي غرق السفنية، لكن لو أخذ مصمم السفينة هذا بعين الإعتبار، فإنه سيقوم ببناء السفينة على شكل مقصورات بحيث لو تأثر أحد المقصورات لن يتأثر باقي المقصورات، ويمكن معالجة الضرر الناتج بسهولة أكبر!، وكذلك الحال في البرامج التي ستقوم ببنائها، إذا قمت ببناء module مستقل لتنفيذ مهام محددة، فإن الخطأ ومعالجته سيكون على مستوى هذا الخطأ داخل هذا ال module...=> الشريحة التالية

Software Design Principles

هذا المبدأ يمكن تقسيمه إلى جزئين:

a. Encapsulation on a method level: بكل بساطة حافظ على ال method بحالة غير قابلة للتغير من أي logic قد يسبب ذلك، ويكون الحل بفصل ال logic القابل للتغير في method أخرى والحفاظ على ال method بشكلها المستقر...شاهد المثال:

في هذا المثال getOrderTotal عبارة عن دالة تقوم بإرجاع السعر النهائي لعملية شراء، المشكلة هي وجود ال TAX، فالضريبة قيمتها تتغير بشكل كبير، ويمكن أن تختلف ضمن ظروف معيارية كثيرة، لذلك فالطريقة الأفضل هي فصلهم بحيث يتم احتساب الضريبة في مكان آخر ومن ثم إرجاع النتيجة لل getOrderTotal بهذا ال getOrderTotal سيكون stable... وبكل بساطة، إذا زاد مستوى التعقيد لحساب الضريبة فبكل سهولة يمكنك نقل العملية ل class...هل أدركت أهمية هذا المفهوم؟

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  if (order.country == "US")
    total += total * 0.07 // US sales tax
  else if (order.country == "EU"):
    total += total * 0.20 // European VAT

  return total
```

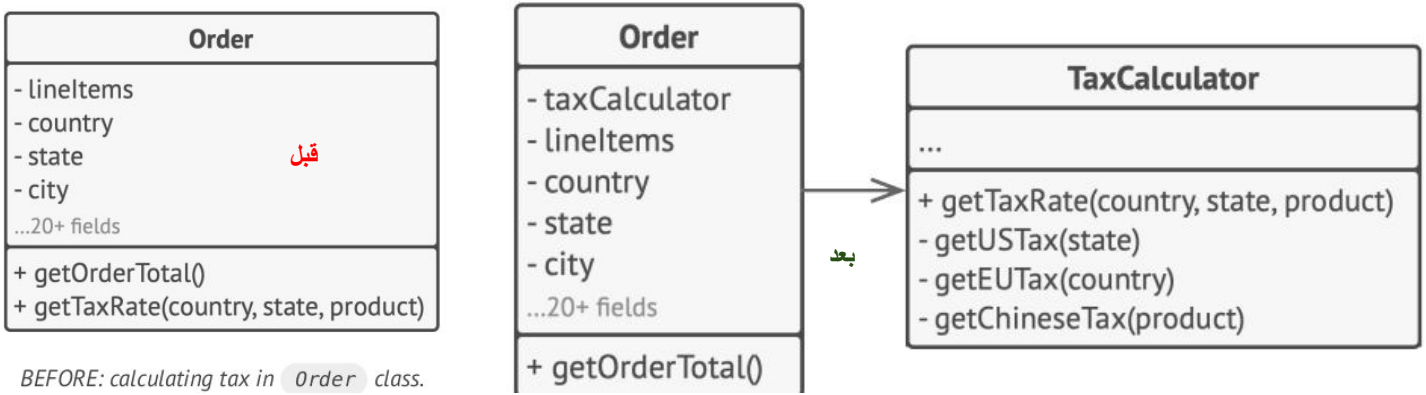
خطأ

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  total += total * getTaxRate(order.country)

  return total

method getTaxRate(country) is
  if (country == "US")
    return 0.07 // US sales tax
  else if (country == "EU")
    return 0.20 // European VAT
  else
    return 0
```

صح

Software Design Principles

b. Encapsulation on a class level: قلنا في المثال السابق، أن مستوى التعقيد لو زاد أكثر فيمكن بكل بساطة فصل هذه ال method إلى class، وهذا هو مفهوم ال Encapsulation on a class level، فبدلا من أن نزيد عدد ال method بنفس ال class لأداء وظائف مختلفة عن طبيعة ال class الحالي، نقوم بفصل هذا ال logic إلى class آخر تربط بينهما علاقة، وهذا الأمر له فائدة كبيرة جدا ومهمة...شاهد المثال:



BEFORE: calculating tax in `Order` class.

لاحظ هنا أن جميع ال method الخاصة باحتساب الضريبة تم نقلها إلى class خاص بحساب الضريبة، تربط هذا الكلاس علاقة مع ال order وبذلك تتم العملية بطريقة صحيحة وقابلة للتعديل وزيادة مستوى التعقيد بسهولة...، تخيل لو أن كل هذه ال methods موجودة داخل ال order؟!

Software Design Principles

2. Program to an Interface, not an Implementation: يعد هذا المبدأ واحد من أهم مبادئ التصميم، وهو يشير إلى استخدام ال interface/abstract وبناء الشيفرة البرمجية بناءا عليهم بدلا من الإعتماد على ال concrete classes لتنفيذ الأفكار البرمجية، وهذا المبدأ إن تم اعتماده في البرمجة فيمكن القول فعلا إن الشيفرة البرمجية مرنة كفاية لأي تعديل!، ومقياس المرونة هو مقدار السهولة في إضافة عنصر جديد دون أن يحدث خلل في الشيفرة البرمجية الحالية أو الحاجة لتعديلها بشكل غير مقبول...

مثال يوضح الفكرة: لو افترضنا وجود two classes، وهما employee و company، أول ما ستفكر فيه بالطريقة التقليدية هو أنهما two classes يمكن الربط بينهما مباشرة من خلال معرفة ال dependency، وهذا هو الأسلوب الذي يحاول معالجته وحله هذا المبدأ!

ملاحظة: ال concrete classes هي ال class التي يتم إنشائها بطريقة يصعب تعديلها -صلبة-، وهي تمثل Entity ذات معنى مثل ال developer class, tester class بداخل company class، ويحتوي في داخله جميع ال method التي يحتاجها بشكل مباشر...(بشكل مبسط، Class مش Abstract class، أو Class عمل implement لكل ال method الي ورثها...)

Software Design Principles

لتحسين النموذج السابق علينا تطبيق عدة نقاط:

i. حدد بالضبط ما يحتاجه أحد ال objects من الآخر، وما هي ال methods التي يتم تنفيذها؟
ii. قم بتعريف هذه ال methods داخل interface/abstract.
iii. نقوم بعمل implementation لل interface على ال class الذي كان يعتمد عليها ال concrete class، ونجعل الاعتمادية على هذه ال interface بدلا من ال concrete class.

مثال: لتطبيق النموذج السابق، دعونا نأخذ Group Company كمثال، الشركة فيها مجموعة الشركات أو متعاقدة مع مجموعة من الشركات، الموظفين يتبع كل منهم الشركة التي أوفدته أو يعمل تحت اسمها، لكن هذه الشركات مهما اختلفت تتشارك في كون ما عندها هو الموظفين، ولهؤلاء الموظفين مجموعة الخصائص ذاتها وإن اختلفت بعض الإضافات أو المميزات أو وسائل العمل وما إلا ذلك...

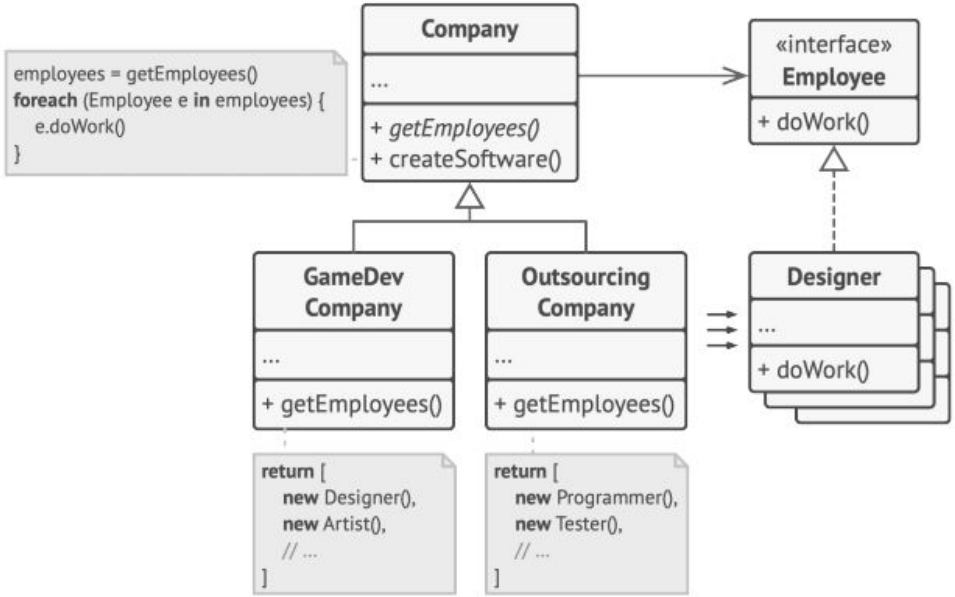
Software Design Principles

الآن، بتطبيق النموذج السابق:

1. الشركة الأم فيها مجموعة من الشركات، هذه الشركات يجب أن يتم تصنيفها وتحديد طبيعة عمل الموظفين الذين ينتمون لها وما هي أبرز ما سيقومون به، فمثلا المصمم تصميم، والمبرمج برمجة وهكذا...
2. سنقوم بإنشاء interface فيه الدوال المشتركة والتي يحتاجها كل object من الآخر، مثلا doWork لأن البرمجة والتصميم هو العمل المراد انجازه، فيشترك الجميع في العمل وتختلف الوظيفة...
3. الشركات ستكون عبارة عن subclass يرث ال superclass وهو الشركة الأم، هذه الشركات ستربط الموظفين بها، وسيقوم الموظفين باختلاف أنواعهم بعمل implement لل interface الخاصة بالموظفين والتي تشترك بها جميع الشركات...

بطريقة أخرى، ما قمنا بفعله هو جعل الشركة الأم ال SuperClass والذي يجب أن ينطوي كل شيء تحته، إحدى أهم الوظائف لهذا ال class هي مثلا getEmployees هذه الدالة سترثها الشركات وستقوم كل شركة بإرجاع موظفينها، كل عمل يقوم به الموظف حسب طبيعة عمله يتبع للشركة التي أوفدته، وكل عمل من هذه الأعمال تقوم بتنفيذ ال method التي أخذها من ال interface والتي تتشابه عند كل الشركات، بحيث يتم تنفيذ هذه ال interface على أنواع الأعمال التي يقوم بها الموظفين وتنفيذ المهام دون للحاجة للإتصال المباشر مع الشركة الأم، بل من خلال ال interface. شاهد ال diagram في الشريحة التالية:

Software Design Principles



AFTER: the primary method of the `Company` *class is independent from concrete employee classes. Employee objects are created in concrete company subclasses.*

Software Design Principles

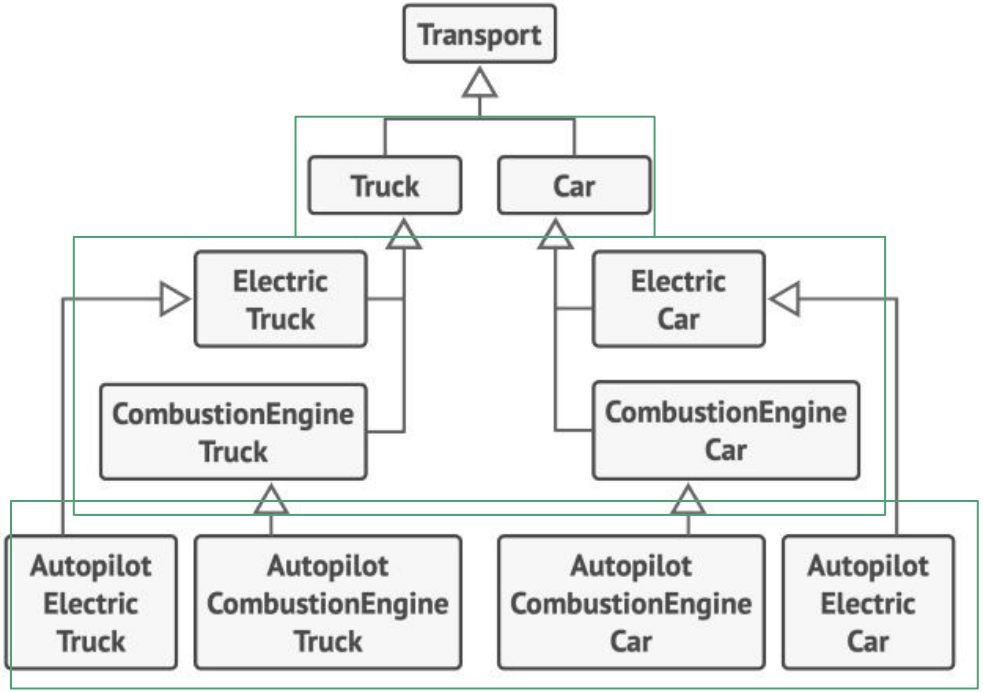
3. Favor Composition Over Inheritance: هذا المبدأ هو المبدأ الثالث من مبادئ التصميم التي سنتحدث عنها، وهو من المبادئ الجميلة والتي فاجئتني عندما قرأت عنها أول مرة، فعادة عند التفكير بطريقة تمكننا من إعادة استخدام الشيفرة البرمجية (reuse code) أول ما نفكر فيه هو ال inheritance، فالوراثة هي الطريقة الأسهل فعليا لمشاركة أي شيفرة برمجية بين two classes مثلا، كل ما عليك فعله وضع ال base code في ال super class، ثم يرث هذا ال class مجموعة أخرى!، لكن، ومع أن هذه طريقة لطيفة وسهلة، لكن لها مشاكل متعددة وخطيرة في المشاريع الكبيرة!، قد تضطرك للنظر لشيء آخر، او تحسين هذا المبدأ ليتوافق مع المشاكل التي قد تطرأ جراء الوراثة، وهذا المفهوم هو ال Composition...
* يشار للوراثة ب "is a" مثل "a car is a transport"، ويشار إلى Composition ب "has a" مثل ال "a car has an engine "

Software Design Principles

قبل الحديث عن ال Composition دعونا نستعرض أهم المشاكل التي قد تواجهك بسبب استخدام الوراثة ولماذا احتجنا للبحث عن مفهوم يحسن طريقة تصميم الشيفرة البرمجية:

- A subclass can’t reduce the interface of the superclass، وجميع ال abstract method الموجودة في ال super class أنت مضطر إلى إعادة كتابة الشيفرة البرمجية الخاصة بها، إلا إذا لم تكن بحاجة لذلك!
- عند ال override method يجب أن تكون حريصا على أن تتوافق طريقة عمل الكود مع ال method الموجود في ال super class، لأن الإختلاف قد تتسبب بخلل ما.
- أي تعديل على ال super class قد يسبب بتعطل لأي functionality داخل ال subclass
- محاولة ال reuse code من خلال استخدام الوراثة قد يتسبب بوجود parallel inheritance hierarchies، هذه النقطة تعتبر النقطة الجوهرية والأهم من وجهة نظري، وذلك لأن الوراثة عادة ما تكون single dimension، لكن ما لو كان هنا أكثر من dimension؟ سيصبح لدينا حينها parallel inheritance hierarchies، وهنا سندخل في حلقة تكرار الكثير من الشيفرة البرمجية في ال sub class ودون حاجة لذلك! وهذا سيقوم بتضخيم حجم الشيفرة البرمجية وعدد ال subclass بشكل طردي، وكل parameter سوف يضاف سيجعل التضخم للنتائج يمثل من خلال subClasses * parameter...

Software Design Principles



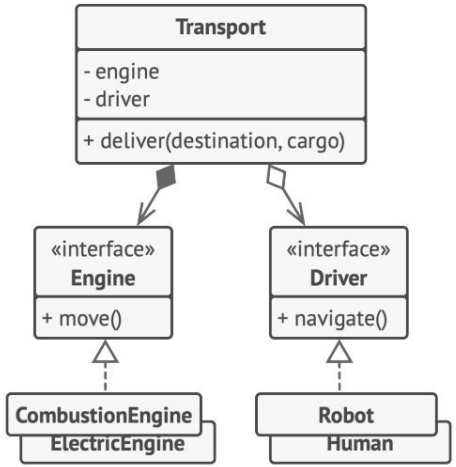
مثال على مشكلة الوراثة في وجود
أكثر من dimension:

تخيل أنك ستبني كتالوج لوسائل المواصلات التي لدينا، فمثلا لو كانت المواصلات عندنا فيها سيارات وشاحنات، السيارات منها كهربائية ومنها على الوقود، ولكل موديل منهم تتوفر القيادة الآلية، ستلاحظ أن الوراثة سببت لنا مشكلة الوراثة الأفقية وعلى أكثر من بعد،
وإذا قمنا بإضافة مزية جديدة مثل waterEngine سيضاف هذا ال subclass تحت ال car, truck وسيضاف sub class له...
وسيصبح المجموع 2 * 3 = 6 تحت ال car, truck
ولو كان لدينا مثلا bus سيصبح المجموع 3 * 3 = 9! وهذا كله بدون حساب ال subclass التي تندرج أسفل من ذلك!

Software Design Principles

الآن، بعد أن استعرضنا المشاكل السابقة للوارثة، لننظر الآن إلى مفهوم ال Composition، في الشرائح السابقة لقد تحدثنا عن ال aggregation relation، وقلنا أن ال composition هي نوع خاص من ال aggregation، والهدف منها بدلا من أن يتم عمل ال implement من خلال نفس ال object، يمكن بكل بساطة تفويض object آخر للقيام بذلك!

مثال: لاحظ كيف اختصر هذا الأسلوب سلسلة الوراثة التي كنا قد واجهناها، وكما تلاحظ هنا فوسائل النقل تختلف تبعا لنوع المحرك وتشترك بالوظيفة وهي مرتبطة بوسائل المواصلات، والسائق كذلك الأمر، ممكن أن يكون آلي وممكن أن يكون بشري!...



النقطة الجميلة في هذا الأسلوب هو أنه يمكن تغيير السلوك في أي وقت لل object أثناء ال run time...

طبعا، هذا الأسلوب سنتحدث عنه بالتفصيل في Strategy pattern بإذن الله...وأثناء التعرف على ال patterns ستلاحظ أهمية ما ذكرناه حتى الآن...

فلْتعلمْ أن الإنسان ما ترك شيئًا لله إلا عوضه الله -سبحانه وتعالى- أفضل مما ترك, وهذا العوض قد يكون من جنس المتروك, وقد يكون من غير جنسه, ولْتعلمْ أن أعظم ما يعوض به الإنسان هو الأنس بالله -سبحانه وتعالى- ومحبته, وطمأنينة القلب والنشاط في طاعته, ورفعة الدرجات في جنات النعيم والتكفير عن الذنوب, وهذا لا ينفرد بل قد يجتمع كله بفضل الله -سبحانه وتعالى- فيكون العوض ظاهرًا مثل البركة وزيادة المال, أو العلم أو الصحة أو القوة إلى آخره, فالحمد لله على كرمه.

SOLID Principles

بعد أن تحدثنا عن المبادئ الأولية لتصميم أي شيفرة برمجية ضمن أي برنامج، سنتحدث الآن خمسة مبادئ أخرى ستجعل الشيفرة البرمجية الخاصة بك أكثر مرونة، ومفهومة أكثر، وأسهل عند الصيانة والتعديل، والهدف منها إعطاء القدرة للتطبيق على التوسع بشكل مرن، فتغير حجم المشروع لن يسبب مشاكل كبيرة كما لو كان من دونها، وفعليا فهي تشير إلى مجموعة الإرشادات التي تحدث عنها روبرت سي مارتن في كتابه Agile Software Development, Principles, Patterns, and Practices، وجاء هذا الإسم (SOLID) من اختصار الأحرف الخمسة الأولى من ال OOD التي تحدث عنها في كتابه، وهي:

- **S** - Single-responsiblity Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

SOLID Principles

- Single-responsibility Principle: يقوم هذا المبدأ على إرشادك لاستخدام ال Class لهدف واحد، وأن التغيير الذي يجب أن يحدث في هذا ال class يجب أن يخدم هذا الهدف فقط!، وبعبارة أخرى، أن ال Class يجب أن يقوم بتأدية وظيفة واحدة فقط.

مثال: لو فرضنا أننا نحتاج إلى حساب المساحة لكل من المربع والدائرة، وطباعة النتائج، ما سيحدث هو أننا سنقوم بإنشاء ال class الخاص بال Circle وال Square وال AreaCalculater، بالأسلوب التقليدي ستكون ال method الخاصة بطباعة نتيجة المساحة موجودة داخل ال AreaCalculater، وهذه هي المشكلة!، لأن أي تعديل على الطباعة سيلزم تعديل على هذا ال Class، مع أن التعديل لا يتعلق بطبيعة عمل ال class ذاتها، فمثلا لو طلب منك إرجاع النتائج على شكل json، أو بناء النتائج على شكل Html معين فستتضطر حينها إنشاء دوال جديدة داخل ال class، مع أنه لا داعي لذلك، والحل من خلال هذا المبدئ، فقط كل ما عليك هو إنشاء كلاس جديد SumCalculatorOutputter ووضع دوال الطباعة بداخله...

SOLID Principles

مثال:

```php
class AreaCalculator
{
    protected $shapes;

    public function __construct($shapes = [])
    {
       ...
    }

    public function sum()
    {
      ...
    }

    public function output()
    {
        return implode('', [
          '',
              'Sum of the areas of provided shapes: ',
              $this->sum(),
          '',
      ]);
    }
}
```

قبل

SOLID Principles

لاحظ هنا أن هذا ال class مصمم لأغراض الطباعة، وبذلك فإن التعديل أو إضافة أي شكل طباعة سيكون سهلا ومباشرا!

```php
class SumCalculatorOutputter
{
    protected $calculator;

    public function __constructor(AreaCalculator $calculator)
    {
        ...
    }

    public function JSON()
    {
        $data = [
          'sum' => $this->calculator->sum(),
      ];

        return json_encode($data);
    }

    public function HTML()
    {
        return implode('', [
          '',
              'Sum of the areas of provided shapes: ',
              $this->calculator->sum(),
          '',
      ]);
    }
}
```

جديد

```php
class AreaCalculator
{
    protected $shapes;

    public function __construct($shapes = [])
    {
        ...
    }

    public function sum()
    {
        ...
    }
}
```

لاحظ هنا أن ال class لا يحتوي داخله إلا الدوال التي تتعلق بطبيعة الوظيفة التي صمم لأجلها!

بعد

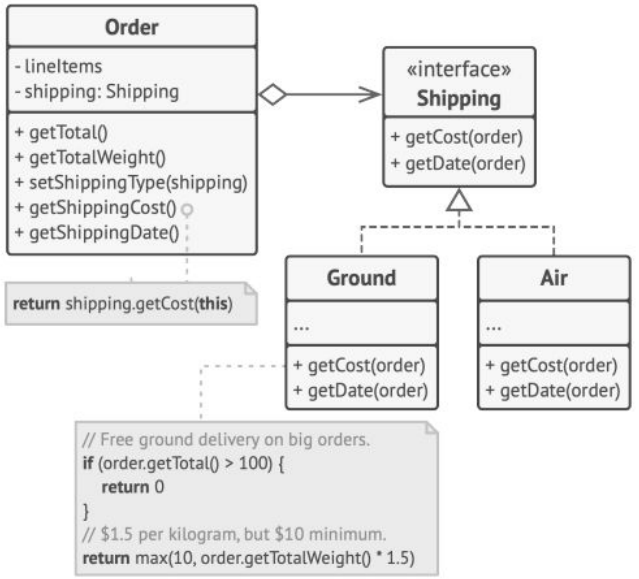
SOLID Principles

- Open-closed Principle: هذا المبدأ يرشد إلى مفهوم مهم، وهو أن ال Class يجب أن يكون قابل للتوسع حسب الحاجة (Open) ولا يُسمح بتعديل ال base code وهذا ما يسمى بال (Close)! والفكرة الأساسية لهذا المبدأ هو منع أي خطأ غير متوقع بسبب تعديل ال base code لل class، لأن ذلك قد يسبب مشكلة كبيرة خصوصا إذا كان هذا ال class مستخدم في أكثر من مكان ومن أكفر من فئة مثل ال classes الموجودة داخل ال frameworks، لذلك فهو يقوم على الحفاظ على الشيفرة الأساسية وإضافة أي method جديدة أو تعديل من خلال subclass مثلا، وبكل تأكيد هذا المبدأ لا ينطبق على ال bugs، فال Bugs يجب أن يتم معالجتها بال class وعدم السماح بمشاركتها لل subclass أو معالجتها بال subclass وترك ال parent class يورث المشكلة لأي sub class جديد!، وأكثر وقت يتم فيه تطبيق هذا المبدأ عند تصدير ال class للعمل الفعلي وفحصه ومراجعته ومشاركته في المشروع وبين أي third party، فتضمن أنت أن الشيفرة لن تتعطل، وبنفس الوقت يتم تطوير ال class وإضافة ال fields, methods الضرورية كما تقتضيه الحاجة!...لنشاهد مثال:

SOLID Principles

مثال:

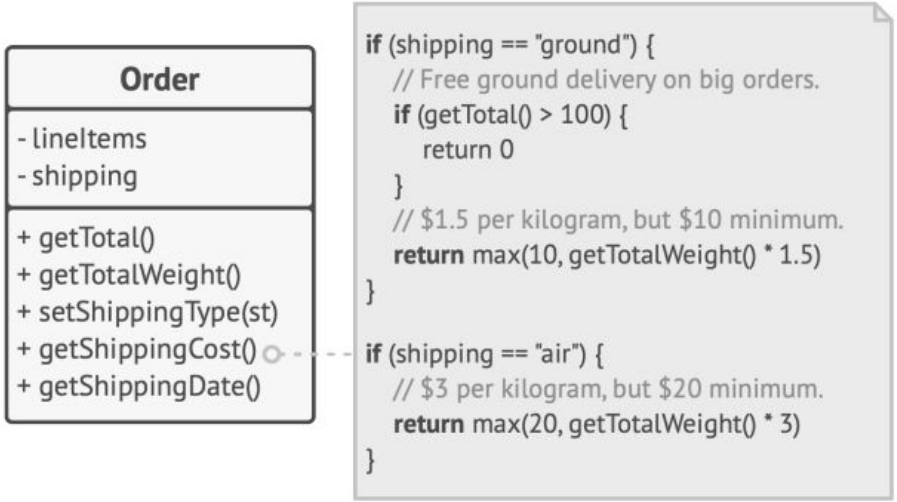
بعد: عند التفكير والعمل على هذا المبدأ، يمكن تطبيق هذه الإستراتيجية، قمنا بانشاء interface، ومن ثم قمنا بإنشاء subclass تطبق ال method التي بداخلها، من خلال هذا المبدأ إضافة ال ground مثلا لن تؤثر شيئا على ال base code، وأصبحت الشيفرة البرمجية أفضل وقابلة للتوسع بشكل ممتاز مع أي إضافة في المستقبل!



AFTER: adding a new shipping method doesn't require changing existing classes.

قبل: في هذا المثال تشاهد أن ال shipping مكتوبة داخل الكلاس hard coded، وأي تعديل مباشر هنا ممكن أن يعمل break لكل ال subs، والحل بدلا من أن نضيف ال shipping هنا ونقوم بتكرار هذه الحركة في كل مرة، نقوم باعتبار ال order على أنه close...

في الصورة لاحظ أنه لو تم اعتباره open كيف شكل ال code وخطورته...shipping == ground ثم مثلا shipping === sea...!!



BEFORE: you have to change the `Order` class whenever you add a new shipping method to the app.

SOLID Principles

- Liskov Substitution Principle: هذا المبدأ يمثل فعليا مجموعة من القواعد التي وضعت للمحافظة على إمكانية التوسع لل class قدر الإمكان، حتى أن البعض يقول أن هذا المبدأ هو مجموعة القواعد التي تضمن وتحافظ على مبدأ ال open-closed principle، هذه القواعد أو المفاهيم تتمحور حول فكرة الاستبدال، بحيث لو تم استبدال ال parent class object بأي من ال sub class object فيجب أن يتم ذلك في البرنامج دون حدوث أي مشكلة!، وهذا يعني الحفاظ على ال behavioral subtyping، فال method التي يرثها ال sub class عليه أن يحافظ على return type، وأن لا يغير من سلوك ال method، وأن يستخدم نفس ال input parameter كما تسمح بها ال method بال superclass وبنفس الوقت يسمح باستبدال ال parameter object لل super class...فمثلا لو كان لدي method وقمت بإرسال feed (Cat obj) فيجب أن يستمر البرنامج بالعمل كما هو متوقع...وذلك لأن ال Cat جزء من ال Animal، والعكس صحيح... feed (Animal obj)

SOLID Principles

قبل البدء بالأمثلة، سنشير إلى القواعد الأساسية والتي شرحنا معظمها في التعريف، وهي:

1. Parameter types in a method of a subclass should match or be more abstract than parameter types in the method of the superclass. (مثال ال Cat, Animal)
2. The return type in a method of a subclass should match or be a subtype of the return type in the method of the superclass ويقصد هنا أن ال return التي يتم الحصول عليها من ال override method يجب أن تتوافق أو أن تكون جزئا من ال type المسموح به في ال method الموجودة بال superclass، مثال: لو كان عندي كلاس Cat وكلاس BlackCat، وقمت بإنشاء دالة اسمها buyCat: BlackCat Obj، فإنها يمكن أن ترجع Cat object أو Black Cat Object، لكن من غير المنطقي أن ترجع مثلا Dog Obj أو Animal Obj، أي أنت ذهبت لشراء قطة وقمت بدفع ثمنها وإذا بالنتيجة حصلت على كلب!

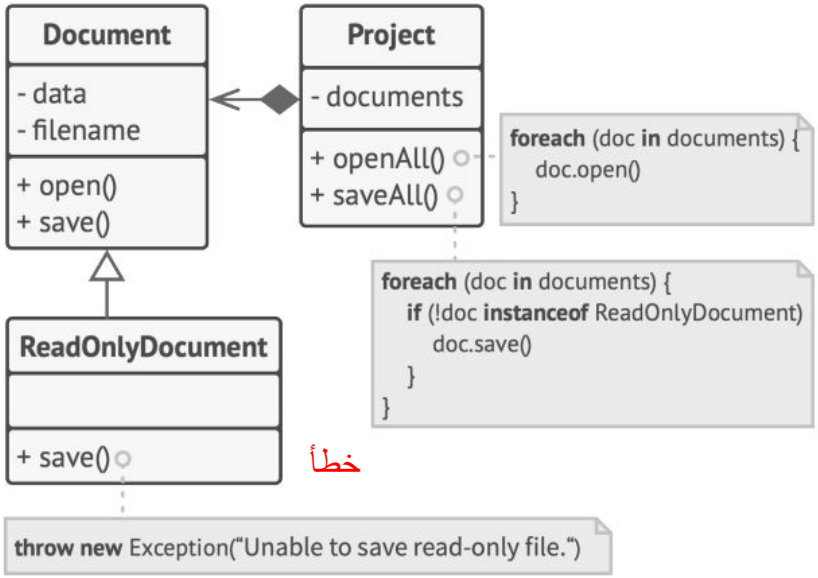
SOLID Principles

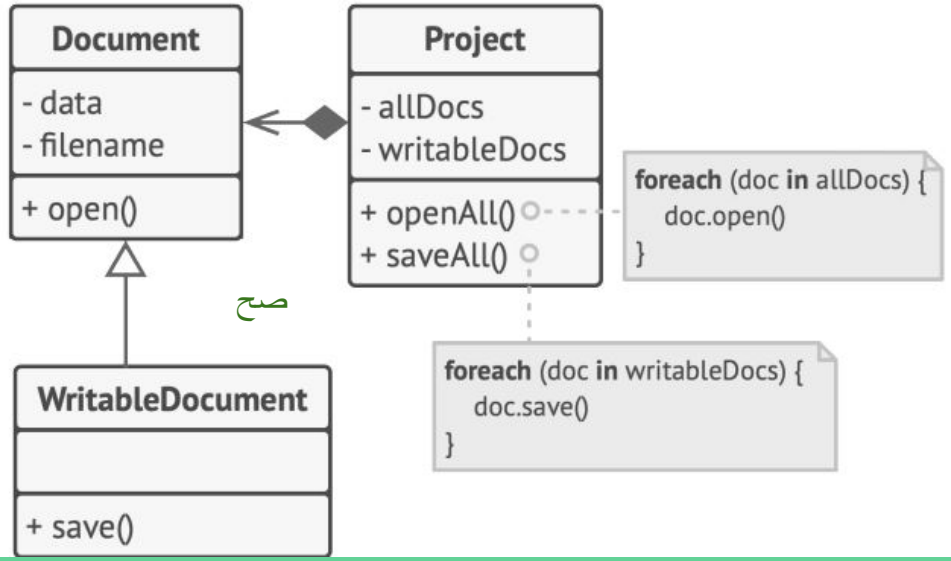
3. A subclass shouldn’t strengthen pre-conditions
4. A subclass shouldn’t weaken post-conditions
5. Invariants of a superclass must be preserved

هذه القواعد الثلاث مهمة جدا، وهي بالأصل جائت من المفهوم Design by contract (DbC)، وهذه المبادئ تعني أن ال sub class يجب أن لا يحتوي شروط ومحددات أقوى من ال super class لل pre-conditions، فلو كان مثلا ال method في ال super class تحتوي على متغير للضريبة والتي يجب أن لا تقل عن 10%، ثم قمت بعمل override للدالة وجعلت الشرط يصبح لا يقل عن 15%، فهنا قد انتهكت هذه القاعدة!، أما في ال post-conditions فهي تخبرك بأن ال subclass عند ال post-conditions يجب أن لا يكون أضعف من ال superclass، فمثلا لو كانت هناك دالة في ال superclass تقوم بإجراء action معين على قواعد البيانات ثم تقوم بإغلاق ال connection بعدها مباشرة، فلا يجب على ال method في ال subclass أن تعدل السلوك فتترك ال connection مفتوح، لأن ال client لن يتوقع أنه أصبح بحاجة لإغلاق ال connection، كما أن هذا التسلسل سيسبب مشاكل أكبر إذا كبرت شجرة الوراثة...، أما ال invariants فهي تشير إلى أن الثوابت الموجودة داخل ال class يجب عدم المساس بها في أي من ال subclass، فلو كان هناك شرط بأن قيمة الراتب يجب أن لا تقل عن 1000 دينار، فيجب على كل ال class أن لا تتجاهل هذه المعلومة ولا تتجاوزها...

SOLID Principles

مثال 1: هذا المثال فيه document، فإذا كان هذا ال doc عبارة عن readOnly فلا يمكننا حفظ ال document، أما إذا كان writable فال save method يجب أن تعمل...، المشكلة في الصورة الأولى هو أننا خالفنا مبدأ Liskov، والسبب في ذلك أن ال ReadOnly ورثت ال save من ال document، وهذا يعني أننا يجب أن نضيف condition عند ال saveAll يحدد نوع ال doc المسموح حفظه!، بينما تم حل المشكلة في الصورة الثانية وذلك من خلال trick بسيط وهو جعل ال save داخل ال writable، وبهذا لا وجود لشروط إضافية تؤثر على ال parent class عند إضافة ال subClass...





SOLID Principles

مثال 2: المثال الثاني سيتحدث عن أشهر مثال يتم طرحه لقاعدة Liskov، وهو مثال المستطيل والمربع، فرياضيا يعد المربع بالأصل هو حالة خاصة من المستطيل تساوت أضلاعه! ، لذلك، لنشاهد مجموعة من الشيفرات البرمجية والمشاكل التي بها
في الشريحة التالية:

```php
class Rectangle
{
    protected $width;
    protected $height;

    public function setHeight($height)
    {
        $this->height = $height;
    }

    public function setWidth($width)
    {
        $this->width = $width;
    }

    public function area()
    {
         return $this->height * $this->width;
    }
}
```

```php
class Square extends Rectangle
{
    public function setHeight($value)
    {
        $this->width = $value;
        $this->height = $value;
    }

    public function setWidth($value)
    {
        $this->width = $value;
        $this->height = $value;
    }
}
```

SOLID Principles

```
class RectangleTest
{
    private $rectangle;

    public function __construct(Rectangle $rectangle)
    {
        $this->rectangle = $rectangle;
    }

    public function testArea()
    {
        $this->rectangle->setHeight(10);
        $this->rectangle->setWidth(6);
        // Expect rectangle's area to be 60
    }
}
```

في الشيفرة البرمجية السابقة قمنا بإنشاء كلاس للمستطيل والمربع، ولاحظ كيف انتهكت قاعدة Liskov فالمربع قام بتغيير السلوك الطبيعي للمستطيل، وأصبح العرض هو نفسه الطول وبهذا لو قمنا مثلا باستبدال ال Obj الخاص بالمستطيل بالمربع أو المربع بالمستطيل قد تظهر لنا أخطاء شنيعة، مثلا، لو قمت ببناء PHP UnitTest للتأكد من أن مساحة المستطيل هي الطول * العرض، وكانت القيم 6 * 10 فإن الناتج لو كان 60 فهو صحيح ومستطيل، أما إن كان الناتج 100 فهذا خطأ!، والمشكلة الثانية أن ال setWidth ستقوم بالتعديل على height، وال setHeight ستقوم بالتعديل على Width، وبهذا قمنا بالتعديل على كل ال setters وجعلها تقوم بنفس العمل، وهذا سيجعل آخر set تعمل...شاهد المثال: لاحظ في المثال أن نتيجة مساحة المستطيل هي 60، وهي صحيحة، لكن لو قمنا باستبدال ال rect obj إلى square فإن ال test سيرجع نتيجة خاطئة وستكون النتيجة هي 36! وتعطل مبدأ الاستبدال هنا...

SOLID Principles

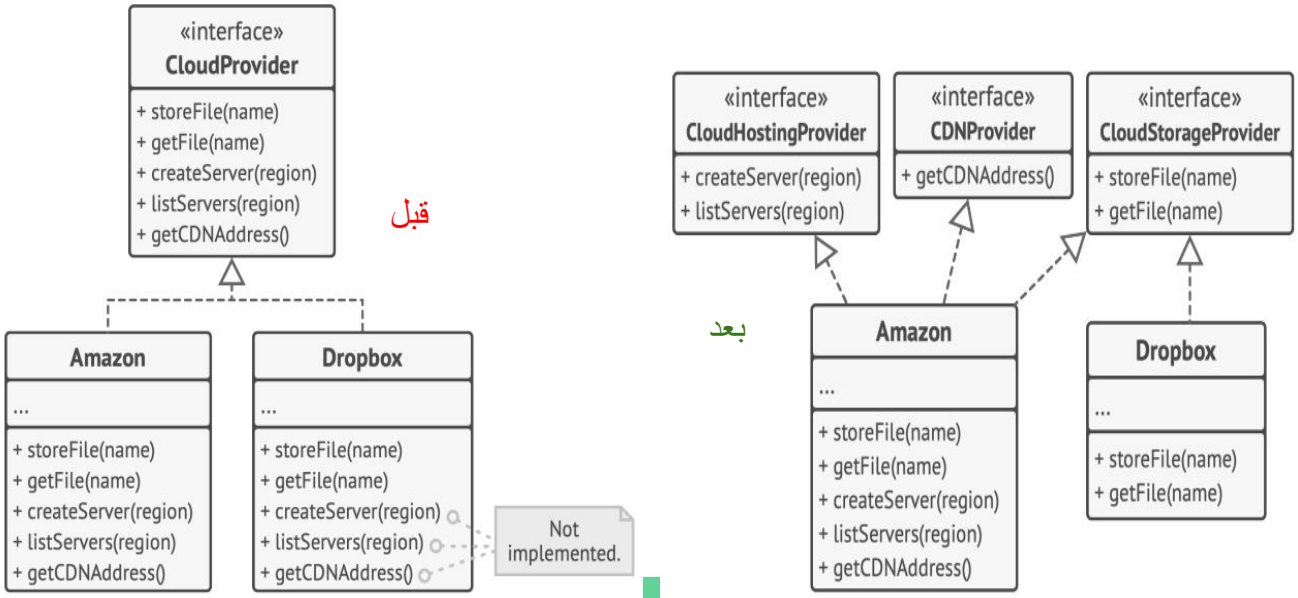
- Interface Segregation Principle: يعد هذا المبدأ من المبادئ السهلة والقوية، فهو يقوم على مبدأ فصل ال interface لأكثر من interface إذا كان ال client لن يستفيد من كل ما في ال interface ولا داعي لوجودها، فمثلا لا تقم بإنشاء interface واحدة تحتوي ال method الخاصة بحساب الوزن مع ال method الخاصة بجلب شعار شركة ما في نفس ال interface file!، والسبب في ذلك أن ال client سيكون ملزم بتنفيذ جميع ال method الموجودة بال interface، ويكون الحل هنا عادة بكتابة ال interface فارغة مثلا ولا تقوم بأي وظيفة!، وهذا الحل ليس لطيفا!، لما نقوم بكتابة جميع ال methods وهي لن تستخدم في معظم الأحيان أو في كثير من ال class؟ بينما يمكننا بكل بساطة فصلها إلى interface مستقلة؟! ...

لنشاهد المثال =>

SOLID Principles

مثال: في هذا المثال تخيل أننا نرغب باستخدام أكثر من ال Cloud Provider، ولنقل أننا اخترنا ال Amazon & Dropbox، إذا نظرنا لمجموعة من ال method المشتركة والتي نرغب في تطبيقها والمتواجدة عند الطرفين معا، ولتكن 5 دوال على سبيل المثال...، لاحظ في ال interface الأول وقبل تطبيق هذا المبدأ، ستشاهد أن الدوال ال 5 تم تطبيقها في ال 2 classes مع أن ال dropbox لا يقدم 3 خدمات من أصل 5، وبهذا كان هذا التكرار دون فائدة!، لكن بعد تطبيق المبدأ لاحظ الفرق!، كل class نفذ ما يقدمه من خدمات فقط!

* ملاحظة: في لغات البرمجة يمكن عمل Implement لأكثر من Interface لنفس ال class

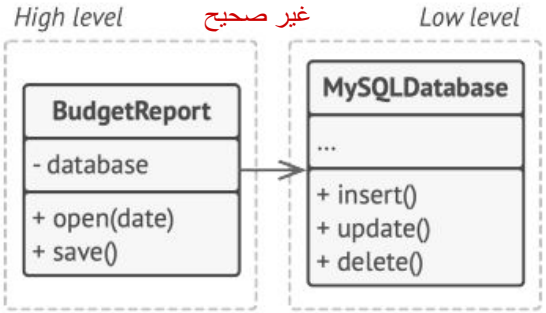


SOLID Principles

- Dependency Inversion Principle: هذا المبدأ يقدم حلول لمشاكل كثيرة يمكن أن تحدث في المستقبل ويعالجها قبل حدوثها، ومع بساطة هذا المبدأ إلى أنه مهم جدا، هذا المبدأ يقوم على رفض وجود أي اعتمادية مباشرة بين ال LOW-LEVEL-CLASS وال HIGH-LEVEL-CLASS، بل يجب على كلاهما الاعتماد على ال Abstraction!، وأشهر مثال يتكرر على هذا المبدأ وكثيرا ما نقوم به قبل البدء بأي مشروع هو ربط ال class التي سنقوم ببنائها مع قاعدة البيانات!، لكن، قبل الحديث عن هذا دعونا نوضح بعض ما ذكرناه بالأعلى:
  - LOW-LEVEL-CLASS: يقصد بها جميع ال class التي تتعامل مباشرة مع العمليات الأساسية لأي وظيفة يحتاجها المشروع، مثل نقل البيانات من خلال ال network أو التعامل مع ال disk أو عمليات الإتصال مع قواعد البيانات ونحو ذلك...
  - HIGH-LEVEL-CLASS; يقصد بها جميع ال class التي تحتوي بثناياها ال Business Logic الخاص بالمشروع والعمليات المعقدة التي تجري بداخله، وتتصل هذه ال class مع ال low-level-class لتنفيذ العمليات الخاصة بها...

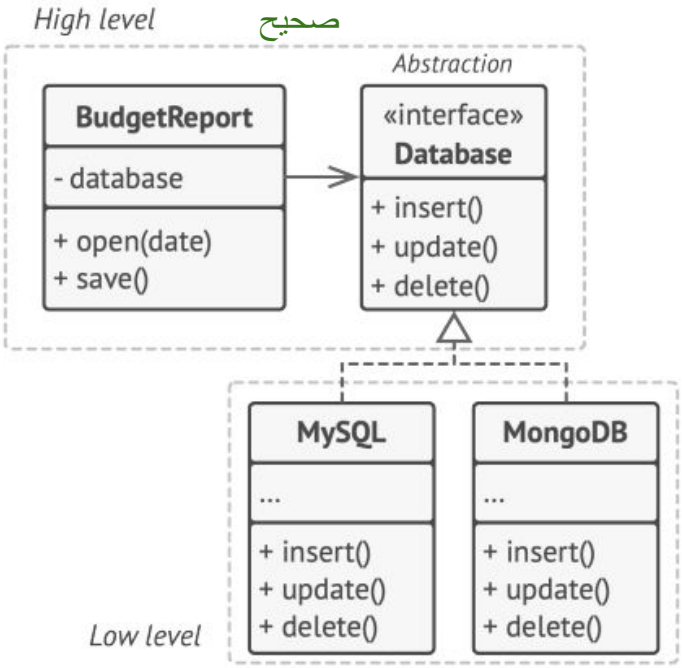
SOLID Principles

مثال: بعد ما تعرفنا على فكرة هذا المبدأ لنحاول تطبيقه على قواعد البيانات، لو فرضنا أن أنك ترغب بإعداد تقارير للميزانية، وستقوم بإدخال البيانات والاستعلام عنها من خلال قواعد البيانات، فماذا ستفعل؟



إذا قمت بالقيام بهذه العملية بالطريقة الموجودة في هذه الصورة، فاعلم أنك خالفت المبدأ الذي نتحدث عنه، خطورة هذه المخالفة تنعكس في العديد من المحاور، لكن قد يكون أهمها، ماذا لو اضطررت لتغير نوع قاعدة البيانات من mysql إلى mongo؟ وماذا سيحدث لو صدر إصدار جديد من قاعدة البيانات وترغب التحديث؟..إلخ هذه الاحتمالات وغيرها ستقودك إلى طريق واحد، هو أن ال High level سيتوقف عن العمل!، ستحتاج إلى تطوير كل مهمة على حدا!، فال High يعتمد على Low بشكل مباشر!، لكن الحل، هو باستخدام مفهوم ال Abstraction، أما الطريقة الصحيحة فهي:

SOLID Principles



في هذه الصورة تم تنفيذ المبدأ بنجاح، وبناءا على ذلك فقد أصبح ال High مرتبط بال Abstraction، وال Low مرتبط بال Abstraction، بهذا التمثيل فيمكن لأي قاعدة بيانات نرغب بإضافتها أو تحديثها من إضافتها بكل سهولة كما يمكن استخدام الدوال المشتركة من ال Abstraction حتى ولو اختلفت الدوال بقواعد البيانات المختلفة!، هذا المبدأ أنقذنا من مشاكل متعددة في مشاريع حقيقة توسعت واحتاجت إلى الكثير من هذه النماذج!، والآن بكل ثقة يمكننا القول أن إضافة أو حذف أو تغيير أي قاعدة بيانات لا يتجاوز تكلفت تغييرها سوى الوقت اللازم لربط الدوال الخاصة بها! والموضوع سهل بشكل لا يوصف!...

فلْتعلمْ أن ما أُمرت به هو ما يتوجب عليك القيام به، ولا فرق بين أن تكون مع الناس أو بمفردك! لهذا لا تلتفت لمن هم حولك وتعلق إيمانك وأفعالك بهم، فإن هذا لن ينفعك بشيء، بل يضرك بامتناعك عن أداء الواجب الذي عليك!

ملاحظات مهمة على ما سبق حول مبادئ التصميم بأنواعها...

1. ما ذكرناه من قواعد ومبادئ هي نماذج من طرق تفكير لحل مشاكل/تحديات معينة، لكنها بذات الوقت ليست عقيدة يجب التمسك بها في كل الأوقات، ولا أظن أن هناك شركة طبقت كل هذه المعايير بكل حيثياتها في كل الأوقات وفي كل الأماكن!، لكن كبرى الشركات والمبرمجين المتميزين يحاولون تطبيق ما أمكنهم حالما تعرضوا لمشاكل/تحديات قد تعيقهم الآن أو في المستقبل، كما أنهم يدركون بناءا على خبرتهم ماذا سيستخدمون من هذه المبادئ عند تعرضهم لأي مشكلة أو تحدي، وهنا يكمن الجمال، وهو استخدام المبدأ المناسب في المكان المناسب والوقت المناسب.
2. قد تجد تداخل بين المبادئ وأن الحلول التي ستقوم بتنفيذها هي فعليا تطبيق لأكثر من مبدأ معا!، بل إن الكثير من المبادئ تتداخل معا!
3. الهدف من المبادئ التي تم ذكرها هو تقليل التعقيد المتوقع وعدد المشاكل المحتملة قبل حدوثها، والتفكير في المستقبل، لكن هذا لا يعني أيضا زيادة التعقيد بشكل كبير أو بشكل لا يستحقه المشروع، فقد تدخل تحت ال "over-engineering".

ملاحظات مهمة على ما سبق حول مبادئ التصميم بأنواعها...

4. وجود Bugs في أحد الأماكن لا يعني تجاوزه ومتابعة العمل من خلاله أو إنشاء class آخر مثلا ووراثة المشكلة مع الخصائص ومن ثم معالجته في ال class الجديد!، لأن ذلك يخالف المبادئ بكل تأكيد، ال Bug يجب أن يتم معالجتها في مكانها وعدم نقلها للأبناء!
5. لكل مبدأ إيجابيات وسلبيات قد تحدث بسبب استخدامه، والتي قد تتغير بناءا على حجم المشروع ونوعه، لذلك يجب أن تدرك ما سيؤول إليه المشروع وما يناسبه من مبادئ قبل اختياره.
6. ال Design Pattern بذاتها عبارة عن مبادئ!، عندما يأتي دور الحديث عنها ستجد أننا نتحدث عن مبادئ وطرق تفكير لحل المشكلات والتحديات التي وجدت لحلها!
7. أنت مبرمج، والمبرمج سيفكر بطريقة برمجية لحل المشكلات، وسيوسع من آفاقه من خلال الاطلاع على حلول الغير وعقولهم، وقد يجد أن الحل هو ضمن أحد المبادئ الموجودة!، وقد تجد أنك بحاجة لمبدئ جديد يناسبك فتقوم بمشاركته والانتفاع من خلاله!

ملاحظات مهمة على ما سبق حول مبادئ التصميم بأنواعها...

8. معظم المبادئ التي تم تصميمها والتي تحدثنا عنها أو سنتحدث عنها ستجعل من بيئة العمل سهلة ومستقرة ومرنة، لكنها لن تعطيك الإجابة عن أفضل طريقة وأفضل مبدأ يمكنك استخدامه لأفضل سهولة أو مرونة أو استقرار!، لأن هذا الأمر يختلف باختلاف طبيعة المشروع ونوعه!، لكن يمكنك أن تجد الإجابة التي تعتقد بأفضليتها بناءا على فهمك لهذه المبادئ ومميزات وعيوب كل واحدة منها...
9. ما تحدثنا عنه من مبادئ وما سنتحدث عنه لا يمثل شيفرة برمجية يمكنك نسخها ومن ثم لصقها!، بل يمثل مفهوم وطريقة تفكير لصياغة الشيفرة البرمجية التي تملكها لتوافق هذه المبادئ!، لذلك لا تتعب نفسك بالبحث عن Copy/Paste لتوافق ما كتبته، بل ابحث عن أمثلة وكتب ومقالات ومدونات تتحدث عن المبدأ وحاول فهمها وتطبيقها، وبهذا تفلح!

فهرس ال Design Patterns

لقد تحدثنا سابقا عن أشهر ثلاث مجموعات لل Patterns، والآن سنحدد ال Patterns الموجودة تحت كل مجموعة والتي سيتم شرحها خلال هذه الشرائح، ثم سننطلق لفهمها بالترتيب، فهل أنت مستعد لهذه المتعة؟!

1. **Creational patterns**:
   a. Factory Method
   b. Abstract Factory
   c. Builder
   d. Prototype
   e. Singleton
2. **Structural patterns**:
   a. Adapter
   b. Bridge
   c. Composite
   d. Decorator
   e. Facade
   f. Flyweight
   g. Proxy
3. **Behavioral patterns**:
   a. Chain of Responsibility
   b. Command
   c. Iterator
   d. Mediator
   e. Memento
   f. Observer
   g. State
   h. Strategy
   i. Template Method
   j. Visitor

Creational Design Patterns

أول مجموعة لدينا من ال Design Patterns هي مجموعة ال Creational، هذه المجموعة تقدم طرق مختلفة ومتنوعة لإنشاء ال object وبأكثر من أسلوب، وهذا بدوره يزيد من مرونة الشيفرة البرمجية والقدرة على إعادة استخدامها في ال Patterns التي تندرج تحت هذه المجموعة...

Creational Design Patterns - Factory Method

ال Factory Method هو أول Pattern سنتحدث عنه في هذه المجموعة، هذا ال Pattern يقدم ال Interface لتكون هي الوسيلة لإنشاء ال Object الخاصة بال superclass ومن خلالها، ولكن في ذات الوقت، يسمح لل subclass من تعديل نوع ال object الذي يرغب بإنشائه.

هذا التعريف ببساطته قد يبدو غامضا وغير مفهوم، لكن مع الأمثلة ستكتشف كم أن هذا ال Pattern مذهل، وسهل الفهم، لذلك، دعنا أولا نتعرف على المشكلة التي جاء لحلها هذا ال Pattern، وكيفية الحل المناسب لهذه المشكلة والتي تحدثنا عنها من خلال التعريف...

Creational Design Patterns - Factory Method

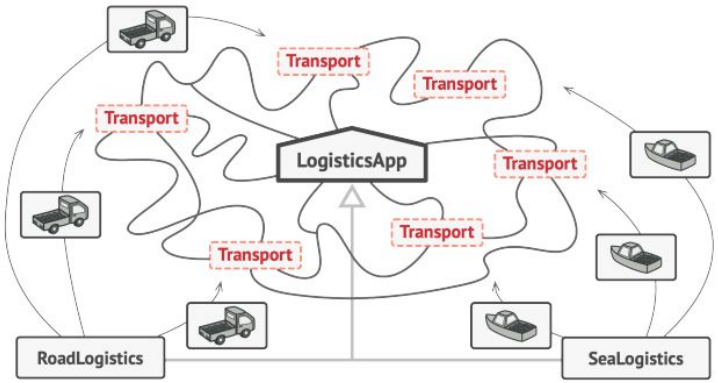
المشكلة:

تخيل أن لديك تطبيق لإدارة الخدمات اللوجستية، الإصدار الأول من هذا التطبيق صمم ودعم النقل من خلال الشاحنات فقط، وذلك يعني الطريق البري، فهذا يعني أن الشيفرة البرمجية الخاصة بهذا التطبيق ستصب وتتركز داخل ال Trucks class!، ومن هنا تبدأ المشكلة!، فعندما تتوسع الشركة وتكبر قد تأتيها عروض للخدمات البحرية، وأنت ملزم بتطبيق هذه التغييرات على المشروع حتى تقدر على مجاراة هذه التغييرات، وستجد نفسك ملزما بالقيام في التعديلات نفسها مرة بعد أخرى، وستقوم بإضافة عدد كبير من الشروط وذلك للتحقق من وسيلة النقل لتقوم بتنفيذ عمليات النقل وإدارتها بشكل صحيح، وهذا سيجعل من الشيفرة البرمجة سيئة ومليئة بالشروط والتي قد تتسبب بمشاكل كثيرة ومتنوعة كلما كبر حجم المشروع...

Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Creational Design Patterns - Factory Method

الحل: ال Factory Method Pattern يقدم حلا جميلا لهذه المشكلة!، والحل يكمن من خلال بناء Factory Method، هذه ال method ستحل محل ال object construction call، فبدلا من أن تعتمد على ال new operator لعمل call خاص بال object بشكل مباشر، نقوم باستخدام ال new operator من خلال ال factory method، لهذا فهناك نقطتان أساسيتان في ال Factory Method، وهي أن ال subclass يمكن أن يقوم بعمل return لأكثر من type من فئة معينة، لذلك يجب أن تكون هذه ال types مشتركة جميعها ب interface أو base class، والنقطة الثانية أن ال factory method الموجودة داخل ال base class يجب أن ترجع نفس ال type الذي تم تعريفه بال interface...،
أعلم أنك لم تفهم ما سبق :P، لذلك دعنا نرى مثالا عمليا على هذا ال pattern...

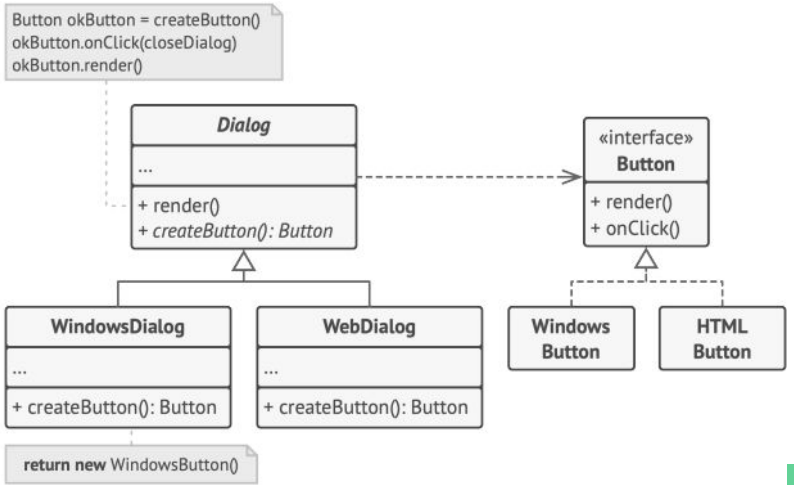


As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.

Creational Design Patterns - Factory Method

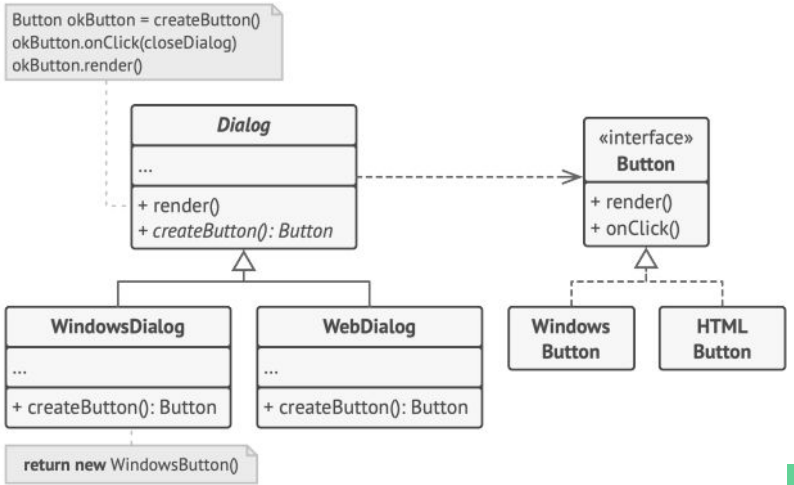
مثال: من الأمثلة الجميلة على ال Factory Method ال Dialog، فمن منا لم يستخدم ال Dialog أو يراها؟!،
لنفرض أنك بحاجة لبناء cross-platform Dialog، وهذا يعني أن هذا ال Dialog سيعمل على الويب،
والموبايل، وأنظمة التشغيل كلينكس وويندوز...، فإذا نظرنا إلى طبيعة هذا المثال وجدنا أنها تقع ضمن المشكلة التي
أقترح ال Factory Method لأجل حلها، ونجد أننا هنا لن نهتم بما سيتم تنفيذه عند ال client بل المهم كيف يمكن
التعامل مع ال dialog إذا اختلف ال type الخاص به بناءا على
ال platform، فجميع هذه ال Dialogs فئات تشترك
جميعها بال type بكونها ال Dialog...، من هنا بدأ
الجمال...في هذه الصورة نجد ما ذكرناه سابقا:

الشرح => الشريحة التالية



Creational Design Patterns - Factory Method

ال Dialog في الصورة يمثل ال Abstract Class، وهو الفئة الأساسية، ويحتوي ال Factory Method وهي ال createButton()، هذه ال factory method ورثها ال WindowsDialog وال WebDialog وأي platform أخرى يمكن إضافتها، وسيتم استخدام ال factory method لإنشاء ال Buttons بما يناسب ال type الخاص بهذه الفئة، وبهذا فإن WindowsDialog سيقوم بإنشاء ال button، وال WebDialog كذلك وكل حسب ما يناسبه، وجميع هذه الأنواع سترجع
ال Button والتي تمثل ال common بين ال Dialogs المختلفة وال Abstract Dialog، والجزء الثاني المهم هو ال Button Interface، فهو يمثل مجموعة ما يشترك به كل من ال abstract class، وال subclass



أعتقد أن الموضوع أصبح سهلا الآن ^_*، لكن لنكتب شيفرة برمجية توضح هذا الكلام لتصبح الرؤيا كاملة...

Creational Design Patterns - Factory Method

1

```php
/**
 * The Creator class declares the factory method
 * 2nees.com
 */
abstract class Dialog
{
    /**
     * factory method.
     */
    abstract public function createButton(): Button;

    public function render(): string
    {
        // Call the factory method to create a Button object.
        $button = $this->createButton();
        // Now, use the button object.
        $result = "My Button: " . $button->showText();
        // return result
        return $result;
    }
}
```

تم إنشاء ال creator الأساسي والذي بدوره يملك ال factory method...

2

```php
/**
 * Concrete Creators override the factory method in order to change the
 * resulting Buttons type.
 * 2nees.com
 */
class WindowsDialog extends Dialog
{
    public function createButton(): Button
    {
        return new WindowsButton();
    }
}

class WebDialog extends Dialog
{
    public function createButton(): Button
    {
        return new HtmlButton();
    }
}
```

تم إنشاء ال classes الخاصة بال platforms المختلفة ومن ثم عمل call لل type المناسب والذي سيجعل من ال factory method تستقبله وتطبقه....

Creational Design Patterns - Factory Method

3

تم إنشاء ال interface والذي بدوره يمتلك جميع ال operation المشتركة بين كل ال objects الخاصة بال creator وال subClass

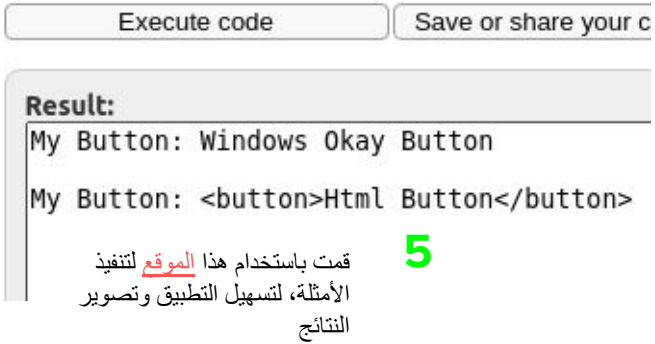
```php
/**
 * 2nees.com
 */
interface Button
{
    public function showText(): string;
}

class WindowsButton implements Button
{
    public function showText(): string
    {
        return "Windows Okay Button";
    }
}

class HtmlButton implements Button
{
    public function showText(): string
    {
        return "<button>Html Button</button>";
    }
}
```

4

الشيفرة البرمجية أصبحت جاهزة، وقمنا بعمل call لل Dialog حتى نرى النتائج ^_^

```php
// Call the client tesult code
function clientCode(Dialog $dialog)
{
    echo $dialog->render();
}

// Create Windows Dialog
clientCode(new WindowsDialog());
echo "\n\n";
// Create Web Dialog
clientCode(new WebDialog());
```

Creational Design Patterns - Factory Method



هل أدركت كم كان الموضوع سهلا وجميلا؟!، شاهد النتائج في الصورة المرفقة...، ويمكنك الحصول على المثال من خلال هذا الرابط

بناءا على ما سبق، يمكن القول أن هذا ال pattern يتكون من 4 أجزاء:

1. Product: وهو يمثل ال interface (ال Button في مثالنا) والذي يكون Common لكل ال Objects التي يتم إنتاجها من خلال creator وال subclass الخاصة به.
2. Concrete Products: وهو يمثل عمليات ال implementations المختلفة لل Product (ال WindowsButton وال HtmlButton في مثالنا)
3. Creator: وهو يمثل ال class الحاوي ال Factory Method، ويجب أن تكون هذه ال Factory Method ترجع نفس ال Type الخاص بال Interface (في مثالنا Dialog Class)
4. Concrete Creators: ال subclass التي تقوم بعمل override لل Factory Method لترجع أنواعا مختلفة من ال Product. (في مثالنا WindowsDialog و WebDialog).

Creational Design Patterns - Factory Method

والسؤال الآن، متى أستخدم هذا ال Pattern؟

الجواب:
استخدم هذا ال Pattern إذا لم يكن لديك معرفة مسبقة بطبيعة ال Types وال Dependency الخاصة بال Object الذي ستقوم بانشائه وستتعامل معه، ومن الأمثلة على ذلك في الحياة العملية:
إذا كنت ترغب في بناء notification داخل تطبيق ولا تدرك أنواع ال notification أو عددها وطبيعتها(SMS, Email...الخ)، ومن الأمثلة كذلك إنشاء اتصال على ال Social Network ومن ثم عمل login و create post...، بهذه الطريقة وفي أي من هذه الحالات ستجد أن هذا ال pattern سيجعل من الموضوع سهلا وسيطبق مجموعة من المبادئ التي تحدثنا عنها سابقا...دعونا نشاهد مميزات وعيوب استخدام هذا ال Pattern:

Creational Design Patterns - Factory Method

المميزات:

- الارتباط بين ال Creator وال Concrete Products ارتباط غير وثيق، أي أن ال Creator وال Concrete Products الاعتمادية فيما بينهم قليلة، والتعديل على أحد هذه ال classes لا يتطلب تعديل على ال class الآخر...، باختصار فإن هذا ال pattern تتجنب ال tight coupling.
- هذا ال Pattern يطبق مفهوم ال Single Responsibility Principle بشكل جميل، مما يجعل عملية ال support سهلة.
- هذا ال Pattern يطبق مفهوم ال Open/Closed Principle بشكل رائع، فأنت يمكنك إضافة أي product دون الخوف من ال breaking code لأي client.

العيوب:

- كل ما زاد عدد ال Product زاد التعقيد، وذلك لأن كل product يلزمك بإنشاء subclasses جديدة...

Creational Design Patterns - Factory Method

والآن نقطة مهمة في أي Design Pattern، إن أهمية فهم ال Design Pattern لا تقل أهمية عن فهم العلاقة بين ال Design Patterns، فمعرفة ما هو أنسب Design Pattern يمكن استخدامه لحل مشكلة معينة والفروق بين هذا ال Pattern مع ال Patterns الأخرى والتي تتشابه فيما بينها إلى حد كبير هو أهم ما يجب أن تجنيه بعد قرائتك لل Design Pattern، ولهذا سنتحدث باختصار عن بعض الملاحظات حول كل Design Pattern سنتطرق له والتي تتعلق بعلاقة هذه ال Pattern مع غيره...

علاقة ال Factory Method مع غيره من ال Pattern:

- الكثير من ال Designs عادة ما تبدأ باستخدام هذا ال Pattern، لأنه مستوى التعقيد الخاص به قليل، ويوفر قدرة عالية على عمل Customization، لكن يتطور هذا ال Design مع الوقت ليصير Abstract Factory أو Prototype أو Builder، والتي ستوفر مرونة أكبر، لكنها ستزيد من مستوى التعقيد

Creational Design Patterns - Factory Method

- غالبا ما تبنا ال Classes الخاصة بال Abstract Factory على مجموعة من Factory Methods، كما يمكن استخدام ال Prototype لتكوين أو تشكيل هذه ال Methods داخل ال Class.
- يمكن استخدام ال Factory Method مع ال Iterator جنبا إلى جنب وذلك لجعل مجموعة من ال subclasses تقوم بإرجاع أنواع مختلفة من ال iterators والتي تتوافق مع مجموعة ال subclasses.
- ال Factory Method مبني على مفهوم الوراثة، لكنه لا يلزمك بال initialization step، بينما ال Prototype لم يبنى اعتمادا على مفهوم الوراثة، لذلك فقد تخلص هذا ال Pattern من عيوب الوراثة، لكن هذا الأمر تطلب زيادة في مستوى التعقيد الخاص بعملية نسخ ال Object عند ال initialization.
- ال Factory Method يمثل نوع مخصص من ال Template Method، لذلك فهمك لهذا ال Pattern سيقدمك خطوة باتجاه ال Template Method.

سبحان الخالق العظيم القدير، الذي خلق هذا الكون المعجز، الذي خلقنا فيه لغاية عظيمة ولم نخلق عبثا -سبحانه وتعالى وجل وعز بقدرته-.

كلما قرأنا أكثر، ورأينا الأساليب التي طرحها البشر؛ لحل المشاكل التي تواجههم في عمارة هذه الأرض، وجدنا أثرا لعظمة الله -سبحانه وتعالى- وقدرته، فتخيل كيفية تنفيذ محاكاة لما خلقه الله -سبحانه وتعالى- ستجعلك عاجزا متعجبا من قدرة الله -سبحانه وتعالى- العظيمة، والتي ستقودك ليقين عظيم؛ بأنه لا يمكن لأي أسلوب من أساليب البشر أن يحاكي الخلق بتعقيداته ومن دون أي خطأ!، وهذا كله عدا عن برمجة ومحاكاة ما خلقه رب العزة جل في علاه!، فكيف ينفي من أدرك هذا وجود خالق عظيم عليم قدير؟!، سبحان الله عما يصفون!، وسبحان من قال في كتابه العزيز: "إِنَّمَا أَمْرُهُ إِذَا أَرَادَ شَيْئًا أَن يَقُولَ لَهُ كُن فَيَكُونُ (82)" وقال تعالى: "هُوَ الَّذِي جَعَلَ لَكُمُ الْأَرْضَ ذَلُولًا فَامْشُوا فِي مَنَاكِبِهَا وَكُلُوا مِن رِّزْقِهِ ۖ وَإِلَيْهِ النُّشُورُ (15)"

لذلك يا أخي، انظر إلى خلق الله -سبحانه وتعالى-، وتفكر وتدبر، ثم أحسن، وإحسانك سيكون نابعا من شعورك بعظمة الله -سبحانه وتعالى-، والذي سيقودك لعبادته والامتثال لأوامره، وسيقودك لعمارة الأرض والتي بها تتحقق المنفعة للبشر!، فالحمد لله رب العالمين.

Creational Design Patterns - Abstract Factory

ال Pattern الثاني ضمن هذه المجموعة هو ال Abstract Factory، هذا ال Pattern يتيح لك إنشاء عائلات من ال objects المترابطة مع بعضها البعض وبدون تحديد ال concrete classes!.

ومن التعريف، فإن العائلات هي جمع العائلة، وكل عائلة ترتبط فيما بينها بقيم مشتركة تختلف عن العائلات الأخرى!، لكن في نفس الوقت، كل العائلات تشترك بأصل المادة!، وإنما الاختلاف فيما بينهم في القيم!، ومن هذا المفهوم، نعلم أن هذا ال Pattern قائم على إنشاء عائلات من ال Object التي تشترك مع بعضها وترتبط فيما بينها ودون تحديدها بفئة معينة تغير من هذا التركيب...

Creational Design Patterns - Abstract Factory

المشكلة:

تخيل أن لديك مجموعة من ال Products، هذه ال Products تباع على شكل مجموعة (عائلة) مثلا كرسي + أريكة + طاولة خدمة، لكن هذه العائلة ليست الوحيدة، فالكرسي والأريكة وطاولة الخدمة يمكن أن توجد ضمن عائلة التصميمات الحديثة أو التصميمات الكلاسيكية أو التصميمات المكتبية...في مثل هذه الحالة، فالمشكلة تكمن في أن الكرسي هو ذاته كرسي والرابط المشترك فيه هو الجلوس، لكن طبيعة أو شكل هذا الكرسي سيختلف بناءا على التصميم (سيختلف باختلاف العائلة)، وكذلك الحال بالنسبة لجميع أفراد العائلة، لذلك، ما نحتاجه هو طريقة نبني من خلالها ال Object بحيث يتطابق هذا ال Object مع ال Objects الأخرى بنفس العائلة، فليس من المنطق أن يأتي لديك Object للكرسي من عائلة التصميم الحديث و Object آخر من عائلة التصميمات الكلاسيكية!، ثم تقول عن هذه التركيبة أنهم عائلة مترابطة!، بناءا على هذه المشكلة أو منعا لظهور هذه المشكلة ظهر الحل المناسب...

Creational Design Patterns - Abstract Factory

الحل: ال Abstract Factory يقدم حلا جميلا لهذه المشكلة، والفكرة التي قدمها تتلخص بإنشاء interfaces لكل Product ضمن عائلة ما، فمثلا الكرسي الحديث والكرسي الكلاسيكي يجب أن يتبعوا ال Interface الخاصة بالكرسي، والأريكة الحديثة والأريكة الكلاسيكية يجب أن تتبع ال Interface الخاصة بالأريكة...وهكذا، بعد هذه الخطوة نقوم بعمل ال Abstract Factory وهو Interface فيه قائمة بكل ال creation methods، مثل createChair و createTable وهكذا...، وهذه ال Method بكل تأكيد ستقوم بإرجاع abstract product type (الكرسي أو الأريكة أو الطاولة)...بهذا نكون ضمنا إنشاء هذه ال Product بأنواعها المختلفة، لكن تبقى علينا خطوة واحدة!، وهي كيف سنضمن أن تكون هذه الأنواع كل حسب عائلتها؟، هنا تأتي النقطة الأخيرة وهي أننا نحتاج إلى إنشاء العائلات، وهي العائلة الحديثة والعائلة الكلاسيكية والعائلة المكتبية...إلى آخره، هذه العائلات هي factory class من ال Abstract Factory interface، مثلا عائلة ال ModernFactory والتي يمكنها فقط إرجاع ModernChair و ModernTable و ModernSofa...والآن، ما رأيك أن نذهب لنشاهد مثالا عمليا يشرح هذه الفكرة، ولنشاهد كم أن هذا ال Pattern شيق؟

Creational Design Patterns - Abstract Factory

مثال (مصنع الأثاث):

في هذا الجزء قمن بانشاء ال interface الخاصة بالعائلة، ويسمى بال Abstract Product

1

```php
/**
 * Interface Chair, Sofa, ServiceTable is the Abstract Product
 ** and all variants of the product must implement this interface
 * 2nees.com
 */
interface Chair {
    public function getChairName() : string;
}

interface Sofa {
    public function getSofaName() : string;
}

interface ServiceTable {
    public function getServiceTableName() : string;
}
```

Creational Design Patterns - Abstract Factory

مثال:

```php
/**
 * Classes ModernChair, ModernSofa, ModernServiceTable, Cla
ClassicalServiceTable
 * Various implementations of abstract products
 * 2nees.com
 */
class ModernChair implements Chair {

    public function getChairName(): string
    {
        return "Modern Chair Name";
    }
}

class ModernSofa implements Sofa {

    public function getSofaName(): string
    {
        return "Modern Sofa Name";
    }
}

class ModernServiceTable implements ServiceTable {

    public function getServiceTableName(): string
    {
        return "Modern Service Table Name";
    }
}
```

2

```php
/**
 * Classes ModernChair, ModernSofa, ModernServiceTable, ClassicalChair, ClassicalSofa,
ClassicalServiceTable
 * Various implementations of abstract products
 * 2nees.com
 */

class ClassicalChair implements Chair {

    public function getChairName(): string
    {
        return "Classical Chair Name";
    }
}

class ClassicalSofa implements Sofa {

    public function getSofaName(): string
    {
        return "Classical Sofa Name";
    }
}

class ClassicalServiceTable implements ServiceTable {

    public function getServiceTableName(): string
    {
        return "Classical Service Table Name";
    }
}
```

في هذا الجزء يتم عمل implementation لل abstract product، والتي يجب أن تنتمي لكل عائلة، وتسمى هذه ال classes ب: Concrete Products

Creational Design Patterns - Abstract Factory

مثال:

في هذا الجزء يتم عمل implementation لل method وكل class حسب النوع الذي يناسبه، وتسمى هذه ال class ب Concrete Factories

4

```php
/**
 * Produce the Object Family - Modern Family
 * 2nees.com
 */
class ModernDesign implements FurnitureFactory
{
    public function createChair(): Chair
    {
        return new ModernChair();
    }
    public function createSofa(): Sofa
    {
        return new ModernSofa();
    }
    public function createServiceTable(): ServiceTable
    {
        return new ModernServiceTable();
    }
}

/**
 * Produce the Object Family - Classical Family
 * 2nees.com
 */
class ClassicalDesign implements FurnitureFactory
{
    public function createChair(): Chair
    {
        return new ClassicalChair();
    }
    public function createSofa(): Sofa
    {
        return new ClassicalSofa();
    }
    public function createServiceTable(): ServiceTable
    {
        return new ClassicalServiceTable();
    }
}
```

يمثل هذا الجزء ال Abstract Factory

3

```php
/**
 * This is AbstractFactory interface
 * 2nees.com
 */
interface FurnitureFactory
{
    public function createChair(): Chair;
    public function createSofa(): Sofa;
    public function createServiceTable(): ServiceTable;
}
```

Creational Design Patterns - Abstract Factory

مثال: والآن، إليك الرابط الخاص بالمثال ^^

Result:

```
Modern Chair Name
Modern Sofa Name
Modern Service Table Name
==========================================

Classical Chair Name
Classical Sofa Name
Classical Service Table Name
==========================================
```

6

النتيجة *_^

5

```php
/**
 * Client Function to print test result code
 * @param FurnitureFactory $factory
 */
function clientCode(FurnitureFactory $factory){
    echo "
            {$factory->createChair()->getChairName()}\n
            {$factory->createSofa()->getSofaName()}\n
            {$factory->createServiceTable()->getServiceTableName()}\n
            ================================================ \n
        ";
}

clientCode(new ModernDesign());
clientCode(new ClassicalDesign());
```

ملاحظة: هذا ال function الغرض منه تبسيط ال call،
لكن يمكنك تنفيذ ال call بطرق أكثر احترافية مثل إنشاء render *_^

Creational Design Patterns - Abstract Factory

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Abstract Products: يتم من خلال هذا الجزء إنشاء ال interface الخاصة بالعائلة والتي تمثل كل منها عنصر مستقل، وفي مثالنا فإن ال Chair, Sofa وال ServiceTable يمثلن هذا الجزء.
2. Concrete Products: في هذا الجزء يتم عمل ال implementation الخاص بال Abstract Product بطريق مختلفة لتأدية المطلوب حسب ما صممت لأجله، والتي يجب أن تطبق على جميع ال AbstractProduct، في مثالنا السابق كانت هذه تمثل ال ModernChair, ModernSofa, ModernServiceTable, ClassicalChair, ClassicalSofa, ClassicalServiceTable
3. Abstract Factory: وهو يمثل ال interface الأساسي الخاص بهذا ال Pattern، والذي يتم من خلاله إنشاء ال Method الخاصة بكل Abstract Product
4. Concrete Factories: في هذه ال Classes يتم عمل implementation لل method التي تم إنشائها في ال Abstract Factory، فمثلا قمنا بعمل ClassicalDesign وال ModernDesign، والتي بدورها قامت بعمل implementation حسب كل عائلة...

Creational Design Patterns - Abstract Factory

متى يمكنني استخدام هذا ال Pattern:

يمكنك استخدام هذا ال Pattern إذا كانت لديك مجموعة من ال Factory Method، فبدلا من إنشاء concrete Class لكل Factory Method، نقوم بإنشاء عائلات تضم هذه ال Factory Method، ومن خلال هذه الحركة الجميلة، يمكننا إنشاء عائلات مختلفة ودون القلق أو الخوف من أن ال Client سيقوم بإنشاء Object لا يتبع العائلة الخاصة به!، لذلك، إذا كان لديك مجموعة من ال Product المختلفة والمترابطة مع بعضها البعض أو كنت ترغب في طريقة تجعل من الشيفرة البرمجية قابلة للتمدد اعتمادا على مجموعة العائلات الخاصة بال Product فعليك باستخدام هذا ال Pattern.

من الأمثلة في الواقع العملي محلات تجميع الحواسيب، مثلا تخيل أن لديك محل يقوم بتجميع أو صيانة لابتوبات من نوع Dell و HP إلى آخره...، أو مصنع يقوم بتصنيع منتجات مقاومة للحرارة وأخرى غير مقاومة...إلى آخره، فالأول سيكون لديك معالج، كرت شاشة، رام..إلى آخره، والثانية كأس، صحن، مقاوم للحرارة وغير مقاوم...

Creational Design Patterns - Abstract Factory

المميزات:

- من خلال هذا ال Pattern ستكون متأكدا من أن جميع ال Products التي قمت بإنتاجها او قام بإنتاجها ال Client من خلال ال Factory تتوافق فيما بينها لأنها من نفس العائلة.
- من خلال هذا ال Pattern فإنك تتجنب ال tight coupling التي يمكن أن توجد بين ال Client وال concrete products.
- يحقق مبدأ ال Single Responsibility Principle بشكل ممتاز، فيمكنك من خلال ال Pattern كتابة وتقسيم الشيفرة البرمجية على شكل فئات كل فئة تخدم هدف معين كما رأينا في المثال.
- يحقق مبدأ ال Open/Closed Principle بشكل ممتاز أيضا، فإنك لن تخشى من إضافة أي Product على الشيفرة البرمجية الحالية لأنها لن تتأثر بالإضافة الجديدة.

العيوب:

- مشكلة هذا ال Pattern أن مستوى التعقيد يزيد طرديا كما زاد عدد ال product، والسبب في ذلك يعود لإنشاء عدد كبير من ال interface وال classes المرتبطة بهذا ال Pattern.

Creational Design Patterns - Abstract Factory

علاقة ال Abstract Factory مع غيره من ال Pattern:

- عادة ما يبدأ العمل على ال Factory Method، لكن يتطور هذا ال Design مع الوقت ليصير Abstract Factory أو Prototype أو Builder، والتي ستوفر مرونة أكبر، لكنها ستزيد من مستوى التعقيد.

- ال Abstract Factory يركز على إنشاء عائلات مترابطة من خلال العلاقات بين ال Objects، ويقوم ال Abstract Factory بإرجاع ال Product مباشرة، بينما ال Builder يركز على بناء ال Object خطوة بخطوة، لذلك فهو يحتاج إلى وقت أكبر للحصول على ال Product.

- فكرة ال Abstract Factory قائمة على وجود مجموعة من ال Factory Method، لكن يمكن استخدام ال Prototype أيضا من خلال عمل compose لل methods داخل ال classes

- يمكن اعتبار ال Abstract Factory بديلا لل Facade عندما تحتاج إلى إخفاء الطريقة التي يتم بها إنشاء ال Objects في ال subSystems من خلال ال Client

Creational Design Patterns - Abstract Factory

- يمكن استخدام ال Abstract Factory مع ال Bridge، هذا الإندماج يعطي مزية جميلة جدا، وهي إمكانية إخفاء التعقيد عن ال Client من خلال ال Abstract Factory التي حوت بداخلها ال Bridge، بشرط أن تكون هذه ال Bridge لها عمل محدد وتطبيق محدد...

- ال Abstract Factories, Builders and Prototypes يمكن بنائهم من خلال ال Singleton.

فلْتعلمْ أن المسلم يستر على أخيه المسلم، فلا يفضحه ولا يتتبع عوراته.

Creational Design Patterns - Builder

ال Pattern الثالث ضمن هذه المجموعة هو ال Builder، هذا ال Pattern يتيح لك بناء Objects معقدة خطوة بخطوة (Step by Step)، بحيث يتم إنتاج ال type المختلفة من ال Object من خلال نفس ال Code الذي قمت ببنائه، وهذا الأمر جميل جدا، لأنك من خلاله ستتمكن من بناء ال Object بناءا على ما يتناسب مع ال representations المراد ومن خلال نفس البناء الذي قمت بتجهيزه لهذا الغرض...

بناءا على ما ذكرنا في التعريف، فإن ما ستقوم فيه في هذا ال Pattern هو بناء الشيفرة البرمجية بطريقة تمكنك من إنشاء ال Object خطوة بخطوة و/أو تبديله بطريقة سهلة أثناء ال Runtime، وبنفس الوقت يتم تجاوز التعقيد الخاص بالشيفرة البرمجية من خلال الإهتمام بال Object بدلا من الاعتماد على ال constructor الخاص بكل class وإضافة شروط معقدة وكبيرة بداخله...، وحقيقة يقدم هذا ال Pattern حلا ممتازا للكثير من التعقيدات التي قد تحصل إذا وجد ال representations في ال constructor...

Creational Design Patterns - Builder

المشكلة: تخيل أن لديك شركة تعمل في مجال البناء، هذه الشركة بكل تأكيد ستقوم ببناء العديد من المنازل بصيغ مختلفة، بناءا على رغبة العميل وإمكانيات الشركة، فمثلا يمكن للشركة أن تقوم ببناء منزل بسيط، أو منزل مع كراج، أو منزل مع مسبح، أو منزل مع مسبح وكراج، أو منزل مع حديقة، أو منزل فيه حديقة وكراج ومسبح، وقد يكون المنزل من الخشب وقد يكون من الطوب...إلى آخره.

الناظر إلى المشكلة السابقة يعلم كم أن الموضوع صعب وعقد لو قمنا بالتفكير في حل هذه المشكلة بالطريقة الإعتيادية، والسبب في ذلك أن كمية الخيارات الممكنة كبيرة جدا، وتتغير من بناء لبناء حسب الحاجة!، والحل الافتراضي سيكون من خلال ال Class وال Sub Class أو من خلال إرسال جميع القيم من خلال ال constructor، هذه الطرق ستجعل من الشيفرة البرمجية معقدة وتزداد تعقيدا مع مرور الوقت، بالإضافة إلى أن جعل ال constructor مليء بالشروط والقيم ال NULL، فمثلا لو اختار أحدهم منزل بسيط ستكون باقي القيم NULL, NULL...، ومع أن هذا الحل قد يخفف من مشكلة التعقيد الخاص بال Sub class، إلا أنه صنع المشكلة الخاصة بال constructor...

وهنا جاء دور ال Builder...

Creational Design Patterns - Builder

الحل: ال Builder يقدم حلا جميلا لمثل هذه المشاكل أو هذه التعقيدات، فالتعقيدات هنا تتركز عند إنشاء ال Object، لذلك، يقترح عليك هذا ال Pattern بفصل ال Object عن ال class الذي ينتمي له، وبناء Builder Objects، وال Builder Object في مثالنا يمثل ال buildDoor وال buildSwim وال buildGarage وال buildWindow...إلى آخره، وكما تلاحظ فإن كل builder منهم سيقوم ببناء ما يتطلب منه، وأن المنزل المراد بناءه هو عبارة عن مجموعة الخطوات التي ستنفذ لبنائه ^_*، والنقطة المهمة والجميلة هنا، أنك لن تحتاج إلى استدعاء كل ال builder!، بل إنك ستقوم باستدعاء ما يلزمك فقط لتغطية ما تريد بناءه، كما أنك ستتمكن من خلال هذه ال builders أن تتحكم بالمادة المراد البناء من خلالها، فمثلا باب حديد أو باب خشب!، وهنا قد تتسائل، هل سيقوم بذلك ال client مباشرة؟ فالجواب، نعم، يمكن لل client أن يقوم باستخدام هذه ال builders مباشرة، لكن إذا أردت الأفضل فعليك استخدام ال Director!، يمثل ال Director كلاس وظيفته بناء الخطوات المطلوبة لإنشاء المطلوب دون الحاجة لكتابتها من خلال ال client، وهذا يقدم الكثير من الفوائد أهمها أن ال client لن يحتاج لمعرفة ال builders، وستقدم له أنت ما ترغب في بنائه بالطريقة التي نظمت خطواتها مسبقا، وهذا يعني أنك يمكنك مشاركة هذه الشيفرة بأكثر من مكان ولأكثر من برنامج بكل سهولة، كل ما عليك فعله فقط استدعاء ال Director...، وحقيقة طريقة التفكير هذه مميزة...

Creational Design Patterns - Builder

مثال: لنقم الآن في بناء مثال يحاكي فكرة إنشاء builder لبناء قلعة وبناء منزل عادي:

ملاحظة: هذا المثال لمحاكاة المبدأ، وبكل تأكيد يمكن تحسين الشيفرة البرمجية في الواقع العملي، لكن الهدف هو عرض جميع الأفكار الممكنة وبأبسط طريقة...

هذا الجزء هو ال builder، والذي يحتوي بداخله جميع ال construction الخاصة بال product والمشتركة بين جميع ال product...

1

```php
/**
 * This is Builder interface
 * 2nees.com
 */
interface HouseBuilderInterface
{
    public function createWall(): void;
    public function createRoof(): void;
    public function createDoor(): void;
    public function createWindow(): void;
    public function createGarage(): void;
    public function createSwimmingPool(): void;
    public function getResult();
}
```

Creational Design Patterns - Builder

2

```php
/**
 * Simple Abstract Class to share common data between Products
 * Note: its just for example, so that no setter or getter...etc
 * 2nees.com
 */
abstract class HouseAbstracts {
    public $wall;
    public $roof;
    public $door;
    public $window;
    public $garage;
    public $swimmingPool;

    public abstract function buildHome() : void;
}

/**
 * This is Product, not require to follow the builder interface in all result
 * 2nees.com
 */
class House extends HouseAbstracts {
    public function buildHome(): void
    {
        echo "Build {$this->wall} walls" . PHP_EOL;
        echo "Build roof" . PHP_EOL;
        echo "Build {$this->door} doors" . PHP_EOL;
        echo "Build {$this->window} window" . PHP_EOL;
        echo "Building House Completed! Thank You!" . PHP_EOL;
    }
}

class Castle extends HouseAbstracts {
    public function buildHome(): void
    {
        echo "Build {$this->wall} walls" . PHP_EOL;
        echo "Build roof" . PHP_EOL;
        echo "Build {$this->door} doors" . PHP_EOL;
        echo "Build {$this->window} window" . PHP_EOL;
        echo "Build Garage" . PHP_EOL;
        echo "Build Swimming Pool" . PHP_EOL;
        echo "Building Castle Completed! Thank You!" . PHP_EOL;
    }
}
```

في هذا الجزء قمنا ببناء ال Product، وال Product تمثل ال result object، والذي يمكن إنشاؤه من أكثر من Builder، ولا يشترط أن يتبع التسلسل الخاص بال interface.

باختصار، إن الناتج الخاص بال Object سيمثل النتيجة النهائية ل Object من class تم إنشاءه بالشكل التقليدي، لكن في ال Builder تم إنشاء ال Object من خلال Common Construction عن طريق مفهوم ال Builder...

لاحظ ال Home, Castle، عبارة عن class يمكنك أن تقوم بعمل أي action ترغب فيه لتنفيذ أو إكمال ما تريد...

ملاحظة: ليس بالضرورة وجود ال Product، لكن هذا مثال لتوضيح جميع الخيارات الممكنة...

```php
/**
 * This is Concrete Bulders
 * 2nees.com
 */
class HouseBuilder implements HouseBuilderInterface {

    private House $house;
    public function __construct()
    {
        $this->reset();
    }
    public function reset()
    {
        $this->house = new House();
    }
    public function createWall(): void
    {
       $this->house->wall = 4;
    }
    public function createRoof(): void
    {
        $this->house->roof = "Done!";
    }
    public function createDoor(): void
    {
        $this->house->door = 1;
    }
    public function createWindow(): void
    {
        $this->house->window = 2;
    }
    public function createGarage(): void
    {
       $this->house->garage = false;
    }
    public function createSwimmingPool(): void
    {
        $this->house->swimmingPool = false;
    }
    public function getResult(): House
    {
        $result = $this->house;
        $this->reset();

        return $result;
    }
}
```

3

```php
/**
 * This is Concrete Builder
 * 2nees.com
 */
class CastleBuilder implements HouseBuilderInterface {

    private Castle $castle;
    public function __construct()
    {
        $this->reset();
    }
    public function reset()
    {
        $this->castle = new Castle();
    }
    public function createWall(): void
    {
       $this->castle->wall = 32;
    }
    public function createRoof(): void
    {
        $this->castle->roof = "Done!";
    }
    public function createDoor(): void
    {
        $this->castle->door = 8;
    }
    public function createWindow(): void
    {
        $this->castle->window = 16;
    }
    public function createGarage(): void
    {
        $this->castle->garage = true;
    }
    public function createSwimmingPool(): void
    {
        $this->castle->swimmingPool = true;
    }
    public function getResult(): Castle
    {
        $result = $this->castle;
        $this->reset();

        return $result;
    }
}
```

في هذا الجزء من الكود، قمنا ببناء ال Concrete Builders، وال Concrete Builders هي المسؤولة عن إنشاء أكثر من implementation في البناء المطلوب، ويمكن أن يتصل هذا ال Concrete Builders مع ال Product class كما في مثالنا الحالي، ويمكن أن لا يتصل وأن يقدم ال builder المطلوب منه من إنشاء ال object بدون أي داعي لوجود أي product... (ستجد مثال آخر بعد هذا المثال)

Creational Design Patterns - Builder

4

```php
/**
 * This is a Director Class which its execute the building steps in particular sequence
 * 2nees.com
 */
class Director
{
    private HouseBuilderInterface $builder;

    public function setBuilder(HouseBuilderInterface $builder): void
    {
        $this->builder = $builder;
    }

    public function buildYourHouse(): void
    {
        $this->builder->createWall();
        $this->builder->createRoof();
        $this->builder->createDoor();
        $this->builder->createWindow();
        $this->builder->createGarage();
        $this->builder->createSwimmingPool();
    }
}
```

هذا الجزء هو ال Director، وهو إضافة جميلة للتحكم في ترتيب الخطوات الخاصة في البناء، دون الحاجة لكتابتها من قبل ال client، وهذه الحركة اللطيفة تقلل من الأخطاء التي يمكن أن تنتج إذا اشترط الترتيب، كما تحتوي بناء أكثر من دالة للتحكم في ترتيب الخطوات، وكل ما يلزم ال client هو عمل call لل method المناسبة...

ملاحظة: لا يشترط وجود ال Director، وفي بعض الحالات لا داعي لوجوده....

Creational Design Patterns - Builder

في هذا الجزء، قمنا بعمل محاكاة لطريقة استخدام ما قمنا ببنائه،
لاحظ كيف قمنا ببناء Home و Castle...، وكيف قمنا باستخدام
ال director...

5

```php
/**
 * Client code, simulate to build castel and home
 * 2nees.com
 */
$director = new Director();
$homeBuilder = new HouseBuilder();
$director->setBuilder($homeBuilder);
$director->buildYourHouse();
$home = $homeBuilder->getResult();
print_r($home);
$home->buildHome();
echo "========================================" . PHP
$castleBuilder = new CastleBuilder();
$director->setBuilder($castleBuilder);
$director->buildYourHouse();
$castle = $castleBuilder->getResult();
print_r($castle);
$castle->buildHome();
```

Result:

```
House Object
(
    [wall] => 4
    [roof] => Done!
    [door] => 1
    [window] => 2
    [garage] =>
    [swimmingPool] =>
)
Build 4 walls
Build roof
Build 1 doors
Build 2 window
Building House Completed! Thank You!
========================================
Castle Object
(
    [wall] => 32
    [roof] => Done!
    [door] => 8
    [window] => 16
    [garage] => 1
    [swimmingPool] => 1
)
Build 32 walls
Build roof
Build 8 doors
Build 16 window
Build Garage
Build Swimming Pool
Building Castle Completed! Thank You!
```

Creational Design Patterns - Builder

مثال 2: في هذا المثال سنتطرق لفكرة من الواقع، هل فكرت يوما في ال SQL Builder؟، ألم تلاحظ أنه Builder؟ دعنا نحاول تطبيق هذه الفكرة في شكلها البسيط، فهي تطبيق ممتاز لمفهوم ال Builder Pattern، وليكن هذا ال builder لل MySql وال PostgreSQL، فكما تعلم ال syntax يتشابه في كثير من الحالات، ويختلف في بعضها، وال builder هنا وسيلة ممتازة لحل التعقيد الموجود هنا.

1

```php
/**
 * This is Builder interface
 * 2nees.com
 */
interface SQLQueryBuilder
{
    public function select(string $table, array $fields): SQLQueryBuilder;

    public function insert(string $table, array $fields): SQLQueryBuilder;

    public function where(string $where): SQLQueryBuilder;

    public function limit(int $start, int $offset): SQLQueryBuilder;

    public function buildSql(): string;
}
```

Creational Design Patterns - Builder

2

```php
/**
 * This is Concrete Builder
 * 2nees.com
 */
class MysqlBuilder implements SQLQueryBuilder
{
    protected stdClass $query;
    protected function reset(): void
    {
        $this->query = new stdClass();
    }
    public function select(string $table, array $fields): SQLQueryBuilder
    {
        $this->reset();
        $this->query->base = "SELECT " . implode(", ", $fields) . " FROM " . $table;
        return $this;
    }
    public function insert(string $table, array $fields): SQLQueryBuilder
    {
        $this->reset();
        $this->query->base = "INSERT INTO {$table} VALUES( " . implode(", ", $fields) . " )";
        return $this;
    }
    public function where(string $where): SQLQueryBuilder
    {
        $this->query->where[] = $where;
        return $this;
    }
    public function limit(int $start, int $offset): SQLQueryBuilder
    {
        $this->query->limit = " LIMIT " . $start . ", " . $offset;
        return $this;
    }
    public function buildSql(): string
    {
        $query  = $this->query;
        $sql    = $query->base;
        if (!empty($query->where)) {
            $sql .= " WHERE " . implode(' AND ', $query->where);
        }
        if (isset($query->limit)) {
            $sql .= $query->limit;
        }
        return $sql . ";";
    }
}
```

في هذا الجزء من الكود، قمنا ببناء ال Concrete Builders الخاص بال MySql، وفيه مجموعة ال method الموجودة بال interface وقد تم تطبيقها...ولأن هذه ال method تشترك مع ال PostgreSQL فإن ال PostgreSQL سيقوم بوراثة ال MySql وعمل override لل method عند اختلاف ال syntax الخاص به، مثل ال Limit...

3

```php
/**
 * This is Concrete Builder
 * 2nees.com
 */
class PostgresBuilder extends MysqlBuilder
{
    public function limit(int $start, int $offset): SQLQueryBuilder
    {
        $this->query->limit = " LIMIT " . $start . " OFFSET " . $offset;

        return $this;
    }
}
```

Creational Design Patterns - Builder

4

```php
/**
 * Use Builders
 * 2nees.com
 */
$mysql = new MysqlBuilder();
echo $mysql->insert("users", ["'Anees'", "'aneeshikmat@2nees.com'", "30"])->buildSql() .
PHP_EOL;
echo $mysql->select("users", ["name", "email", "age"])
    ->where("age > 18")
    ->where("name LIKE '%Anees%'")
    ->limit(1, 5)
    ->buildSql();

echo PHP_EOL . "=========================================" . PHP_EOL;

$postgres = new PostgresBuilder();
echo $postgres->insert("users", ["'Anees'", "'aneeshikmat@2nees.com'", "30"])->buildSql() .
PHP_EOL;
echo $postgres->select("users", ["name", "email", "age"])
    ->where("age > 18")
    ->where("name LIKE '%Anees%'")
    ->limit(1, 5)
    ->buildSql();
```

لاحظ أن هذا المثال لم يحتاج لوجود Product ولا وجود Director

يمكنك مشاهدة الأمثلة مباشرة من هنا: (قمت برفع 3 أمثلة لتطبيق 3 أفكار مختلفة):

المثال الأول

المثال الثاني

المثال الثالث

5

Result:

```
INSERT INTO users VALUES( 'Anees', 'aneeshikmat@2nees.com', 30 );
SELECT name, email, age FROM users WHERE age > 18 AND name LIKE '%Anees%' LIMIT 1, 5;
=========================================
INSERT INTO users VALUES( 'Anees', 'aneeshikmat@2nees.com', 30 );
SELECT name, email, age FROM users WHERE age > 18 AND name LIKE '%Anees%' LIMIT 1 OFFSET 5;
```

Creational Design Patterns - Builder

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. ال Builder: وهو ال interface الذي يحتوي بداخله ال methods الخاصة ببناء ال object، أو بمعنى آخر، مجموعة الخطوات اللازمة لإنتاج product معين ويشترك مع كل ال builders الأخرى في هذه الخطوات، في مثالنا "HouseBuilderInterface"
2. ال Concrete Builders: يتم من هذه ال builders عمل implementation لل method الموجودة داخل ال Builder، وليس بالضرورة أن يلتزم ال Concrete builder فقط في Builder Interface، في مثالنا: "HouseBuilder"
3. ال Products: تمثل ال Products النتيجة الخاصة بال Object، لذلك، ليس من الضرورة أن يتبع جميع ال Products نفس التسلسل الخاص بال classes أو ال interface، فمثلا يمكن أن نستخدم ال builder في تصميم السيارات وفي بناء manual لها!، وذلك ببساطة من خلال تنفيذ اللازم داخل ال builder، والاستفادة من تسلسل كل Product، في مثالنا: "House".

Creational Design Patterns - Builder

4. Director: يمثل ال Director المكان المناسب للاحتفاظ بالترتيب المناسب لأي Object سيتم بناؤه بناءا على ما تم تعريفه مسبقا، وهذا الأمر جميل جدا، فلا داعي لمشاركة ال method من ال builder بذاتها إلى ال client، بل قد يكفي مشاركة ال director والذي يحتوي بداخله ال order المناسب للبناء، ولا يشترط أن يكون موجودا في كل الحالات. في مثالنا: "Director".
5. ال Client: يمثل الطريقة التي سيتم من خلالها بناء ال object بشكله النهائي وبما يتناسب مع متطلبات العمل، ويختلف شكل الشيفرة البرمجية عند ال client باختلاف طريقة بناء ال builder -مثل وجود Director أو أكثر من Product...- كما شاهدنا في الأمثلة الثلاث السابقة.

Creational Design Patterns - Builder

متى يمكنني استخدام هذا ال Pattern؟

استخدم هذا ال Pattern عند وجود تعقيدات كثيرة أو كبيرة عند بناء ال Object، والتي ستتسبب بإنشاء العديد من ال Parameters والتي ستلزمك بوضع الكثير من الشروط أو عمل ال overload في لغات البرمجة التي تسمح بذلك!، كما قم باستخدام هذا الأسلوب عند حاجتك لبناء أكثر من شكل للبيانات لمجموعة من ال product مثل (بيت من خشب أو حجر، بناء السيارة وبناء ال manual للسيارة)، ولذلك لأنك تستطيع أن تتحكم في خطوات البناء والتي يمكن أن ترتبط بأكثر من product، كما يستخدم للتقليل من تعقيد ال Composite tree، فهو يمكنك من بناء ال Object step by step، ويمكنك تأجيل أو إعادة استخدام نفس ال method أكثر من مرة...، لذلك، إن صادفت من مثل هذه التعقيدات عند إنشاء ال Object، ففكر في هذا ال Pattern.

Creational Design Patterns - Builder

المميزات:

- يمكن بناء ال Object خطوة بخطوة، كما يمكن التحكم في ال Order الخاص بالبناء.
- يمكن استخدام نفس خطوات البناء للوصول لتمثيل مختلف لكل Product حسب حاجته
- يحقق مبدأ ال Single Responsibility Principle بشكل جميل، فهو ينزع التعقيد من ال Class الخاص بال Product.

العيوب:

- التعقيد الخاص بهذا ال Pattern، ويزداد مستوى التعقيد لهذا ال Pattern مع كل زيادة في مستوى تعقيد البزنس، وسبب وجود مشكلة التعقيد هو أنك بحاجة لإنشاء العديد من ال classes الجديدة لتغطية أي إضافة جديدة.

Creational Design Patterns - Builder

علاقة ال Builder مع غيره من ال Pattern:

- عادة ما يبدأ العمل على ال Factory Method، لكن يتطور هذا ال Design مع الوقت ليصير Abstract Factory أو Prototype أو Builder، والتي ستوفر مرونة أكبر، لكنها ستزيد من مستوى التعقيد.
- ال Abstract Factory يركز على إنشاء عائلات مترابطة من خلال العلاقات بين ال Objects، ويقوم ال Abstract Factory بإرجاع ال Product مباشرة، بينما ال Builder يركز على بناء ال Object خطوة بخطوة، لذلك فهو يحتاج إلى وقت أكبر للحصول على ال Product.
- يمكن استخدام ال Builder عند بناء ال Composite tree لأنك ستكون قادرا على بناء الخطوات step-by-step بشكل متكرر.
- يمكنك أن تجمع بين ال Builder وال Bridge، لأن ال Director class سيقوم بتشكيل الوظائف الخاصة بال Abstraction، بينما تحتوي ال builders المختلفة على ال implementations المناسب لها.
- كل من ال Abstract Factories وال Builders وال Prototype يمكنك تنفيذها على أنها Singletons.

يا أخي، أدبّ لسانك!، فلا يخرج منه قبيح القول، سيئه ومنكره، كمن يشتم الناس، ويقذفهم بالسوء ونحو ذلك.

Creational Design Patterns - Prototype

ال Pattern الرابع ضمن هذه المجموعة هو ال Prototype، هذا ال Pattern بسيط جدا، والفكرة المنبثقة منه جميلة ومفيدة في الحالات التي نرغب فيها باستنساخ Object دون الحاجة لتكرار شيفرة برمجية للقيام بعملية النسخ هذه، ودون الحاجة لاستخدام "new"، ومن اسم هذا ال Pattern، فهو يمثل نموذج مسبق الصنع لل Object المراد إنشائه، بعد ذلك يتم التعديل على هذا ال Object الافتراضي، ويمكن متابعة العمل في وقت لاحق للتعديل على المعلومات، ومن الأمثلة على ذلك، إنشاء Object خاص في موظف جديد، ففي هذه الحالة مثلا يمكنك إضافة هذا الموظف في معلومات افتراضية ومثلا كتابة الإسم الحقيقي للموظف فقط، وبعد ذلك سيقوم الموظف بالتعديل على معلوماته...

Creational Design Patterns - Prototype

المشكلة: تخيل أن لديك موظف ترغب في إنشائه، ال Object الخاص بهذا الموظف لديه معلومات افتراضية يجب أن يتم تعبئتها، لكن، هذه المعلومات سيتم تعبئتها من خلال user، لكن بعضها سيتم تعديله من قبل موظف ال Hr، الآن، لو أردت نسخ المعلومات الافتراضية في كل مرة باستخدام ال new operator، وهذه هي الطريقة المباشرة لنسخ القيم!، وهذا ممكن حقيقة!، لكن هناك مشكلتين في هذا الأسلوب: الأولى أنك قد لا تستطيع نسخ القيم الموجودة داخل ال Private fields، والثانية أن يجب أن تربط عملية النسخ هذه بال Class الذي ستقوم بإنشاء النسخة منه، فأي إضافة على هذا ال class يمكن أن تسبب بخلل ما، كما أنك ليس بالضرورة أن تعلم ما هو التطبيق الموجود داخل هذا ال class، فماذا لو كان interface method؟!..، وتستقبل أكثر من نوع من ال object؟

Creational Design Patterns - Prototype

الحل: يقدم ال Prototype حلا جميلا وبسيطا لهذه المشكلة، وذلك من خلال نقل عملية نسخ المعلومات الموجودة في ال Object من الطريقة المباشرة إلى ال Object المراد استنساخه فعلا، ويتم هذا بكل بساطة من خلال إنشاء method اسمها clone داخل interface مثلا ومن ثم وراثتها لكل ال Object التي سيسمح باستنساخها، وبهذا فإن التنفيذ والتطبيق عادة يكون متشابها في كل مكان، وتستطيع الوصول لكل المعلومات التي تحتاجها بما فيها ما بداخل ال private fields! هذا ال Object الذي يمكنك أن تستنسخه اسمه Prototype ^^، وبهذا بدلا من إنشاء العديد من ال subclass للوصول لشكل معين، يمكنك بكل بساطة نسخ ال Object المراد ومن ثم تغير القيم الخاصة به كما يتناسب معك...

ملاحظة: في ال PHP عناك Magic Method اسمها __clone()، يمكنك استخدامها لتحقيق هذا المبدأ، ويمكنك إنشاء method خاصة بال clone اذا كانت لغة البرمجة التي تعمل عليها لا تقدم مثل هذه ال method.

Creational Design Patterns - Prototype

```php
class EmployeesPrototype
{
    private string $name;
    private int $age;
    private Salary $salary;
    private PrivilegesTypes $privilegesTypes;

    public function __construct(string $name, int $age, Salary $salary, PrivilegesTypes $privilegesTypes)
    {
        $this->name = $name;
        $this->age = $age;
        $this->salary = $salary;
        $this->privilegesTypes = $privilegesTypes;
    }

    public function setName(string $name): void
    {
        $this->name = $name;
    }

    public function setAge(int $age): void
    {
        $this->age = $age;
    }

    public function __clone()
    {
        $this->setAge(18);

        $this->salary->setSalary(300);
        $this->salary->setTax(0.1);

        $this->privilegesTypes->clearPrivilegesTypes();
        $this->privilegesTypes->addPrivilegesTypes("view");

        return $this;
    }
}
```

1

هذا ال Class يمثل ال Concrete Prototype والذي بدوره يقوم بعملية النسخ لل Object الأصلي، والذي يمكننا أيضا من خلاله تعديل القيم كما نشاء، لاحظ في هذا المثال، قمت بوضع قيم افتراضية لل object الجديد، تتناسب مع ما هو موجود ومطلوب من ال class...ويمكن أيضا تركها بدون أي تطبيق مختلف وستبقى القيم المنسوخة كما هي...

مثال: لنقم الآن بتطبيق المثال الخاص بالموظف، والذي تحدثنا عنه سابقا:

Creational Design Patterns - Prototype

```php
class Salary {
    private float $salary;
    private float $tax;

    public function setSalary(float $salary): void
    {
        $this->salary = $salary;
    }

    public function setTax(float $tax): void
    {
        $this->tax = $tax;
    }
}

class PrivilegesTypes {
    private array $privilegesTypes;

    public function clearPrivilegesTypes(): void
    {
        $this->privilegesTypes = [];
    }

    public function addPrivilegesTypes(string $privilegesType): void
    {
        $this->privilegesTypes[] = $privilegesType;
    }
}
```

2

هذه ال Class التي قمنا بإنشائها لزيادة مستوى التعقيد في ال class الخاص بالموظف، والذي أظهر لنا فائدة ال clone بطريقة بسيطة وجميلة.

ملاحظة: هذه ال classes ليس لها علاقة مباشرة بال pattern، ال pattern يهتم بال clone كما ذكرنا، وال class التي تتواجد بها هذه ال method.

Creational Design Patterns - Prototype

3

هنا قمنا باستخدام الفكرة من ال pattern، انظر كيف قمنا بإنشاء موظف، وانظر كيف قمنا بنسخه...وشاهد نتائج ذلك في الشرحة التالية

```php
$salary = new Salary();
$salary->setSalary(500);
$salary->setTax(0.16);

$pr = new PrivilegesTypes();
$pr->addPrivilegesTypes("add");
$pr->addPrivilegesTypes("update");

$newEmployee = new EmployeesPrototype("Anees", 29, $salary, $pr);
echo PHP_EOL . "============== Print New Employee Data ==========================" . PHP_EOL;
print_r($newEmployee);
echo PHP_EOL . "============== Print Cloned Employee Data ==========================" . PHP_EOL;
$clonedEmployee = clone $newEmployee;
$clonedEmployee->setName("Hikmat");
print_r($clonedEmployee);
```

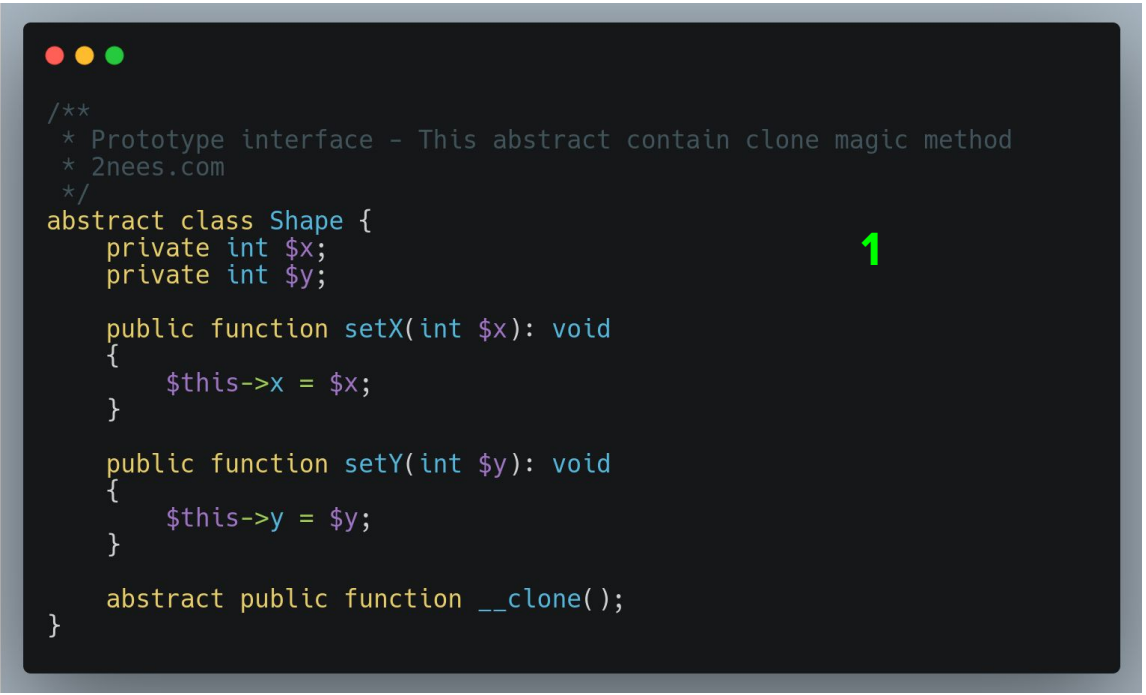
Creational Design Patterns - Prototype

```
=============== Print New Employee Data ==========================
EmployeesPrototype Object
(
    [name:EmployeesPrototype:private] => Anees
    [age:EmployeesPrototype:private] => 29
    [salary:EmployeesPrototype:private] => Salary Object
        (
            [salary:Salary:private] => 500
            [tax:Salary:private] => 0.16
        )

    [privilegesTypes:EmployeesPrototype:private] => PrivilegesTypes Object
        (
            [privilegesTypes:PrivilegesTypes:private] => Array
                (
                    [0] => add
                    [1] => update
                )

        )

)
=============== Print Cloned Employee Data ==========================
EmployeesPrototype Object
(
    [name:EmployeesPrototype:private] => Hikmat
    [age:EmployeesPrototype:private] => 18
    [salary:EmployeesPrototype:private] => Salary Object
        (
            [salary:Salary:private] => 300
            [tax:Salary:private] => 0.1
        )

    [privilegesTypes:EmployeesPrototype:private] => PrivilegesTypes Object
        (
            [privilegesTypes:PrivilegesTypes:private] => Array
                (
                    [0] => view
                )

        )

)
```

4

والآن، إليك الرابط الخاص بالمثال ^^

Creational Design Patterns - Prototype



مثال 2: تخيل أن لدينا لعبة مثلا، ونريد أن ننشئ بداخلها نسخا مكررة من المربعات والمستطيلات والدوائر، بعضها بنفس القيم، وأخرى بقيم عشوائية، هذا ال pattern سيجعل من هذا ال trick سهلا، بكل بساطة clone في interface ترثها جميع ال shapes...، شاهد المثال:

Creational Design Patterns - Prototype

```php
/**
 * Simple Class for simulate Concrete Prototype
 * 2nees.com
 */
class Rectangle extends Shape {
    public function __clone()
    {
        return new Rectangle();
    }
}

class Circle extends Shape {
    public int $radius;

    public function __clone()
    {
        return new Circle();
    }
}

class Square extends Shape {
    private int $position;

    public function __construct() {
        $this->position = 1;
        $this->setX(1);
        $this->setY(1);
    }

    public function __clone()
    {
        $size = rand(10, 20);
        $this->position = rand(2, 40);
        $this->setX($size);
        $this->setY($size);
        return $this;
    }
}
```

2

لاحظ أن المستطيل والدائرة والمربع كل منها ورث ال Shape، وكل منها لديه ال clone الخاص به، وبهذا، فجميع هذه الأشكال قامت بعمل clone بالطريقة المناسبة لها

```php
$shapes = [];

$newCircle = new Circle();
$newCircle->radius = 20;
$newCircle->setX(5);
$newCircle->setY(5);

$newRect = new Rectangle();
$newRect->setX(10);
$newRect->setY(10);

$newSq = new Square();

$shapes[] = $newCircle;
$shapes[] = $newRect;
$shapes[] = $newSq;

for ($i = 1; $i <= 3; $i++){
    $shapes[] = clone $newCircle;
    $shapes[] = clone $newRect;
    $shapes[] = clone $newSq;
}

print_r($shapes);
```

3

لاحظ هنا أن ال Client قام بكل بساطة بعمل clone لكل Object، ومن ثم قمنا بطباعة ال shapes

Creational Design Patterns - Prototype

```
Array
(
    [0] => Circle Object
        (
            [radius] => 20
            [x:Shape:private] => 5
            [y:Shape:private] => 5
        )

    [1] => Rectangle Object
        (
            [x:Shape:private] => 10
            [y:Shape:private] => 10
        )

    [2] => Square Object
        (
            [position:Square:private] => 1
            [x:Shape:private] => 1
            [y:Shape:private] => 1
        )

    [3] => Circle Object
        (
            [radius] => 20
            [x:Shape:private] => 5
            [y:Shape:private] => 5
        )

    [4] => Rectangle Object
        (
            [x:Shape:private] => 10
            [y:Shape:private] => 10
        )

    [5] => Square Object
        (
            [position:Square:private] => 21
            [x:Shape:private] => 14
            [y:Shape:private] => 14
        )

    [6] => Circle Object
```

4

والآن، إليك الرابط الخاص بالمثال ^^

Creational Design Patterns - Prototype

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. ال Prototype: ال interface الذي يحتوي ال clone method، هذا ال interface ليس بالضرورة أن يكون موجودا، كما في مثالنا الأول، لكن في المثال الثاني كان لابد من وجوده.
2. ال Concrete Prototype: وهو ال Class الذي يقوم بعمل implementation لل clone method.
3. ال Client

Creational Design Patterns - Prototype

متى يمكنني استخدام هذا ال Pattern؟

استخدم هذا ال Pattern عند وجود حاجة لنسخ ال Object وبدون الاعتماد على concrete class، فمثلا أي third party library تحتاج إلى نسخ ال object منها، هنا يمكنك إنشاء ال clone لهذه ال object، لان ال concrete class غير معلومة!، كما أن هذا ال pattern يستخدم لتقليل عدد ال subclass التي يمكن أن تنشئ للتحكم بال configuration الخاص بال object، فبدلا من ذلك، يتم تعريف ال configuration ومن ثم نسخها!، وهذا ال Pattern هو الحل لمشاكل النسخ المباشر من القيم من خلال ال new keywords!

Creational Design Patterns - Prototype

المميزات:

- يمكن نسخ ال Object دون أن يقترن ذلك بال concrete classes.
- يمكن التخلص من مشكلة ال inilization لكل object في كل مرة نرغب فيها باستدعاء ال Object وذلك من خلال ال clone الذي سيحقق هذه العملية.
- بسبب وجود ال clone، فإن إنشاء object معقده لن يكون مشكلة عند الرغبة بالحصول على القيم، لأن ال clone سيجعل من ال object الموجود يتشكل بما يلائم حاجتنا.
- قد يكون هذا الأسلوب بديلا عن الوراثة في بعض الحالات، فبدلا من القيام بالوراثة من class إلى آخر، يمكنك بناء object بالإعدادات المطلوبة من خلال ال clone، وطبعا ذلك على حساب التعقيد الموجود في ال object.

العيوب:

- عملية النسخ ل compex object في حالة وجود circular references قد تكون صعبة جدا، لأن مكونات ال Object الخاص بك قد تحتاج إلى قيمة من جزئية أخرى، وتلك الجزئية تحتاج النتيجة من الجزئية الأولى!

Creational Design Patterns - Prototype

علاقة ال Prototype مع غيره من ال Pattern:

- يمكن استخدام ال Prototype مع ال Command عند الحاجة أو الرغبة في نسخ ال Command Pattern إلى ال History.
- التصاميم التي تستخدم ال Composite and Decorator بشكل كبير يمكن أن تستخدم ال Prototype pattern والاستفادة من خاصية ال clone لنسخ ال complex object الموجودة بدلا من إعادة بنائها في كل مرة.
- ال Prototype لا يعتمد على inheritance، لذلك فسيئات الوراثة ليست موجودة لهذا ال pattern، لكن في ذات الوقت، فإن التعقيد ينتقل إلى ال initialization الخاص بال clone object، بينما ال Factory method يعتمد على الوراثة لكنه لا يتطلب initialization step.
- في بعض الحالات يمكن أن يكون ال Prototype بديلا بسيطا عن ال Memento، ويكون هذا الأمر عند الرغبة بحفظ ال state داخل ال history، إذا كان ال Object غير مرتبط ب external resource link، أو من السهل إعادة بناء ال links.
- كل من ال Abstract Factories وال Builders وال Prototype يمكنك تنفيذها على أنها Singletons.

اللهم أنت ربي لا إله إلا أنت خلقتني وأنا عبدك وأنا على عهدك ووعدك ما استطعت, أعوذ بك من شر ما صنعت, أبوء لك بنعمتك علي وأبوء بذنبي, فاغفر لي فإنه لا يغفر الذنوب إلا أنت

Creational Design Patterns - Singleton

ال Pattern الخامس ضمن هذه المجموعة هو ال Singleton، هذا ال Pattern الجميل يقدم خدمة أسطورية للمبرمجين، فهو يضمن لهم أن ال Class سيكون له instance واحدة فقط، ويمكن الوصول لها من أي مكان داخل التطبيق (Global Access)!

إذا باختصار، فأنت أنشأت instance واحدة يمكن الوصول إليها من أكثر من مكان!، وبهذا فأنت تضمن أن ما رغبت في الوصول إليه من خلال هذه ال instance سيكون متاحا للجميع، ولا يمكنهم التغيير على هذه ال instance.

Creational Design Patterns - Singleton

المشكلة: تخيل أن لديك الإعدادات الخاصة بالاتصال بقواعد البيانات، هذه الإعدادات أن تتمكن جميع الملفات من الوصول إليها، وبنفس الوقت، يجب أن تضمن أن هذه الإعدادات يجب أن لا تتغير من قبل ال client أو عن طريق الخطأ! فماذا ستفعل؟ إنشاء هذه الإعدادات من خلال ال constructor أمر غير مقبول أو غير ممكن!، لأن كل عملية call لهذا ال constructor تعني object جديد، وإضافة Global variable أمر خطير لأن هذه القيم يمكن تعديلها في أي وقت...إذا ما الحل؟!

الحل: قدم ال Singleton Pattern حلا جميلا لهذه المشاكل، وذلك من خلال جعل ال constructor private، وإنشاء ال method التي تتعامل مع ال constructor بنوع static، بهذا أنت تضمن أن هذه ال instance يمكن الوصول إليها Global، ولا يمكن تعديلها ^^، ومن الحركات اللطيفة التي يمكنك القيام بها أيضا، هي جعل ال clone private!، وبهذا فأنت تمنع نسخ هذه ال instance ^^. ملاحظة: يمكن استخدام protected بدلا من private في حال الرغبة بمشاركة ال instance مع ال sub class للقيام ببعض الوظائف عند الحاجة

Creational Design Patterns - Singleton

```php
class DatabaseConfig {
    private static string $host;
    private static int $port;
    private static string $dbUser;
    private static string $dbPassword;
    private static ?self $database = null;
    private function __clone(){}
    private function __construct()
    {
        self::$host         = "localhost";
        self::$port         = 80;
        self::$dbUser       = "anees";
        self::$dbPassword   = "password";
    }
    public static function DB(): self
    {
        if(is_null(self::$database)){
            self::$database = new self();
        }

        return self::$database;
    }
    public static function getDBConfig(): array
    {
        return [
            self::$host,
            self::$port,
            self::$dbPassword,
            self::$dbUser
        ];
    }
    public function getNeededConfig($key)
    {
        return self::$$key ?? null;
    }
}

$db = DatabaseConfig::DB();
print_r($db::getDBConfig());
echo PHP_EOL;
print_r($db->getNeededConfig("dbUser"));
echo PHP_EOL;
print_r(DatabaseConfig::DB()->getNeededConfig("dbUser"));
```

```
Array
(
    [0] => localhost
    [1] => 80
    [2] => password
    [3] => anees
)

anees
anees
```

مثال: هذا المثال يوضح ال Singleton، وفعليا كما ترى، المفهوم بسيط جدا، والفائدة عظيمة، لاحظ هنا أن هذا ال class ما ميزه فعلا هو ال وجود ال instance تسمى $database، وال private على construct وال clone، وال static method لكي نستطيع الوصول لل instance وهي DB، وبهذا تكون قد بنيت هذا ال Pattern.

والآن، إليك الرابط الخاص بالمثال ^^

Creational Design Patterns - Singleton

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Singleton Instance: والتي تمثل ال Private field، وفي مثالنا هي $database
2. Private Construct
3. Static Method: والتي بدورها تمثل الحلقة المسؤولة عن التحكم في ال access للوصول إلى ال Singleton instance، وفي مثالنا هي DB()

إذا، متى يمكنني استخدام هذا ال Pattern؟
استخدم هذا ال Pattern عند حاجتك لوجود instance واحدة لكل ال Client، مثل مشاركة ال database object بين كل ال client، كما استخدم هذا ال Pattern عند حاجتك لوضع قيود ومحددات على ال Global Variable.

Creational Design Patterns - Singleton

المميزات:

- كل class لديه instance واحدة
- يمكن الوصول إلى ال Singleton instance من أي مكان في المشروع Global
- ال Singleton Object يتم عمل initialized له فقط عند أول request.

العيوب:

- ينتهك هذا ال Pattern ال Single Responsibility Principle، والسبب في ذلك لأنه يقوم بالتحكم في ال create الخاص به، وفي ال life-cycle الخاصة به.
- لأن ال singleton لديها instance واحدة يمكن مشاركتها مع كل ال client، هذا قد يتسبب بإخفاء العيوب الموجودة بالتصميم، لأن ما في داخل هذه ال instance مشارك مع الجميع.

يتبع =>

Creational Design Patterns - Singleton

- هذا ال Pattern يتطلب عمل إضافي في بيئة العمل التي تدعم ال multithreaded، لأنك لا تريد بالسماح لكل thread بإنشاء ال singleton object أكثر من مرة.
ملاحظة: هناك العديد من الحلول لحل هذه المشكلة، مثل استخدام Double Check على ال instance إذا كانت null أم لا، كذلك عمل lock في اللغات التي تسمح بذلك، أو استخدام ال sync باللغات التي تسمح بذلك...وغيرها الكثير!، ولكل حل عيوبه ومميزاته، فانظر إلى اللغة التي تستخدمها، وما هو أنسب حل لهذه المشكلة!، طبعا هذا في حال وجود تطبيق multithreaded.

- من الصعب بناء unit test له، ولذلك لأن معظم ال unit test تعتمد على الوراثة، وهنا لا يوجد وراثة، وفوق هذا ال construct بمثل private.

Creational Design Patterns - Singleton

علاقة ال Singleton مع غيره من ال Pattern:

- غالبا ما يتحول ال Facade class إلى Singleton لأن facade object واحد كافي في معظم الحالات.

- ال Flyweight شبيه بال Singleton إذا قمت بإدارة جميع ال state من خلال Object واحد، لكن هناك اختلافين جوهريين، الأول أن ال Singleton لا يحتوي إلا instance واحدة، بينما ال Flyweight يمكن أن يحتوي أكثر من instance، والثاني أن ال Singleton يمكن تعديل ال object الخاص به، بينما ال flyweight غير قابل للتعديل.

- كل من ال Abstract Factories وال Builders وال Prototype يمكنك تنفيذها على أنها Singletons.

لا إله إلا اللّٰه العظيم الحليم . لا إله إلا اللّٰه رب العرش العظيم . لا إله إلا اللّٰه رب السماوات ورب الأرض ورب العرش الكريم

Structural Design Patterns

ثاني مجموعة لدينا من ال Design Patterns هي مجموعة ال Structural، في هذه المجموعة تركز ال Patterns على كيفية تجميع ال Objects وال Classes في structures أكبر، مع الحفاظ على مرونة وكفاءة هذه ال structures.

Structural Design Patterns - Adapter

ال Pattern الأول ضمن هذه المجموعة هو ال Adapter، هذا ال Pattern فكرته قائمة على جعل ال Objects المختلفة والغير متوافقة مع بعضها البعض بأن تعمل معا وكأن شيئا لم يكن^^، أي أنك ستحاول من خلال هذا ال pattern جعل ال Objects التي لا تتوافق فيما بينها مع ال interfaces الخاصة بهم تتعاون حتى لا يتم يتم إجراء تغييرات على ال Process النهائية الخاصة بال Object عند ال client، ومن الإسم Adapter الذي يعني محول أو مهيئ أو مشترك كهربائي، فكلها تشير إلى معنى واحد تهيئة أو تحويل أمر ما ليعمل بما يتناسب مع المنظومة الحالية.

Structural Design Patterns - Adapter

المشكلة: تخيل أنك تتعامل مع موقع من المواقع التي تقدم خدمات إرسال رسائل ال SMS، وقمت ببناء الشيفرة البرمجية الخاصة بك بناءا على ما تتطلبه هذه الشركة، ثم قررت الإدارة الإنتقال من هذه الشركة إلى شركة أخرى لأي سبب مثل جودة أعلى أو سعر أفضل أو نحو ذلك، في هذه الحال وللنظرة الأولى ستقول، بإمكاننا إعادة كتابة الشيفرة البرمجية التي لدينا لتتناسب مع الشركة الجديدة!، وهذا له مشاكل كثيرة خصوصا إذا كان هناك client يعتمدون على هذه الخدمة بما قدمته لهم من interface!، كما أن الوقت المستغرق للتعديل وال risk الناتج من هذه العملية قد يكون كبيرا!، ثم قد تقول، يمكنني التعديل على library الجديدة لجعلها تعمل كما تعمل الشركة القديمة، وهذا أيضا سيتسبب في مشاكل أخرى، مثل أنك قد تخسر بعض ال funcinalty الموجودة داخل هذه ال library، ثم ليس بالضرورة أن الشيفرة البرمجة سيسمح لك بتعديلها، بالإضافة إلى ذلك قد تفقد القدرة على متابعة التحديثات الجديدة لهذه ال library...إذا، ما الحل؟

Structural Design Patterns - Adapter

الحل: يقترح ال Adapter Pattern حلا جميلا لمثل هذه المشاكل، فيقول بدلا من التعديل على الشيفرة البرمجية بذاتها أو التعديل على ال Library بذاتها، نقوم بتهيئة أو تحويل طريقة العمل لل Library الجديدة للتناسب مع ال interface الخاصة بالشركة القديمة، بعبارة أخرى، إنك تقوم ببناء interface خاصة بالشركة الجديدة تتعامل مع أدواتها الخاصة داخل ال method المستخدمة مع ال interface القديمة، وبهذا فأنت يمكنك الإنتقال من شركة إلى شركة دون الحاجة إلى تعديل الشيفرة البرمجية، سوى سطر واحد وهو ال class الذي سترغب في مناداته فقط!، وباختصار فإن ال Adapter هو special object وظيفته تحويل ال interface الخاصة ب object معين إلى شكل يفهمه ال object الآخر ويستطيع التعامل معه، فيقوم ال adapter بعمل wrapping لإخفاء التعقيد الموجود في ال object الحاصل بسبب هذا النوع من التغييرات، وبدون الإهتمام بما سيقوم به ال adapter فيما بعد، على سبيل المثال قد تكون الشركة الجديدة ترجع الداتا على شكل JSON، بينما ال Adapter سيقوم بأخذ هذا ال JSON من ال Object وتحويله ل XML ليتوافق مع البنية الخاصة به!، بهذه الطريقة يكون ال adapter متوافقا مع ال interface وبذلك فيتوافق مع ال objects، ويمكن لل objects من الوصول لل methods الخاصة بال adapter...، ومن الحركات اللطيفة أيضا هي إمكانية إنشاء two-way-adapter، فبدلا من القيام بإنشاء adapter واحد مرتبط بالشركة الجديدة، يمكن إنشاء adapter يعمل كمحول من القيم القديمة للقيم الجديدة والعكس صحيح، مثلا من XML to JSON أو من JSON to XML!

Structural Design Patterns - Adapter

مثال: والآن لنقم بتطبيق عملي، لنأخذ مثلا أننا نتعامل مع شركة لإرسال رسائل SMS، وأسمينا هذه الشركة OldSMSCompany، وعندنا interface فيها يظهر طريقة التنفيذ أو التعامل مع هذه الشركة، ثم بعد ذلك جاءت الشركة NewSMSCompany...

1

ال interface المبنية في المشروع والتي تحتوي في داخلها شكل وآلية التنفيذ لأداء المهمة المطلوبة...

```php
/*
* Client Interface
* 2nees.com
*/
abstract class SMSClient
{
    private string $number;
    private string $message;

    public function setNumber(string $number): void
    {
        $this->number = $number;
    }

    public function setMessage(string $message): void
    {
        $this->message = $message;
    }

    public function getNumber(): string
    {
        return $this->number;
    }

    public function getMessage(): string
    {
        return $this->message;
    }

    abstract public function sendMessage();
}
```

2

هذا ال Class يمثل الشركة القديمة التي تعاقدت معها الشركة...
والتي نرغب في تغييرها...،

```php
class OldSMSCompany extends SMSClient
{
    public function sendMessage()
    {
        echo "SMS Message Sent to {$this->getNumber()}: {$this->getMessage()}!" . PHP_EOL;
    }
}
```

Structural Design Patterns - Adapter

هذا ال class يمثل ال Adaptee للشركة الجديدة، وهو يمثل ال interface الخاصة بالشركة الجديدة والغير متوافقة مع ال interface الخاصة بنا...
ملاحظة: ال interface هنا يقصد بها الواجهة التي سنتعامل معها بغض النظر عن كونها class أو abstract أو interface...

3

```php
/**
 * This is Adaptee Class
 * 2nees.com
 */
class NewSMSCompanyAdaptee
{
    private string $connect;
    private string $oath;
    private string $num;
    private string $text;

    public function __construct(bool $connect, string $oath, string $num, string $text)
    {
        $this->connect = $connect ? "connect" : "disconnect";
        $this->oath = $oath;
        $this->num = $num;
        $this->text = $text;
    }

    public function send()
    {
        if($this->isConnected()){
            $rand = rand(1, 50);
            echo "SMS Sent #{$rand}:  {$this->num}: {$this->text}!". PHP_EOL;
        }else {
            echo "SMS NOT SEND: {$this->connect}:{$this->oath}". PHP_EOL;
        }
    }

    private function isConnected(): bool
    {
        return $this->oath === "2nees.com" && $this->connect;
    }
}
```

مثال: لاحظ أن الشركة الجديدة لديها بعض التفاصيل المهمة قبل إرسال الرسالة، مثل التحقق من oauth، والاتصال!، ولاحظ أن الرسائل النصية تغيرت... ومع هذا فإن هذا لن يؤثر على العمل، لأن ال Adapter سيقوم باللازم...

Structural Design Patterns - Adapter

مثال:

```php
class NewSMSCompanyAdapter extends SMSClient {
    public function sendMessage()
    {
        $textRandom  = ["2nees.com", "Anees"][rand(0, 1)];
        $randConnect = rand(0, 5);
        $newSmsCompanyAdaptee = new NewSMSCompanyAdaptee(
          $randConnect > 1,
          $textRandom,
          $this->getNumber(),
          $this->getMessage()
        );
        $newSmsCompanyAdaptee->send();
    }
}
```

هذا ال Class يمثل ال Adapter، فوظيفته تكمن في أخذ شكل الإجراءات الخاصة بالشركة القديمة، وصياغتها لتتناسب مع الشركة الجديدة..

ملاحظة: هذا المثال في هذا الشكل لتوضيح الفكرة و عمل محاكاة للتغير على القيم

Structural Design Patterns - Adapter

مثال:

ال client code للشركة القديمة...

```php
// Company changed? Just update company class ^^, and keep your
client code in loop as is.
$sms = new OldSMSCompany();
//$sms = new NewSMSCompanyAdapter();
for ($i = 1; $i <= 5; $i++){
    $sms->setNumber("00962000{$i}");
    $sms->setMessage("SMS send from 2nees.com+{$i}");
    $sms->sendMessage();
}
```

```
SMS Message Sent to 009620001: SMS send from 2nees.com+1!
SMS Message Sent to 009620002: SMS send from 2nees.com+2!
SMS Message Sent to 009620003: SMS send from 2nees.com+3!
SMS Message Sent to 009620004: SMS send from 2nees.com+4!
SMS Message Sent to 009620005: SMS send from 2nees.com+5!
```

والآن، إليك الرابط الخاص بالمثال ^^

5

ال client code للشركة الجديدة...لاحظ أن التغيير لم يحدث إلا على مستوى ال call الخاص بال company ^^

```php
// Company changed? Just update company class ^^, and keep your
client code in loop as is.
//$sms = new OldSMSCompany();
$sms = new NewSMSCompanyAdapter();
for ($i = 1; $i <= 5; $i++){
    $sms->setNumber("00962000{$i}");
    $sms->setMessage("SMS send from 2nees.com+{$i}");
    $sms->sendMessage();
}
```

```
SMS Sent #44:  009620001: SMS send from 2nees.com+1!
SMS NOT SEND: disconnect:Anees
SMS NOT SEND: disconnect:Anees
SMS Sent #31:  009620004: SMS send from 2nees.com+4!
SMS Sent #24:  009620005: SMS send from 2nees.com+5!
```

Structural Design Patterns - Adapter

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Client interface: وهذا يمثل المكان الذي يحتوي على طريقة تنفيذ الإجراءات بالشكل الذي يتعامل معه ال client، وهذا يمثل ال SMSClient في مثالنا، وهذا يمكن أن يكون interface, abstract, class.

2. Adaptee (Service): وهو يمثل العنصر الجديد الذي نرغب بإضافته للشيفرة البرمجية، لكن ال client لا يمكنه الاتصال معه مباشرة بسبب اختلاف ال interface، وفي مثالنا هو ال Adaptee.

3. Adapter: وهو ال class الوسيط، والذي يمثل حلقة الوصل بين ال client وال Adaptee الجديدة، ويمكنه التعامل مع الجزئين، وبكل بساطة فوظيفته تكمن في أخذ البيانات من ال client ومن ثم ترجمتها للشكل الذي يتعامل معه ال Adaptee.

4. Client: وهو الجزء الذي يقوم بعمل call لل class المناسب لتنفيذ المطلوب، ويجب أن لا تتغير الشيفرة البرمجية أكثر من تبديل ال class أو إنشاء object وتبديل ال class بحيث يستقبل ال object...أي أن التعديلات لا تتعدى ال call.

Structural Design Patterns - Adapter

إذا، متى يمكنني استخدام هذا ال Pattern؟

يمكنك استخدام هذا ال Pattern عند حاجتك لاستخدام بعض ال class الموجودة في المشروع ومن الصعب أو من الخطر تعديل السلوك للشيفرة الحالية، وبنفس الوقت ال interface الجديدة غير متوافقة مع التطبيق الحالي، كما يمكنك استخدامه في حال وجود العديد من ال subclass والتي من الصعب وضع ال funcinalty الخاصة بها بال superclass، وبنفس الوقت أن لا ترغب بتكرار الشيفرة البرمجية في كل مرة، فيمكن بناء ال adapter ليعمل wrapping لهذه ال subclass وبنفس الوقت يتبع ال super class (هذا قريب كم ال Decorator)، كما أن هذا ال pattern مهم جدا في التعامل مع ال third party library والتي سيتم ضمها للشيفرة البرمجية، والتي ستحتاج إلى استخدامها وليس لك القدرة على تعديل ال methods التي بداخلها، ولا يمكن تحمل خطورة أو مشاكل ذلك، باختصار، عندما تحتاج إلى middleware بين ال client وال service فاعلم أنك بحاجة لل Adapter.

ملاحظة: في اللغات التي تدعم ال multiple inheritance، يمكنك بكل بساطة استخدامها لإنشاء ال adapter، ويكون ذلك بكل بساطة من خلال وراثة ال current class وال new class في class جديد ليكون ال Adapter، وال trick في الموضوع هو استخدام ال override لل method، ولا حاجة لأكثر من ذلك.

Structural Design Patterns - Adapter

المميزات:

- يحقق مبدأ ال Single Responsibility Principle، لأنه يقوم بفصل ال interface أو ال code المسؤول عن تحويل البيانات في class جديد.
- يحقق مبدأ ال Open/Closed Principle، وهذا من النقاط الجميلة لهذا ال pattern، فبهذا التعديل أنت حافظت على ال logic القديم ولم تعديل على الشيفرة الخاصة به، وبنفس الوقت قمت بإضافة adapters جديدة حسب حاجتك، وبهذا فانت تخلصت من ال risk المحتمل.

العيوب:

- يزيد مستوى التعقيد للشيفرة البرمجية لحاجتك بإنشاء interfaces او classes جديدة، لذلك، إذا لم تتحقق المنفعة من استخدامه كما ذكرنا في دواعي الاستخدام، فقد يكون تعديل الكلاس الحالي أسهل...

Structural Design Patterns - Adapter

علاقة ال Adapter مع غيره من ال Pattern:

- الاستخدام الشائع لل Adapter يكون عند الحاجة لجعل classes غير متوافقة مع بعضها أن تعمل مع بعضها، بينما صمم ال Bridge ليقوم ببناء أجزاء التطبيق بشكل مستقل عن الأجزاء الأخرى.

- في ال Adapter فإننا نقوم بعمل تغيير على مستوى ال interface لل current object، بينما يقوم ال Decorator بعمل تحسين على ال object الحالي بدون تغييره، وهو يدعم ال recursive composition بينما لا يدعمه ال Adapter.

- ال Adapter يمكن لأكثر من interface أن تعمل wrap لل object، بينما في ال Decorator يحسن على ال interface الموجودة، وال Proxy يقدم نفس ال interface.

- في ال Adapter فإنك تحاول جعل ال interface الحالية قابلة للاستخدام وذلك يظهر من خلال طريقة عمله، بينما ال Facade يقوم بتعريف interface جديدة لل current object، وهذا يعني أن ال Adaper يعمل مع ال object ك wrapper له، بينما يتعامل ال Facade مع كامل مكونات ال object.

Structural Design Patterns - Adapter

- ال Bridge وال State وال Strategy وإلى حد ما ال Adapter جميعها Pattern قائمة أو تعتمد على ال composition، فهي تنقل العمل إلى objects أخرى، وحقيقة؛ فإن ال structures الخاص بهم متقارب، لكن كل واحد من هذه ال pattern يقدم حلا لمشكلة معينة، وهنا يكمن جمال ال pattern، فهو لا يمثل فقط حلا أو طريقة لكتابة الشيفرة البرمجية، بل يخبر المبرمجين الآخرين عند رؤيتهم للشيفرة البرمجية ما هي المشكلة التي تم حلها باستخدام هذا الpattern، وهذا يعيدنا في الذكريات إلى الوراء قليلا

وَاشكُر فَضَائِلَ صُنعِ اللّٰهِ إذ جُعِلَت **** إلَيكَ, لَا لَكَ عِندَ النَّاسِ حَاجَاتُ
قَد مَاتَ قَومٌ وَمَا مَاتَت فَضَائِلُهُم **** وَعَاشَ قَومٌ وَهُم فِي النَّاسِ أمواتُ

الإمام الشافعي -رحمه اللّٰه-

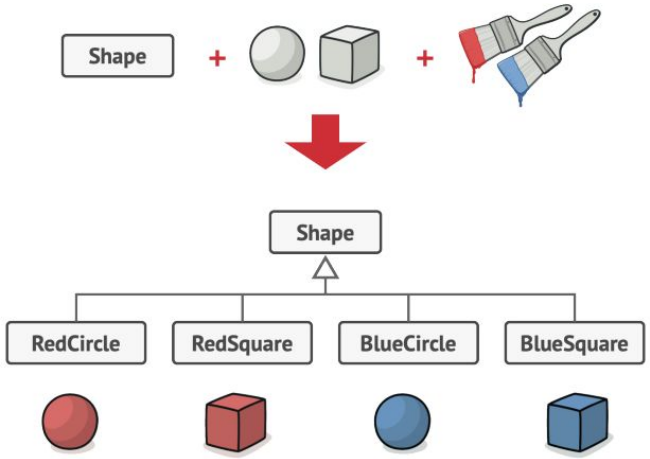
Structural Design Patterns - Bridge

ال Pattern الثاني ضمن هذه المجموعة هو ال Bridge، هذا ال Pattern فكرته قائمة على تجزئة ال Large class أو ال classes المترابطة مع بعضها البعض بشكل وثيق إلى جزئين منفصلين بحيث يمكن تطوير كل جزء دون اعتماد على الجزء الآخر، وهذا يعني أننا نتحدث عن تسلسل هرمي مختلف لكل جزء، الأول سيكون لل Abstraction والثاني سيكون لل implementation، وبهذا يمكنك أن تتوسع دون أن يتأثر أي جزء من الآخر، وبهذا سمي هذا ال Pattern بال Bridge، أي أنك ستبني جسرا يصل الجزئين معا ودون أن يهتم هذا الجسر بمقدار التوسع أو التغييرات على أي جانب.

ملاحظة: ال Abstraction وال implementation هنا لا يقصد بهم ال interface أو ال abstract keys words الموجودة في لغات البرمجة، بل المراد هو المفهوم بأن ال Abstraction layer وظيفته تفويض المهام المطلوبة إلى ال implementation layer، وأعتقد أننا نوهنا إلى ذلك سابقا في الأمثلة أو الشرائح، لكن من باب التأكيد على المعنى لأهميته في هذا الجزء.

Structural Design Patterns - Bridge

المشكلة: تخيل أن لديك Shape، هذا ال Shape لديه subclass كالدائرة والمربع، ولكل shape منهم مجموعة من ال color، ولتكن مثلا الأحمر والأزرق، لتنفيذ هذا الشكل، أول ما يتبادر لذهنك هي الوراثة!، والوراثة فعلا ستحل هذه المشكلة، لكنها ستخلق تعقيدا كبيرا، كما أنها ستتسبب في مشكلة من مشاكل الوراثة وهي التمدد، فسيتضاعف عدد ال classes وال subclass أضعافا مضاعفة، ولتخيل المشكلة بشكل أوضح شاهد هذه الصورة:



لاحظ كمية ال classes الموجودة في هذه الصورة
والتي ستزيد طرديا مع إضافة أي Shape جديد
أو مع إضافة أي color جديد!، تخيل مقدار التعقيد!
ماذا لو أضفنا مستطيل ومثلث وأصفر وبرتقالي !_!

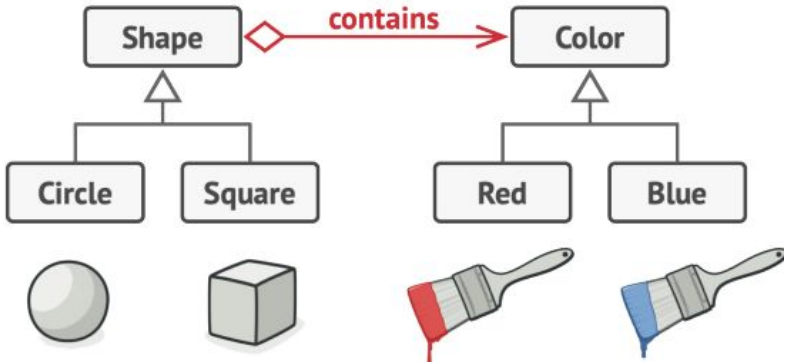
Structural Design Patterns - Bridge

الحل: قدم ال Bridge حلا جميلا لهذه المشكلة، فالمشكلة تكمن في أن بعدين مختلفين من الوظائف تم وضعهم ضمن نفس التسلسل الهرمي، لذلك، فإن ال Bridge يقوم على فصل هذا التسلسل الهرمي لجزئين أحدهما يمثل ال Abstraction، والآخر يمثل ال Implementation، وبهذا فأنت تخلصت من الوراثة في مقابل عمل composition لل object، ومن خلال هذا، فيمكن حل المشكلة السابقة من خلال فصل الألوان إلى classes خاصة بها، ومن ثم يقوم ال shape بعمل pointer إلى أحد ال Colors object، وبهذا فإن أي shape يمكنه الإتصال بأي color من خلال field الذي تم إنشائه

ليكون ك pointer بين كل تسلسل منهم،

هذا ال reference الذي قمنا بإنشائه بين هذان التسلسلان اسمه Bridge.

لاحظ من خلال هذا الأسلوب، أن الإضافة ستكون على أحد جوانب التسلسل، برتقالي وأصفر، وال Shapes لن تتأثر، بس ستتمكن من الوصول للألوان الجديدة.



Structural Design Patterns - Bridge

مثال:

1

يمثل هذا ال abstract class ال Abstraction Part، والذي يمثل أحد أجزاء التسلسل الهرمي، ويمثل ال Color ال pointer الذي سيقوم ببناء الجسر بين هذا الجزء المتعلق بال Shape والجزء الآخر المتعلق بال Color.

```php
/**
 * Abstraction Part for handling one of two hierarchies of classes
 * 2nees.com
 */
abstract class Shape
{
    private string $name;
    protected Color $color;

    public function __construct(Color $color)
    {
        $this->changeColorRenderer($color);
    }

    public function changeColorRenderer(Color $color): void
    {
        $this->color = $color;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setName(string $name): void
    {
        $this->name = $name;
    }

    abstract public function shapeDetails(): void;
}
```

2

تمثل هذه ال Classess ال Concrete Class، والتي ترث ال Shape ومن ثم تبني التمثيل الخاص بها.

```php
/**
 * Concrete Abstraction represents Circle
 * 2nees.com
 */
class Circle extends Shape
{
    public function __construct(Color $color)
    {
        parent::__construct($color);
        $this->setName("Circle");
    }

    public function shapeDetails(): void
    {
        echo "{$this->getName()} Filled with this
color: {$this->color->getFillColor()}" . PHP_EOL;
    }
}

class Rect extends Shape
{
    public function __construct(Color $color)
    {
        parent::__construct($color);
        $this->setName("Rect");
    }

    public function shapeDetails(): void
    {
        echo "{$this->getName()} Filled with this
color: {$this->color->getFillColor()} With Border
Color: {$this->color->getBorderColor()}" . PHP_EOL;
    }
}
```

Structural Design Patterns - Bridge

مثال:

3 يمثل هذا الجزء ال Implementation الخاصة بال interface، والذي يحتي على ال method التي يمكن أن يصل إليها ال part Abstraction.

```php
/**
 * The Implementation Part for declare needed method which can access via Abstraction
 * Interface Color
 */
interface Color
{
    public function getFillColor(): string;

    public function getBorderColor(): string;
}
```

4 يمثل هذا الجزء ال Concrete implementation، والذي سيقوم كل واحد منهم بعمل render لل colors التي يمثلها

```php
/**
 * Concrete Implementation render Red Colors
 * 2nees.com
 */
class Red implements Color
{
    public function getFillColor(): string
    {
        return "#ff0000";
    }

    public function getBorderColor(): string
    {
        return "#000000";
    }
}

class Blue implements Color
{
    public function getFillColor(): string
    {
        return "#0000ff";
    }

    public function getBorderColor(): string
    {
        return "#ffff00";
    }
}
```

Structural Design Patterns - Bridge

5

```php
// Call Concrete Implementation To Passing it to
Concrete Abstraction
$redColor = new Red();
$blueColor = new Blue();

// Create Circle
$myCircleShape = new Circle($redColor);
$myCircleShape->shapeDetails();
// Change Color on Run time
$myCircleShape->changeColorRenderer($blueColor);
$myCircleShape->shapeDetails();

// Create Rect
$myRectShape = new Rect($redColor);
$myRectShape->shapeDetails();
// Change Rect on Run time
$myRectShape->changeColorRenderer($blueColor);
$myRectShape->shapeDetails();
```

ال Client: في هذا الجزء قمنا بإنشاء الألوان التي سنستخدمها ومن ثم قمنا إنشاء الأشكال المطلوبة، وإرسال هذه الألوان لها...وبكل بساطة، سيتم الربط بين هذان التسلسلان، كما يمكن تغيير الألوان في ال real time بسهولة...

مثال: هل لاحظ كم أن الفكرة سهلة لكنها مهمة، وهل لاحظت كيف تم التخلص من مشكلة التمدد التي كانت ستحصل لو أبقينا كل الاعتمادية في مكان واحد...؟!

والآن، إليك الرابط الخاص بالمثال ^^

6

```
Circle Filled with this color: #ff0000
Circle Filled with this color: #0000ff
Rect Filled with this color: #ff0000 With Border Color: #000000
Rect Filled with this color: #0000ff With Border Color: #ffff00
```

وهذا مثال ثاني: الرابط يحاكي تطبيق آخر على هذا ال Pattern لتتضح الفكرة أكثر

Structural Design Patterns - Bridge

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. ال Abstraction: وهو يمثل الجزء الذي يعتمد على ال implementation لتنفيذ المهام المطلوبة منه، وهو أحد أجزاء التسلسل الهرمي، في مثالنا كان ال Shape يمثل نقطة البداية لل Abstraction
2. ال Implementation: وهو يمثل الجزء الثاني من التسلسل الهرمي، ويحتوي بداخله الشيفرة البرمجية الخاصة بالدوال التي يعتمد عليها ال Abstraction، ويمثل ال Color نقطة البداية لل Implementation.
3. Concrete Implementations: يمثل التطبيق الخاص بشيفرة برمجية معينة ليخدم هدف محدد، مثل ال Red, Blue classes في مثالنا.
4. Concrete Abstraction: تمثل مجموعة مختلفة من ال Classes التي تحتوي التطبيق الخاص بها معتمدة على ال Parent class، وخاضعة لل Implementation interface.

Structural Design Patterns - Bridge

إذا، متى يمكنني استخدام هذا ال Pattern؟

يمكنك استخدام هذا ال Pattern عند وجود مجموعة من ال classes المتجانسة فيما بينها وفيها العديد من ال functionality المختلفة، فبفصل هذه ال Classes إلى أكثر من تسلسل هرمي، يصبح التعديل أقل خطورة وأسهل، كما يمكنك استخدام هذا ال Pattern عندما ترى أن ال class يقوم بكل الوظائف بنفسه، وأن الوراثة ستتسبب في مشكلة التمدد، لأنك ستنقل العمل من ال class إلى ال object معتمدا على التسلسل الهرمي، بالإضافة إلى هذا فإنه يمكنك بكل بساطة تغيير ال implementation لل Object أثناء ال runtime إذا رغبت بذلك...جميع هذه الاحتياجات قد تجعلك تفكر في هذا ال Pattern.
من الأمثلة العملية على هذا ال Pattern إذا كان class معين يعمل مع أكثر من database، أو أن لديك مجموعة من الصفحات والتي لكل صفحة render مختلف ب type مختلف...ونحو ذلك.

Structural Design Patterns - Bridge

المميزات:

- يحقق مبدأ ال Single Responsibility Principle
- يحقق مبدأ ال Open/Closed Principle
- يمكن إضافة وإنشاء العديد من ال classes دون أن يؤثر ذلك على التطبيق، ويتخلص من مشكلة التمدد في الوراثة، ويتعامل ال client مع ال abstraction مباشرة دون الحاجة للعلم بتفاصيل ال implementation.

العيوب:

- مستوى التعقيد يزداد بشكل طردي مع كبر حجم ال class ومقدار التماسك بين مكوناته

Structural Design Patterns - Bridge

علاقة ال Bridge مع غيره من ال Pattern:

- الاستخدام الشائع لل Adapter يكون عند الحاجة لجعل classes غير متوافقة مع بعضها أن تعمل مع بعضها، بينما صمم ال Bridge ليقوم ببناء أجزاء التطبيق بشكل مستقل عن الأجزاء الأخرى.

- ال Bridge وال State وال Strategy وإلى حد ما ال Adapter جميعها Pattern قائمة أو تعتمد على ال composition، فهي تنقل العمل إلى objects أخرى، وحقيقة؛ فإن ال structures الخاص بهم متقارب، لكن كل واحد من هذه ال pattern يقدم حلا لمشكلة معينة، وهنا يكمن جمال ال pattern، فهو لا يمثل فقط حلا أو طريقة لكتابة الشيفرة البرمجية، بل يخبر المبرمجين الآخرين عند رؤيتهم للشيفرة البرمجية ما هي المشكلة التي تم حلها باستخدام هذا الpattern، وهذا يعيدنا في الذكريات إلى الوراء قليلا

- يمكن استخدام ال Abstract Factory مع ال Bridge، وهذا الأمر مفيد جدا إذا كان ال abstraction يعتمد على implementation محدد، فيمكن هنا عمل encapsulate حتى يتم إخفاء التعقيد عن ال client.

- يمكن استخدام ال Builder مع ال Bridge وذلك مثلا من خلال إنشاء director class، وذلك لأن ال director يمكنه أن يحتوي مجموعة ال role التي قمنا بإنشائها لل Abstraction، وكل builder بداخل ال director يمثل ال implementation، هذه حركة جميلة حقيقة... شاهد المثال (هذا المثال نسخة عن المثال الثاني، مع تعديل بسيط بإضافة ال director، وتغيير على client)

قال تعالى في سورة الحج: "ذَٰلِكَ وَمَن يُعَظِّمْ شَعَائِرَ اللَّهِ فَإِنَّهَا مِن تَقْوَى الْقُلُوبِ (32)"

فلتعلم يا أخي أن تعظيم شعائر الله صادر من تقوى القلوب، فالمعظم لها يبرهن على تقواه وصحة إيمانه، لأن تعظيمها، تابع لتعظيم الله وإجلاله.

Structural Design Patterns - Composite

ال Pattern الثالث ضمن هذه المجموعة هو ال Composite، هذا ال Pattern قائم على فكرة تشكيل ال Objects على شكل شجرة، ثم العمل مع هذا التشكيل (الشجرة) وكأن كل جزء منها individual object، ويكون السبب المنطقي الوحيد لاستخدام مثل هذا ال structure هو أن يكون ال core الخاص بال model عبارة عن شجرة، لذلك تحتاج إلى Pattern يمكنه تمثيل هذه الشجرة وتغطية متطلباتها، ومن متطلباتها أن تستطيع التعامل كل ورقة ضمن هذه الشجرة بشكل مستقل، وهذا ما يتحقق بال Composite.

Structural Design Patterns - Composite

المشكلة: تخيل أنك تريد كتابة شيفرة برمجية لتمثيل شجرة العائلة الخاصة بك، شجرة العائلة هذه تبدأ من الجد وثم تذهب باتجاه الأوراق حتى تصل إلى آخر ورقة في آخر غصن، وهو آخر حفيد ولد، ولنفرض أن المعلومات التي تحتاجها هي أن تعرف معلومات كل ورقة في هذه الشجرة، كاسمه ومكان عمله وعنوان سكنه، وترغب في جمع عدد الأبناء لهذه العائلة وكل فرع منها...، هنا ستجد أنك دخلت في شجرة لو رغبت بتنفيذها من خلال استخدام ال Concrete class، ستجد أن الموضوع معقد وصعب جدا، فجلب هذه البيانات كلها من مجموعة ال classes المختلفة وبناء البيانات الخاص بها لتتناسب مع كل عملية ستكون كبيرة وتحتاج إلى جهد كبير، بل قد يكون من المستحيل بنائها لأنك قد تدخل ب level متداخلة كثيرة جدا...إذا ما الحل؟

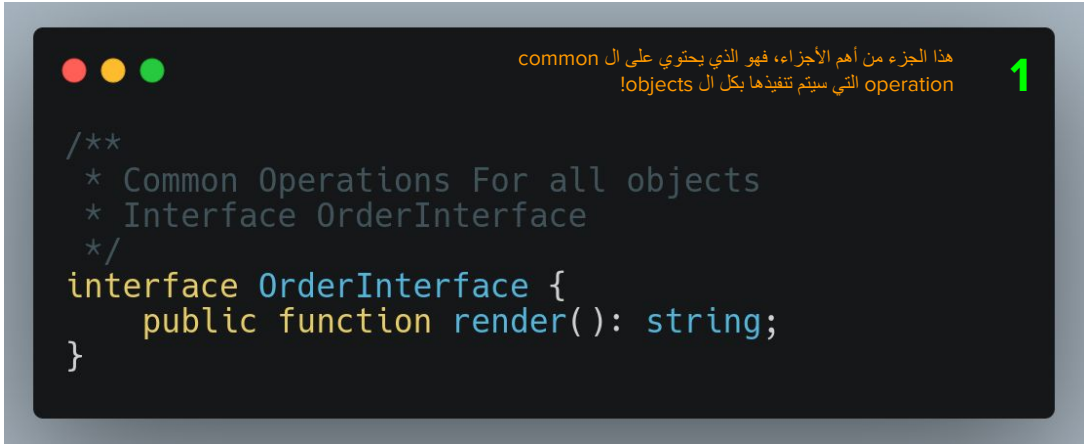
Structural Design Patterns - Composite

الحل: هذا ال Pattern يقترح الحل لهذه المشكلة، وبأسلوب بسيط وجميل، وإن سألت كيف هذا، فسيكون الجواب من خلال إنشاء interface تكون مستخدمة في جميع الأوراق ضمن هذه الشجرة، وبهذا فأنت تضمن أن كل ورقة لديها نفس ال methods المشتركة، لكن لكل ورقة منها التنفيذ الخاص بها، لهذا، فأنت لست محتاجا للنظر ما إذا كان ال Object معقد أو بسيط، ولا تريد معرفة أي شيء عن ال concrete class، لأنك ستتعامل مع ال method نفسها والتي هي من ال common interface.

ما زالت الفكرة غامضة؟!، لنذهب ونرى مثالا...

Structural Design Patterns - Composite

مثال: لنتخيل أننا نرغب ببناء شجرة من ال orders، بحيث يتم عرض المنتج وصاحب المنتج، لذلك، دعونا نشاهد طريقة التطبيق: -ملاحظة هذا مثال افتراضي، وفقط لتبسيط شرح العناصر المكونة لهذا ال Pattern، وستجد مثالا آخر بعده أكثر واقعية يمكنك الإطلاع عليه وتم رفعه مع الأمثلة-



Structural Design Patterns - Composite

مثال:

هذه ال Classes تمثل ال Leaf class، بحيث أن كل واحدة منها تقوم بتنفيذ العمليات الخاصة بها ومن ثم إرجاعها.

2

```php
/**
 * Leaf class
 * Class OrderItems
 * 2nees.com
 */
class OrderItems implements OrderInterface {
    private string $itemName;
    private string $itemID;
    private OrderOwner $orderOwner;

    public function __construct(string
$itemName, string $itemID, $orderOwner)
    {
        $this->itemName = $itemName;
        $this->itemID = $itemID;
        $this->orderOwner = $orderOwner;
    }

    public function render(): string
    {
        return "The {$this->orderOwner-
>render()} Buy: {$this->itemID}: {$this-
>itemName}";
    }
}
```

```php
/**
 * Leaf class
 * Class OrderOwner
 * 2nees.com
 */
class OrderOwner implements OrderInterface {
    private string $name;
    private string $phone;

    public function __construct(string $name,
string $phone)
    {
        $this->name = $name;
        $this->phone = $phone;
    }

    public function render(): string
    {
        return "Order Owner: {$this->name}:
{$this->phone}";
    }
}
```

Structural Design Patterns - Composite

```php
class Composite implements OrderInterface {
    private SplObjectStorage $members;

    public function __construct()
    {
        $this->members = new \SplObjectStorage();
    }

    public function addMember(OrderInterface $member){
        $this->members->attach($member);
    }

    public function removeMember(OrderInterface $member){
        $this->members->detach($member);
    }

    public function render(): string
    {
        $result = [];
        foreach ($this->members as $member){
            array_push($result, $member->render());
        }

        return json_encode($result, JSON_PRETTY_PRINT)
    }
}
```

هذا ال class يعتبر ال Container الخاص بأي Leaf تم إنشاءها أو أي Container آخر! بشرط أن يتبع نفع ال interface.

3

مثال:

ملاحظة: ال SplObjectStorage المستخدم في المثال ليس له علاقة بال Pattern، بل هو أحد ال PHP classes والمسؤولة عن تقديم map للتحويل Objects إلى Data أو شكل يتجاهل ما يحتويه من data، والفائدة منه في هذا المثال هو التحكم بالإضافة أو الحذف بسهولة، بالإضافة للتعامل مع ال objects بشكل مباشر...^^، لاحظ أنني في المثال أرسلت ال Object إلى ال removeMember، وسيتم حذفه مباشرة من خلال ال detach...

Structural Design Patterns - Composite

مثال:

والآن، إليك الرابط الخاص بالمثال ^^

هذا الجزء ال Clinet, لاحظ هنا أمرا مهما، أننا استطعنا إنشاء أكثر من Composite، وأضفنا Leaf أو أكثر بكل واحدة منها، واستطعنا إضافة Composite 2 إلى Composite جديد، مشكلين شجرة...

4

```php
$comp1 = new Composite();
$comp1->addMember(new OrderItems("Car", 20, new OrderOwner("Anees", 1234687)));
print_r($comp1->render());
echo PHP_EOL . "================================================" . PHP_EOL;

$comp2 = new Composite();
$comp2->addMember(new OrderItems("Bike", 15, new OrderOwner("Taher", 66234)));
print_r($comp2->render());
echo PHP_EOL. "================================================" . PHP_EOL;

$comp3  = new Composite();
$comp3->addMember($comp1);
$comp3->addMember($comp2);
$comp3->addMember(new OrderItems("OOO", 15, new OrderOwner("&_&", 4123)));
$comp3->addMember(new OrderItems("BBB", 16, new OrderOwner("0_)", 123)));
$comp3->addMember(new OrderItems("AAA", 12, new OrderOwner("*_*", 1532)));
print_r($comp3->render());

echo PHP_EOL. "================================================" . PHP_EOL;
$comp3->removeMember($comp2);
print_r($comp3->render())
```

```
[
    "The Order Owner: Anees:1234687 Buy: 20: Car"
]
================================================
[
    "The Order Owner: Taher:66234 Buy: 15: Bike"
]
================================================
[
    "[\n    \"The Order Owner: Anees:1234687 Buy: 20:
Car\"\n]",
    "[\n    \"The Order Owner: Taher:66234 Buy: 15:
Bike\"\n]",
    "The Order Owner: & &:4123 Buy: 15: OOO",
    "The Order Owner: 0_):123 Buy: 16: BBB",
    "The Order Owner: *_*:1532 Buy: 12: AAA"
]
================================================
[
    "[\n    \"The Order Owner: Anees:1234687 Buy: 20:
Car\"\n]",
    "The Order Owner: & &:4123 Buy: 15: OOO",
    "The Order Owner: 0_):123 Buy: 16: BBB",
    "The Order Owner: *_*:1532 Buy: 12: AAA"
]
```

Structural Design Patterns - Composite

```
==================================================
Array
(
    [0] => Array
        (
            [Id] => 4
            [Name] => Anees
            [Children] => Array
                (
                )

            [Address] => Array
                (
                    [street] => st. three
                    [city] => West Zarqa
                )

            [Details] => Array
                (
                    [Gender] => Male
                    [Age] => 30
                    [Job] => Array
                        (
                            [jobTitle] => Engineer
                            [salary] => 656
                        )

                )

        )

    [1] => Array
        (
```

مثال 2: المثال الثاني أكثر واقعية لتمثيل الفكرة بشكل صحيح، وهو التمثيل الخاص بشجرة العائلة، والذي لم أستخدمه بشرح المثال الأول لتتضح الفكرة بشكل سهل...، الآن أريد منك الإطلاع على هذا المثال، وبعد ذلك، يمكنك المتابعة في الشرائح لشرح مكونات هذا ال Pattern كما ذكر في الأمثلة.

Structural Design Patterns - Composite

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Component interface: ويمثل ال interface الذي يحتوي جميع ال Method المشتركة بين جميع العناصر ضمن هذه الشجرة.

2. Leaf: وهو ال Class الذي يمثل مكان العمل الحقيقي على الجزئية المراد إنجازها والحصول منها على النتائج المتوقعة.

3. Composite: هذا الجزء يمثل ال class الخاص بال Composite والذي يحتوي على أي Leaf تم إنشاءها أو أي Composite آخر! بشرط أن يتبع نفع ال interface.

Structural Design Patterns - Composite

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند حاجتك لبناء شجرة من المعلومات التي تعتمد على بعضها البعض، مثل شجرة العائلة، أو التسلسل الإداري في شركة ما أو مؤسسة ما كالجيش مثالا، كما يمكنك استخدام هذا الأسلوب عند حاجتك للتعامل مع Objects بسيطة أو معقدة لكن بنفس الشكل...، والسبب في هذا يعود لوجود interface مشتركة، فال method مشتركة، والنتائج موحدة المضمون مختلفة في الشكل.

Structural Design Patterns - Composite

المميزات:

- يحقق مبدأ ال Open/Closed Principle

- من خلال هذا ال Pattern يمكنك التعامل ال Tree Structure حتى ولو كانت معقدة، كما يمكن الاستفادة من خصائص كال polymorphism وال recursion

العيوب:

- من الصعب في بعض الأحيان من بناء interface واحدة مشتركة لكل الوظائف خصوصا بوجود functionality بعيدة عن بعضها البعض في ال Classes.

- قد يصعب فهم هذا ال Pattern إذا تنوعت الوظائف واختلفت ضمن ال Classes

Structural Design Patterns - Composite

علاقة ال Composite مع غيره من ال Pattern:

- يمكن استخدام ال Builder مع هذا ال Pattern، وذلك للتحكم بال Composite tree، لأنك من خلال ال Builder ستتمكن من التحكم بخطوات بناء الشجرة، ومن ثم إعادتها بشكل متكرر دون أي مخاوف.

- غالبا ما يستخدم ال Chain of Responsibility Pattern مع ال Composite، وفي هذه الحالة، فإن ال Chain of Responsibility سيقوم بتمرير ال request الخاص بأي leaf من جميع ال Parents إلى ال root tree.

- يمكنك استخدام ال Iterators للوصول ال Composite trees المطلوبة.

- يمكنك استخدام ال Visitor لتنفيذ عمليات على كامل ال Composite tress.

Structural Design Patterns - Composite

- يمكنك استخدام ال Flyweights مع ال composite للتخفيف من استهلاك ال RAM من خلال عمل implementation لل shared leaf من خلال هذا ال Flyweights.

- ال Decorator وال Composite يتشابهون كثير في ال Structure، فكلاهما يعتمدان على ال Recursive composition ومع هذا فهناك فرق جوهري وهو أن ال Decorator يضيف مسؤوليات إضافية لل Object، كما أنه لا يملك إلا child component واحدة فقط...

- بالنسبة لل Patterns مثل ال Decoder وال Composite والتي يعد ال Structure الخاص بها معقد ويصعب إنشائها في كل مرة من الصفر، يمثل ال Prototype Pattern حلالا جميلا لهذه المشكلة، فمن خلاله يمكنك نسخ ال Object بدلا من إعادة بنائه.

اللهم لا تجعل نفوسنا إلا طيبة، في أجساد طيبة، لا تخرج إلا حميدة، مبشرة بروح وريحان، ورب غير غضبان، برحمتك يا أرحم الراحمين يا اللّٰه.

Structural Design Patterns - Decorator

ال Pattern الرابع ضمن هذه المجموعة هو ال Decorator، هذا ال Pattern يقوم على فكرة إضافة سلوك جديد لل Object من خلال تعريف wrapper لل Objects التي تحتوي بداخلها السلوك الجديد، أي أنك تقوم بتشكيل ال Object لينفذ أو يقوم بمهام معينة، كل Object من هذه ال Objects يكون بداخل ال Wrapper، عملية إعادة التشكيل أو التصميم هذه، تسمى ب Decorator Design Pattern، ومن خلال إعادة التشكيل هذه، فأنت قد تمكنك من إضافة وظائف جديدة إلى أي class، ودون الحاجة إلى إجراء أي تغيير على ال classes الأخرى.

النقطة المهمة عند رغبتنا باستخدام هذا ال Pattern هي العلم بأننا نرغب بالتحكم بمجموعة متعددة أو متغيرة من المَسالِك والتي لا تنتمي إلينا بالأصل، أو ليست قطعة منا!، والتي يمكننا بأي لحظة أن نتخلص من إحداها، وهذا كمثل الملابس التي ترتديها في الشتاء، كلما كنت تشعر بالبرد أكثر، كلما ارتديت من الملابس ما يزيد من دفئك، ومع ذلك، فالملابس ليست قطعة منك بذاتها، ومن هذه الجزئية، يهتم هذا ال Pattern بأن يعمل delegate لل work من خلال ال Objects، ويتخلص من ال inheritance، وذلك لعدة أسباب، أهمها أننا سنتمكن من تغيير ال type أثناء ال runtime، والتخلص من مشكلة التمدد، لأنك في أغلب لغات البرمجة لن تستطيع وراثة أكثر من class مرة واحدة، وبهذا لن ترث إلا السلوك الخاص بال parent class...، وبهذا فإن هذا ال Pattern قائم على ال Aggregation وال Composition.

Structural Design Patterns - Decorator

المشكلة: تخيل أن لديك موقع إلكتروني تعرض فيه خدماتك لإنشاء المواقع الإكترونية، عندما بدأت هذا الموقع، كان يقدم خدمة تصميم موقع إلكتروني بسيط، فيه الصفحة الرئيسية والسعر، ومجموعة من الخيارات لهذا الموقع البسيط، ثم بعد ذلك تعلمت إنشاء المواقع من خلال drupal وال Yii، كما تعلمت كيف يمكنك إنشاء موقع إلكتروني بأكثر من لغة، وإضافة موقع للتقارير خاص يمكنك ربطها مع المواقع التي ستقوم بتصميمها، والآن أنت ترغب بإضافة هذه الخيارات لموقعك الإلكتروني، فما هي الخيارات المتاحة لديك لتنفيذ هذه الخيارات؟ علما أن المستخدم سيقوم بإضافة الخيارات من خلال ال UI، مثلا إضافة أكثر من لغة، أو إضافة صفحة للتقارير، واختيار نوع الموقع، أهو بسيط، أو drupal CMS أو Yii framework، هذه الخيارات كلها ستقوم بالإضافة والتعديل على السعر بشكل تراكمي، مثلا الموقع البسيط سعره 500، إذا رغب المستخدم باستخدام Yii، يضاف على المبلغ 1200، وهكذا...،علما أنك عند تصميمك الموقع، كان لديك method بسيطة، تقوم بإرجاع المعلومات المطلوبة...فما هو الحل الذي تتوقعه؟

Structural Design Patterns - Decorator

الحل: الحل لهذه المشكلة ممكن أن يكون من خلال الوراثة، لكن كما تحدثنا سابقا، فإن الوراثة في مثل هذه الحالات ستتسبب لنا في مشاكل عديدة كما ذكرنا سابقا، وكما ذكرنا في صفحة التعريف الخاص بهذا ال Pattern، بناءا على هذا، يكون الحل من خلال ال Aggregation أو ال Composition، أي أننا سنقوم بعمل delegate للعمل من خلال ال Objects، في الحالة التي طرحناها، يمكن بكل بساطة عمل Decorator، بحيث نعيد تشكيل ال Objects كما نريد، بكل بساطة، ما سنقوم بفعله هو المحافظة على ال Concrete Component والذي يمثل ال Basic or default implementation للمواقع الإلكتروني التي سنعمل عليها، ومن ثم تصميم Common interface لمشاركة ال method بين جميع العناصر، وعمل Base Decorator class ك Wrapper لجميع ال Object والتي سيتم إنشاء ال decorator class الجديدة من خلال هذا ال class، وبهذا الكل يشترك بنفس ال interface، والكل له نفس ال method، وكل واحد يقوم بتنفيذ الأعمال التي يريدها، لكنها بذات الوقت ترتبط فيها بينهما، لنذهب لنشاهد مثالا عمليا يوضح الفكرة، وهي سهلة جدا...

Structural Design Patterns - Decorator

مثال:

```php
/**
 * Interface WebsiteComponent - Component
 * 2nees.com
 */
interface WebsiteComponent {
    public function getDetails(): array;
}
```

هذا الجزء يمثل ال Component والي سيتم مشاركتها مع جميع objects

1

```php
/**
 * Class BasicWebsite - Concrete Component
 * 2nees.com
 */
class BasicWebsite implements WebsiteComponent {
    private float $price;

    /**
     * BasicWebsite constructor.
     * @param float $price
     */
    public function __construct(float $price)
    {
        $this->price = $price;
    }

    public function getDetails(): array
    {
        return [
            "Price" => $this->price,
            "Type" => "Basic",
            "Options" => ["HomePage", "EnLanguage", "ContactUs-EN", "AboutUS-EN"],
            "Description" => "Your looking to Basic Website and this is price is: {$this->price}$"
        ];
    }
}
```

هذا الجزء يمثل ال Concrete Component، وهو يمثل السلوك الافتراضي قبل التعديل من أي decorator في المستقبل...

2

Structural Design Patterns - Decorator

مثال:

يمثل هذا الجزء ال Base Decorator، وهو أحد أهم المكونات في هذا ال Pattern، وفكرته بكل بساطة هي عمل reference field لل Wrapped Object ، وكما تلاحظ في ال method، فتم عمل delegate للوظيفة المطلوبة لنفس ال object.

3

```php
/**
 * Class WebsiteBaseDecorator - Base Decorator
 * 2nees.com
 */
class WebsiteBaseDecorator implements WebsiteComponent {
    protected WebsiteComponent $websiteComponent;

    public function __construct(WebsiteComponent $websiteComponent)
    {
        $this->websiteComponent = $websiteComponent;
    }

    public function getDetails(): array
    {
        return $this->websiteComponent->geDetails();
    }
}
```

يمثل هذا الجزء ال Concrete Decorator، وهو يمثل مكان التشكيل أو إعادة التصميم، ويتم ذلك من خلال إضافة method جديدة أو عمل override لل method الموجودة...

4-أ

```php
/**
 * Class MultiLanguageWebsite - Concrete Decorators
 * 2nees.com
 */
class MultiLanguageWebsite extends WebsiteBaseDecorator {
    public function getDetails(): array
    {
        $base = parent::getDetails();
        $price = 600 + $base["Price"];

        return [
            "Price" => $price,
            "Type" => "{$base['Type']} + MultiLanguage",
            "Options" => array_merge($base['Options'],
["HomePage-AR","ContactUs-AR", "AboutUS-AR", "ArLanguage"]),
            "Description" => "Your looking to Basic
MultiLanguage Website and this is price is: {$price}$"
        ];
    }
}
```

Structural Design Patterns - Decorator

```php
/**
 * Class ReportsDashboardWebsite - Another Concrete
Decorators
 * 2nees.com
 */
class ReportsDashboardWebsite extends WebsiteBaseDecorator {
    public function getDetails(): array
    {
        $base = parent::getDetails();
        $price = 444 + $base["Price"];

        $options = $base['Options'];
        array_push($options, "DashBoard-En");
        if(in_array("ArLanguage", $options)){
            array_push($options, "DashBoard-AR");
        }

        return [
            "Price" => $price,
            "Type" => "{$base['Type']} +
AddReportsDashboard",
            "Options" => $options,
            "Description" => "Your looking to Basic Report
Website and this is price is: {$price}$"
        ];
    }
}
```

يمثل هذا الجزء جميع ال Concrete Decorator المستخدمة في هذا المثال بالإضافة إلى 4-أ

ب-4

```php
/**
 * Class DrupalWebsite - Another Concrete Decorators
 * 2nees.com
 */
class DrupalWebsite extends WebsiteBaseDecorator {
    public function getDetails(): array
    {
        $base = parent::getDetails();
        $price = 200 + $base["Price"];

        return [
            "Price" => $price,
            "Type" => "{$base['Type']} + Drupal",
            "Options" => array_merge($base['Options'],
["user1"]),
            "Description" => "Your looking to Drupal CMS
Website and this is price is: {$price}$"
        ];
```

```php
/**
 * Class YiiWebsite - Another Concrete Decorators
 * 2nees.com
 */
class YiiWebsite extends WebsiteBaseDecorator {
    public function getDetails(): array
    {
        $base = parent::getDetails();
        $price = 1200 + $base["Price"];

        return [
            "Price" => $price,
            "Type" => "{$base['Type']} + Yii",
            "Options" => array_merge($base['Options'],
["AdminPage", "useDesignPattern", "WorkWithMySql",
"WorkWithRedis"]),
            "Description" => "Your looking to Yii Website and
this is price is: {$price}$"
        ];
    }
}
```

Structural Design Patterns - Decorator

5

يمثل هذا الجزء ال Client، لاحظ مقدار الجمال في هذا التشكيل، بكل بساطة، ال Client لزمه فقط أن يرسل ال decorator المطلوب، وسيحصل على النتائج التي يرغب، وإن لم يرسل أي شيء، الشيفرة البرمجية ستعمل على الخيار الافتراضي لأن كل ال decorator ليست من صلب ال basic website، كما أنك تتحكم في الشروط والنتائج، ويمكنك إضافة العديد من الخطوات الأخرى أو حذفها...

```php
// Fake Checkbox/Radio from UI
$miltiLangauge          = true;
$linkWithReportWebsite  = true;
$websiteType = 3;// 1 or 2, 3

// Call Basic or Default Implementation
$website = new BasicWebsite(500);
// Add Multi MultiLanguage
if($miltiLangauge) {
    $website = new MultiLanguageWebsite($website);
}
// Link website with report website dashboard
if($linkWithReportWebsite) {
    $website = new ReportsDashboardWebsite($website);
}
// Create Professional Website Drupal or Yii
if($websiteType === 2) {
    $website = new DrupalWebsite($website);
}elseif ($websiteType === 3) {
    $website = new YiiWebsite($website);
}

// Print final result (Note: You can print result in any
place above and you can use result as you want)
print_r($website->getDetails());
```

مثال:

والآن، إليك الرابط الخاص بالمثال ^^

* يمكنك تجربة إنشاء مثال للتحكم بإرسال الإشعارات، وليكن الافتراضي ال Email، وعند الحالات الطارئة sms، وemail، مع إمكانية إرسال إشعارات slack و whatsapp عند حدوث Error.

6 النتائج

```
Array
(
    [Price] => 2744
    [Type] => Basic + MultiLanguage + AddReportsDashboard + Yii
    [Options] => Array
        (
            [0] => HomePage
            [1] => EnLanguage
            [2] => ContactUs-EN
            [3] => AboutUS-EN
            [4] => HomePage-AR
            [5] => ContactUs-AR
            [6] => AboutUS-AR
            [7] => ArLanguage
            [8] => DashBoard-En
            [9] => DashBoard-AR
            [10] => AdminPage
            [11] => useDesignPattern
            [12] => WorkWithMySql
            [13] => WorkWithRedis
        )

    [Description] => Your looking to Yii Website and this is price is: 2744$
)
```

Structural Design Patterns - Decorator

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Component interface: ويمثل ال interface الذي يحتوي جميع ال Method المشتركة بين جميع العناصر لل Wrappers & Wrapped Objects، وفي مثالنا WebsiteComponent

2. Concrete Component: يمثل هذا الجزء ال Default class والذي يحتوي بداخله السلوك الافتراضي للعمل، وبداخله مجموعة الدوال القابلة للتعديل من خلال ال decorators، في مثالنا: BasicWebsite

3. Base Decorator: وهو ال class الذي يحتوي بداخله reference لل field الذي يمثل ال wrapper object، هذا ال decorator أهم ما يقوم به هو عمل delegate للوظائف المطلوبة، بحيث يقوم كل جزء على تنفيذ الدالة الخاصة به بما فيها من شيفرة، في مثالنا: WebsiteBaseDecorator

4. Concrete Decorators: وهو يمثل ال class الذي يرث ال Base Decorator، ووظيفته إعادة تشكيل وعمل override لل methods الأساسية حسب احتياج كل Decorator، أي أن هذا الجزء هو المسؤول عن تغيير السلوك الأصلي، ومن مثالنا نذكر ال YiiWebsite.

Structural Design Patterns - Decorator

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند وجود أكثر من Layer (كما في مثالنا Yii, Drupal, Languages, Reports)، كل Layer يُلزمك بإضافة أو تغيير السلوك الخاص ب Object أثناء ال runtime ودون الخوف من ال breaking لل code إذا تغير أو اختلف أي layer (كما في مثالنا، عند التحكم بإضافة اللغات أو التقارير، او إضافة دروبال أو Yii كود)، وهذا كله ممكن لأن ال pattern يعتمد على ال common interface، كما أن هذه ال Pattern مفيد جدا عندما يكون استخدام الوراثة غير ملائم أو في غير محله كما في الحالة التي تطرقنا إليها، أو عند عدم إمكانية وراثة السلوك مثل استخدام final والتي ستمنعك من التعديل على السلوك الخاص بال method أو على مستوى ال class ككل.

Structural Design Patterns - Decorator

المميزات:

- من خلال هذا ال Pattern يمكنك وراثة السلوك الخاص بال Object دون الحاجة إلى إنشاء sub classes جديدة لتغطية جميع الاحتمالات الممكنة
- يمكن إضافة أو حذف العديد من ال layers أثناء ال runtime.
- يمكن استخدام ال object مع أكثر من decorator، كما يمكنك object واحد لحفظ جميع النتائج الخاصة بالطبقات.
- يحقق مبدأ ال Single Responsibility Principle

العيوب:

- من الصعب حذف ال Wrapper لأن التأثير سيكون على كل ما تحته.
- من الصعب إضافة decorator لا يعتمد سلوكه على ال order الخاص بالتنفيذ لل decorators

Structural Design Patterns - Decorator

علاقة ال Decorator مع غيره من ال Pattern:

- في ال Adapter فإننا نقوم بعمل تغيير على مستوى ال interface لل current object، بينما يقوم ال Decorator بعمل تحسين على ال object الحالي بدون تغييره، وهو يدعم ال recursive composition بينما لا يدعمه ال Adapter.

- ال Adapter يمكن لأكثر من interface أن تعمل wrap لل object، بينما في ال Decorator يحسن على ال interface الموجودة، وال Proxy يقدم نفس ال interface.

- ال Chain of Responsibility وال Decorator قريبين من بعضهم البعض بشكل كبير، لكن هناك فرق خطير، فالأول يمكنه إيقاف ال process عند أي نقطة، بينما ال decorator لا يمكنه القيام بذلك.

Structural Design Patterns - Decorator

- ال Decorator وال Composite يتشابهون كثير في ال Structure، فكلاهما يعتمدان على ال Recursive composition ومع هذا فهناك فرق جوهري وهو أن ال Decorator يضيف مسؤوليات إضافية لل Object، كما أنه لا يملك إلا child component واحدة فقط...

- بالنسبة لل Patterns مثل ال Decoder وال Composite والتي يعد ال Structure الخاص بها معقد ويصعب إنشائها في كل مرة من الصفر، يمثل ال Prototype Pattern حلالا جميلا لهذه المشكلة، فمن خلاله يمكنك نسخ ال Object بدلا من إعادة بنائه.

- ال Decorator وال Proxy كلاهما متشابهان أيضا ويستخدمون نفس المبدأ، لكن الفرق الأساسي بينهم أن ال Proxy يتحكم بال life cycle لل object من خلاله، بينما تتم هذه العملية في ال decorator من خلال ال client.

فلْتعلمْ أننا أمرنا أن نقيم الصلاة بأركانها وشروطها وسننها، ولْتعلمْ بأن الصلاة راحة وطمأنينة للنفس والقلب، وفيها قول رسول الله -صلى الله عليه وسلم-: "يا بلالُ أقمِ الصلاةَ، أرِحْنا بها"

Structural Design Patterns - Facade

ال Pattern الخامس ضمن هذه المجموعة هو ال Facade، هذا ال Pattern يقدم فكرة لإخفاء التعقيدات عن ال Client، فبدلا من أن "يدوخ" المبرمج أثناء ربط ال Classes وال interfaces وتذكر تسلسلها والقيام بهذه العملية مرارا وتكرارا، يقوم هذا ال Pattern على إنشاء واجهة تقوم بإخفاء التعقيدات الموجودة لأي مكتبة أو framework أو مجموعة من ال classes بداخلها، وكل ما على المبرمج القيام به هو عمل call لهذه ال method، وبهذا تكون فكرة هذا ال Pattern تبسيط الشيفرة البرمجية من خلال interface عن طريق إخفاء التعقيدات.

Structural Design Patterns - Facade

المشكلة: تخيل أنك ستستخدم third-party library داخل المشروع الخاص بك، هذه المكتبة معقدة وتحتاج إلى ترتيب معين لتنفيذ الإجراءات المطلوبة، في هذه الحالة سيكون لديك العديد من ال Objects، وهذه ال Objects ستستخدم في كل مرحلة حسب الترتيب المناسب لها ضمن السلسلة، هذا الأمر وبناءا على ال Business Logic سيجعل من الشيفرة البرمجية مترابطة بشكل محكم مع هذه المكتبة، وهذا سيجعل من صيانة الشيفرة البرمجية أو تعديلها في المستقبل أمرا صعبا جدا!، كما أن الأخطاء التي قد تنتج عن ال client قد تكون كبيرة كلما زاد تعقيد وعدد ال calls، إذا ما الحل؟

الحل: يقدم هذا ال Pattern حلا جميلا، وذلك من خلال إنشاء class يمثل interface لأي Complex Code يحتوي بداخله الكثير والعديد من ال methods وال classes، لذلك، وبناءا على هذا، فإننا نقوم بإنشاء Facade ومن ثم إنشاء دوال تقوم بتنفيذ ما نحتاجه فقط بالترتيب والتسلسل الصحيح، ودون الحاجة إلى استخدام كل ما في المكتبة أو ال classes، ودون الحاجة لمشاركة هذه التعقيدات مع ال Client.

Structural Design Patterns - Facade

مثال: التطبيقات التي يمكن تطبيقها على هذا ال Pattern كثيرة، لكن لنأخذ فكرة ال Share على وسائل التواصل الاجتماعي!، نحن ندرك تماما أن فكرة ال share بسيطة، كل ما يزلمنا القيام به هو استدعاء المكتبة المطلوبة ومن ثم عمل share، وقد يكون لدينا بالأساس interface مشتركة لل social media، نستخدمها لعمل implement حسب الاحتياج، لكننا بنفس الوقت، نقوم بعمل نفس هذا ال implement لكل وسائل التواصل حينما نحتاج إلى عمل share، لذلك، دعونا أن نقوم بتجربة لاستخدام ال Facade ونرى النتائج ^^.

1 يمثل هذا الجزء ال subsystem، أي الجزء المعقد الذي نريد اختزال تعقيده، وكما تلاحظ فلدي مجموعة من وسائل التواصل التي نستخدمها لعمل share.

```php
// 2nees.com
abstract class SocialMedia {
    protected string $message;

    public function setMessage(string $message): void
    {
        $this->message = $message;
    }

    abstract public function share(): void;
}
```

```php
// 2nees.com
class Facebook extends SocialMedia {
    public function share(): void
    {
        echo "Facebook Shared this message: {$this->message}" . PHP_EOL;
    }
}

class Twitter extends SocialMedia {
    public function share(): void
    {
        echo "Twitter Shared this message: {$this->message}" . PHP_EOL;
    }
}

class LinkedIn extends SocialMedia {
    public function share(): void
    {
        echo "LinkedIn Shared this message: {$this->message}" . PHP_EOL;
    }
}
```

Structural Design Patterns - Facade

هذا الجزء يمثل طريقة عمل share لكل وسائل التواصل...لكننا نرغب في اختزال هذا الجزء...لذلك سنقوم بالاستغناء عنه وإنشاء Facade.

```php
// 2nees.com
$facebook = new Facebook();
$facebook->setMessage($message);
$facebook->share();

$twitter  = new Twitter();
$twitter->setMessage($message);
$twitter->share();

$linkedIn = new LinkedIn();
$linkedIn->setMessage($message);
$linkedIn->share();
```

2

هذا ال Class يمثل ال Facade، والذي بدوره سيكون الواجهة التي سنستخدمها لتنفيذ المهام المطلوبة بعد إخفاء التعقيدات...

```php
class SocialMediaFacade {
    protected Facebook $facebook;
    protected Twitter $twitter;
    protected LinkedIn $linkedIn;

    public function __construct(Facebook $facebook,
Twitter $twitter, LinkedIn $linkedIn)
    {
        $this->facebook = $facebook;
        $this->twitter = $twitter;
        $this->linkedIn = $linkedIn;
    }

    public function setMessage($message): void {
        $this->facebook->setMessage($message);
        $this->twitter->setMessage($message);
        $this->linkedIn->setMessage($message);
    }

    public function share(): void {
        $this->facebook->share();
        $this->twitter->share();
        $this->linkedIn->share();
    }
}
```

3

لاحظ أننا قمنا بتنفيذ الشيفرة البرمجية عند ال client بالشكل المطلوب، ودون الحاجة للدخول بكافة التعقيدات أو الوقوع بأي أخطاء غير موقعة...شاهد الفرق بين هذا التنفيذ والذي بالأعلى...

```php
// 2nees.com
$facade = new SocialMediaFacade(new Facebook(), new Twitter(), new LinkedIn());
$facade->setMessage("{$message} (We Are Using Facade Now)");
$facade->share();
```

Structural Design Patterns - Facade

```php
// 2nees.com
class SocialMediaFacade {
    protected Facebook $facebook;
    protected Twitter $twitter;
    protected LinkedIn $linkedIn;

    public function __construct()
    {
        $this->facebook = new Facebook();
        $this->twitter  = new Twitter();
        $this->linkedIn = new LinkedIn();
    }

    public function setMessage($message): void {
        $this->facebook->setMessage($message);
        $this->twitter->setMessage($message);
        $this->linkedIn->setMessage($message);
    }

    public function share(): void {
        $this->facebook->share();
        $this->twitter->share();
        $this->linkedIn->share();
    }
}
```

ملاحظة: يمكنك بناء ال Facade يما يلبي احتياجاتك، فمثلا في الصورة السابقة قمنا بإرسال الشركات من خلال ال Facade إلى ال construct، ويمكن كما في الصورة أن نستغني عن ذلك!، ويمكنك أن تجعل هذه القيم اختيارية، كل ذلك حسب ما يقتضيه سير العمل.

```php
// 2nees.com - New Way (After use Facade)
$facade = new SocialMediaFacade();
$facade->setMessage("{$message} (We Are Using Facade Now)");
$facade->share();
```

Structural Design Patterns - Facade

```
Facebook Shared this message: 2nees.com Facade Design Pattern.
Twitter Shared this message: 2nees.com Facade Design Pattern.
LinkedIn Shared this message: 2nees.com Facade Design Pattern.
=====================================================
Facebook Shared this message: 2nees.com Facade Design Pattern. (We Are Using Facade Now)
Twitter Shared this message: 2nees.com Facade Design Pattern. (We Are Using Facade Now)
LinkedIn Shared this message: 2nees.com Facade Design Pattern. (We Are Using Facade Now)
```

4

والآن، إليك الرابط الخاص بالمثال ^^

لاحظ في هذا المثال مع بساطته وقلة مستوى التعقيد في الجزء المراد اختزاله، إلا أن عدد الأسطر البرمجية التي تم التخلص منها كان ممتازا، تخيل لو أن لديك عشرة أو أكثر من وسائل تواصل!، وقس على ذلك!

ومن التطبيقات التي يمكن أن تجرب بنائها، مثالا اختزال التعقيد في عملية ال upload لملف فيديو ومن ثم تحويله إلى ملف صوتي، ومن ثم ضغطه وحفظه...، لاحظ هنا أننا سنعتمد على العديد من المكاتب، لكنك يمكنك اختزال التعقيد بكل بساطة من Facade class، وإنشاء دالة اسمها مثلا convert!

ملاحظة: يمكنك إنشاء أكثر من Facade إذا كانت ال method المطلوبة غير مرتبطة بنفس ال Facade class الذي ستقوم ببنائه، وهذا سيحافظ على مبادئ التصميم، وسيجعل من أمر الصيانة أسهل، كما أن ال Facade class الجديد يمكن استخدامه داخل Facade class أو من خلال ال client.

Structural Design Patterns - Facade

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Facade: وهو ال Class المسؤول عن اختزال التعقيد وتقديم interface لمجموعة من الوظائف المحددة والتي نحتاجها لتنفيذ مجموعة المهام التي لدينا
2. Additional Facade (اختياري): يمكنك إضافة Facade إضافية لتنفيذ مهام أخرى لا تتعلق بنفس ال Facade الأول، وهذا يحافظ على مبادئ التصميم التي لدينا، ويقلل من مستوى التعقيد الخاص بال Facade مما يسهل صيانته في المستقبل.
3. Complex Subsystem: أي جزء يحتوي على مجموعة كبيرة من ال Objects المترابطة فيما بينها للقيام بمهمة معينة، هذا الجزء هو ما نريد اختزاله.
4. ال Client: بدلا من أن يتم استخدام ال sub system مباشرة، يستم استخدام ال Facade class الذي تم إنشاؤه.

Structural Design Patterns - Facade

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند وجود مجموعة مترابطة من ال classes والتي ترغب بضمان تنفيذها أو الحفاظ على ترتيبها واختزال التعقيد منها في أكثر من مكان لمجموعة معينة من ال methods، وعادة ما يكون الهدف منه اختصار عمليات ال calls وما يتم عليها من process بعد ذلك، والتي ستزداد تعقيدا حتى بوجود واستخدام ال patterns مع تزايد حجم التطبيق وعدد ال classes وال sub classes ونحو ذلك، ويكون ال Facade حلا جميلا لهذه المشكلة، كما يمكنك من خلاله بناء layers بحيث تربط أكثر من Facade مع بعضهم البعض ليصبح التواصل بينهم اختزالا للتعقيد على مستوى أكثر من sub system!...فكرة بسيطة، لكنها مفيدة جدا!

Structural Design Patterns - Facade

المميزات:

- عزل الشيفرة البرمجية الخاصة بي والمسؤولة عن تنفيذ المهمات للجزء المعقد عن ال client.
- عند حدوث أي مشكلة أو تحديث أو تعديل، سيتم ذلك على ال Facade مباشرة.

العيوب:

- هذا ال Pattern يمكن أن يتسبب بجعل ال Object يعرف كثيرا أو يقوم بعمل الكثير، وهذه تعد واحدة من الأمور التي تندرج تحت باب ال code-smell والتي تنتهك مبادئ التصميم anti-pattern.

Structural Design Patterns - Facade

علاقة ال Facade مع غيره من ال Pattern:

- في ال Adapter فإنك تحاول جعل ال interface الحالية قابلة للاستخدام وذلك يظهر من خلال طريقة عمله، بينما ال Facade يقوم بتعريف interface جديدة لل current object، وهذا يعني أن ال Adaper يعمل مع ال object ك wrapper له، بينما يتعامل ال Facade مع كامل مكونات ال object.

- ال Abstract Factory يمكن أن يكون بمثابة بديل لل Facade وذلك إن كان المطلوب هو فقط إخفاء عملية ال create لل Objects.

- ال Facade يقوم على فكرة جعل Object واحد يمثل أكبر قدر من البيانات ويختزل أكبر قدر من التعقيد، بينما ال Flyweight يقوم على فكرة بناء الكثير من عدد قليل من ال Objects.

Structural Design Patterns - Facade

- ال Facade وال Mediator قريبين من بعضهم، فوظيفة كل واحد منهما هي محاولة تنظيم وبناء علاقة بين مجموعة ال Classes، لكن ال Facade يقدم interface جديدة لكنه لا يقدم وظائف جديدة، فال Objects لديك وأنت تقوم باستدعاء ما تطلبه من ال sub system مباشرة، بينما ال Mediator يكون هو حلقة الوصل بين ال Component المختلفة، ولا يمكن لل component من الوصول المباشر ل component أخرى، إلا عن طريق ال Mediator والذي يمثل حلقة الوصل فيما بينهم.

- يمكن تحويل ال Facade إلى Singleton، لأن ال Facade لديه single object، وعادة ما يكون هذا Object كاف لتنفيذ المطلوب ويمكن استخدامه على مستوى كل المشروع.

- ال Proxy مشابه لل Facade من ناحية تخزين أو إعداد ال Object، لكن ال Proxy لديه نفس ال interface الخاصة بال Object مما يجعل هذا ال Object قابل للتبادل مع ال interface الخاصة بال Proxy وال Service Object.

قال بعض البلغاء: "دولة الجاهل عبرة العاقل"،
وقال بعض الحكماء: "الحاجة إلى العقل أقبح من الحاجة إلى المال"
كتاب أدب الدين والدنيا - ص 55

قلت، والعاقل من أبصر مواطن الجهل في نفسه قبل غيره، والعاقل من اتعظ بجهل غيره، والعاقل من سعى لإثراء عقله قبل جيبه!، والعاقل من أدرك أن كل ما مضى ما هو إلى وسيلة للتقرب إلى اللّٰه - سبحانه وتعالى- وفقه ما أنزل إلينا. الحمد لله!

Structural Design Patterns - Flyweight

ال Pattern السادس ضمن هذه المجموعة هو ال Flyweight، هذا ال Pattern قائم على فكرة مشاركة البيانات أو جزء منها مع ال Objects المختلفة، بحيث تمثل هذه البيانات قيمة مشتركة يمكن أن تتكرر في أكثر من Object، وهذا كله بدلا من أن نقوم بإنشاء Object في كل مرة يحتوي على نفس البيانات...

هذا الأمر له تأثير مهم على ال RAM، فبدلا من أن نضيع مساحة كبيرة من ال RAM في بناء Object يحتوي على بيانات تم إضافتها من قبل، نقوم بإنشاء هذه البيانات المشتركة وإرجاعها في كل مرة يتم طلب Object فيه نفس هذه الخصائص، وبناءا على هذه التفاصيل، فإن هذا ال Pattern يكثر استخدامه في الألعاب وقد يستخدم في بعض التطبيقات، لكن في مجال الويب، نادرا ما يستخدم، فمثلا ال PHP single thread، وبهذا فهو لن يقوم بحفظ كل ال Application داخل Object ولن يقوم بحفظ هذا ال Object داخل ال Memory ل single thread واحدة!...

Structural Design Patterns - Flyweight

المشكلة: تخيل أن لديك لعبة، هذه اللعبة فيها صور أشجار وسيارات ونحو ذلك، وأنت ترغب في تكرار ظهور السيارات بعدة أنواع محددة مسبقا مثل مرسيدس و تويوتا وكيا، وكذلك ترغب بتكرار زراعة الشجر في اللعبة ولعدة أنواع مختلفة، مثل التفاح والمشمش والخوخ والتين، للوهلة الأولى، ستقوم بإنشاء جملة دوران وطباعة عدد الشجر المطلوب، وعدد السيارات المطلوب...، لكن سرعان ما ستكتشف أنك وقعت في كارثة ستتسبب في القضاء على ال RAM عندك، فمثلا لو طلبت منك أن تتواجد في هذه اللعبة 10 آلاف سيارة، 90 ألف شجرة...، لو فرضنا أن حجم كل صورة هي فقط 10KB ودون احتساب أي خصائص أخرى مثل اللون او الحجم أو أو السرعة أو أي شيء آخر، فستكون النتيجة هي 1Giga!، ستستهلك 1 جيجا من ال RAM فقط على هذه الخصائص، ماذا لو قلنا أن لديك الشارع والمنازل والبنادق وصور الشخصيات...إلى آخره، بكل تأكيد، ستجد كل الأجهزة تحترق "وتتمنى أن لا ترى شيفرة برمجية كهذه بعد ذلك" ٨٨، إذا ما الحل؟

Structural Design Patterns - Flyweight

الحل: يقدم هذا ال Pattern حلا جميلا لهذه المشكلة، فبدلا من إنشاء ال Object في كل مرة وتكرار البيانات، يقترح هذا ال Pattern بفصل الجزء الخاص بالبيانات التي ستكرر دوما دون أن تتغير قيمتها، فمثلا بدلا من إنشاء 100 ألف شجرة، سنقوم ببساطة بإنشاء شجرة واحدة ومن نجعل جميع الأرض في اللعبة تستعدي هذا ال Object، وبدلا من إنشاء 10 آلاف سيارة، يمكن إنشاء سيارة واحدة من كل نوع، أو 10 سيارات لكل نوع، وكل سيارة لديها 10 ألوان!، بالنهاية النتيجة ستكون صادمة، فمثلا قلنا أن 90 ألف شجرة = 900MB، فإن شجرة واحدة = 10KB!، هل تتخيل كم الفرق!، وإن سألت كيف يمكن هذا، فنقول أن هذا ممكن من خلال إنشاء مصفوفة فيها قائمة العناصر التي تم إنشاؤها من ال State التي يمكن أن تتشارك، مثلا الشجرة والسيارة، فإذا تم إرسال سيارة لون أحمر، يتم التأكد أولا هل السيارة ذات اللون الأحمر موجودة أم لا، فإن كانت موجودة فنرجعها نفسها، وإلا نقوم بإنشاء سيارة جديدة ونضيفها للقائمة، فكرة بسيطة، لكنها مفيدة...، هذا العملية تلزم وجود عدة أجزاء، دعونا نتعرف عليها من خلال مثال عملي...

Structural Design Patterns - Flyweight

مثال: لنأخذ الشجر من مثالنا السابق
ولنحاول تمثيل نفس الفكرة من
خلال ال Pattern.

```php
/**
 * Class TreeType - This flyweight class which its contain a unique state
 ** When you look to this simulate its a big size in KB nested of B
 * 2nees.com
 */
class TreeType {
    const TREE_NAME_LIST = [
        "Apple",
        "Orange",
        "Banana"
    ];

    const TREE_SIZE_LIST = [
        "10x10",
        "10x15",
        "20x20"
    ];

    private string $name;
    private string $size;
    private string $image;

    public function __construct(string $name, string $size, string $image)
    {
        $this->name     = $name;
        $this->size     = $size;
        $this->image    = $image;
    }

    public function draw(): string {
        return "name: {$this->name}, size: {$this->size}, img: {$this->image}";
    }
}
```

هذا ال Class يمثل ال Flyweight، فهو الحاوي لجميع ال state data التي تشترك بها جميع الأشجار في اللعبة الخاصة بنا....

1

Structural Design Patterns - Flyweight

مثال:

```php
/**
 * Class TreeFactory - Flyweight factory which its used to create a flyweight object or re-use exists one
 * 2nees.com
 */
class TreeFactory {
    /**
     * @var TreeType[]
     */
    private static array $treeTypes = [];

    public static function getTreeType(string $name, string $size, string $image): TreeType
    {
        $key = md5($name.$size.$image);
        if(!array_key_exists($key, self::$treeTypes)){
            self::$treeTypes[$key] = new TreeType($name, $size, $image);
        }

        return self::$treeTypes[$key];
    }

    public static function getAllTreesType(): void {
        echo "Total Count For Unique Object Saved on RAM: " . count(self::$treeTypes) . PHP_EOL;
        print_r(array_map(fn(TreeType $treeType) => $treeType->draw(), self::$treeTypes));
    }
}
```

يمثل هذا ال Class ال Flyweight Factory، وهو الجزء المسؤول عن إنشاء أي Flyweight object جديد إذا لم يكن موجوداً، أما اذا كان موجودا فنعيده نفسه..

2

Structural Design Patterns - Flyweight

مثال:

```php
/**
 * Class Tree - This is context which its create contextual Object, and you
can create a lot and alot of these object
 ** and you will be safe since its too small...
 */
class Tree {
    private int $x;
    private int $y;
    private TreeType $treeType;

    public function __construct(int $x, int $y, TreeType $treeType)
    {
        $this->x = $x;
        $this->y = $y;
        $this->treeType = $treeType;
    }

    public function draw(): string {
        return  "x: {$this->x}, y: {$this->y}, {$this->treeType->draw()}}";
    }
}
```

يمثل هذا الجزء ال Context الخاص بإنشاء مجموعة ال Object ذات الخصائص المتغيرة مضافة إليها العناصر ال Unique، بحيث يربتط كل Object من هذا ال context مع ال Flyweight class، ويمكنك هنا إنشاء الملايين منه وأنت مطمئن ^^

3

Structural Design Patterns - Flyweight

مثال:

```php
/**
 * Class GameLand - This is the Flyweight for client, which its return or represent data for client
 * 2nees.com
 */
class GameLand {
    private array $trees;

    public function draw(): array
    {
        return array_map(fn(Tree $tree) =>
           $tree->draw()
        , $this->trees);
    }

    public function addTree($x, $y, $name, $size, $image): void
    {
        $treeType = TreeFactory::getTreeType($name, $size, $image);
        $this->trees[] = new Tree($x, $y, $treeType);
    }
}
```

4

يمثل هذا الجزء المكان الذي سيناديه ال Client، وهو يمثل مجموعة ال Flyweight التي تم بنائها مع ال context الخاصة بها...، وأهم فكرة في هذا ال class هو طريقة إضافة العناصر الجديدة، لاحظ أننا قبل إضافة الشجرة قمنا بجلب ال treeType، وبهذا تترابط جميع الأطراف.

5

ال Client Call، محاكاة إنشاء 100 ألف شجرة...

```php
// Client Code
$land = new GameLand();
// Simulate to add 100K of trees
for($i = 0; $i < 100000; $i++){
    $land->addTree(
        $i,
        $i * 2,
        TreeType::TREE_NAME_LIST[rand(0, 2)],
        TreeType::TREE_SIZE_LIST[rand(0, 2)],
        md5(rand(0, 2))
    );
}

// Call Draw Land
print_r($land->draw());
echo "========================" . PHP_EOL;
// Here get Total Count For Unique Object Saved on RAM (To validate our work)
TreeFactory::getAllTreesType();
```

Structural Design Patterns - Flyweight

```
    [99991] => x: 99991, y: 199982, name: Banana, size: 10x15, img: c81e728d9d4c2f636f067f89cc14862c}
    [99992] => x: 99992, y: 199984, name: Orange, size: 10x10, img: c4ca4238a0b923820dcc509a6f75849b}
    [99993] => x: 99993, y: 199986, name: Orange, size: 20x20, img: c81e728d9d4c2f636f067f89cc14862c}
    [99994] => x: 99994, y: 199988, name: Apple, size: 20x20, img: cfcd208495d565ef66e7dff9f98764da}
    [99995] => x: 99995, y: 199990, name: Banana, size: 20x20, img: c4ca4238a0b923820dcc509a6f75849b}
    [99996] => x: 99996, y: 199992, name: Banana, size: 10x10, img: c81e728d9d4c2f636f067f89cc14862c}
    [99997] => x: 99997, y: 199994, name: Banana, size: 10x10, img: c81e728d9d4c2f636f067f89cc14862c}
    [99998] => x: 99998, y: 199996, name: Apple, size: 10x10, img: cfcd208495d565ef66e7dff9f98764da}
    [99999] => x: 99999, y: 199998, name: Orange, size: 10x10, img: cfcd208495d565ef66e7dff9f98764da}
)
================================================
Total Count For Unique Object Saved on RAM: 27
Array
(
    [6e4b50023145bb17eb4038e20ad477fa] => name: Orange, size: 10x15, img: c81e728d9d4c2f636f067f89cc14862c
    [7181ece89323ef29b60e650fababba5b] => name: Banana, size: 10x10, img: cfcd208495d565ef66e7dff9f98764da
    [e04e1d69ce61f5ab9a5c95bb9223b518] => name: Apple, size: 10x15, img: cfcd208495d565ef66e7dff9f98764da
    [538f5af398bb25592eb18c569b83f680] => name: Banana, size: 20x20, img: c81e728d9d4c2f636f067f89cc14862c
    [af0de537fd83e33b15a0dd450dde88c1] => name: Apple, size: 20x20, img: c4ca4238a0b923820dcc509a6f75849b
    [72d008e6747c0e5ff8d053ea144aad83] => name: Orange, size: 20x20, img: cfcd208495d565ef66e7dff9f98764da
    [f4dab2aa7bf1dc50a2b6b7284c8f171f] => name: Orange, size: 10x10, img: cfcd208495d565ef66e7dff9f98764da
    [f85b26599e56542a534e5bd59533bc88] => name: Orange, size: 10x15, img: cfcd208495d565ef66e7dff9f98764da
    [7a7eb412504e4c79fd7d894d453fab22] => name: Banana, size: 10x15, img: c81e728d9d4c2f636f067f89cc14862c
    [170f41dd2cc000f7a912b327115f6228] => name: Orange, size: 10x15, img: c4ca4238a0b923820dcc509a6f75849b
    [1c61a2d48e43cf2d4f5017785e63a80d] => name: Banana, size: 10x15, img: cfcd208495d565ef66e7dff9f98764da
    [270694dc15069c5254eef53031f4226d] => name: Apple, size: 10x10, img: cfcd208495d565ef66e7dff9f98764da
    [02d05f34882e0f615a7517b9d3fd606c] => name: Orange, size: 20x20, img: c4ca4238a0b923820dcc509a6f75849b
    [e41fc6e23688061af0233a8f4b06c4e3] => name: Apple, size: 20x20, img: cfcd208495d565ef66e7dff9f98764da
    [88aae94acae237e6f2cd13f2da7a8938] => name: Orange, size: 10x10, img: c4ca4238a0b923820dcc509a6f75849b
    [a3e6606e9f315443ae235cab0ae537d2] => name: Banana, size: 10x10, img: c81e728d9d4c2f636f067f89cc14862c
    [afa455d74f2b14fbc8d8a23fa4d39a1c] => name: Apple, size: 10x10, img: c81e728d9d4c2f636f067f89cc14862c
    [7d145f09f7a0fc0055bf2a41820948d6] => name: Orange, size: 20x20, img: c81e728d9d4c2f636f067f89cc14862c
    [31fef90595a08e3b2a9c3a0e7aa8f2e2] => name: Banana, size: 10x10, img: c4ca4238a0b923820dcc509a6f75849b
    [c23b45824e2a72e44bd7abcca351a16d] => name: Apple, size: 10x15, img: c81e728d9d4c2f636f067f89cc14862c
    [89b27bcfed5abaad79d6029182ba138d] => name: Banana, size: 10x15, img: c4ca4238a0b923820dcc509a6f75849b
    [dde56c47c752f96c6004ebe5f11a952f] => name: Orange, size: 10x10, img: c81e728d9d4c2f636f067f89cc14862c
    [98bfc65d114a4562149cec6fd007a92f] => name: Banana, size: 20x20, img: cfcd208495d565ef66e7dff9f98764da
    [f73e63a0a309b2b37fa2809f5861c98b] => name: Apple, size: 10x15, img: c4ca4238a0b923820dcc509a6f75849b
    [43844e7add85c3b4fbed8c66bd948b68] => name: Banana, size: 20x20, img: c4ca4238a0b923820dcc509a6f75849b
    [f916e581c920d349092787c85c929d4b] => name: Apple, size: 20x20, img: c81e728d9d4c2f636f067f89cc14862c
    [a52d1165207caeb0ee6f21b2a7627822] => name: Apple, size: 10x10, img: c4ca4238a0b923820dcc509a6f75849b
)
```

النتائج: لاحظ من 100 ألف، فقط Flyweight 27 **6**

مثال:

ملاحظة قبل تنفيذ المثال، يمكنك تصغير الرقم من 100 ألف إذا كان الجهاز لديك لن يتحمل ذلك الرقم على المتصفح.

والآن، إليك الرابط الخاص بالمثال ^^

Structural Design Patterns - Flyweight

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. ال Flyweight class: وهو ال Class الذي يحتوي بداخله مجموعة ال State التي يمكن مشاركتها مع أكثر من Object، وال State الذي يحتويها هي Unique، في مثالنا: TreeType.

2. Flyweight factory: وهذا يمثل ال Class الذي سيقوم بعملية إنشاء flyweight object جديد أو إعادة استخدام flyweight object تم إنشائه من قبل، في مثالنا: TreeFactory.

3. Context: وهو يمثل ال contextual Object الذي يحتوي بداخله ال State التي سيتم إرسالها إلى ال flyweight method، وبمجرد الحصول على flyweight object يمكن عرض جميع البيانات الخاصة بهذا ال object، ويمكن طباعة أعداد كبيرة جدا هنا بدون خوف لأن ال unique state ستكون مشاركة مع كل ال object، في مثالنا: Tree.

4. Flyweight Clients; وهذا يمثل ال Class الذي سيتم استدعاؤه لبناء ال Objects بما يتناسب مع البيانات التي سيتم إرسالها، ليتم عرض البيانات وتمثيلها بشكل صحيح، في مثالنا: GameLand.

Structural Design Patterns - Flyweight

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عندما تجد أن لديك عدد كبير جدا من ال Objects والتي ستستهلك الرام ولا تترك شيئا في الرام إلا استحوذت عليه، لذلك، إذا وجدت أن لديك عدد كبير من ال Objects وعدد كبير منهم متشابه فعليك باستخدام هذا ال Pattern لأن ذلك سيستزف الرام لا محالة، وأشهر التطبيقات على ذلك الألعاب.

من التطبيقات العملية التي واجهتني سابقا في إحدى البرامج الطبية إضافة بعض الأدوات داخل غرفة افتراضية بعد تحليل صور ال CT, CTA, MRI وتحويلها إلى 3D، إضافة هذه الأدوات كانت حركة خطيرة، فكل أداة عبارة عن Object يمكن أن يتكرر أو يوجد الكثير منه، ولكبر حجم البيانات فإن أي إضافة غير ضرورية أو أي اضافة تستحوذ على الرام ستكون كفيلة بعمل Crash.

Structural Design Patterns - Flyweight

المميزات:

- حفظ ال RAM من خلال منع تكرار ال Object نفسه.

العيوب:

- يزيد من مستوى التعقيد للشيفرة البرمجية، ودون توثيق جيد قد يصعب فهم لماذا تمت مشاركة ال State بهذه الطريقة.

Structural Design Patterns - Flyweight

علاقة ال Flyweight مع غيره من ال Pattern:

- ال Facade يقوم على فكرة جعل Object واحد يمثل أكبر قدر من البيانات ويختزل أكبر قدر من التعقيد، بينما ال Flyweight يقوم على فكرة بناء الكثير من عدد قليل من ال Objects.

- يمكنك استخدام ال Flyweights مع ال composite للتخفيف من استهلاك ال RAM من خلال عمل implementation لل shared leaf من خلال هذا ال Flyweights.

- ال Facade يقوم على فكرة جعل Object واحد يمثل أكبر قدر من البيانات ويختزل أكبر قدر من التعقيد، بينما ال Flyweight يقوم على فكرة بناء الكثير من عدد قليل من ال Objects.(كما في مثالنا، 100 ألف لكن ال flyweight 27).

- ال Flyweight شبيه بال Singleton إذا قمت بإدارة جميع ال state من خلال Object واحد، لكن هناك اختلافين جوهريين، الأول أن ال Singleton لا يحتوي إلا instance واحدة، بينما ال Flyweight يمكن أن يحتوي أكثر من instance، والثاني أن ال Singleton يمكن تعديل ال object الخاص به، بينما ال flyweight غير قابل للتعديل.

عن المقدام بن معدي كرب -رضي الله عنه- عن النبي -صلى الله عليه وسلم- قال:
"ما ملأ آدميٌّ وعاءً شرًّا مِن بطنٍ , بحسبِ ابنِ آدمَ أُكُلاتٌ يُقمنَ صُلبَهُ , فإن كانَ لا محالةَ فثُلثٌ لطعامِهِ وثُلثٌ لشرابِهِ وثُلثٌ لنفَسِهِ"

Structural Design Patterns - Proxy

ال Pattern السابع والأخير ضمن هذه المجموعة هو ال Proxy، هذا ال Pattern يقوم على استبدال Object مكان Object آخر، بحيث يمثل ال Object الجديد Proxy يتحكم في الوصول إلى ال Object الأصلي، وذلك غالبا ما يستخدم للقيام بإجراء معين قبل أو بعد معالجة أي request قادم إلى ال Original Object، بمعنى آخر، لقد قمت بتفويض Object للقيام بمجموعة من الإجراءات قبل وصول أي أمر للتنفيذ لل Object الأصلي أو بعده، فمثلا لو رغبت بالتحقق من صلاحيات هذا ال Object للوصول إلى Application معين ومعالجة البيانات أو القيام بإجراء، فذلك يمكن أن يتم من خلال ال Proxy، فال Object الأصلي مهمته هي القيام مثلا ب myOriginalObject->play، بينما ال Proxy سيقوم بعمل isAllowedToAccess ثم تحويل العمل ال original object إذا كانت لديه صلاحية، وفي حال لم يكن له صلاحية يمكن عمل أي إجراء مثل كتابة log.

Structural Design Patterns - Proxy

المشكلة: تخيل أن لديك بطاقة ائتمانية، هذه البطاقة مربوطة في حسابك البنكي، فكيف يمكن أن تستخدم هذا النقد الموجود في حسابك وتحويله لكل عميل قمت بالشراء منه؟ أو ليكن لديك قاعدة بيانات يطلب ال client منها بعض البيانات والتي تستهلك resource، لكن لا يوجد سبب للإبقاء على قاعدة البيانات هذه تعمل طوال الوقت، بل فقط عند الطلب، وكيف يمكن حفظها مثلا من خلال استخدام ال cache؟

الحل: حقيقة هناك العديد من الحلول لحل المشاكل السابقة، لكن بعض الحلول قد لا تعمل في كل الأوقات، فمثلا وجود third party قد يمنعك من تنفيذ lazyload، لكن من ضمن الحلول الكثيرة، يأتي ال Proxy بحل ينقذنا من هذه المشكلة بطريقة جميلة، هذا الحل ببساطة يكون من خلال إنشاء Proxy Class يقوم بتطبيق نفس ال interface الخاصة بال Object الأصلي، وبهذا يمكننا من تمرير ال Proxy Object إلى جميع ال Objects الأصلية دون خوف من أي مشكلة، وبهذا قمنا بعمل delegate للوظيفة المطلوبة، وهنا النقطة الرئيسية، أي أن الفائدة التي يقدمها هذا ال Pattern عن بقية الحلول الممكنة هي إمكانية تنفيذ مجموعة من الأوامر قبل الإنتهاء من ال Logic الأساسي لل Class الأصلي أو بعده، وهذه نقطة مهمة، لأنك لن تحتاج إلى التعديل على ال class الأصلي، لأنك بتنفيذك لل interface في كلا ال object بإمكانك تمرير ال Proxy دون أن يحدث مشاكل عند ال client، فمثلا البطاقة الإئتمانية تمثل تطبيقا واقعيا على فكرة ال Proxy، فهي تمثل النقد الموجود بالحساب، لذلك كلا ال objects يطبق ال pay interface مثلا، وال object الأصلي هو النقد، بينما ال Proxy هو البطاقة الإئتمانية.

Structural Design Patterns - Proxy

مثال: لنأخذ مثالا بسيطا يشرح الفكرة،
ولنفترض أنني أرغب بإنشاء Proxy ل
Class موجود لدي لإضافة Role، بحيث لا
يسمح لمن هو read فقط من الكتابة فوق
الملف...

2

```php
/**
 * Class TxtFile - Original Server which its a concrete implementation
 * 2nees.com
 */
class TxtFile implements TextFilesControl {
    private $file;
    private string $fileName;
    public function __construct(string $fileName)
    {
        $this->fileName = $fileName;
    }
    public function read(): void
    {
        echo file_get_contents("{$this->fileName}.txt") ?: "Empty" . PHP_EOL;
    }
    public function write(string $txt): void
    {
        fwrite($this->file, $txt);
    }
    public function open(): void
    {
        $this->file = fopen("{$this->fileName}.txt", "w+");
    }
    public function close(): void
    {
        fclose($this->file);
    }
    public function getFile()
    {
        return $this->file;
    }
}
```

1

```php
/**
 * Interface TextFilesControl - The interface for original service and proxy
 * 2nees.com
 */
interface TextFilesControl {
    public function read(): void;
    public function write(string $txt): void;
    public function open(): void;
    public function close(): void;
    public function getFile();
}
```

Structural Design Patterns - Proxy

مثال:

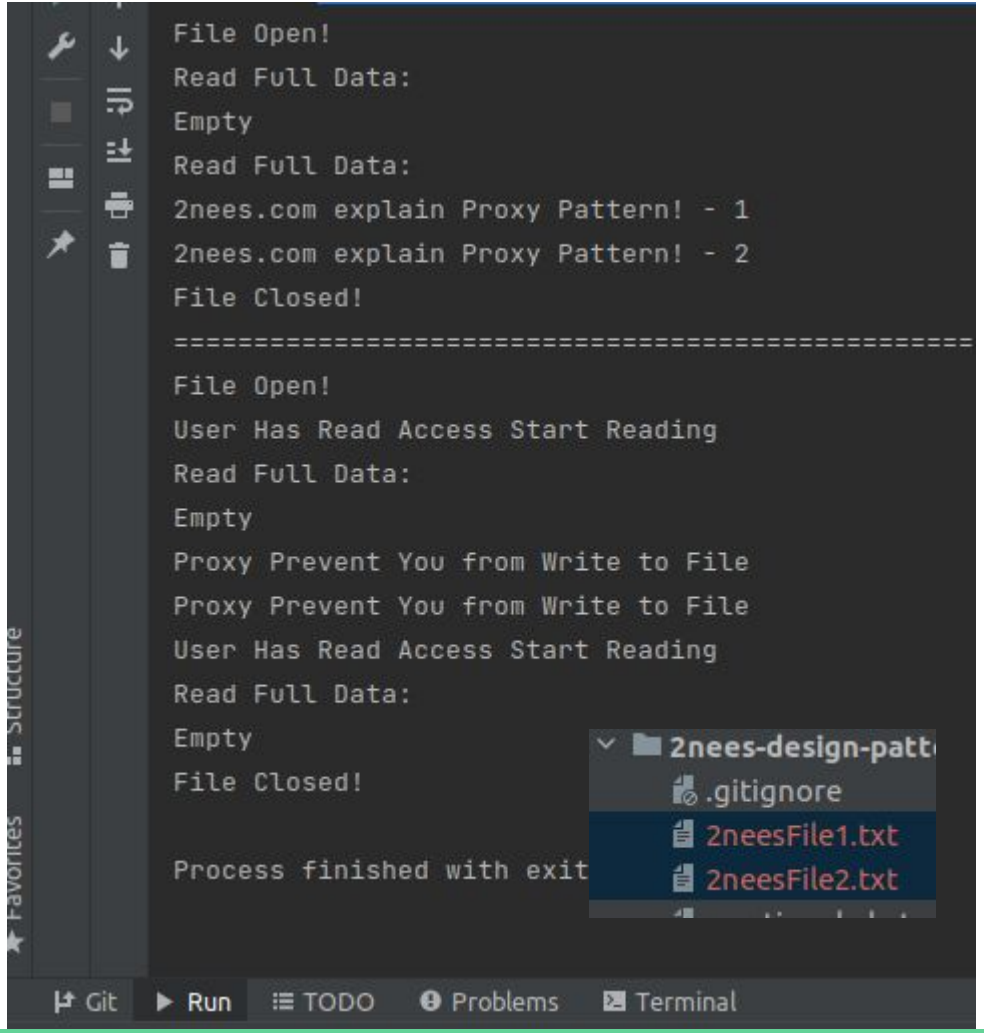
```php
/**
 * Class TxtFileProxy - This is the Proxy class which implements the same interface for Original Class
 * 2nees.com
 */
class TxtFileProxy implements TextFilesControl {
    private TextFilesControl $textFilesControl;
    private int $role;
    public function __construct(TextFilesControl $textFilesControl, int $role)
    {
        $this->textFilesControl = $textFilesControl;
        $this->role = $role;
    }
    public function read(): void
    {
        if($this->role !== 1){
            echo "User Has Read Access Start Reading" . PHP_EOL;
        }

        echo "Read Full Data: " . PHP_EOL;
        $this->textFilesControl->read();
    }
    public function write(string $txt): void
    {
        if($this->role === 1){
            $this->textFilesControl->write($txt);
        }else {
            echo "Proxy Prevent You from Write to File" . PHP_EOL;
        }
    }
    public function open(): void
    {
        if(!$this->getFile()){
            $this->textFilesControl->open();
            echo "File Open!" . PHP_EOL;
        }else {
            echo "File Already Open!" . PHP_EOL;
        }
    }
    public function close(): void
    {
        $this->textFilesControl->close();
        echo "File Closed!" . PHP_EOL;
    }
    public function getFile()
    {
        return $this->textFilesControl->getFile();
    }
}
```

3

هذا ال Class يمثل ال Proxy، والفكرة الأساسية كما تلاحظ أننا قمنا بربط ال service مع ال interface، ويستقبل هذا ال Proxy ال Original object....، لاحظ أن الفكرة هنا بإضافة شرط تمثل صلاحية الوصول للكتابة، وأخرى جملة طباعة لتمثيل ال LOG...

```php
$file1 = new TxtFile("2neesFile1");
$fileProxy1 = new TxtFileProxy($file1, 1);
$fileProxy1->open();
$fileProxy1->read();
$fileProxy1->write("2nees.com explain Proxy Pattern! - 1\n");
$fileProxy1->write("2nees.com explain Proxy Pattern! - 2\n");
$fileProxy1->read();
$fileProxy1->close();
echo "================================================" . PHP_EOL;
$file2 = new TxtFile("2neesFile2");
$fileProxy2 = new TxtFileProxy($file2, 2);
$fileProxy2->open();
$fileProxy2->read();
$fileProxy2->write("2nees.com explain Proxy Pattern! - 1\n");
$fileProxy2->write("2nees.com explain Proxy Pattern! - 2\n");
$fileProxy2->read();
$fileProxy2->close();
```

4



Structural Design Patterns - Proxy

مثال:

في هذا المثال قد تعرفنا على تطبيق متخيل لهذا ال Pattern، وقمت أيضا بإضافة مثال آخر، وذلك فقط لإخبارك بإمكانية استخدام بعض الوسائل للتعامل مع ال proxy بالطريقة التي تراها مناسبة، وطبعا ما هو موجود في المثال الثاني هو فقط للمحاكاة وإظهار بعض الحركات التي يمكنك القيام بها...، بعد إطلاعك عليها، دعنا نشاهد المثال في الصفحة التالية...

والآن، إليك الرابط الخاص بالمثال ^^

رابط المثال الثاني ^^

Structural Design Patterns - Proxy

مثال 2: في هذا المثال نريد عمل محاكاة لتطبيق فكرة من الحياة العملية، وهي مثلا جلب معلومات عن فيديو معين من ال Youtube وتحميله، في الحالة الطبيعية قد يكون لديك ال class وال interface والأمور لطيفة، باستثناء أنك تحتاج في كل عملية download إلى الاتصل مجددا بسيرفرات ال Youtube وجلب الفيديو، وهذا أمر مكلف بالإضافة لكونه يأخذ وقتا أكثر، فلو قمنا بوضع Proxy لعمل Cache لل videos ومعلوماتها والتي تم تحميلها، سيكون هذا توفيرا ممتازا لنا...لنشاهد المثال:

```php
/**
 * Interface YouTubeLib - The interface for original service and proxy
 * 2nees.com
 */
interface YouTubeLib {
    public function getVideoInfo(): string;
    public function downloadVideo(): string;
    public function setSelectedVideoId(string $vid): void;
    public function getSelectedVideoId(): string;
}
```

مثال2:

Structural Design Patterns - Proxy

```php
/**
 * Class YouTubeClass - Original Server which its a concrete implementation
 * 2nees.com
 */
class YouTubeClass implements YouTubeLib {
    private string $selectedVideoId;

    public function getVideoInfo(): string
    {
        $seen = rand(500, 2000);
        return "Video {$this->selectedVideoId} seen: {$seen}" . PHP_EOL;
    }

    public function downloadVideo(): string
    {
        $size = rand(1, 500);
        return "Video {$this->selectedVideoId} size: {$size}MB" . PHP_EOL;
    }

    public function setSelectedVideoId(string $vid): void{
        $this->selectedVideoId = $vid;
    }

    public function getSelectedVideoId(): string{
        return $this->selectedVideoId;
    }
}
```

```php
/**
 * Class YouTubeProxy - This is the Proxy class which implements the same interface for Original Class
 * 2nees.com
 */
class YouTubeProxy implements YouTubeLib {
    private YouTubeLib $youTubeLib;
    private array $cachedInfos = [];
    private array $cachedDownloadedFiles = [];
    public function __construct()
    {
        $this->youTubeLib = new YouTubeClass();
    }
    public function getVideoInfo(): string
    {
        $key = $this->getSelectedVideoId();
        if(!array_key_exists($key, $this->cachedInfos)) {
            $this->cachedInfos[$key] = $this->youTubeLib->getVideoInfo();
        }else {
            echo "Get Info from Cache!" . PHP_EOL;
        }
        return $this->cachedInfos[$key];
    }
    public function downloadVideo(): string
    {
        $key = $this->getSelectedVideoId();
        if(!array_key_exists($key, $this->cachedDownloadedFiles)) {
            $this->cachedDownloadedFiles[$key] = $this->youTubeLib->downloadVideo();
        }else {
            echo "Downloaded Video from Cache!" . PHP_EOL;
        }
        return $this->cachedDownloadedFiles[$key];
    }
    public function setSelectedVideoId(string $vid): void
    {
        $this->youTubeLib->setSelectedVideoId($vid);
    }
    public function getSelectedVideoId(): string
    {
        return $this->youTubeLib->getSelectedVideoId();
    }
}
```

Structural Design Patterns - Proxy

مثال 2:

```php
// 2nees.com
$youtube = new YouTubeProxy();
$youtube->setSelectedVideoId("2nees-A");
echo $youtube->getVideoInfo();
echo $youtube->downloadVideo();
echo "=========================1=======================" . PHP_EOL;
$youtube->setSelectedVideoId("2nees-B");
echo $youtube->getVideoInfo();
echo $youtube->downloadVideo();
echo "=========================2=======================" . PHP_EOL;
$youtube->setSelectedVideoId("2nees-A");
echo $youtube->getVideoInfo();
echo $youtube->downloadVideo();
echo "=========================3=======================" . PHP_EOL;
$youtube->setSelectedVideoId("2nees-C");
echo $youtube->getVideoInfo();
echo $youtube->downloadVideo();
echo "=========================4=======================" . PHP_EOL;
$youtube->setSelectedVideoId("2nees-B");
echo $youtube->getVideoInfo();
echo $youtube->downloadVideo();
```

```
Video 2nees-A seen: 1472
Video 2nees-A size: 121MB
=========================1=======================
Video 2nees-B seen: 963
Video 2nees-B size: 309MB
=========================2=======================
Get Info from Cache!
Video 2nees-A seen: 1472
Downloaded Video from Cache!
Video 2nees-A size: 121MB
=========================3=======================
Video 2nees-C seen: 1151
Video 2nees-C size: 484MB
=========================4=======================
Get Info from Cache!
Video 2nees-B seen: 963
Downloaded Video from Cache!
Video 2nees-B size: 309MB
```

والآن، إليك الرابط الخاص بالمثال الأخير ^^

Structural Design Patterns - Proxy

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Interface
2. Original Service: وهو ال Class الذي يحتوي بداخله مجموعة الدوال المفيدة لذات ال class، وتقوم بتنفيذ الوظائف المطلوبة منها على أتم وجه.
3. Proxy Class: وهو ال Class الذي يحتوي بداخله ال reference field، وهو المسؤول عن ربط ال Proxy مع ال Original service، وفيه تتم العمليات التي نرغب بالقيام فيها ودون التعديل على ال Original Class، مثل ال Cache.
4. Client: يجب أن يعمل بدون أي مشاكل باستخدام ال Proxy او ال Original class، كما يمكنه استخدام ال Proxy object في أي مكان بدلا من ال original class عند الحاجة لذلك.

Structural Design Patterns - Proxy

إذا، متى يمكنني استخدام هذا ال Pattern؟
بكل بساطة يمكنك استخدام هذا ال Pattern في الكثير من الحالات، وقد قمنا في المثالين السابقين بمحاكاة نموذجين مختلفين، الأول لل protection proxy (ل Access Control) -قمنا بتنفيذ ال role كمحاكاة بسيطة-، إن أشهر caching proxy والآخر لل استخدامات ال Proxy تندرج تحت 6 أبواب، وهن ال virtual proxy وال protection proxy وال remote proxy وال logging proxy وال caching proxy وال smart reference. (البعض يعتبر هذه التقسيمات 4 من دون ال caching وال logging).

- Virtual proxy: تمثل ال Lazy initialization، أي بدلا من أن ننشئ ال Object مباشرة مع تشغيل البرنامج، لا نقوم بإنشائه إلا عند الحاجة إليه.
- Protection proxy: يستخدم عند الحاجة لوضع شروط لمنع الوصول إلى ال Original Object إلا بتحققها، ويمكن تطبيق أي ACL لذلك، مثلا object لخاصية في لعبة، لا يجب أن يصل لها إلا admin.
- Remote proxy: هذا يستخدم عند وجود ال original object على remote server أو على جهاز آخر.
- Logging proxy: عند الحاجة لعمل log لكل شيء إكمال العمل.
- Caching proxy: عند الحاجة لعمل caching للنتائج والتحكم بال life cycle لهذا ال cache.
- Smart reference: يستخدم لإغلاق جمع ال Objects إذا لم يعد هناك users يستخدمونه.

Structural Design Patterns - Proxy

المميزات:

- يحقق مبدأ ال Open/Closed Principle
- ال Proxy يمكنه التحكم بال Original object وإضافة العمليات دون أن يهتم أو يعلم ال Client بذلك.

العيوب:

- يزيد من مستوى التعقيد للشيفرة البرمجية بسبب كمية ال classes الجديدة التي من الممكن إنشائها والمرتبطة بعدد ال Proxies.

Structural Design Patterns - Proxy

علاقة ال Proxy مع غيره من ال Pattern:

- ال Adapter يمكن لأكثر من interface أن تعمل wrap لل object، بينما في ال Decorator يحسن على ال interface الموجودة، وال Proxy يقدم نفس ال interface.

- ال Proxy مشابه لل Facade من ناحية تخزين أو إعداد ال Object، لكن ال Proxy لديه نفس ال interface الخاصة بال Object مما يجعل هذا ال Object قابل للتبادل مع ال interface الخاصة بال Proxy وال Service Object.

- ال Decorator وال Proxy كلاهما متشابهان أيضا ويستخدمون نفس المبدأ، لكن الفرق الأساسي بينهم أن ال Proxy يتحكم بال life cycle لل object من خلاله، بينما تتم هذه العملية في ال decorator من خلال ال client.

وَمَنْ تَأَمَّلَ أَحْوَالَ الرُّسُلِ مَعَ أُمَمِهِمْ: وَجَدَهُمْ كَانُوا قَائِمِينَ بِالْإِنْكَارِ عَلَيْهِمْ أَشَدَّ الْقِيَامِ. حَتَّى لَقُوا اللَّهَ تَعَالَى, وَأَوْصَوْا مَنْ آمَنَ بِهِمْ بِالْإِنْكَارِ عَلَى مَنْ خَالَفَهُمْ وَأَخْبَرَ النَّبِيُّ - صَلَّى اللهُ عَلَيْهِ وَسَلَّمَ -: أَنَّ الْمُتَخَلِّصَ ه
وَبَالَغَ فِي الْأَمْرِ بِالْمَعْرُوفِ وَالنَّهْيِ عَنِ الْمُنْكَرِ أَشَدَّ الْمُبَالَغَةِ, حَتَّى قَالَ «إِنَّ النَّاسَ إِذَا تَرَكُوهُ: أَوْشَكَ أَنْ يَعُمَّهُمُ ا

ج3 - ص128 - كتاب مدارج السالكين بين منازل إياك نعبد وإياك نستعين

لذلك يا أخي, لا تداهن ولا تجامل في الباطل, بل أنكر الباطل ما استطعت, فإن لم تستطع, فأقل ذلك إنكار القلب!

Behavioral Design Patterns

ثالث مجموعة لدينا من ال Design Patterns والأخيرة هي مجموعة ال Behavioral، هذه المجموعة تهتم بسلوك ال Objects فيما بينها وبنقل المسؤوليات التي تقع على عاتق كل واحدة منها، وبطريقة أخرى فإن هذه المجموعة تهتم بخوارزمية او طريقة تطبيق الكود الذي لدينا...

Behavioral Design Patterns - Chain of Responsibility

ال Pattern الأول في هذه المجموعة هو ال Chain of Responsibility، هذا ال Pattern يقوم على فكرة إنشاء سلسلة (Chain) من المسؤوليات التي تنتقل من Object إلى آخر حتى تنتهي السلسلة أو تتوفق لسبب ما، كلما نجح Object أو إنتهى من العمل على المسؤوليات الخاصة به، أرسل أمرا للانتقال للحلقة التي بعدها من السلسلة، ويمكن في أي لحظة عند حدوث أي خطأ قطع السلسلة ومنعها من متابعة العمل.

Behavioral Design Patterns - Chain of Responsibility

المشكلة: تخيل أن لديك موقع بيع إلكتروني، عند قيام المستخدم بطلب Order فإنك مضطر إلى القيام بمجموعة من الإجراءات، مثلا التحقق من أنك Logged-in، والتحقق من عمرك فوق ال 18، والتحقق من النقود التي لديك في المحفظة، والتحقق من وجود شركات شحن يمكنها أن تصل إلى المنطقة التي تتواجد فيها...إلى آخره، كل هذه العمليات كما تلاحظ سيتم التحقق منها بشكل متتالي، سنتأكد من أن المستخدم logged-in، ثم سنتأكد من عمره، ثم سنتأكد من رصيده، ثم سنتأكد من إمكانية الشحن إليه..إلخ، ماذا لو فرضنا أنك قمت بكتابة الشيفرة البرمجية في مكان واحد، ثم ظهرت مجموعة شروط جديدة بناءا على البيزنس؟1، إضافتها ستكون خطيرة بل وكارثية!، ماذا لو أردنا إضافة log و cache بعد مراحل معينة في هذه السلسلة؟!، سيصبح الموضوع صعبا ومزعجا بشكل كبير، وسكون هناك انتهاك صارخ للعديد من مبادئ التصميم...اذا ما الحل؟!

Behavioral Design Patterns - Chain of Responsibility

الحل: يقترح هذا ال Pattern حلا جميلا لهذه المشكلة، وهي من استغلال أن هذه الشروط أو/و العمليات يتم تنفيذها بشكل متتابع، وذلك من خلال بناء سلسلة من ال Objects، كل سلسلة تقوم بتنفيذ المسؤوليات المناطة بها، ومن ثم الإنتقال للعنصر التالي في السلسلة، أو إيقافها عند حدوث حطأ ما، عملية نقل السلوك هذه لكل Object على حدى هو ما يقوم عليه ال Pattern، ويسمى ال Object الذي تلقى هذا السلوك ب Handler، هذا ال handler سيحتوي بداخله reference لل next handler، وتنتهي عندما يكون ال next handler غير موجود!، بناءا على هذا الطرح، نكتشف أننا بحاجة إلى Common Interface لنتمكن من بناء هذه السلسلة.

Behavioral Design Patterns - Chain of Responsibility

مثال: لنقم بتجربة محاكاة لبناء سلسلة من المسؤوليات، ويمكن أن نأخذ مثال ال Order من الواقع العملي...

```php
/**
 * Interface OrderHandler - This is Handler
 * 2nees.com
 */
interface OrderHandler {
    public function setNextHandler(OrderHandler $orderHandler): OrderHandler;
    public function handle(Order $order);
}
```

تمثل هذه ال interface ال handler لكل ال handlers class.

1

```php
andler - Optional, but its useful to share
 */
abstract class OrderBaseHandler implements OrderHandler {
    private ?OrderHandler $nextHandler = null;

    public function setNextHandler(OrderHandler $orderHandler): OrderHandler
    {
        $this->nextHandler = $orderHandler;

        return $this->nextHandler;
    }

    public function handle(Order $order)
    {
        if($this->nextHandler){
            return $this->nextHandler->handle($order);
        }

        return null;
    }
}
```

2

يمثل هذا ال abstract ال base handler، وهو خيار اختياري، نستخدمه لوضع السلوك المعتاد الخاص بال handlers class.

Behavioral Design Patterns - Chain of Responsibility

```php
/**
* Concrete Handlers Class contain actual code for processes request
* 2nees.com
*/
class CheckIsLoggedInUser extends OrderBaseHandler {
    public function handle(Order $order)
    {
        if($order->getUser()->getIsLoggedIn()){
            parent::handle($order);
        }else {
            throw new Exception("User not Logged In");
        }
    }
}
class CheckUserRole extends OrderBaseHandler {
    public function handle(Order $order)
    {
        if($order->getUser()->getRole() === 1){
            parent::handle($order);
        }else {
            throw new Exception("{$order->getUser()->getName()} Cant Access This Order!");
        }
    }
}
class CheckProductQuantity extends OrderBaseHandler {
    public function handle(Order $order)
    {
        $notAvailable = false;
        foreach ($order->getProducts() as $product){
            if($product->getQuantity() === 0){
                $notAvailable = $product;
            }
        }

        if($notAvailable === false){
            parent::handle($order);
        }else {
            throw new Exception("{$notAvailable->getName()} not available since its contain
({$notAvailable->getQuantity()})Item");
        }
    }
}

class AddLog extends OrderBaseHandler {
    public function handle(Order $order)
    {
        echo "Add Order Log for Order ID: #{$order->getOrderId()}" . PHP_EOL;
        parent::handle($order);
    }
}

class OrderSummery extends OrderBaseHandler {
    public function handle(Order $order)
    {
        echo "Order Owner: #{$order->getUser()->getId()} - {$order->getUser()->getName()}" .
PHP_EOL;
        echo "Order Id: #{$order->getOrderId()}".PHP_EOL;
        parent::handle($order);
    }
}
```

3

تمثل هذه ال classes جميعها ال Concrete handlers؛ فهي الكود الفعلي لمعالجة ما وصل، فمثلا بعض منها class للتحقق، وبعض منها لإضافة لوج، وبعض منها لطباعة summary...فالسلسلة ليست بالضرورة للتحقق فقط، بل يمكنك إضافة أعمال ترغب بإنجازها في أي مرحلة من المراحل...

مثال:

4

هذا ال class وما يليه 1-4 , 2-4 عبارة عن classes موجودة في المشروع، لإمكانية بناء المثال (Service Classes).

```php
class Order {
    private Users $user;
    private array $products;
    private int $orderId;

    public function __construct(Users $user, array
$products, int $orderId)
    {
        $this->user = $user;
        $this->products = $products;
        $this->orderId = $orderId;
    }
    public function getUser(): Users
    {
        return $this->user;
    }
    public function getProducts(): array
    {
        return $this->products;
    }
    public function getOrderId(): int
    {
        return $this->orderId;
    }
}
```

Behavioral Design Patterns - Chain of Responsibility

مثال:

4-1

```php
class Users {
    private bool $isLoggedIn;
    private string $name;
    private int $id;
    private int $role;
    public function __construct(string $name, int $id, int $role, bool $isLoggedIn)
    {
        $this->name = $name;
        $this->id = $id;
        $this->role = $role;
        $this->isLoggedIn = $isLoggedIn;
    }
    public function getName(): string
    {
        return $this->name;
    }
    public function getId(): int
    {
        return $this->id;
    }
    public function getRole(): int
    {
        return $this->role;
    }
    public function getIsLoggedIn(): bool
    {
        return $this->isLoggedIn;
    }
}
```

4-2

```php
class Products {
    private string $name;
    private int $id;
    private int $quantity;
    public function __construct(string $name, int $id, int $quantity)
    {
        $this->name = $name;
        $this->id = $id;
        $this->quantity = $quantity;
    }
    public function getName(): string
    {
        return $this->name;
    }
    public function getId(): int
    {
        return $this->id;
    }
    public function getQuantity(): int
    {
        return $this->quantity;
    }
}
```

Behavioral Design Patterns - Chain of Responsibility

مثال:

```php
// Init Data For Simulate User Request order - 2nees.com
$user1 = new Users("Anees", 1, 1, true);
$user2 = new Users("Taher", 2, 1, true);
$user3 = new Users("Saed", 3, 2, true);
$user4 = new Users("Ibraheem", 4, 1, false);

$prod1 = new Products("Car", 1, 40);
$prod2 = new Products("Computer", 2, 10);
$prod3 = new Products("Gold Ring", 3, 0);

$order1 = new Order($user1, [$prod1, $prod2], 1);
$order2 = new Order($user2, [$prod3], 2);
$order3 = new Order($user3, [$prod1], 3);
$order4 = new Order($user4, [$prod1, $prod2], 4);
```

5

```php
$checkIsLoggedInUser    = new CheckIsLoggedInUser();
$checkUserRole          = new CheckUserRole();
$checkProductQuantity   = new CheckProductQuantity();
$AddLog                 = new AddLog();
$orderSummery           = new OrderSummery();

// Set order of handler execute
$checkIsLoggedInUser
    ->setNextHandler($checkUserRole)
    ->setNextHandler($checkProductQuantity)
    ->setNextHandler($AddLog)
    ->setNextHandler($orderSummery);

try {
    $checkIsLoggedInUser->handle($order1);
} catch (Exception $e) {
    echo $e->getMessage() . PHP_EOL;
}
echo "================================================" . PHP_EOL;
try {
    $checkIsLoggedInUser->handle($order2);
} catch (Exception $e) {
    echo $e->getMessage() . PHP_EOL;
}
echo "================================================" . PHP_EOL;
try {
    $checkIsLoggedInUser->handle($order3);
} catch (Exception $e) {
    echo $e->getMessage() . PHP_EOL;
}
echo "================================================" . PHP_EOL;
try {
    $checkIsLoggedInUser->handle($order4);
} catch (Exception $e) {
    echo $e->getMessage() . PHP_EOL;
}
```

هذا الجزء الأخير من الشيفرة البرمجية، لاحظ أننا قمنا بتحديد ال next handler، ثم أرسلنا ال order...

والآن، إليك الرابط الخاص بالمثال ^^

6

```
Add Order Log for Order ID: #1
Order Owner: #1 - Anees
Order Products:
Order Id: #1
================================================
Gold Ring not available since its contain (0)Item
================================================
Saed Cant Access This Order!
================================================
User not Logged In
```

Behavioral Design Patterns - Chain of Responsibility

ملاحظة على المثال السابق: أنا قمت بوضع Exception في حال لم يتحقق الشرط، لكن يمكنك وضع أو إرجاع ما ترغب به، ويمكنك التحكم متى يذهب لل next handler بسهولة من خلال ال base handler...، الفكرة بطريقة التفكير لبناء السلسلة ضمن هذه المكونات...

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Handler: هو ال interface الذي يجب أن يطبق ال implementation الخاص به في جميع ال Concrete Handler Class.
2. Base Handler: وهو abstract class يمكنك استخدامه لحفظ ال referance الخاص بال handler وجعله المكان الموحد أو المشترك الذي تتعامل أو تتواصل معه جميع ال Concrete Handler
3. ال Concrete Handler: وهو يمثل مجموعة ال Classes التي تقوم حقيقة بتنفيذ المهام المطلوبة منها ومن ثم إتمام السلسلة أو قطعها...
4. Client: مكان تعريف أو إنشاء السلسلة، ويتم تحديد ال handlers التي سيتم وضعها في السلسلة في هذا المكان، وطبعا لا يشترط استخدام كل ال handlers أو بترتيب معين، بل ما يحكم ذلك هو البيزنس logic.

Behavioral Design Patterns - Chain of Responsibility

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند وجود العديد من العمليات التي ترغب من معالجتها والتحقق منها أو القيام بمجموعة من ال Process الأخرى بحيث تكون أنواع هذه العمليات مختلفة، كما أن هذه الحالة عادة ما تكون sequences، فإن كانت كذلك، فعليك بهذا ال Pattern، كما أن هذا ال Pattern مفيد عند رغبتك بترتيب العمليات بناءا على ال order الذي ترسله، فكما لاحظت نحن من نتحكم من هو ال next handler!، كما يمكنك من إضافة مجموعة من ال method لحذف أو إضافة handler جديدة أثناء ال runtime، لكن عادة لا نحتاج لذلك (immutable)، ومن الأمثلة العملية لاستخدام هذا ال Pattern بناء ال Order وال HTTP Middleware وأي متسلسلة من المهام...

Behavioral Design Patterns - Chain of Responsibility

المميزات:

- يحقق مبدأ ال Open/Closed Principle
- يحقق مبدأ ال Single Responsibility Principle
- بإمكانك التحكم في بناء السلسلة وترتيبها حسب البزنس Logic.

العيوب:

- تحتاج إلى معالجة جميع ال request، وقد يتم قطع السلسلة فلا تصل بعض ال request إلى النهاية ولم تتم معالجة نتائجها، لذلك يجب على ال client التحقق من ال handling لكل النتائج المتوقعة، وعلى المطور يضمن أن كل حلقة من السلسلة يمكنها بعد معالجة بياناتها الانتقال لما بعدها أو إرجاع ما يشير إلى قطع السلسلة.

Behavioral Design Patterns - Chain of Responsibility

علاقة ال Chain of Responsibility مع غيره من ال Pattern:

- ال Chain of Responsibility وال Command وال Mediator وال Observer كلها Patterns بطريقة إرسال واستقبال ال object request، لكن ال CoR يتعامل هذه الطلبات على شكل متسلسلة، كلما انتهت حلقة انتقلت إلى الحلقة التي تليها، بينما ال command يقوم بفتح unidirectional بين ال object المرسل والمستقبل، وال Mediator يفتح اتصال مباشر بين ال object المرسل والمستقبل لكن لا يتم ذلك إلا من خلال ال mediator object، وال Observer يكون فيه استقبال ال request لل subscribe وال unsubscribe بشكل dynamic.

- غالبا ما يستخدم ال Chain of Responsibility Pattern مع ال Composite، وفي هذه الحالة، فإن ال Chain of Responsibility سيقوم بتمرير ال request الخاص بأي leaf من جميع ال Parents إلى ال root tree.(لأن بإمكانك أخذ سلسلة من إحدى السلاسل من الشجرة، مثلا في شجرة العائلة يمكنك أخذ أنيس و حكمت و أنيس الجد من عائلة أبو حميد...)

Behavioral Design Patterns - Chain of Responsibility

- يمكن عمل implement لل handlers class الموجودة بال CoR من خلال ال Command، في هذه الحالة يمكنك القيام بالعديد من المهام من خلال نفس ال Object context، كما يمكن عمل العكس، أن نربط مجموعة من ال Commands object context بسلسلة مستخدمين هذا ال Pattern.

- ال Chain of Responsibility وال Decorator قريبين من بعضهم البعض بشكل كبير، لكن هناك فرق خطير، فالأول يمكنه إيقاف ال process عند أي نقطة، بينما ال decorator لا يمكنه القيام بذلك.

فلتعلم يا أخي أن الفرق بين الجُود والسَّرَف هو في كون أنَّ الجواد حكيم يضع العطاء مواضعه, والمسرف مبذر, قد يصادف عطاؤه موضعه, وكثيرا لا يصادفه.

كتاب الروح - ص 661 - ابن القيم -رحمه الله-

Behavioral Design Patterns - Command

ال Pattern الثاني في هذه المجموعة هو ال Command، هذا ال Pattern يقوم على فكرة تحويل ال Request إلى Object مستقل بذاته فيه جميع المعلومات الخاصة بهذا ال request، عملية التحويل هذه تتيح لك إرسال مجموعة مختلفة من ال requests من خلال ال Param، كما تتيح لك عمل delay لل request أو وضعه داخل queue!، كما يمكنك دعم ال operation الغير قابلة للتنفيذ(مثل ال Undo).

Behavioral Design Patterns - Command

المشكلة: تخيل أن لديك Text Editor، هذا ال Editor فيه العديد من ال Buttons، وكل Buttons ستقوم بتنفيذ عملية معينة، كما أن كل Button من هذه ال Buttons يمكن أن تنفذ من خلال اختصار على ال menu، كما يمكن تنفيذ ما في داخلها من خلال اختصارات ال keyboard، مثل هذه العمليات قد تجعل الشيفرة البرمجية قبيحة جدا!، والسبب في ذلك أن طريقة العمل التقليدية ستكون من خلال عمل sub class تتوارث مثلا ال button أو تتوارث ال actions، وهذا سيحدث لدينا كم كبير من ال sub classes، وكم كبير من التكرار للشيفرة البرمجية، تخيل أن هذا الأمر ستفعله مع كل عملية موجودة عندك في ال Text editor، ما هي النتائج التي تتوقعها؟!، تخيل أنك ترغب في إضافة history لهذه العمليات مثل العودة لنص سابق وهكذا؟!، ستجد أن الأمر معقد جدا، وستجد امتدادا في ال sub classes بشكل غير مسبوق، تخيل في هذه اللحظات المريرة، حدثت إضافة على أحد ال parent classes، فما هو حجم التعديل الذي تتخيله؟! كارثة!، لذلك، جاء هذا ال Pattern لحل هذه المشكلة...

Behavioral Design Patterns - Command

الحل: هذا ال Pattern اقترح حلا جميلا لهذه المشكلة، الحل في تخيل أنك تتعامل مع أكثر من Layer، ال Layer العليا هي ما تشاهده، وال Layer السفلى هي التي تحتوي على Logic، في الحالة الطبيعية فإن ال Layer العليا سترسل ال request مباشرة إلى ال Layer السفلى، وهنا جاء دور هذا ال Pattern، فقال، بدلا من إرسال ال Request من ال Layer العليا إلى السفلى مباشرة، نقوم بأخذ كل المعلومات التي تلزمنا من هذا ال Request ونضعها في method مستقلة، وهنا سيتشكل لدينا Command Object، هذا ال Object هو من سيتولى إرسال ال Request، وال Layer العليا لن تهتم بطبيعة ال Object الموجودة في ال Logic في الطبقة السفلى، لأن كل ما ستقوم به هو إطلاق Trigger لأحدى ال Command!، لو عكسنا هذا الأمر على المشكلة التي تطرقنا إليها، سنكتشف أننا سنجعل ال Button وال Keyboard وال menu shortcut تقوم بعمل trigger لل Copy مثلا!، وبهذا، فإننا لم نكرر ال Code، ولم ننشئ الكثير من ال sub classes...

Behavioral Design Patterns - Command

مثال: لنأخذ مثالا تصوريا بأن لدينا تلفاز افتراضي نريد برمجة بعض الأوامر له، مثل تشغيل وإغلاق، ورفع الصوت وخفضه، ويمكن القيام بهذه العمليات من خلال الضغط على remote control أو على button موجودة على التلفاز مباشرة...لنشاهد كيف يمكن تطبيق هذا ال Pattern;

```php
/**
 * Interface TVControlCommand - This is the Command Interface which its
contain all common command method
 * 2nees.com
 */
interface TVControlCommand {
    public function execute();
}

/**
 * Class TVBaseCommand - This is optional abstract class, I used it since
this declaration common for all Concrete Command Class
 */
abstract class TVBaseCommand implements TVControlCommand {
    protected TVControl $control;

    public function __construct(TVControl $control)
    {
        $this->control = $control;
    }
}
```

تمثل هذه ال interface ال Command لكل ال Concrete Command class. 1

هذا الجزء اختياري، قمت ببنائه لأن هذا الجزء سيكون مشتركا لكل ال Commands في مثالنا هذا...

Behavioral Design Patterns - Command

مثال:

```php
/**
 * Class TVControl - Receiver Class, this class represent the real place for doing actual work
 * 2nees.com
 */
class TVControl {
    private int $volume;
    private bool $tvTurnStatus;
    public function __construct(int $volume, bool $tvTurnStatus)
    {
        $this->volume = $volume;
        $this->tvTurnStatus = $tvTurnStatus;
    }
    public function turnOff(): bool {
        $this->tvTurnStatus = false;
        echo "TV is turned OFF" . PHP_EOL;
        return $this->tvTurnStatus;
    }
    public function turnOn(): bool {
        $this->tvTurnStatus = true;
        echo "TV is turned ON" . PHP_EOL;
        return $this->tvTurnStatus;
    }
    public function decreaseVolume(): void
    {
        $volume = $this->volume - 1;
        if($volume >= 0){
            $this->volume = $volume;
        }
    }
    public function increaseVolume(): void
    {
        $volume = $this->volume + 1;
        if($volume <= 7){
            $this->volume = $volume;
        }
    }
    public function isTvTurnStatus(): bool
    {
        return $this->tvTurnStatus;
    }
    public function getVolume(): int
    {
        return $this->volume;
    }
}
```

هذا الجزء يمثل ال Receiver، وهو المكان الفعلي الذي تتم فيه العمليات...(لا جديد هنا)

2

Behavioral Design Patterns - Command

مثال:

```php
// 2nees.com - Concrete Command Classes which its used to execute specific work
class TVTurnOnControlCommand extends TVBaseCommand {
    public function execute()
    {
        if($this->control->isTvTurnStatus()){
            return;
        }
        $this->control->turnOn();
    }
}
class TVTurnOffControlCommand extends TVBaseCommand {
    public function execute()
    {
        if(!$this->control->isTvTurnStatus()){
            return;
        }
        $this->control->turnOff();
    }
}
class TVVolumeUpControlCommand extends TVBaseCommand {
    public function execute()
    {
        if(!$this->control->isTvTurnStatus()){
            echo "TV is OFF!, Turn TV ON to change volume UP!" . PHP_EOL;
            return;
        }
        $this->control->increaseVolume();
        echo "Your Volume: {$this->control->getVolume()}" . PHP_EOL;
    }
}
class TVVolumeDownControlCommand extends TVBaseCommand {
    public function execute()
    {
        if(!$this->control->isTvTurnStatus()){
            echo "TV is OFF!, Turn TV ON to change volume Down!" . PHP_EOL;
            return;
        }
        $this->control->decreaseVolume();
        echo "Your Volume: {$this->control->getVolume()}" . PHP_EOL;
    }
}
```

هذه ال Classes تمثل ال Concrete Command، وهي ال Classes المسؤولة عن تنفيذ وظيفة معينة بالتحديد وكل واحد منها لديه الوظيفة الخاصة به...

3

Behavioral Design Patterns - Command

```php
/**
 * Class Application - Sender (Invoker)
 * 2nees.com
 */
class Application {
    private TVTurnOffControlCommand $turnOffControlCommand;
    private TVTurnOnControlCommand $turnOnControlCommand;
    private TVVolumeUpControlCommand $volumeUpControlCommand;
    private TVVolumeDownControlCommand $volumeDownControlCommand;
    public function __construct(
        TVTurnOffControlCommand $turnOffControlCommand,
        TVTurnOnControlCommand $turnOnControlCommand,
        TVVolumeUpControlCommand $volumeUpControlCommand,
        TVVolumeDownControlCommand $volumeDownControlCommand
    ) {
        $this->turnOffControlCommand = $turnOffControlCommand;
        $this->turnOnControlCommand = $turnOnControlCommand;
        $this->volumeUpControlCommand = $volumeUpControlCommand;
        $this->volumeDownControlCommand = $volumeDownControlCommand;
    }
    public function on() {
        $this->turnOnControlCommand->execute();
    }
    public function off() {
        $this->turnOffControlCommand->execute();
    }
    public function up() {
        $this->volumeUpControlCommand->execute();
    }
    public function down() {
        $this->volumeDownControlCommand->execute();
    }
    public function doSomethingElseViaTheseCommand() {
        $this->on();
        $this->up();
        $this->up();
        $this->up();
    }
}
```

هذا ال Class يمثل ال Invoker لدينا، هذا ال Class يقوم بعمل init لل request، ويجب أن يحتوي بداخله reference لل commands كما في الصورة...، وكما تلاحظ فإن ما يقوم به هو عمل trigger لل command.

4

مثال:

ملاحظة: قد تتسائل هنا، لما قمنا بإنشاء ال Invoker، مع أننا فعليا يمكننا مباشرة عند ال client عمل ال execute، فالجواب ينقسم لعدة أجزاء، أولا في هذا الشكل من التصميم أو الأمثلة، فعلا لن تحتاج له، لكن، من المعتمد وجوده وأن تتم العملية من خلاله كما هو الشكل الخاص بال Pattern، ثانيا: غالبا ما سيكون لديك history من الأوامر أو أمور ترغب بمشاركتها والقيام بها، كما أن هذا ال Invoker قد يحتوي أكثر من نوع من ال Command، كل هذه الأسباب ستجعل منه مهما لتتم العملية من خلاله، ولا تنسى أن مجموعة ال Actions من الممكن بناؤها أو إضافتها من خلال هذا ال Invoker، كما يمكن مشاركة هذا ال Object في أكثر من مكان لاستخدام ما في داخله...

Behavioral Design Patterns - Command

```php
// Client - 2nees.com
$tv         = new TVControl(2, true);
$app        = new Application(
                    new TVTurnOffControlCommand($tv),
                    new TVTurnOnControlCommand($tv),
                    new TVVolumeUpControlCommand($tv),
                    new TVVolumeDownControlCommand($tv)
                );

/**
 * This loop to simulate we are press on remote or TV internal button, we share
invoker as you see and can we use and put it any where...
 */
for ($i = 0; $i < 10; $i++){
    switch (rand(1, 8)):
        case 1:
        case 2:
            $app->on();
            break;
        case 3:
        case 4:
            $app->off();
            break;
        case 5:
        case 6:
            $app->up();
            break;
        case 7:
        case 8:
            $app->down();
            break;
    endswitch;
}
echo "================================================" . PHP_EOL;
$app->doSomethingElseViaTheseCommand();
```

5

تمثل هذه الشيفرة البرمجية ال Client call، وطبعا هي عملية محاكاة تصورية لل Action...

مثال:

```
Your Volume: 1
Your Volume: 0
Your Volume: 1
Your Volume: 2
TV is turned OFF
TV is OFF!, Turn TV ON to change volume UP!
TV is turned ON
================================================
Your Volume: 3
Your Volume: 4
Your Volume: 5
```

6

والآن، إليك الرابط الخاص بالمثال ^^

Behavioral Design Patterns - Command

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Command Interface: وفيه ال Method الخاصة بال Execute، وعادة ما تكون هذه ال Method لوحدها في هذه ال interface، ويمكن إضافة Methods أخرى عند الحاجة.

2. Base Command: وهو Abstract اختياري يمكن استخدامه إذا رغبنا بمشاركة شيء مشترك بين كل ال Concrete Commands

3. Concrete Commands: وهي ال Classes التي تقوم بتنفيذ عدة أنواع من ال requests، ولا تقوم هذه ال Command بتنفيذ المهام بنفسها، بل يتم ذلك من خلال ال Receiver.

4. Receiver: ويمثل ال Class الذي يحتوي على ال Business Logic، وهو المكان الحقيقي للقيام بالأعمال، وفعليا وظيفة ال Command هي التحكم بال Request التي ستصل إلى هذا ال Class

5. Invoker (Sender): يمثل هذا ال Class الذي سيتم عمل init لل request من خلاله، ويحتوي عادة بداخله field تربط ال Command Object، يقوم هذا ال Class بالتحكم وعمل trigger لل command المطلوب تنفيذها، وعادة ما يتم عمل ال init لل Command object من خلال ال constructor.

6. Client: وهو المكان الذي يتم فيه إعداد ال Concrete Command وال Receiver وال Sender...

Behavioral Design Patterns - Command

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern في الكثير من الحالات، مثل رغبتك بعمل queue للمهمات التي ترغب بإنجازها أو عند رغبتك بتقديم او تأخير التنفيذ لبعض المهام لقدرتك على التحكم في ذلك، كما قد يكون هذا ال Pattern الأسلوب الشائع في تطبيق ال undo/redo، كما أن هذا ال Pattern مهم جدا عند حاجتنا لإرسال ال Command إلى Method معينة على شكل Argument، كما يمكن تخزينه داخل Object آخر أو وتبديل ال command أثناء ال run time، وهذا كله مفيد في الحالات الي تتطلب أكثر من Action للقيام بنفس العمل كمثال ال Editor التي تحتوي button, keyboard, shortcut...

Behavioral Design Patterns - Command

المميزات:

- يحقق مبدأ ال Open/Closed Principle
- يحقق مبدأ ال Single Responsibility Principle
- يمكن القيام بال undo/redo من خلاله
- يمكن تأجيل بعض الأعمال للقيام بها لاحقا

العيوب:

- قد يكون هذا ال Pattern معقدا إلى حد ما، وذلك لأنك قمت بتعريف Layer جديدة بين ال Sender وال Recevier

Behavioral Design Patterns - Command

علاقة ال Command مع غيره من ال Pattern:

- ال Chain of Responsibility وال Command وال Mediator وال Observer كلها Patterns بطريقة إرسال واستقبال ال object request، لكن ال CoR يتعامل هذه الطلبات على شكل متسلسلة، كلما انتهت حلقة انتقلت إلى الحلقة التي تليها، بينما ال command يقوم بفتح unidirectional بين ال object المرسل والمستقبل، وال Mediator يفتح اتصال مباشر بين ال object المرسل والمستقبل لكن لا يتم ذلك إلا من خلال ال mediator object، وال Observer يكون فيه استقبال ال request لل subscribe وال unsubscribe بشكل dynamic.

- يمكن عمل implement لل handlers class الموجودة بال CoR من خلال ال Command، في هذه الحالة يمكنك القيام بالعديد من المهام من خلال نفس ال Object context، كما يمكن عمل العكس، أن نربط مجموعة من ال Commands object context بسلسلة مستخدمين هذا ال Pattern.

Behavioral Design Patterns - Command

- يمكنك استخدام ال Memento مع ال Command عند العمل على ال undo، لأن ال command يمكنه القيام بالعمليات التي نحتاجها، وسيوفر لنا ال Memento حفظ ال State الخاصة بال object قبل تنفيذ أي Command، وهذه نقطة مهمة تحل أحد المشاكل المتعلقة بال private state في حالة ال undo.
- ال Command وال Strategy قد يبدوان متشابهان جدا، لكن هناك اختلافات مهمة بينهما، فال Command يقوم على تحويل جميع ال Operation إلى Command، وهذا يسمح لنا بعمل queue او delay...إلى آخره، بينما ال Strategy قائم على فكرة القيام بالعمل في أكثر من طريقة والتبديل بين هذه الطرق ضمن context واحد.
- يمكنك استخدام ال Prototype لأخذ نسخة من ال Object وحفظها بال history.(يمكن عمل serialize لل object مثلا وحفظه داخل sqlite ومن ثم جلبه وإعادة استخدامه)
- يمكنك التعامل مع ال Visitor وكأنه Command في نسخة مطورة ذو صلاحيات أعلى، فيمكنك من خلاله تنفيذ العمليات بين مجموعة مختلفة من ال Objects من classes مختلفة.

رَبِّ أَوْزِعْنِي أَنْ أَشْكُرَ نِعْمَتَكَ الَّتِي أَنْعَمْتَ عَلَيَّ وَعَلَىٰ وَالِدَيَّ وَأَنْ أَعْمَلَ صَالِحًا تَرْضَاهُ وَأَدْخِلْنِي بِرَحْمَتِكَ فِي عِبَادِكَ الصَّالِحِينَ

Behavioral Design Patterns - Iterator

ال Pattern الثالث في هذه المجموعة هو ال Iterator، هذا ال Pattern يقوم على فكرة بسيطة ومهمة، وهي كيف يمكنني الوصول إلى العناصر التي أرغب في الحصول عليها ضمن Collection معينة ودون الاهتمام بالتمثيل الأساسي في ذاته، مثل ال list وال tree...إلخ

بالتأكيد أثناء عملك على مشاريع متنوعة، واجهتك مثلا tree ترغب بالحصول على إحدى المعلومات التي حوتها، وكنت تقوم بعمل جُمل الدوران مثل foreach أو while حتى تصل إلى البيانات التي تريدها في كل مرة تحتاج إلى ذلك، كما أنك ستجد من الصعوبة أن تقوم بإدارة أكثر من نوع من البيانات أثناء الدوران، لكن، هذا ال Pattern سيقوم ببناء Class يقوم بهذه الوظيفة، وبكل بساطة سيكون التكرار والوصول إلى البيانات المطلوبة من خلال عمل تطبيق لهذا ال Pattern، ويمكن بناء أكثر من نوع من ال Iterator حسب ما تقتضيه المصلحة، وعادة ما يكون هناك Iterator واحد...

Behavioral Design Patterns - Iterator

المشكلة: تخيل أنك تحتاج للحصول على عنصر معين من tree، وكانت طريقة الحصول على هذا العنصر في أول الأمر من خلال المرور على كل ال breadth-first ثم احتجت أن تحصل على العنصر أيضا من خلال المرور من خلال ال depth first، ماذا لو احتجت لجلب عنصر بطريقة عشوائية، ماذا لو أردت حذف أو القيام بعمليات معينة افتراضية أثناء الدوران؟ ماذا لو أردت السماح بأكثر من شكل أو نوع للبيانات...إلخ، ما ستفعل؟

الحل: يقترح هذا ال Pattern حلا جميلا وبسيطا، فبدلا من أن يعتمد ال Object على نفسه أو شكل ال Collection التي هو بها بشكل مباشر، يتم نقل هذه المهمة لل Iterator وعمل implementation من خلاله، أي عملية Encapsulate لطريقة الحصول على البيانات المطلوبة...، وكل ما تحتاجه فعليا للمشكلة السابقة، هو إنشاء Iterator لكل حالة منها بكل بساطة، وستنتهي المشكلة ^_*...

Behavioral Design Patterns - Iterator

```php
// 2nees.com
class Countries implements \Iterator {
    private int $pointer = 0;
    private array $countriesList;
    public function __construct(array $countriesList)
    {
        $this->countriesList = $countriesList;
    }
    public function current()
    {
        echo "CURRENT VALUE" . PHP_EOL;
        return  $this->countriesList[$this->pointer];
    }
    public function next()
    {
        echo "NEXT VALUE" . PHP_EOL;
        $this->pointer += 1;
    }
    public function key()
    {
        echo "CURRENT KEY" . PHP_EOL;
        return $this->pointer;
    }
    public function valid()
    {
        echo "VALIDATE VALUE" . PHP_EOL;
        return isset($this->countriesList[$this->pointer]);
    }
    public function rewind()
    {
        echo "RESET VALUE" . PHP_EOL;
        $this->pointer = 0;
    }
}
```

1 هذا Class قام بتنفيذ Iterator.

مثال: سنأخذ مثالا بسيطا وذلك من خلال ال PHP، وسنقوم ببناء Iterator لينفذ جملة دوران بسيطة...

لاحظ في المثال أننا قمنا بالتحكم في بال life cycle، البداية والتحقق والتنقل والنهاية وعمل reset...

ملاحظة: في ال PHP، ال Iterator interface موجود بشكل افتراضي، وفيه 5 دوال كما تشاهد في الصورة، وطبعا يمكنك بناء ال Iterator واستخدامه حتى لو لم تكن هذه الinterface موجودة عندك...

Behavioral Design Patterns - Iterator

لاحظ في النتائج أن أول تنفيذ كان ال rest، ثم التحقق، ثم جلب القيمة الحالية ثم الإنتقال للقيمة التالية والتحقق...

```php
/**
* Client Code
* 2nees.com
*/
$countriesList = new Countries([
    "Jordan", "Egypt", "Palestine", "Syria"
]);

foreach ($countriesList as $country){
    echo "^_^ => " . $country . PHP_EOL;
}
```

2

هذا يمثل ال client code، لاحظ أن ال foreach الموجودة والمستخدمة في هذا المثال، ستقوم بتنفيذ ال iterator الذي قمنا ببنائه بشكل افتراضي في الصورة الأولى...

```
RESET VALUE
VALIDATE VALUE
CURRENT VALUE
^_^ => Jordan
NEXT VALUE
VALIDATE VALUE
CURRENT VALUE
^_^ => Egypt
NEXT VALUE
VALIDATE VALUE
CURRENT VALUE
^_^ => Palestine
NEXT VALUE
VALIDATE VALUE
CURRENT VALUE
^_^ => Syria
NEXT VALUE
VALIDATE VALUE
```

3

والآن، إليك الرابط الخاص بالمثال ^^

مثال 2: الرابط

Behavioral Design Patterns - Iterator

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Iterator Interface: وهو يمثل ال interface التي ستستخدم في كل ال Concrete Iterator
2. ال Concrete Iterator هو ال Class الذي يحتوي بداخله الخوارزمية أو الطريقة التي صممت بها طريقة جلب المعلومات المطلوبة، مثل الدوران التسلسلي أو من خلال البحث العشوائي أو من خلال الأحرف أو البحث داخل tree...إلخ، كل واحد يمثل وسيلة واحدة مستقلة.
3. Collection Interface، وهي interface مصممصة لترجع واحد أو أكثر من نوع من ال Iterator، والتي تتناسب مع ال collection...(مجموعة ال Method التي قد ترجع أكثر من نوع من ال Concrete Iterator التي تم إنشاؤها)
4. Concrete Collections: يقوم بإرجاع instance معينة ل iterator معين بناء على طلب ال Client.

مثال 3: الرابط

Behavioral Design Patterns - Iterator

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند وجود شكل من البيانات معقد التركيب، مثل ال tree، فتحتاج إلى الوصول إلى قيمة معينة، وبنفس الوقت نحتاج إلى إخفاء التعقيد عن ال Client لعدة أسباب كجعل الأمر أسهل!، كما أن هذا ال Pattern مفيد جدا في حالات وجود جملة تكرار سينم استخدامها أكثر من مرة وفي أكثر من مكان، وهذا مرتبط بالجزئية الأولى، فجمل الدوران الاعتيادية والتي تسير على ال normal list لا تحتاج إلى كثير من العناء، بينما من تحتاج إلى الدخول مثلا داخل Tree فالأمر مختلف...، وهذا ال Pattern سيجعل من طريقة الوصول إلى العناصر المطلوبة أسهل، وأجمل، وأفضل للشيفرة البرمجية الخاصة بالتطبيق، خصوصا أنه يمكنك بناء collection وربط أكثر من iterator بها، لدعم أكثر من نوع من البيانات.

Behavioral Design Patterns - Iterator

المميزات:

- يحقق مبدأ ال Open/Closed Principle
- يحقق مبدأ ال Single Responsibility Principle
- كل Iterate لها State خاصة بها، فيمكن بناء أكثر من Iterate بشكل متوازي
- يمكن القيام بالعديد من العمليات أثناء ال Iterate مثلا delay عند شرط معين، والمتابعة فيما بعد.

العيوب:

- استخدام هذا ال Pattern مع collection بسيطة مثل ال normal list قد يكون مبالغة لا داعي لها من المطورين، خصوصا إذا كان التطبيق كله لا بحتاج أنواع معقدة من collection!
- قد يكون هذا ال Pattern أقل كفائة من الأسلوب الاعتيادي، خصوصا إذا كانت طريقة تنفيذ الشيفرة البرمجية ليست بتلك الكفاءة.

Behavioral Design Patterns - Iterator

علاقة ال Iterator مع غيره من ال Pattern:

- يمكنك استخدام ال Iterators للوصول ال Composite trees المطلوبة.
- يمكن استخدام ال Factory Method مع ال Iterator جنبا إلى جنب وذلك لجعل مجموعة من ال subclasses تقوم بإرجاع أنواع مختلفة من ال iterators والتي تتوافق مع مجموعة ال subclasses.
- يمكنك استخدام ال Memento مع ال Iterator لأخذ ال Current state أثناء الدوران، هذا سيمكننا من العودة للخلف عند الحاجة.
- يمكن استخدام ال Visitor مع ال Iterator للحصول على البيانات المطلوبة من ال Complex Data Structure، كما يمكن تنفيذ العديد من المهمات أثناء الحصول على هذه البيانات.

لماذا لم يُهزم نبيّ قط؟

قُتل من الأنبياء من قُتل، ويأتي بعض الأنبياء يوم القيامة بلا أتباع؛ لأنّ أممهم قد كذّبتهم ونفرت منهم.. ومع ذلك لم يُهزم نبيّ قط.. والسبب أنّ المعركة الكبرى للأنبياء ليست جمع الأتباع بكلّ سبيل، وإنّما هي إعلان البراءة من "أوثان" العصر، سواء كانت هذه "الأوثان" من حجر، أو لحم ودم، أو فكرة جاهليّة يركع الناس في محرابها.. وقد جهر الأنبياء كلّهم بالكفر "بالأوثان"، وأعلنوا براءتهم منها، ونادوا بإفراد الربّ بالطاعة..

والهزيمة كلّ الهزيمة، أن يتصالح المرء مع أوثان عصره، أو أن يلتقي عند نقطة "وسط" معها.. أو أن يوهم الناس أنّ "الوثن" قد يكون حليفًا يوم ما لأجل مصلحة أعلى..!

ولا نصر لمؤمن اليوم، إلا أن يكفر "بوثن" العصر، وعنوان جاهليته: العالمانية التي صرفت الناس عن إفراد الله بالطاعة؛ إلى اتخاذ الأهواء معبودًا يُطاع، ويدًا تشكّل حياة الناس على الصورة التي تريد...

النصر، سبيله التواصي بالبراءة من "الأوثان"، والصبر على ذلك.. تلكّ سنة الأنبياء..

كتاب العالمانية طاعون العصر - الدكتور سامي عامري

Behavioral Design Patterns - Mediator

ال Pattern الرابع في هذه المجموعة هو ال Mediator، هذا ال Pattern يمثل وسيط بين مجموعة من ال Objects، فكرة هذا الوسيط هي منع الاتصال المباشر بين ال objects وجعل ذلك يتم فقط من خلال ال Mediator، وهذا الأمر مهم جدا عندما يكون عدد العلاقات الخاصة بال object كبيرة أو مترابطة مع بعضها البعض بشكل فوضوي يصعب تتبعه لكثرته، كما أنه مهم في حال وجود هذا الشكل من ال objects مع إمكانية الزيادة الطردية في العلاقات...، لذلك، هذا ال Pattern سيقوم ببناء وسيط تتعامل معه كل ال objects، فيتم التواصل من خلاله، دون اتصال مباشر فيما بينها...

Behavioral Design Patterns - Mediator

المشكلة: تخيل أنك تريد إدارة حركة الطائرات بأنواعها المختلفة أثناء الهبوط والإقلاع، وهذا يشمل إدارة المدرج وتنظيم الأوقات بين هذه الطائرات لمنع التصادم، في الطريقة التقليدية، ستقول أن كل طيار عليه التواصل الطائرات الأخرى مباشرة لضمان عدم حدوث أي مشكلة! ، أو ستقول أن على كل طيار الإهتمام بما يراه حوله وتجنب الاصطدام من خلال المراقبة والاتصال بالطائرات القريبة، هذه العملية في هذا الشكل سترفع من معدل الحوادث بين الطائرات بشكل كبير!، لذلك لزم وجود حل لهذه المشكلة!

الحل: الحل بكل بساطة هو بناء وسيط بين الطائرات، بحيث لا تتواصل الطائرات مع بعضها البعض، بل تتواصل مع الوسيط حتى ينظم عمليات الهبوط والإقلاع، ودون الحاجة لأن تهتم كل طائرة بكمية وعدد الطائرات الموجودة الأخرى!، والسبب في ذلك يعود لأن الوسيط هو من سيقوم بإدارة هذه العملية من خلال السماح بالهبوط أو الإقلاع، وتنظيم الوقت اللازم ذلك، وبناء التسلسل أو الجدول لهذه الطائرات، ويمكن القول بأننا نحتاج إلى Class يمثل الوسيط، هذا ال class يتم التواصل معه من كل ال Object، كل هذه ال Object تقوم بعمل implement ل interface فيها مثلا action method، فيصبح لدينا override حسب حاجة كل class و encapsulate لل relation بين ال objects في الوسيط.

Behavioral Design Patterns - Mediator

مثال: لنقم ببناء مثال تخيلي يحاكي الفكرة، وليكن لدينا مجموعة من الطيارين الذين سنعلمهم من خلال البرج عن إمكانية الإقلاع والهبوط...

1

قمنا بتعريف ال Interface، هذه ال interface تحتوي مجموعة ال method هي التي سيتم استدعائها من خلال ال Component.

```php
/**
 * Interface PilotsMediator - Mediator Interface
 * 2nees.com
 */
interface PilotsMediator {
    public function notify(BaseComponent $sender, string $event);
}

/**
 * The Base Component provides the basic functionality of storing a
mediator's
 */
abstract class BaseComponent
{
    protected ?ControlTower $controlTower;

    public function setControlTower(ControlTower $controlTower): void
    {
        $this->controlTower = $controlTower;
    }
}
```

في هذا الجزء قمنا بتعريف Base class لمشاركة ال common functionality بين جميع ال component.

Behavioral Design Patterns - Mediator

```php
// Components Class - 2nees.com
class Helicopter extends BaseComponent {
    public function helicopterUP()
    {
        $this->controlTower->notify($this, UP);
    }
    public function helicopterDOWN()
    {
        $this->controlTower->notify($this, DOWN);
    }
    // Do needed job base on actions or base on needed details for this class...
}
class Airbus extends BaseComponent {
    public function airbusUP()
    {
        $this->controlTower->notify($this, UP);
    }
    public function airbusDOWN()
    {
        $this->controlTower->notify($this, DOWN);
    }
    // Do needed job base on actions or base on needed details for this class...
}
class Emergency extends BaseComponent {
    private bool $isSafe = true;
    public function fireAlarm(){
        $this->isSafe = false;
        $this->controlTower->notify($this, FIRE);
    }
    public function isSafe(): bool
    {
        return $this->isSafe;
    }
    public function setIsSafe(bool $isSafe): void
    {
        $this->isSafe = $isSafe;
    }
}

class SafetyTeam extends BaseComponent {
    public function safeAlarm(){
        $this->controlTower->notify($this, SAFE);
    }
    // Do needed job base on actions or base on needed details for this class...
}
```

هذه ال Classes تمثل مجموعة من العناصر المختلفة في وظائفها والتي ترتبط فيما بينها بطريقة أو بأخرى، وبدلا من كتابة وبناء العلاقات بين هذه ال Classes هنا، سنقوم باستخدام ال Meditor

2

```php
/**
 * Class ControlTower - Concrete Mediators which its contain relations between various components
 * 2nees.com
 */
class ControlTower implements PilotsMediator {
    private Helicopter $helicopter;
    private Airbus $airbus;
    private Emergency $emergency;
    private SafetyTeam $safetyTeam;
    public function __construct(Helicopter $helicopter, Airbus $airbus, Emergency $emergency, SafetyTeam $safetyTeam)
    {
        $this->helicopter = $helicopter;
        $this->helicopter->setControlTower($this);
        $this->airbus = $airbus;
        $this->airbus->setControlTower($this);
        $this->emergency = $emergency;
        $this->emergency->setControlTower($this);
        $this->safetyTeam = $safetyTeam;
        $this->safetyTeam->setControlTower($this);
    }
    public function notify(BaseComponent $sender, string $event)
    {
        if($sender instanceof Airbus){
            if(!$this->emergency->isSafe()){
                echo "Airbus keep fly!, no action since we have emergency Alarm!" . PHP_EOL;
                return;
            }
            if($event === UP){
                echo "Airbus: Wait until road is empty!" . PHP_EOL;
            }else {
                $roadNumber = rand(100, 400);
                echo "Airbus: Go to Road Number #{$roadNumber}" . PHP_EOL;
            }
        }else if($sender instanceof Helicopter){
            if(!$this->emergency->isSafe()){
                echo "Helicopter keep fly!, no action since we have emergency Alarm!" . PHP_EOL;
                return;
            }
            if($event === UP){
                echo "Helicopter: Go Direct" . PHP_EOL;
            }else {
                echo "Helicopter: Helicopter Circle is empty!" . PHP_EOL;
            }
        }else if($sender instanceof Emergency){
            if($event === FIRE){
                echo "We have an Emergency case..." . PHP_EOL;
            }
        }else if($sender instanceof SafetyTeam){
            if($event === SAFE){
                $this->emergency->setIsSafe(true);
                echo "Its Safe now, you can complete your job!!" . PHP_EOL;
            }
        }
    }
}
```

هذا ال Class يمثل ال Concrete Mediator، وهو المسؤول عن احتواء جميع ال relation بين ال component المختلفة...

3

Behavioral Design Patterns - Mediator

Behavioral Design Patterns - Mediator

Client ال 4

```php
/**
* Client Code Sample - 2nees.com
*/
$helicopter = new Helicopter();
$airbus = new Airbus();
$emergency = new Emergency();
$safetyTeam = new SafetyTeam();
$controlTower = new ControlTower($helicopter, $airbus, $emergency, $safetyTeam);

$helicopter->helicopterUP();
echo "==========================================" . PHP_EOL;
$emergency->fireAlarm();
$helicopter->helicopterDOWN();
echo "==========================================" . PHP_EOL;
$safetyTeam->safeAlarm();
$helicopter->helicopterDOWN();
echo "==========================================" . PHP_EOL;
$emergency->fireAlarm();
echo "==========================================" . PHP_EOL;
$airbus->airbusDOWN();
$airbus->airbusUP();
echo "==========================================" . PHP_EOL;
$safetyTeam->safeAlarm();
$airbus->airbusDOWN();
$airbus->airbusUP();
$airbus->airbusDOWN();
```

والآن، إليك الرابط الخاص بالمثال ^^

```
Helicopter: Go Direct
==========================================
We have an Emergency case...
Helicopter keep fly!, no action since we have emergency Alarm!
==========================================
Its Safe now, you can complete your job!!
Helicopter: Helicopter Circle is empty!
==========================================
We have an Emergency case...
==========================================
Airbus keep fly!, no action since we have emergency Alarm!
Airbus keep fly!, no action since we have emergency Alarm!
==========================================
Its Safe now, you can complete your job!!
Airbus: Go to Road Number #281
Airbus: Wait until road is empty!
Airbus: Go to Road Number #113
```

5

Behavioral Design Patterns - Mediator

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Mediator Interface: وهي ال interface التي تحتوي جميع ال Method التي سيتم استخدامها أثناء التواصل مع ال components، والتي عادة ما تكون method واحدة.

2. Components: تمثل ال Component مجموعة ال classes المختلفة والتي ستقوم بتنفيذ مهمة معينة، والتي تترابط أو تتقاطع مع Component أخرى، وكل component من هذه لها reference مع ال Mediators، والجميل في هذه الجزئية، أن هذه ال Component يمكن أخذها واستخدامها في مشروع آخر بسهولة، وكل ما عليك هو فقط ربطها بال Mediator الجديد، وهذا يعني أن ال Component لا تهتم بما يحتويه ال Mediator class.

3. Concrete Mediators: وهو المكان الذي يتم عمل encapsulate لجميع ال relation الخاصة بال components، ويحتوي reference لكل component، في مثالنا: ControlTower

Behavioral Design Patterns - Mediator

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern إذا رأيت أنك قمت بإنشاء الكثير من ال sub classes لوراثة عدد معين أو قليل من الخصائص للقيام بمهام معينة أو محددة، حينها فكر من إمكانية إنشاء الوسيط واستخدام الخصائص المطلوبة من خلال استغلال العلاقات بين ال component وإدارتها من خلال الوسيط، كما أن هذا ال Pattern مفيد إذا لم تكن قادرا على إعادة استخدام class معين بسبب كثرة ال dependency التي يعتمد عليها، فبوجود الوسيط، كل ما يلزمك هو تبديل class الوسيط، فإن كان من الصعب تعديل أي class وفيه عدد كبير من العلاقات المترابطة فيما بينها، فإن إنشاء الوسيط سيجعل من هذا الأمر سهلا وبسيطا، والسبب في ذلك أن جميع العلاقات التي نحتاجها ستكون متاحة من خلاله...

Behavioral Design Patterns - Mediator

المميزات:

- يحقق مبدأ ال Open/Closed Principle
- يحقق مبدأ ال Single Responsibility Principle
- يقلل من الاعتمادية المباشرة بين ال Classes
- يجعل من إعادة استخدام class معين أسهل

العيوب:

- هذا ال Pattern يمكن أن يتسبب بجعل ال Object يعرف كثيرا أو يقوم بعمل الكثير، وهذه تعد واحدة من الأمور التي تندرج تحت باب ال code-smell والتي تنتهك مبادئ التصميم anti-pattern.

Behavioral Design Patterns - Mediator

علاقة ال Mediator مع غيره من ال Pattern:

- ال Chain of Responsibility وال Command وال Mediator وال Observer كلها Patterns بطريقة إرسال واستقبال ال object request، لكن ال CoR يتعامل هذه الطلبات على شكل متسلسلة، كلما انتهت حلقة انتقلت إلى الحلقة التي تليها، بينما ال command يقوم بفتح unidirectional بين ال object المرسل والمستقبل، وال Mediator يفتح اتصال مباشر بين ال object المرسل والمستقبل لكن لا يتم ذلك إلا من خلال ال mediator object، وال Observer يكون فيه استقبال ال request لل subscribe وال unsubscribe بشكل dynamic.

- ال Facade وال Mediator قريبين من بعضهم، فوظيفة كل واحد منهما هي محاولة تنظيم وبناء علاقة بين مجموعة ال Classes، لكن ال Facade يقدم interface جديدة لكنه لا يقدم وظائف جديدة، فال Objects لديك وأنت تقوم باستدعاء ما تطلبه من ال sub system مباشرة، بينما ال Mediator يكون هو حلقة الوصل بين ال Component المختلفة، ولا يمكن لل component من الوصول المباشر ل component أخرى، إلا عن طريق ال Mediator والذي يمثل حلقة الوصل فيما بينهم.

Behavioral Design Patterns - Mediator

علاقة ال Mediator مع غيره من ال Pattern:

- التشابه بين ال Observer وال Mediator كبير جدا، لكن الفرق الجوهري أن ال Mediator يمثل Single Object تتم من خلاله جميع العمليات، بينما ال Observer يقوم على فكرة إنشاء one-way connection بطريقة dynamic، وكطريقة بسيطة للتفريق بين الإثنين، إذا كانت كل العلاقات بين ال Objects لا تتم إلا من خلال ال Single object فهذا Mediator، أما اذا كان هناك طريقة يتواصل بها أحد ال Object مع غيره فهذا Observer.

اللهم إني ظلمت نفسي ظلما كثيرا ولا يغفر الذنوب إلا أنت فاغفر لي مغفرة من عندك وارحمني إنك أنت الغفور الرحيم

Behavioral Design Patterns - Memento

ال Pattern الخامس في هذه المجموعة هو ال Memento، هذا ال Pattern يهتم بطريقة حفظك لل State وإعادة استرجاعها ل Object معين، بحيث تتم هذه العملية دون الاهتمام بتفاصيل العملية وبدون الحاجة لأخذ جميع ال State من جميع ال Object، فمثلا لو كان لدي أكثر من Object تجتمع خصائصهم عند class معين للتنفيذ، فإننا نقوم ببناء ال Memento في هذا ال Class، ونهتم بحفظ ال State الخاص به ولا نهتم بكل ال Object الأخرى، وهذا يقدم حلا مهما للعديد من المشاكل، فمثلا بهذه الطريقة يمكنني حفظ ال State الخاص بال Object واسترجاعها دون الحاجة لتحويل ال Private إلى Public، وبهذا، فإن ال Memento أيضا يجب أن لا يصل إلى المعلومات الخاصة به إلا من خلال ال Class التي حُفظت ال state له، ويسمى ال Originator، بينما يمكن مشاركة بعض ال MetaData مع ال Objects الأخرى عند الحاجة مثل متى تم إنشاء هذه النسخة أو اسمها!

في الويب عموما، قد لا تظهر أهمية هذا ال Pattern أو قد لا تجد تطبيقات كثيرة لاستخدمه في الويب، فمثلا بال PHP أو ال Typescript لديك ال serialization والتي يمكنك استخدامها للقيام بهذه العملية، لكن قد تكمن أهميتها في الجافا أو C++ مثلا، بالإضافة إلى ذلك، فإن هذا ال Pattern يمكن بناؤه من خلال ال Nested class مباشرة في اللغات التي تدعم هذه المزية، بينما في ال PHP تختلف طريقة التفكير للوصول لنفس النتيجة، ويتم ذلك من خلال وجود interface memento و concrete class ينفذ هذه ال interface...

Behavioral Design Patterns - Memento

المشكلة: تخيل أنك ترغب ببناء خاصية undo لأحد ال Classes التي تعمل عليها، فإنك أول ما ستفكر فيه، أن كل Object سيأتي سأقوم بحفظ ال state الخاص به، ومن ثم استرجاع هذه ال state كلها، ومن ثم إعادة إرسالها إلى مكان التنفيذ، أي أنك ستعتمد على ال command معين لتقوم بحفظ ال state، لكن هذا الأسلوب له مشاكل كثيرة، أهمها أن state ليست بالضرورة أن تكون كلها public، ثم إذا قلت أنك ستحولها إلى public فهنا انتهكت ال access الخاص بالمتغيرات فيزداد الخطر، بالإضافة إلى ذلك، فإن أي حاجة للتعديل أو التغيير مستقبلا على أي class ضمن الحلقة سيتسبب بمصيبة، والسبب أنك تحتاج إلى البحث والتعديل في كل جزئية مرتبطة بهذه الحلقة، وبهذا تصل لفكرة مفادها أن ال وجود ال Private سيمنعك من أخذ نسخة متكاملة، وجعل كل شي public ومن ثم التعديل فإن ذلك يعني unsafe...، إذا ما الحل؟

Behavioral Design Patterns - Memento

الحل: يقترح ال Memento بكل بساطة عمل delegate للمكان الذي سيتم فيه إنشاء نسخة ال state المطلوبة، هذا المكان هو ال Owner الحقيقي لهذه ال state، وهو ال originator كما ذكرنا في التعريف، وبهذا، فبدلا من محاولة عمل copy لل state من خارج هذا الكلاس أو عن طريق مجموعة ال Objects المختلفة، فإننا سنقوم بعمل النسخة من داخل ال originator class، وبهذا يمكننا الوصول إلى جميع ال state، وبنفس الوقت لا يمكن لل classes الأخرى من الوصول والتعديل على هذه ال state لأنها لا تملكها...، النسخة التي سيتم نسخها وحفظها ستكون داخل Memento Object، والذي يمكن من خلاله مشاركة بعض المعلومات كما ذكرنا آنفا مثل ال created date...

Behavioral Design Patterns - Memento

مثال: لنتخيل أن لدينا نظام حجز تذاكر، وأنت ترغب إذا وقع أي موظف بأي خطأ غير مقصود على التذاكر أن تقوم بإرجاعها للقيم السابقة...، وسنقوم باستخدام ال Memento لهذا الغرض:

```php
// Fake Status List
const STATUS = [
    "ACTIVE",
    "PENDING",
    "REJECTED",
    "APPROVED",
    "RUNNING",
    "IN_PROGRESS",
    "DELETED",
    "HOLD"
];

/**
 * Memento Interface which its used to retrive meta data only
 * 2nees.com
 */
interface MementoTicket
{
    public function getTicketName(): string;
    public function getCreatedAt(): string;
}
```

1

قمنا بتعريف ال Interface، هذه ال interface تحتوي مجموعة ال method هي التي سيتم استخدامها لاسترجاع ال Meta data، المهم أن تبقى في ذهنك أن هذه ال interface يجب أن لا ترجع أي بيانات تخص ال state التي حفظت بداخلها...

Behavioral Design Patterns - Memento

يمثل هذا ال Class ال Originator، وهو ال Class الحاوي لل State التي نرغب بحفظها واسترجاعها عند الحاجة...

2

```php
class TicketOriginator
{// 2nees.com
    private int $ticketNumber;
    private string $ticketOwner;
    private string $ticketStatus;
    public function __construct(int $ticketNumber, string $ticketOwner)
    {
        $this->ticketNumber = $ticketNumber;
        $this->ticketOwner = $ticketOwner;
        $this->ticketStatus = STATUS[rand(0, 7)];
    }
    public function getTicketStatus(): string
    {
        return $this->ticketStatus;
    }
    public function setTicketStatus(string $ticketStatus): void
    {
        $this->ticketStatus = $ticketStatus;
    }
    public function getTicketNumber(): int
    {
        return $this->ticketNumber;
    }
    public function setTicketNumber(int $ticketNumber): void
    {
        $this->ticketNumber = $ticketNumber;
    }
    public function getTicketOwner(): string
    {
        return $this->ticketOwner;
    }
    public function setTicketOwner(string $ticketOwner): void
    {
        $this->ticketOwner = $ticketOwner;
    }
    public function takeSnapShot(): MementoTicket
    {
        return new TicketConcreteMemento(clone $this);
    }
    public function restoreState(MementoTicket $mementoTicket)
    {
        $state = $mementoTicket->getState();
        $this->setTicketNumber($state->getTicketNumber());
        $this->setTicketOwner($state->getTicketOwner());
        $this->setTicketStatus($state->getTicketStatus());
    }
}
```

```php
class TicketConcreteMemento implements MementoTicket
{// 2nees.com
    private TicketOriginator $state;
    private $date;

    public function __construct(TicketOriginator $state)
    {
        $this->state = $state;
        $this->date = date('Ymd');
    }

    public function getState(): TicketOriginator
    {
        return $this->state;
    }

    /**
     * Some of metadata may used from Originator
     */
    public function getTicketName(): string
    {
        return "memento-state-{$this->date}";
    }

    public function getCreatedAt(): string
    {
        return $this->date;
    }
}
```

3

هذا ال Class يمثل ال Concrete Memento، والذي يقوم بعمل implement لل Memento interface (ال Metadata)، وفيه البنية أو الطريقة التي ستحفظ من خلالها ال state...

Behavioral Design Patterns - Memento

```php
class TicketActions
{// 2nees.com
    private ?array $statesHistory;
    private TicketOriginator $ticketOriginator;

    public function __construct(TicketOriginator $ticketOriginator)
    {
        $this->ticketOriginator = $ticketOriginator;
    }

    public function backup()
    {
        $state = $this->ticketOriginator->takeSnapShot();
        $this->statesHistory[] = $this->ticketOriginator->takeSnapShot();
        $this->printDetails("Backup Saved", $state);
    }

    public function restore()
    {
        if(!count($this->statesHistory)){
            echo "You cant restore empty data" . PHP_EOL;
            return;
        }

        $state = array_pop($this->statesHistory);
        $this->ticketOriginator->restoreState($state);
        $this->printDetails("Restored", $state);
    }

    private function printDetails($action, $state): void
    {
        echo "{$action}:
            #{$state->getCreatedAt()}/{$state->getTicketName()}
            - {$this->ticketOriginator->getTicketNumber()}/{$this->ticketOriginator->getTicketOwner()}
            - With Status {$this->ticketOriginator->getTicketStatus()}" . PHP_EOL;
    }
}
```

4

هذا ال Class يمثل ال Caretaker، وهو المسؤول عن الإهتمام بال history الخاص بال Originator state، وهذا ال class يمكنه الوصول إلى metadata الخاصة بال memento، لكن لا يمكنه الوصول لنفس ال state التي داخل ال memento، وليس له صلاحية للتعديل عليه، بل يتم ذلك من ال Originator، وبهذا فإن هذا ال class هو الذي يأخذ ال action مثل CTRL+Z، لعمل restore.

Behavioral Design Patterns - Memento

```php
// Client - 2nees.com
$ticketOriginator = new TicketOriginator(rand(1, 200), "Anees");
$ticketActions = new TicketActions($ticketOriginator);

$ticketActions->backup();// Save init data
$ticketOriginator->setTicketNumber(rand(200, 400));
$ticketOriginator->setTicketOwner("Hikmat");
$ticketOriginator->setTicketStatus(STATUS[rand(0, 7)]);
$ticketActions->backup();// Save first change
$ticketOriginator->setTicketStatus(STATUS[rand(0, 7)]);
$ticketOriginator->setTicketOwner("2nees.com");
$ticketActions->backup();// Save second change
$ticketOriginator->setTicketNumber(rand(500, 600));// not saved, else if you uncomment bellow...
//$ticketActions->backup(); // Uncomment if you like to save latest random number

echo "Latest Update: " . $ticketOriginator->getTicketNumber() . PHP_EOL;// will print new value, but not saved since latest backup is commented

// OOPS, its mistake, lets restore old data...
$ticketActions->restore();
$ticketActions->restore();
$ticketActions->restore();
$ticketActions->restore();
```

5

هنا ال client code، لاحظ بكل بساطة أننا في أي مرحلة يمكننا حفظ واسترجاع ال state الخاصة بال object، ولاحظ أننا اذا لم نقم بحفظ ال state، فإنها لن تسترجع...

```
Backup Saved:
            #20210810/memento-state-20210810
            - 172/Anees
            - With Status RUNNING
Backup Saved:
            #20210810/memento-state-20210810
            - 271/Hikmat
            - With Status APPROVED
Backup Saved:
            #20210810/memento-state-20210810
            - 271/2nees.com
            - With Status REJECTED
Latest Update: 541
Restored:
            #20210810/memento-state-20210810
            - 271/2nees.com
            - With Status REJECTED
Restored:
            #20210810/memento-state-20210810
            - 271/Hikmat
            - With Status APPROVED
Restored:
            #20210810/memento-state-20210810
            - 172/Anees
            - With Status RUNNING
You cant restore empty data
```

6

والآن، إليك الرابط الخاص بالمثال ^^

Behavioral Design Patterns - Memento

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Memento Interface: تحتوي مجموعة ال Method التي يمكن الوصول إليها لطباعة بعض ال Metadata مثل ال createdAt، والتي ستستخدم لغرض ما من خلال ال Caretaker.
2. Originator: وهو يمثل ال Class المالك لل State، وهو المسؤول عن حفظ واسترجاع هذه ال state من خلال إنشاء الدوال مثل save, restore والتي توجه أو تعتمد على ال Memento.
3. Concrete Memento: هذا ال class يمثل المكان الذي يحتوي على التطبيق الخاص بال metadata لإرجاعها عند الحاجة، كما يحتوي على البنية والطريقة الخاصة بحفظ ال state واستدعائها، ويجب التأكيد على أن هذه ال state الخاص بال memento يجب أن لا يصل إليها أحد إلا ال Originator، بينما يمكن الوصول لل metadata.
4. Caretaker: وهو المكان الذي سيتم حفظ أو استرجاع ال memento object من خلاله عن طريق Action/command معين، مثل CTRL+Z، ولا يستطيع قراءة ما في ال Memento object باستثناء ال metadata.

Behavioral Design Patterns - Memento

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند حاجتك لأخذ نسخة من Object بما فيه من state private or public، وقد يكون أشهر مثالا في أذهاننا هو ال undo، لكن هذه العملية أو هذا ال Pattern نجد له فائدة كبيرة في كثير من الأحيان، مثلا عند حاجتنا لعمل roll back عند وجود error معين.

Behavioral Design Patterns - Memento

المميزات:

- يمكننا أخذ نسخة من ال Object ودون أن ننتهك المفهوم الخاص بال Encapsulation
- بسبب وجود ال caretaker فإننا نتخلص من بعض التعقيد الموجود بال originator، لأن ال caretaker هو من سيهتم ويحتوي الشيفرة البرمجية الخاصة بال action الذي يستوجب أن نقوم بعمل save أو restore.

العيوب:

- استهلاك ال RAM الكبير إذا كان عدد ال state الذي يتم حفظه من المستخدم كبير.
- يجب أن يخضع ال caretaker لل originator حتى يضمن تدمير ال states القديمة بناءا على ال lifecycle الخاص بال originator، كما أن اللغات التي تندرج تحت ال dynamic programing مثل ال PHP قد لا تضمن أن ال Memento لن يتم المساس به في مرحلة ما...

Behavioral Design Patterns - Memento

علاقة ال Memento مع غيره من ال Pattern:

- يمكنك استخدام ال Memento مع ال Command عند العمل على ال undo، لأن ال command يمكنه القيام بالعمليات التي نحتاجها، وسيوفر لنا ال Memento حفظ ال State الخاصة بال object قبل تنفيذ أي Command، وهذه نقطة مهمة تحل أحد المشاكل المتعلقة بال private state في حالة ال undo.
- يمكنك استخدام ال Memento مع ال Iterator لأخذ ال Current state أثناء الدوران، هذا سيمكننا من العودة للخلف عند الحاجة.
- في بعض الحالات يمكن أن يكون ال Prototype بديلا بسيطا عن ال Memento، ويكون هذا الأمر عند الرغبة بحفظ ال state داخل ال history، إذا كان ال Object غير مرتبط ب external resource link، أو من السهل إعادة بناء ال links بين ال recource.

فلْتعلمْ أن طريق العلم ليس بيسير، وأن من يريد أن يتعلم ويرتقي بالعلوم، فعليه تحمل مشاق ذلك الطريق، ولْتعلمْ أن مشقة العلم أهون من مشقة الجهل.

Behavioral Design Patterns - Observer

ال Pattern السادس في هذه المجموعة هو ال Observer، هذا ال Pattern مهم جدا ومفيد جدا ومستخدم بكثرة، ووظيفته باختصار هي إنشاء subscription تعمل بطريقة معينة لإطلاق إشعار يُعلم مجموعة مختلفة من ال Object بحصول event معين، بحيث تكون جميع هذه ال Objects خاضعة للمراقبة ومستعدة لأي event، لذلك سمي ب Observer أي المراقبة، ويسمى أيضا هذا ال Pattern ب Event-Subscriber, Listener، وأظنك استخدمت هذا المصطلح أثناء عملك في مشروع ما أو بحثت عنه لحاجة عندك، فمثلا إذا قام مستخدم بتسجيل الدخول فقد تقوم بإطلاق event تسمعه عدة classes وتنتظر هذا ال event لتنفيذ أمر ما...، هل علمت لماذا قلت لك بأنه شاسع الاستعمال؟، لأن عدد ال الحالات الممكنة لتطبيق هذا ال Pattern كثيرة...

Behavioral Design Patterns - Observer

المشكلة: تخيل أن لديك مطعم، هذا المطعم بمجرد دخول زبون يجب أن يصل إشعار إلى النادل والطباخ والمحاسب، في الحالة التقليدية، ستقوم باستدعاء جميع ال objects ومن ثم القيام بعمل معين حسب احتياج كل object، فمثلا النادل سيجهز الطاولة، والطباخ سيبدأ بتحضير المقبلات، والمحاسب سيقوم بتسجيل فاتورة افتتاحية للطاولة...، هذا الأسلوب سيتسبب في مشاكل كثيرة، وهي أن كل object يحتاج إلى هذا ال events ستكون ملزما بإضافته في نفس المكان، بالإضافة لذلك أنت قد تحتاج إلى أنواع مختلفة من ال events في كل مرحلة...، وماذا عن التعامل مع أكثر من زبون؟...تلاحظ من طريقة التفكير التقليدية أن المشاكل التي يمكن أن تطرأ أو نفكر بها كثيرة...إذا ما الحل؟

Behavioral Design Patterns - Observer

الحل: يقدم ال Observer حلا جميلا لهذه المشكلة، فبدلا من التعامل التقليدي مع ال objects، يمكنك بناء ال Class الذي يحتوي ال State المهمة باعتباره Publisher (Subject)، وهذا يعني أن هذا ال Class لديه شيء مثير للإهتمام ترغب بقية ال Classes بالحصول على هذا الشيء، وتسمى ال objects الأخرى التي تعمل مراقبة لل track ب subscribers، هذه العملية تتم بسهولة بالغة، فكل ما يلزمك Array لل subscriber، وحملة دوران عليهم لتنفيذ ال method المطلوبة، ويتم تنفيذ هذا الأمر من خلال ال publisher عند حدوث سبب ما يدعو لإعلام كل object يراقب ال state التي تغيرت...، مثل دخول الزبائن للمطعم...، وبكل تأكيد، يجب علينا لتحقيق هذه الغاية من بناء interface تحتوي جميع ال method المشتركة بين ال subscriber، والتي تشترك أيضا مع ال publisher في أنه لن يتم عمل communicate بينهم إلا من خلال هذه ال interface...الموضوع سهل جدا، لكن لنرى مثلا تتضح الفكرة من خلاله...

Behavioral Design Patterns - Observer

يسمى هذا ال Class بال Publisher، ويحتوي بداخله ال state المهمة والتي سيشاركها هند أي تحديث مع ال Observers

1

```php
// 2nees.com
class Restaurant implements SplSubject {
    private SplObjectStorage $observers;
    private int $tableNumber = 0;
    private int $numberOfCustomer = 0;
    public function __construct()
    {
        $this->observers = new SplObjectStorage();
    }
    public function attach(SplObserver $observer)
    {
        $this->observers->attach($observer);
    }
    public function detach(SplObserver $observer)
    {
        $this->observers->detach($observer);
    }
    public function notify()
    {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }
    public function customerSetOnTable(){
        $this->numberOfCustomer = rand(1, 5);
        $this->tableNumber += 1;

        $this->notify();
    }
    public function getTableNumber(): int
    {
        return $this->tableNumber;
    }
    public function getNumberOfCustomer(): int
    {
        return $this->numberOfCustomer;
    }
}
```

تشمل هذه ال Classes ال Concrete Subscribers والتي تراقب أي تحديث من ال Publisher

2

```php
class Waiter implements SplObserver {
    public function update(SplSubject $subject)
    {
        echo "Hay!, We have a new customer here,
check Table Number #{$subject->getTableNumber()}" .
PHP_EOL;
    }
}//2nees.com
```

مثال يحاكي فكرة المطعم السابقة...

```php
class Kitchen implements SplObserver {
    public function update(SplSubject $subject)
    {
        echo "Hay!, We have an order for
         Table Number #{$subject->getTableNumber()}
         And We have {$subject-
>getNumberOfCustomer()} Customer" . PHP_EOL;
    }
}//2nees.com
```

```php
class Accountant implements SplObserver {
    public function update(SplSubject $subject)
    {
        $numberOfCustomer = $subject->getNumberOfCustomer();
        $initPrice = $numberOfCustomer * 0.5;
        echo "Hay!, Invoice Start for
         Table Number #{$subject->getTableNumber()}
         And We have {$numberOfCustomer} Customer
         And total init price is {$initPrice} JOD
         " . PHP_EOL;
    }
}// 2nees.com
```

Behavioral Design Patterns - Observer

ملاحظة: تقدم ال PHP ثنائي من ال interface الجاهزة لخدمة ال Observer Pattern، وهي ال SplSubject وال SplObserver، بحيث يستخدم الأول مع ال Publisher والثاني مع ال Subscriber

3

يمثل هذا الجزء ال Client side، لاحظ أننا قمنا باضافة جميع ال class التي نرغب إخطارها عند وجود\ أي تحديث...

```php
// Client - 2nees.com
$restaurant = new Restaurant();
$kitchen = new Kitchen();
$waiter= new Waiter();
$accountant = new Accountant();

$restaurant->attach($waiter);
$restaurant->attach($kitchen);
$restaurant->attach($accountant);

$restaurant->customerSetOnTable();
echo "====================================" . PHP_EOL;
$restaurant->customerSetOnTable();
echo "====================================" . PHP_EOL;
$restaurant->customerSetOnTable();
```

4

```
Hay!, We have a new customer here, check Table Number #1
Hay!, We have an order for
        Table Number #1
        And We have 4 Customer
Hay!, Invoice Start for
        Table Number #1
        And We have 4 Customer
        And total init price is 2 JOD

====================================
Hay!, We have a new customer here, check Table Number #2
Hay!, We have an order for
        Table Number #2
        And We have 1 Customer
Hay!, Invoice Start for
        Table Number #2
        And We have 1 Customer
        And total init price is 0.5 JOD

====================================
Hay!, We have a new customer here, check Table Number #3
Hay!, We have an order for
        Table Number #3
        And We have 1 Customer
Hay!, Invoice Start for
        Table Number #3
        And We have 1 Customer
        And total init price is 0.5 JOD
```

والآن، إليك الرابط الخاص بالمثال ^^

مثال 2: وفيه تمثيل لفكرة ال Events: الرابط الخاص بالمثال ^^

Behavioral Design Patterns - Observer

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Publisher: وهو ال Class الذي يحتوي بداخله البنية التركيبية الخاصة بالتعامل مع Subscribers من إضافة أو حذف، كما أنه يملك ال State المهمة، والتي تجعل من ال Subscribers يهتمون بمراقبة أي تحديث يطرأ على ال State...، في مثالنا: Restaurant

2. Subscriber: وهي تمثل ال interface الخاصة بال Concrete Subscribers، وتحتوي بداخلها الدالة التي ستساهم بعمل ال logic الخاص بأي notification، وعادة ما تكون method واحدة يمكنها استقبال أكثر من param. (شاهد المثال 1 و 2)، في مثالنا: SplObserver

3. Concrete Subscribers: وهي مجموعة ال Classes التي قامت بتنفيذ ال subscriber interface، وهي تعمل على مراقبة أي تغيير يتم الإشعار به من خلال ال Publisher.

4. Client: في هذا الجزء يتم تعريف جميع ال Classes التي نحتاجها، ومن ثم نقوم بتربيطها معا وذلك من خلال عمل attach لل Concrete Subscribes داخل ال Publisher، كما يمكن القيام بأي Logic بعدها...

Behavioral Design Patterns - Observer

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند حاجتك إلى مراقبة أي تغيير يحصل على State ما داخل Object وترغب بناءا على ذلك من القيام بمجموعة من المهام المختلفة بناءا على التغيير الذي حصل، فمثلا عندما جلس الزبون داخل المطعم، قمنا باخطار النادل والمطبخ والكاشير بوجود زبون جديد، وكل منهم قام بعمل معين، وهذا الأمر مفيد جدا في ال Graphical user interface، فمثلا لو قمت بتصميم موقع إلكتروني وقمت بإضافة كبسة معينة ويمكن لل client من تحديد السلوك الخاص بها أو يمكن إضافة أكثر من مهمة مرتبطة بما سيحدث من خلالها، فسيكون هذا ال Pattern المناسب لك بكل تأكيد، كما أن هذا ال Pattern مفيد جدا لو أردت أن تجعل ال Object يراقب التغييرات لفترة مؤقتة، لأنك بكل بساطة يمكنك عمل detach، وبهذا تنتهي المراقبة وبكل سهولة!

Behavioral Design Patterns - Observer

المميزات:

- يحقق مبدأ ال Open/Closed Principle
- يمكنك إنشاء relations بين ال objects أثناء ال runtime

العيوب:

- ال Subscribers يتم إشعارهم بحدوث التغير بترتيب عشوائي...، أي ليس هناك ضمان من تنفيذ ال events بتسلسل كما هو مراد، لكن هذا الأمر يمكن إدارته من خلال الشيفرة البرمجية الخاصة بك...، وبعض الأفكار قد تقوم بإضافة متغير Priority لمحاولة السيطرة على هذه المشكلة في حال ظهورها، وعادة ما تحصل إن كان هناك Observer يعتمد على ال Observer آخر...

Behavioral Design Patterns - Observer

علاقة ال Observer مع غيره من ال Pattern:

- ال Chain of Responsibility وال Command وال Mediator وال Observer كلها Patterns بطريقة إرسال واستقبال ال object request، لكن ال CoR يتعامل هذه الطلبات على شكل متسلسلة، كلما انتهت حلقة انتقلت إلى الحلقة التي تليها، بينما ال command يقوم بفتح unidirectional بين ال object المرسل والمستقبل، وال Mediator يفتح اتصال مباشر بين ال object المرسل والمستقبل لكن لا يتم ذلك إلا من خلال ال mediator object، وال Observer يكون فيه استقبال ال request لل subscribe وال unsubscribe بشكل dynamic.

- التشابه بين ال Observer وال Mediator كبير جدا، لكن الفرق الجوهري أن ال Mediator يمثل Single Object تتم من خلاله جميع العمليات، بينما ال Observer يقوم على فكرة إنشاء one-way connection بطريقة dynamic، وكطريقة بسيطة للتفريق بين الإثنين، إذا كانت كل العلاقات بين ال Objects لا تتم إلا من خلال ال Single object فهذا Mediator، أما اذا كان هناك طريقة يتواصل بها أحد ال Object مع غيره فهذا Observer.

فضحايا "الاستبداد السياسي" يغالون في مفهوم الطاعة السياسية فوق القدر الشرعي المأمور به، وينفرون من الاحتساب الولائي، ويميلون للتفهم الشرعي لكل أمر سلطاني، ويبحثون في النصوص الشرعية أو التراث الإسلامي لتبرير وتسويغ أوامر المستبد، حتى لو كان بنوع من التطويع، ويتحدثون عن المستبد بعبارات الانبهار والتفخيم، وكل ذلك نتيجة (انهيارهم النفسي) أمام نفوذ المستبد السياسي

إبراهيم السكران - كتاب سلطة الثقافة الغالبة - ص25

Behavioral Design Patterns - State

ال Pattern السابع في هذه المجموعة هو ال State، هذا ال Pattern من ال Patterns الشائع استخدامها والذي قد لا يخلو مشروع فيه States من حاجة إلى هذا ال Pattern، وفكرته قائمة على تغيير السلوك الخاص بال Object عند حصول تغيير على قيمة ال State، وهذه الطريقة حقيقة تتم من خلال عمل encapsulate لمجموعة مختلفة من ال state داخل نفس ال Object...

Behavioral Design Patterns - State

المشكلة: تخيل أن لديك قسم لبيع منتجاتك من خلال موقعك الإلكتروني، عند بنائك لهذه الصفحات وجدت أن لديك مجموعة مختلفة من الحالات، مثلا Create كأول state افتراضية، وهي تمثل أول صفحة بعد الدخول من عربة التسوق، بعد ذلك يذهب المستخدم ل Payment بحيث تمثل هذه الحالة المنطقة التي تظهر فيها إعدادات الدفع الإلكتروني، وهنا لدينا حالتين، إما أن يتابع المستخدم فتصبح الحالة Payment-Processing وإما أن يلغي العملية فتصبح الحالة Canceled، في حالة ال Payment-Processing إذا تمت العملية بنجاح فإنه يذهب إلى ال Done، وإلا سيذهب إلى ال Fail...، تخيل كمية هذه ال States وكمية ال Logic الذي ستقوم بكتابته للقيام بهذه المهمات، بالطريقة التقليدية ستفكر من خلال بناء function و switch case ومن ثم تقوم بكتابة ال logic الخاص بك، كل هذا له مشاكل كبيرة وكثيرة، أولها أن الشيفرة البرمجية ستصبح معقدة وكبيرة ويصعب تتبعها، ومن الصعب صيانتها، كما أن أي تغيير قد يطرأ على ال state مثلا لو كإضفنا Archived state في منتصف المراحل سيستوجب أخذ نظرة على كل ال state التي تقود للمرحلة التي تليها والتي تسبقها لاستبدالها وإسقاط الحالة الجديدة مكانها...إذا ما الحل؟

Behavioral Design Patterns - State

الحل: هذا ال Pattern يقترح حلا جميلا لهذه المشكلة، فبدلا من أن تتم كتابة جميع الشيفرة البرمجية في مكان واحد، فإننا نقوم بنقل ال state والسلوك الخاص بها والمراد تطبيقه إلى Class مستقل، كل Class يمثل ال state مختلفة في مرحلة ما، مثل class لل state الخاصة بال create، وكذلك لل done...إلخ، ومن ثم إنشاء reference يحفظ فيه ال current state object، ويتم عمل delegate للوظيفة المطلوب تنفيذها بناءا على هذا ال reference، وهذا لا يتم بكل تأكيد إلا بوجود interface مشتركة لجميع ال State Class...، من الأمثلة الواقعية التي تراها أو تتعامل معها، ال Buttons الموجودة على هاتفك المحمول، فمثلا كبسة الإغلاق إذا الهاتف مغلقا فتستخدم للتشغيل، وإذا كان قيد التشغيل قد تستخدم لعمل lock للشاشة، وإذا كانت الشاشة lock فهو يحذف ال lock عنها...إلى آخره، لاحظ أنك تنظر إلى state مختلفة، يختلف السلوك الخاص لل button بناءا على ال state التي هو بها...

Behavioral Design Patterns - State

مثال: الآن سنقوم بتطبيق محاكاة للفكرة السابقة، وهي بناء مجموعة من الخطوات التي تمثل States مختلفة، لكل state منها وظيفة معينة...

1

يعتبر هذا ال Class بمثابة ال Context الذي يحتوي بداخله ال Referance الخاص بال State Class، ويعتبر هذا ال Class بمثابة الواجهة التي سيتعامل ال Client معها، كالتنقل بين ال State...

```php
const CREATE_STATE = "CREATE";
const CANCELED_STATE = "CANCELED";
const PAYMENT_STATE = "PAYMENT";
const PAYMENT_PROCEED_STATE = "PAYMENT_PROCEED";
const DONE_STATE = "DONE";
const FAIL_STATE = "FAIL";
const SUCCESS_STATE = "SUCCESS";
class MyStore {
    private State $state;// This is the reference state
    private string $processStatus;// Normal variable
    public function __construct()
    {
        $this->transitionTo(new CreateState());
    }
    public function transitionTo(State $state): void
    {
        $this->state = $state;
        $this->state->setMyStore($this);
    }
    public function nextStep(): void
    {
        $this->state->nextStep();
    }
    public function getProcessStatus(): string
    {
        return $this->processStatus;
    }
    public function setProcessStatus(string $processStatus)
    {
        $this->processStatus = $processStatus;
    }
}
```

2

يمثل هذا ال Abstract ال interface التي تحتوي جميع ال method التي ستتم مشاركتها معها ال Concrete State، وهو الذي يحتوي على ال Context Reference بحيث يمكن ربطها مع ال State، ولأن هذه العملية مشتركة مع كل ال Concrete State، فقمنا باستخدام ال abstract...

```php
// 2nees.com
abstract class State {
    protected MyStore $myStore;

    public function setMyStore(MyStore $myStore)
    {
        $this->myStore = $myStore;
    }

    abstract public function nextStep(): void;
}
```

Behavioral Design Patterns - State

تمثل هذه ال Classes مجموعة ال Concrete State , وكما تلاحظ، فكل Class يمثل State مختلفة، وكل State لها السلوك الخاص بها، ويمكن إضافة الشرط المناسب لكل حالة...

3

```php
// 2nees.com
class CreateState extends State {
    public function nextStep(): void
    {
        echo "CREATED STEP!" . PHP_EOL;
        $this->myStore->setProcessStatus(CREATE_STATE);
        $this->myStore->transitionTo(new PaymentState());
    }
}

class PaymentState extends State {
    public function nextStep(): void
    {
        echo "PAYMENT STEP!" . PHP_EOL;
        $this->myStore->setProcessStatus(PAYMENT_STATE);
        if(rand(0, 5) > 2){
            $this->myStore->transitionTo(new PaymentProceedState());
        }else {
            $this->myStore->transitionTo(new CanceledState());
        }
    }
}

class PaymentProceedState extends State {
    public function nextStep(): void
    {
        echo "PAYMENT PROCEED STEP!" . PHP_EOL;
        $this->myStore->setProcessStatus(PAYMENT_PROCEED_STATE);

        if(rand(0, 5) > 2){
            $this->myStore->transitionTo(new SuccessState());
        }else {
            $this->myStore->transitionTo(new FailState());
        }
    }
}
```

```php
class FailState extends State {
    public function nextStep(): void
    {
        echo "FAIL STEP!" . PHP_EOL;
        $this->myStore->setProcessStatus(FAIL_STATE);
        $this->myStore->transitionTo(new DoneState());
    }
}

class SuccessState extends State {
    public function nextStep(): void
    {
        echo "SUCCESS STEP!" . PHP_EOL;
        $this->myStore->setProcessStatus(SUCCESS_STATE);
        $this->myStore->transitionTo(new DoneState());
    }
}

class CanceledState extends State {
    public function nextStep(): void
    {
        echo "CANCELED STEP!" . PHP_EOL;
        $this->myStore->setProcessStatus(CANCELED_STATE);
        $this->myStore->transitionTo(new DoneState());
    }
}

class DoneState extends State {
    public function nextStep(): void
    {
        if($this->myStore->getProcessStatus() === DONE_STATE){
            return;
        }

        echo "DONE STEP!" . PHP_EOL;
        $this->myStore->setProcessStatus(DONE_STATE);
        // No need to do any transition, since its last step...
    }
}
```

Behavioral Design Patterns - State

4

يمثل هذا الكود ال Client، ولاحظ بساطة التنفيذ، كل ما يلزم للتنقل بين المراحل هو استدعاء ال nextStep...، وطبعا يمكنك بناء ال Logic الخاص بك كما تريد، مثلا لو احتجت next & prev أو specific action...فكل ذلك
الأمر سهل جدا...، لاحظ كيف قل التعقيد...

```php
// Client...- 2nees.com
for ($i = 1; $i <= 5; $i++){
    $myStore = new MyStore();
    $myStore->nextStep();
    $myStore->nextStep();
    $myStore->nextStep();
    $myStore->nextStep();
    $myStore->nextStep();
    echo "====================================" . PHP_EOL;
}
```

5

```
CREATED STEP!
PAYMENT STEP!
PAYMENT PROCEED STEP!
SUCCESS STEP!
DONE STEP!
====================================
CREATED STEP!
PAYMENT STEP!
PAYMENT PROCEED STEP!
FAIL STEP!
DONE STEP!
====================================
CREATED STEP!
PAYMENT STEP!
PAYMENT PROCEED STEP!
SUCCESS STEP!
DONE STEP!
====================================
CREATED STEP!
PAYMENT STEP!
CANCELED STEP!
DONE STEP!
====================================
CREATED STEP!
PAYMENT STEP!
CANCELED STEP!
DONE STEP!
====================================
```

والآن، إليك الرابط الخاص بالمثال ^^

Behavioral Design Patterns - State

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Context: وهو المكان الذي يحتوي ال Reference الخاصة بال State، هذا ال Reference يشير إلى ال Class State الحالي، ويتم من خلال هذا ال reference عمل delegate للعمل المطلوب...

2. State: وهو ال interface الخاصة بجميع ال concrete state، وفيه ال methods اللازمة للقيام ب logic معين، بحيث يتم كتابة هذا ال logic من خلال ال state class.

3. Concrete State: وهو ال State class لحالة واحدة فقط، مثل ال Create أو Done، وفي هذه ال Classes يتم كتابة السلوك الخاص بهذا ال state، وما هي ال state القادمة...

Behavioral Design Patterns - State

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عندما يتواجد لديك أكثر من State، كل State لها سلوك خاص ووظيفة مرتبطة بها، كما يمكنك استخدامها في حال وجود عدد كبير من ال State والذي قد يزيد من مستوى التعقيد ويخالف مبادئ التصميم بسبب التركيز في كتابة ال logic كله في مكان واحد، كما يمكنك استخدام هذا ال pattern إذا كانت ال state تتغير بشكل مستمر من وإلى...، هذا كله مهم للتقليل من عدد الشروط التي يمكن أن تتواجد في مكان واحد أو داخل Switch case واحدة!

Behavioral Design Patterns - State

المميزات:

- يحقق مبدأ ال Single Responsibility Principle
- يحقق مبدأ ال Open/Closed Principle
- يبسط الشيفرة البرمجية بشكل رهيب، لأنك لن تدوخ وأنت تبحث داخل ال switch case حول طريقة تنفيذ الشيفرة البرمجية وآلية الانتقال...

العيوب:

- في بعض الأحيان يكون استخدام هذا ال Pattern أمرا مبالغا فيه!، وقد يبالغ في استخدامه المطورون، ويكون استخدامه عيبا اذا كان عدد ال State قليل أو نادرا ما تتغير ال State.

Behavioral Design Patterns - State

علاقة ال State مع غيره من ال Pattern:

- ال Bridge وال State وال Strategy وإلى حد ما ال Adapter جميعها Pattern قائمة أو تعتمد على ال composition، فهي تنقل العمل إلى objects أخرى، وحقيقة؛ فإن ال structures الخاص بهم متقارب، لكن كل واحد من هذه ال pattern يقدم حلا لمشكلة معينة، وهنا يكمن جمال ال pattern، فهو لا يمثل فقط حلا أو طريقة لكتابة الشيفرة البرمجية، بل يخبر المبرمجين الآخرين عند رؤيتهم للشيفرة البرمجية ما هي المشكلة التي تم حلها باستخدام هذا الpattern، وهذا يعيدنا في الذكريات إلى الوراء قليلا

- يمكن اعتبار ال State على أنه جزء من ال Strategy، فكلاهما يعتمد على عمل delegate للوظيفة المطلوبة إلى ال Object المناسب، وهذا الأمر يحصل لأننا بحاجة لتنفيذ سلوكيات مختلفة بناءا على كل حالة، لكن الفرق يكمن في أن ال Strategy يجعل هذه ال Object مستقلة عن بعضها تماما ولا يهتم أحدهما بالآخر.

قَالَ الْإِمَامُ أَحْمَدُ رَضِيَ
اللَّهُ عَنْهُ: النَّاسُ إِلَى الْعِلْمِ أَحْوَجُ مِنْهُمْ إِلَى الطَّعَامِ وَالشَّرَابِ. لِأَنَّ الرَّجُلَ يَحْتَاجُ إِلَى الطَّعَامِ وَالشَّرَابِ فِي الْيَوْمِ مَرَّ

كتاب مدارج السالكين في إياك نعبد وإياك نستعين - ج2 - 490 - منزلة العلم

Behavioral Design Patterns - Strategy

ال Pattern الثامن في هذه المجموعة هو ال Strategy، هذا ال Pattern يسمح لك بالقيام بوظيفة معينة بأكثر من طريقة بحيث تكون قابلة للتبديل أثناء ال Runtime!، بمعنى آخر، فإنك قد تقوم ببناء وظيفة معينة يمكن تطبيقها بأكثر من طريقة، كل طريقة من هذه الطرق يمكن استخدامها محل الأخرى دون حصول أي مشكلة، ودون أن تعرف كل طريقة عن الأخرى!، ودون الحاجة لأي علاقة فيما بينهم سوى interface مشتركة!، فكرة بسيطة وسهلة، خصوصا بعد تعرفنا على ال State قبلها ^^، وأعتقد أنك من التعريف لاحظت الفرق بينهم...

Behavioral Design Patterns - Strategy

المشكلة: تخيل أن لديك نص، هذا النص تحتاج إلى تشفيره، عملية التشفير هذه يمكن أن تكون من خلال md5 أو من خلال ال SHA أو أي استراتيجية أخرى، كل استراتيجية من هذه الاستراتيجيات لها طريقتها الخاصة عند التنفيذ، وكتابتهم في مكان واحد أمر سيء، ماذا لو أردنا إضافة 10 استراتيجيات أخرى للتشفير؟!، هل سنقوم بكتابتها أيضا ضمن Switch case؟!، ماذا لو أردت استبدال نوع مكان نوع آخر أثناء ال runtime؟! هل تتخيل كمية التعقيد التي يمكن أن تحدث؟!، كمية الأخطاء وال conflict؟!...بنفس الفكرة، تخيل أن لديك أكثر من طريقة للدفع، مثلا Credit Card أو Paypal، طرق الدفع هذه لكل واحدة منها form خاص بها، وطريقة يتم بنائها لأجلها، فكيف ستقوم ببناء ذلك؟!، ماذا لو أضفنا طرق أخرى؟!، كذلك، لو أردت الذهاب إلى المطار، وتوفر لك الشركة عدة طرق للذهاب هناك، تختلف بها المدة الزمنية والتكلفة، فكيف ستدير الشركة كل هذه الخيارات؟!...، ما الحل؟!

Behavioral Design Patterns - Strategy

الحل: يقدم هذا ال Pattern فكرة بسيطة وجميلة لحل هذه المشكلة، وهي النظر إلى ال Class الذي يحتوي على العديد من الاستراتيجيات لتنفيذ مهمة معينة، ومن ثم فصل كل استراتيجية يمكن تنفيذها داخل class مستقل، ويتم عمل delegate للعمل لها، ودون أن يهتم ال Context بما يجري فعلا داخل هذه ال Strategy classes، كما أن عملية اختيار الاستراتيجية المطلوبة تتم من خلال ال Client، من خلال هذا، ستجد مثلا في المشكلة السابقة أن هناك class لتشفير ال md5، و class آخر لتشفير ال sha...، كل واحدة تؤدي المهمة المطلوبة منها...

Behavioral Design Patterns - Strategy

مثال:

1

Strategy Interface وتستخدم لمشاركة ال method بين جميع ال Concrete Strategy class، والدالة التي بداخلها ستستخدم من قبل ال context للوصول لطريقة التنفيذ الخاصة بال Concrete Strategy...

```php
// 2nees.com
interface EncryptStrategy {
    public function encrypt($text): string;
}
```

2

Context; هذا ال Class يستخدم كواجهة للتعامل، بحيث يتم طلب المهمة المراد إنجازها من خلال ال client، ويقوم هذا ال context بتنفيذ الاستراتيجية المناسبة بناءا على ما أرسله ال client، ويساعده في ذلك امتلاكه لل strategy reference...

```php
// 2nees.com
class EncryptedContext {
    private EncryptStrategy $encryptStrategy;
    private string $text;

    public function __construct(EncryptStrategy $encryptStrategy)
    {
        $this->encryptStrategy = $encryptStrategy;
    }

    public function encryptText(): void {
        echo $this->encryptStrategy->encrypt($this->text) . PHP_EOL;
    }

    public function setText(string $text): void
    {
        $this->text = $text;
    }

    public function setEncryptStrategy(EncryptStrategy $encryptStrategy):
void
    {
        $this->encryptStrategy = $encryptStrategy;
    }
}
```

Behavioral Design Patterns - Strategy

مثال:

تمثل هذه ال classes مجموعة ال Concrete Strategy، ففي كل واحد من هذه ال Classes تحتوي سلوكا مستقلا عن غيره...

3

```php
// 2nees.com
class MD5Encrypt implements EncryptStrategy {
    public function encrypt($text): string
    {
        return "MD5: " . md5($text);
    }
}

class SHA1Encrypt implements EncryptStrategy {
    public function encrypt($text): string
    {
        return "SHA1: " . sha1($text);
    }
}

class Haval160Encrypt implements EncryptStrategy {
    public function encrypt($text): string
    {
        return "haval160,4: " . hash("haval160,4", $text);
    }
}
```

ال client code....

4

```php
// Client...2nees.com
$encrypt = new EncryptedContext(new SHA1Encrypt());
$encrypt->setText("2nees.com");
$encrypt->encryptText();
$encrypt->setEncryptStrategy(new MD5Encrypt());
$encrypt->encryptText();
$encrypt->setEncryptStrategy(new Haval160Encrypt());
$encrypt->encryptText();
```

5

```
SHA1: c63ef2414f554db43d8a8057b7e0137d136f9450
MD5: 353b652f7a32143b2aad78f9f0e43dfe
haval160,4: 21ebdfa5498283bd112d186c1645119367f728f0
```

والآن، إليك الرابط الخاص بالمثال ^^

Behavioral Design Patterns - Strategy

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Strategy Interface: وهي ال interface التي سيتم استخدامها في جميع ال Concrete Strategy، والتي تحتوي بداخلها الدالة المميزة في آلية عملها والتي تختلف من Class إلى Class آخر!، هذه الدالة هي التي سيتم استدعائها من خلال ال context، في مثالنا: encrypt.

2. Context: يمثل هذا ال Class الواجهة التي يتعامل معها ال client، ويحتوي هذا ال Class بداخله على reference الخاص بال Strategy المراد تنفيذها، والتي يتم تحديدها من خلال client، أو يتم تبديلها من خلال ال client.

3. Concrete Strategy: وهي مجموعة ال Classes التي تمثل كل واحدة منها طريقة معينة لتنفيذ الجزء المطلوب، مثل ال md5 أو ال sha1...، والتي يستخدمها ال context ودون أن يدرك ما هي طبيعة الاستراتيجية التي تم تنفيذها، فهو مهتم بالنتائج!، كل ما يهمه هو ال reference الذي سيتصل من خلاله مع الاستراتيجية...، والتي يتم وضعها من خلال ال constructor أو ال setStrategy

Behavioral Design Patterns - Strategy

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند حاجتك لتنفيذ أكثر من سلوك (استراتيجية) للوصول إلى نتيجة معينة بغض النظر عن الطريقة التي تم تنفيذها بها (مثال ال encrypt)، يضاف إلى ذلك أيضا عند حاجتنا للقدرة على تغيير الاستراتيجية أثناء ال runtime!، كما يمكنك استخدام هذا ال Pattern عند حاجتك لتطبيق أكثر من استراتيجية لكنها غير مرتبطة بشكل مباشر مع ال context، وبهذا فبفصل ال implementation عن ال context نكون قد قللنا مستوى التعقيد، ونجحنا في تطبيق مبادئ التصميم، وهذا كله سيقلل من عدد الشروط الموجودة عند التعامل مع أكثر من استراتيجية، وهذا يعني أنك إن رأيت نفسك تكتب مجموعة من الشروط التي تحتوي في ثناياها logic كبير أو معقد، وان مستوى التعقيد يزداد فعليك التفكير بال Strategy (راعي الفرق بينها وبين ال state) ^_*

Behavioral Design Patterns - Strategy

المميزات:

- يمكن تغيير السلوك أثناء ال runtime
- يحقق مبدأ ال Open/Closed Principle

العيوب:

- في بعض الأحيان يكون استخدام هذا ال Pattern أمرا مبالغا فيه!، وقد يبالغ في استخدامه المطورون، ويكون استخدامه عيبا اذا كان عدد ال Strategy قليل أو نادرا ما ستتغير طريقة تنفيذ هذه الاستراتيجيات!
- يجب على ال Client معرفة الاستراتيجيات وأماكن أو كيفية استخدامها!

Behavioral Design Patterns - Strategy

علاقة ال Strategy مع غيره من ال Pattern:

- ال Bridge وال State وال Strategy وإلى حد ما ال Adapter جميعها Pattern قائمة أو تعتمد على ال composition، فهي تنقل العمل إلى objects أخرى، وحقيقة؛ فإن ال structures الخاص بهم متقارب، لكن كل واحد من هذه ال pattern يقدم حلا لمشكلة معينة، وهنا يكمن جمال ال pattern، فهو لا يمثل فقط حلا أو طريقة لكتابة الشيفرة البرمجية، بل يخبر المبرمجين الآخرين عند رؤيتهم للشيفرة البرمجية ما هي المشكلة التي تم حلها باستخدام هذا الpattern، وهذا يعيدنا في الذكريات إلى الوراء قليلا

- ال Command وال Strategy قد يبدوان متشابهان جدا، لكن هناك اختلافات مهمة بينهما، فال Command يقوم على تحويل جميع ال Operation إلى Command، وهذا يسمح لنا بعمل queue او delay...إلى آخره، بينما ال Strategy قائم على فكرة القيام بالعمل في أكثر من طريقة والتبديل بين هذه الطرق ضمن context واحد.

Behavioral Design Patterns - Strategy

- يمكن اعتبار ال State على أنه جزء من ال Strategy، فكلاهما يعتمد على عمل delegate للوظيفة المطلوبة إلى ال Object المناسب، وهذا الأمر يحصل لأننا بحاجة لتنفيذ سلوكيات مختلفة بناءا على كل حالة، لكن الفرق يكمن في أن ال Strategy يجعل هذه ال Object مستقلة عن بعضها تماما ولا يهتم أحدهما بالآخر.

- ال Template Method يعمل من خلال الوراثة، بينما يعمل ال Strategy من خلال ال composition، لذلك، عندما يقوم بال Template Method بالتغيير على سلوك معين فإنه يحتاج إلى إنشاء sub class، بينما ال Strategy يقوم بذلك من خلال تقديم ال Strategy المطلوبة، وهذا يجعل الأول يعمل من خلال ال Class level، بينما يعمل الثاني من خلال ال Object level، وبهذا فإن الأول static، والثاني يمكنه تغيير الاستراتيجية أثناء ال runtime.

بعض الشباب المتطلع للثقافة اليوم يتوهم أنه يعيش انفتاحا وتجديدا بين مجموعة من المنغلقين، ولا يعلم أنه ضحية لمؤامرة سياسية كبرى تستهدف بطرق متنوعة ومن خلال نوافذ مختلفة إعادة ترميم الإسلام لينسجم مع مصالح اللاعبين الكبار، باعتبارها ثقافة الغالب. وأطرف ما في الأمر أن بعض هؤلاء الشباب ينعى على علماء ودعاة أهل السنة ضعف الوعي السياسي، ولا يعلم أنه هو الذي يدار بخيوط السياسة من بعيد، ويطبخ في قدر الهيمنة الغربية وهو لا يدري.

إبراهيم السكران - كتاب سلطة الثقافة الغالبة - ص (19-20)

Behavioral Design Patterns - Template Method

ال Pattern التاسع في هذه المجموعة هو ال Template Method، هذا ال Pattern يقوم على فكرة بناء الهيكل الخاص بالشيفرة البرمجية الخاصة بتنفيذ مهمة معينة في ال SuperClass، ولكن، في نفس الوقت يسمح في الوراثة ويجعل من ال subclass المكان المناسب لعمل implement للشيفرة البرمجية بما يتناسب مع الهدف لهذا ال subclass، وذلك من خلال عمل override لل method التي يمكن تعديلها، لكن، لا يمكن لهذا subclass أن يقوم بتعديل ال structure الخاص بال superclass، فمثلا لو افترضت ترتيب معين لتنفيذ ال method، فلا يمكن تغيير هذا الترتيب من خلال ال subclass، وحقيقة هذا ال pattern يشبه ال strategy pattern، لكن بفرق جوهري هو أنه يعتمد على الوراثة بدلا من ال composition، وهذا ينعكس على أماكن تطبيق هذا ال Pattern، فمثلا تجد أن هذا ال Pattern مفيد جدا إذا كنت مطور لمكتبة أو ext معينة وترغب في مشاركتها مع مطورين آخرين، تريد إعطائهم الحرية في عمل ال implementation لل method مع منهم من التغيير على ال superclass الذي قمت ببنائه وبالشروط التي وضعتها...

هل لاحظت كيف أن اختلاف التكنولوجي وأماكن استخدامها أو طريقة استخدامها هو ما يحدد الأفضل؟!، الوراثة أمر رائع، وال composition كذلك، لكن، ليكون كلاهما رائعا عليك باستخدام المنافع التي يقدمونها في الوقت المناسب، غير ذلك ستقع ضحية للمشاكل التقنية التي اخترتها لأنها لم تكن ملائمة لما طبقته من عمل...

Behavioral Design Patterns - Template Method

المشكلة: تخيل أن لديك ماكينة لإعداد القهوة، هذه الماكينة يمكنها أن تنتج العديد من الأنواع ضمن تصنيفات القهوة أو الشوكولاتة، هذه الماكنة لديها أصناف رئيسية، لكن جميع هذه الأصناف تشترك بأن الماكينة يجب أن تقوم بترتيب محدد إضافة جميع المكونات، وبنفس الوقت ستسمح للمطورين من إضافة أو التحكم بطريقة تنفيذ أو إضافة المكونات حسب كل صنف، فماذا ستفعل؟!

أكبر مشكلة قد تواجهك إذا فكرت بطريقة تقليدية كمية الشروط التي يتوجب عليك إضافتها وكيف يمكنك الحفاظ على الترتيب الذي ترغب بإضافته دون تعديل، ودون تكرار الكود في كل مكان!، أما إذا فكرت في طريقة تعتمد من خلالها على interface وعمل polymorphism، فسأقول لك...أحسنت، لنذهب إلى الحل ^_*

Behavioral Design Patterns - Template Method

الحل: يقدم هذا ال Pattern حلا بسيطا ومفيدا لهذه المشكلة، وهي بناء template لا يمكن تعديله من قبل ال subclass، مع الحفاظ على إمكانية تعديل ال implementation لل methods، وهذا يعني أنك ستقوم بإضافة الخطوات الخاصة بإعداد أي صنف حسب شروط ماكينة القهوة داخل method لا يمكن عمل override لها، وبهذا فأنت تضمن أن الترتيب الذي قمت بوضعه سيتم تنفيذه دون خوف، ومهما كان عدد الأصناف الجديدة...، كل صنف جديد سيمثل subclass يرث ال abstract class وهو الماكينة، ومن ثم سيقوم بالتعديل بنائا على احتياجاته، ويمكن إضافة default methods إذا كان التعديل على method ما قليل أو غير مستخدم في أغلب الحالات...

الموضوع سهل؟ نعم، إذا لنشاهد المثال ^^

Behavioral Design Patterns - Template Method

مثال: لنقم بمحاولة محاكاة للمثال السابق...

يمثل هذا ال Abstract Class الجزء المشترك والحاوي لل Template Method، بالإضافة إلى ال methods التي سيقوم بتنفيذها جميع ال Concrete SubClass

1

```php
// 2nees.com
abstract class CoffeeMachine {
    // This is the Template Method
    final public function prepareCoffee(): void {
        echo "Preparing Start..." . PHP_EOL;
        echo $this->addCup() . PHP_EOL;
        echo $this->addWater() . PHP_EOL;
        echo $this->addMilk() . PHP_EOL;
        echo $this->addSugar() . PHP_EOL;
        echo "Done! you can got your Cup!" . PHP_EOL;
    }

    abstract protected function addWater(): string;
    abstract protected function addSugar(): string;
    abstract protected function addMilk(): string;
    // Step, but can override!, this is the default cub
    protected function addCup(): string {
        return "Add Carton Cub!";
    }
}
```

Behavioral Design Patterns - Template Method

تمثل هذه ال Subclass ال Concrete الذي يقوم بعمل وراثة لل Abstract، ومن ثم القيام بالعمليات اللازمة...

2

```php
class Coffee extends CoffeeMachine {// 2nees.com
    protected function addWater(): string
    {
        return "Add Hot Water to Coffee";
    }
    protected function addSugar(): string
    {
        return "Add Sugar to Coffee";
    }
    protected function addMilk(): string
    {
        return "skip milk...=>";// No milk on coffee
    }
}
class HotChocolate extends CoffeeMachine {
    protected function addWater(): string
    {
        return "Add Hot Water to HotChocolate";
    }
    protected function addSugar(): string
    {
        return "skip sugar...=>";// No Sugar on HotChocolate
    }
    protected function addMilk(): string
    {
        return "Add Milk";
    }
}
class IceCoffee extends CoffeeMachine {
    protected function addWater(): string
    {
        return "Add Cold Water to IceCoffee";
    }
    protected function addSugar(): string
    {
        return "Add a little bit of Sugar";
    }
    protected function addMilk(): string
    {
        return "Add Milk";
    }
    protected function addCup(): string {
        return "Add Plastic Cub!";
    }
}
```

3

```php
// Client - 2nees.com
$coffee = new Coffee();
$coffee->prepareCoffee();
echo "============================" . PHP_EOL;
$hotChocolate = new HotChocolate();
$hotChocolate->prepareCoffee();
echo "============================" . PHP_EOL;
$iceCoffee = new IceCoffee();
$iceCoffee->prepareCoffee();
```

لاحظ طريقة العمل عند ال client والنتائج الظاهرة، لاحظ أن الأصناف الثلاثة تمت معالجتها بنفس الترتيب وكما هو متوقع...

4

```
Preparing Start...
Add Carton Cub!
Add Hot Water to Coffee
skip milk...=>
Add Sugar to Coffee
Done! you can got your Cup!
============================
Preparing Start...
Add Carton Cub!
Add Hot Water to HotChocolate
Add Milk
skip sugar...=>
Done! you can got your Cup!
============================
Preparing Start...
Add Plastic Cub!
Add Cold Water to IceCoffee
Add Milk
Add a little bit of Sugar
Done! you can got your Cup!
```

والآن، إليك الرابط الخاص بالمثال

^^

Behavioral Design Patterns - Template Method

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Abstract Class: وهو الجزء الحاوي لجميع الدوال المشتركة والتي سيتم استخدامها في ال Concrete Class، ويطلق على هذه الدوال اسم steps، بالإضافة إلى امتلاكه ال Template Method والتي تحتوي بداخلها ال logic الذي لا يمكن تعديله.

2. Concrete SubClass: مجموعة ال classes التي ترث ال Abstract وتقوم بتنفيذ العمليات المطلوبة على ال steps من خلال عمل override عند الحاجة، مع عدم القدرة على تعديل ال template method

3. Client: المكان المخصص لتنفيذ الشيفرة البرمجية الخاصة بهذه الأنواع، ويمكن الوصول للنتائج من خلال عمل call لل template method حتى نحصل على النتيجة المتوقعة...

Behavioral Design Patterns - Template Method

إذا، متى يمكنني استخدام هذا ال Pattern؟
بكل بساطة يمكنك استخدام هذا ال Pattern عند حاجتك للحفاظ على طريقة عمل الكود أو ترتيبه من أي تعديل، وبنفس الوقت تريد إعطاء صلاحية لل client في كتابة التنفيذ الخاص به للوظائف التي يرغب بإنجازها، كما أن هذا ال Pattern مفيد في حالة وجود العديد من ال class التي ستقوم بوظيفة محددة تتشابه فيما بينها ما اختلاف بسيط بين هذه ال Classes، فبدلا من تكرار الشيفرة البرمجية يمكنك الحفاظ عليها ومنع التكرار من هذا ال Pattern، باختصار استخدم هذا ال Pattern عند حاجتك للسماح في التوسع للشيفرة البرمجية مع رغبتك بعدم السماح للتعديل على هيكلية التنفيذ، ومن الأمثلة على ذلك، إذا كنت تنوي بناء framework وفيه بعض الخصائص التي سيقوم بها المطورون لكنك لا ترغب في السماح لهم بتعديل ال structure الخاص بال framework...، والكثير من الحالات أو الأمثلة المشابهة لهذا المفهوم.

Behavioral Design Patterns - Template Method

المميزات:

- يقلل من ال code duplication لأنك قادر على نقل هذه الشيفرة البرمجية لل superclass
- ال client قادر على تبديل أو تعديل جزء معين، وهذا يعني تقليل مستوى الخطر من تعديل الشيفرة البرمجية

العيوب:

- كما أنها ميزة أن تحدد الجزء القابل للتعديل، فهية قد تكون من المحددات عند حاجة ال client للتعديل عليها...
- مع طريقة التنفيذ هذه، قد تنتهك مبدأ Liskov

Behavioral Design Patterns - Template Method

علاقة ال Template Method مع غيره من ال Pattern:

- ال Template Method يعمل من خلال الوراثة، بينما يعمل ال Strategy من خلال ال composition، لذلك، عندما يقوم بال Template Method بالتغيير على سلوك معين فإنه يحتاج إلى إنشاء sub class، بينما ال Strategy يقوم بذلك من خلال تقديم ال Strategy المطلوبة، وهذا يجعل الأول يعمل من خلال ال Class level، بينما يعمل الثاني من خلال ال Object level، وبهذا فإن الأول static، والثاني يمكنه تغيير الاستراتيجية أثناء ال runtime.
- ال Factory Method يمثل نوع مخصص من ال Template Method، في ال Method Template ذات الحجم الكبير، يمكن استخدام ال Factory Method باعتباره أحد ال steps داخل هذا ال template أو لخدمة هذا ال template بالطريقة المناسبة.

من أساليب الهداية القرآنية إلى العلوم الكونية، أن يعرض علينا القرآن صوراً من العالم العلوي والسفلي، في بيان بديع جذاب، يشوقنا إلى التأمل فيها، والعمق في أسرارها.

وهنا يذكر لنا ما خبأه في السموات والأرض لنشتاق إليه، وننبعث في البحث عنه، واستجلاء حقائقه، ومنافعه؛ بدافع غريزة حب الاستطلاع، ومعرفة المجهول.

وبمثل هذا انبعث أسلافنا في خدمة العلم، واستثمار ما في الكون، إلى أقصى ما استطاعوا، ومهدوا بذلك السبيل لمن جاء بعدهم، ولن نعز عزهم إلاّ إذا فهمنا الدين فهمهم وخدمنا العلم خدمتهم.

تفسير ابن باديس - من تفسير سورة النمل - تشويق القرآن إلى علوم الأكوان

Behavioral Design Patterns - Visitor

ال Pattern العاشر والأخير في هذه المجموعة هو ال Visitor، هذا ال Pattern يعد من ال Patterns الأكثر تعقيدا والأقل انتشارا، ومع ذلك، فهو يقدم حلا لمشكلة حميلة في عالم البرمجة، وسيكون هذا ال Pattern أحد الحلول الجميلة لها، فكرة هذا ال Pattern قائمة على مبدأ فصل الطريقة التي سيتم فيها كتابة الشيفرة البرمجية عن ال Object نفسه، بحيث يقوم كل Object فقط بعمل call لل method المناسبة والتي ترثها مجموعة من ال classes المختلفة، والتي تحتوي دوال مختلفة!، ممم، إذا لم يصل إليك المعنى من هذا التعريف...، لنذهب ونشاهد المشكلة وحلها ^^

ملاحظة: هذا ال Pattern قد يستغل أحد الأساليب التي تدعمها بعض لغات البرمجة وهو ال overloading، هذا الأسلوب يسمح لك بإنشاء أكثر من دالة بنفس
الإسم لكن ب Parameters مختلفة، هذا الأسلوب ليس مدعوما بكل لغات البرمجة، وقد يحتاج لحل مشكلة معينة للحفاظ على ref لذلك، يمكن الإعتماد على اسم الدالة بدلا من ال Param، مثال:

```
// Overloading
visit(One);
visit(Two);

// Overloading Not Support
visitOne(One);
visitTwo(Two);
```

ملاحظة 2: يمكن استخدام ال PHP magic function لبناء شيء مشابه، لكنني أفضل المثال كما في الصورة...(من خلال ال method naming)

Behavioral Design Patterns - Visitor

المشكلة: تخيل أن لديك شركة تأمين، شركة التأمين هذه يمكنها أن تتعامل مع عدة أنواع من الشركات، وأن تقدم عروض تأمين بما يتناسب مع الشركات، لنفترض أن شركة التأمين قد بدأت بتقديم عروض التأمين للمنازل والسيارات فقط، ثم بعد ذلك جاء قسم المبيعات وأخبرك بوجود عميل جديد، كالبنك مثلا، ثم الميناء، ثم المستشفيات...إلى آخره، ثم أخبرك برغبة المبيعات بإضافة خيار تصدير للتقارير الشهرية للبنوك والمستشفيات، مع العلم أن كل واحدة من هذه الشركات لها نظام خاص بالتأمين، فطريقة التأمين التي تنطبق على البنوك من السرقة تختلف عن السيارات وتختلف عن المنازل...، فإذا قلت يمكن حل هذه المشكلة بسهولة من خلال ال Polymorphism، سأقول لك، أحسنت، فهذا حل جميل، لكنه قد يتسبب بمشاكل كثيرة قد يرفض الكود لأجلها!، مثلا إذا الشيفرة البرمجية الحالية على سيرفر ال Production، فإن التعديل هناك سيكون خطرا، لأن العملاء يتعاملون مع هذا النظام حاليا، وإضافة سلوكات مختلفة كما ذكرنا بالأعلى سيزيد من التعقيد وال risk، بالإضافة إلى ذلك، في مثل هذا السلوك من التعديل، قد تتعدد مسؤوليات ال class الواحد، فبدلا أن يقوم بتنفيذ ما يحتاجه فقط، سيحتوي implementation خارج فكرة ال Class...، ثم كيف يمكننا إضافة شروط مؤقتة عند النوازل؟!، مثلا في كورونا أضيفت بعض الشروط أو تغيرت بناءا على نوع التأمين، وهذا التعديل سيكون لفترة مؤقتة، فكيف ستقوم بهذا ودون خسارة الشيفرة البرمجية الأصلية؟!، ما الحل برأيك؟

Behavioral Design Patterns - Visitor

الحل: يقدم هذا ال Pattern حلا جميلا لهذه المشكلة، وذلك من خلال فصل كل فئة إلى class مستقل بذاته يطلع عليه Visitor، فبدلا من إضافة الشيفرة البرمجية على نفس ال Class، سيستقبل هذا ال Visitor ال Original Object، وهذا يعني أن ال Visitor يستطيع الوصول إلى جميع المعلومات التي يرغب في الحصول عليها، جميع ال Visitors يشتركون في Interface واحدة تحتوي جميع ال methods المرتبطة بال Component Class، وكل Component Class لديها interface مشتركة فيما بينها فيها method وظيفتها عمل accepting لل Visitor، والتي تم تعريفها في ال interface الأولى...

لاحظ أننا في هذا التعديل قمنا بإسقاط التعديل على ال class الأصلي لمرة واحدة فقط، وهو إضافة ال method على ال classes الموجودة، وهذا لن يتكرر مجددا لأننا سنقوم بإضافة ال Visitor بسهولة بعدها...، وهكذا تخلصنا من المشكلة الأولى والثانية، وبسبب هذا الأسوب، فنحن قادرون على إضافة Visitors مستقلة بذاتها للقيام بوظائف تختلف شروطها عن الشيفرة الأصلية، وبهذا فإنك قادر على إزالة أو إضافة أي خاصية في أي وقت ودون الخوف من التعديل على ال core، وبهذا تكون المشاكل التي تحدثنا عنها قد ذهبت بإذن الله تعالى ^_^...فكرة جميلة...لنحاول تطبيقها ^^

Behavioral Design Patterns - Visitor

مثال: لنحاول محاكاة المثال السابق، وليكن سبب إضافة ال Visitor هو فيروس كورونا -عافانا الله وإياكم من كل سوء-...

Behavioral Design Patterns - Visitor

2-A

```php
class HomeInsurance implements InsuranceContracts {// 2nees.com
    public function getCost(): int
    {
        return rand(200, 500);
    }

    public function getContracts(): string
    {
        return "Your Home Insurance Contracts Is Approved for 1 year: {$this->getCost()}";
    }

    public function accept(Visitor $visitor): string
    {
        return "Home Insurance: " . $visitor->visitHome($this);
    }
}
```

تمثل هذه ال Classes من a-2 إلى d-2 ال Concrete Element class، وأهم جزئيتين في هذه ال Classes هما ال implement الخاص بال accept، لأن من خلال هذه الدالة سيتم إرجاع ال Object الخاص بهذا ال class إلى ال visitor...، والثانية أن كل Concrete من هذه ال classes d حتوي دوال مختلفة عن الأخرى، وطرق معالجة بيانات تختلف عن الأخرى، ومع ذلك، يستطيع ال Visitor الوصول إلى كل ال method التي يحتاجها بناءا على نوع كل class....

2-B

```php
class CarInsurance implements InsuranceContracts {
    private int $engineSize;
    public function __construct()
    {
        $this->engineSize = 1;
    }
    public function getCost(): int
    {
        return 100 * $this->getEngineSize();
    }
    public function getEngineSize(): int
    {
        return $this->engineSize;
    }
    public function setEngineSize(int $engineSize): void
    {
        $this->engineSize = $engineSize;
    }
    public function getContracts(): string
    {
        return "Your Car Insurance Contracts Is Approved for 1 year: {$this->getCost()}
         And Your Engine is: {$this->getEngineSize()}";
    }
    public function accept(Visitor $visitor): string
    {
        return "Car Insurance: " . $visitor->visitCar($this);
    }
}
```

Behavioral Design Patterns - Visitor

2-C

```php
class BankInsurance implements InsuranceContracts {// 2nees.com
    private int $numberOfEmployee;
    private int $cost;
    public function __construct()
    {
        $this->numberOfEmployee = 50;
        $this->cost = 250;
    }
    public function getCost(): int
    {
        return $this->cost * $this->numberOfEmployee;
    }
    public function setCost($cost): void{
        $this->cost = $cost;
    }
    public function getContracts(): string
    {
        return "
                ******************
             Bank Contracts:
             Total Cost: {$this->getCost()}
             Number OF Employee: {$this->numberOfEmployee}
             And Cost For every one is: {$this->cost}
                ******************
         ";
    }
    public function accept(Visitor $visitor): string
    {
        return "Bank Insurance: " . $visitor->visitBank($this);
    }
}
```

2-D

```php
class HospitalInsurance implements InsuranceContracts {// 2nees.com
    private int $numberOfRoom;
    private int $numberOfBeds;
    public function __construct()
    {
        $this->numberOfRoom = 20;
        $this->numberOfBeds = 50;
    }
    public function getCost(): int
    {
        return 1000 * $this->numberOfBeds * $this->numberOfRoom;
    }
    public function getContracts(): string
    {
        return "
                ******************
                 Hospital Contracts:
                 Total Cost: {$this->getCost()}
                ******************
         ";
    }
    public function accept(Visitor $visitor): string
    {
        return "Hospital Insurance: " . $visitor->visitHospital($this);
    }
    public function getNumberOfRoom(): int
    {
        return $this->numberOfRoom;
    }
    public function getNumberOfBeds(): int
    {
        return $this->numberOfBeds;
    }
    public function setNumberOfRoom(int $numberOfRoom): void
    {
        $this->numberOfRoom = $numberOfRoom;
    }
    public function setNumberOfBeds(int $numberOfBeds): void
    {
        $this->numberOfBeds = $numberOfBeds;
    }
}
```

avioral Design Patterns - Visitor

4 يسمى هذا ال class بال Concrete Visitor، وهو المكان المسؤول عن تطبيق الجديد بناءا على شرط معين..
وبكل تأكيد يمكن بناء أكثر من Visitor واستخدامه...

```php
class CovidCondition implements Visitor {// 2nees.com
    private bool $includeCovid;
    public function __construct(bool $includeCovid)
    {
        $this->includeCovid = $includeCovid;
    }
    public function visitHome(HomeInsurance $homeInsurance)
    {
        echo "You are in Covid Visitor Conditions, and your contract cant
        contain Covid!" . PHP_EOL;
        echo $homeInsurance->getContracts() . PHP_EOL;
    }
    public function visitCar(CarInsurance  $carInsurance)
    {
        echo "You are in Covid Visitor Conditions, and your contract cant
        contain Covid!" . PHP_EOL;
        echo $carInsurance->getContracts() . PHP_EOL;
    }
    public function visitBank(BankInsurance $bankInsurance)
    {
        if($this->includeCovid){
            echo "You are in Covid Visitor Conditions, and your company pay
            to include Covid in your Insurance!" . PHP_EOL;
            $bankInsurance->setCost(125);
        }else {
            echo "You are in Covid Visitor Conditions, and your company not pay
            to include this type of disease!" . PHP_EOL;
        }

        echo $bankInsurance->getContracts() . PHP_EOL;
    }
    public function visitHospital(HospitalInsurance $hospitalInsurance)
    {
        if($hospitalInsurance->getNumberOfRoom() < 25){
            echo "Sorry, After Covid small hospital not supported!" . PHP_EOL;
            return;
        }

        if($this->includeCovid){
            echo "You are in Covid Visitor Conditions, and your company pay
            to include Covid in your Insurance!" . PHP_EOL;
            $contracts = $hospitalInsurance->getContracts();
            $contracts .= "Number of beds Support:
            {$hospitalInsurance->getNumberOfBeds()}" . PHP_EOL;
            $contracts .= "Number of Room Support:
            {$hospitalInsurance->getNumberOfRoom()}" . PHP_EOL;
            echo $contracts . PHP_EOL;
        }else {
            echo "Sorry, Your Hospital Must be include Covid Addon For
            its insurance Contract!!" . PHP_EOL;
        }
    }
}
```

3 تمثل هذه ال Interface ال Visitor Interface، وهي المكان الذي تتواجد به جميع الدوال الخاصة بال Concrete Class...(هنا يمكن استخدام ال overloading)

```php
interface Visitor {// 2nees.com
    public function visitHome(HomeInsurance $homeInsurance);
    public function visitCar(CarInsurance $carInsurance);
    public function visitBank(BankInsurance $bankInsurance);
    public function visitHospital(HospitalInsurance $hospitalInsurance);
}
```

4 Client Code، لاحظ أن كل ما قمنا بفعله هو إنشاء بعمل call لل accept...

```php
// Clint - 2nees.com
$homeInsurance = new HomeInsurance();
$carInsurance = new CarInsurance();
$bankInsurance = new BankInsurance();
$hospitalInsurance = new HospitalInsurance();
$covid1 = new CovidCondition(true);
$homeInsurance->accept($covid1);
$carInsurance->accept($covid1);
$bankInsurance->accept($covid1);
$hospitalInsurance->accept($covid1);
echo "====================End 1==========================" . PHP_EOL;
$covid2 = new CovidCondition(false);
$homeInsurance->accept($covid2);
$carInsurance->accept($covid2);
$bankInsurance->accept($covid2);
$hospitalInsurance->accept($covid2);
echo "====================End 2==========================" . PHP_EOL;
$covid3 = new CovidCondition(true);
$homeInsurance->accept($covid3);
$carInsurance->setEngineSize(20);
$carInsurance->accept($covid3);
$bankInsurance->accept($covid3);
$hospitalInsurance->setNumberOfRoom(50);
$hospitalInsurance->setNumberOfBeds(100);
$hospitalInsurance->accept($covid3);
echo "====================End 3==========================" . PHP_EOL;
```

```
You are in Covid Visitor Conditions, and your contract cant contain Covid!
Your Home Insurance Contracts Is Approved for 1 year: 433
You are in Covid Visitor Conditions, and your contract cant contain Covid!
Your Car Insurance Contracts Is Approved for 1 year: 100
        And Your Engine is: 1
You are in Covid Visitor Conditions, and your company pay to include Covid in your Insurance!

                ******************
            Bank Contracts:
            Total Cost: 6250
            Number OF Employee: 50
            And Cost For every one is: 125
                ******************

Sorry, After Covid small hospital not supported!
====================End 1==========================
You are in Covid Visitor Conditions, and your contract cant contain Covid!
Your Home Insurance Contracts Is Approved for 1 year: 254
You are in Covid Visitor Conditions, and your contract cant contain Covid!
Your Car Insurance Contracts Is Approved for 1 year: 100
        And Your Engine is: 1
You are in Covid Visitor Conditions, and your company not pay to include this type of disease!

                ******************
            Bank Contracts:
            Total Cost: 6250
            Number OF Employee: 50
            And Cost For every one is: 125
                ******************

Sorry, After Covid small hospital not supported!
====================End 2==========================
You are in Covid Visitor Conditions, and your contract cant contain Covid!
Your Home Insurance Contracts Is Approved for 1 year: 394
You are in Covid Visitor Conditions, and your contract cant contain Covid!
Your Car Insurance Contracts Is Approved for 1 year: 2000
        And Your Engine is: 20
You are in Covid Visitor Conditions, and your company pay to include Covid in your Insurance!

                ******************
            Bank Contracts:
            Total Cost: 6250
            Number OF Employee: 50
            And Cost For every one is: 125
                ******************

You are in Covid Visitor Conditions, and your company pay to include Covid in your Insurance!

                ******************
                 Hospital Contracts:
                 Total Cost: 5000000
                ******************
        Number of beds Support: 100
Number of Room Support: 50

====================End 3==========================
```

هنا قمنا باستخدام ال visitor...

6

Behavioral Design Patterns - Visitor

```
Your Home Insurance Contracts Is Approved for 1 year: 412
Your Car Insurance Contracts Is Approved for 1 year: 100
        And Your Engine is: 1

                ******************
            Bank Contracts:
            Total Cost: 12500
            Number OF Employee: 50
            And Cost For every one is: 250
                ******************
```

في حالة التنفيذ المباشر دون وجود أي Visitor، ولنفترض أنه ال Core or Default code....

5

```
                ******************
                 Hospital Contracts:
                 Total Cost: 1000000
                ******************

====================Default Implementation==========================
```

والآن، إليك الرابط الخاص بالمثال ^^

Behavioral Design Patterns - Visitor

بناءا على المثال السابق، يمكننا أن نستنتج أن مكونات هذا ال Pattern هي:

1. Element Interface: وهي ال interface التي تحتوي بداخلها ال accept method، وعن طريق دالة ال accept يتم استقبال ال Visitor Object...

2. Visitor Interface: وهي ال interface التي تحتوي بداخلها جميع ال methods الخاصة بال Concrete Element.

3. Concrete Element: وهي ال Core Class والتي تحتوي بداخلها على ال implementation الخاص بال accept method، داخل ال implementation يتم استخدام ال Visitor Object للوصول إلى ال Method الخاصة بهذا ال Class، ومن إرسال البيانات الخاصة بهذا ال class إلى ال Visitor ليستطيع الوصول لل state المطلوبة.

4. Concrete Visitor: وهو كل Class يقوم بتنفيذ أو إضافة عناصر أو خصائص شروط بطريقة مختلفة أو جديدة، ويمكن أن يكون واحد أو أكثر حسب الحاجة، هذا ال Class هو الحاوي لل implementation الخاص بال method والتي تتناسب مع ال Concrete Element.

Behavioral Design Patterns - Visitor

إذا، متى يمكنني استخدام هذا ال Pattern؟

بكل بساطة يمكنك استخدام هذا ال Pattern عند حاجتك لفصل الوظائف الأساسية في التطبيق الخاص بك عن ال Main class لتبقى أكثر عمقا، وأكثر أمانا، كما يمكنك استخدامها عند وجود Structure معقد ل object ما، بحيث يمكنه التعامل مع أكثر من class، فكما لاحظت في المثال السابق، تم التعامل مع 4 classes مختلفة ودون الاهتمام بذلك!، فال accept ستتولى المهمة عنك...، كما يمكن استخدام هذا ال Pattern عند حاجتك لإضافة شروط أو إضافات طارئة على المشروع وليست من صلب ال classes الموجودة، وقد تكون لفترة مؤقتة أو لمعالجة حالة خاصة...، كل هذه الأسباب قد تجعلك تفكر بهذا ال Pattern...

Behavioral Design Patterns - Visitor

المميزات:

- يحقق مبدأ ال Open/Closed Principle
- يحقق مبدأ ال Single Responsibility Principle
- أسلوب جيد في التعامل مع ال Complex Object Structure

العيوب:

- أي حذف أو إضافة على ال Elements Class، ستحتاج إلى عكس ذلك على Visitors.
- يعد مستوى التعقيد لهذا ال Pattern مرتفع...

Behavioral Design Patterns - Visitor

علاقة ال Visitor مع غيره من ال Pattern:

- يمكنك التعامل مع ال Visitor وكأنه Command في نسخة مطورة ذو صلاحيات أعلى، فيمكنك من خلاله تنفيذ العمليات بين مجموعة مختلفة من ال Objects من classes مختلفة.

- يمكنك استخدام ال Visitor لتنفيذ عمليات على كامل ال Composite trees.

- يمكن استخدام ال Visitor مع ال Iterator للحصول على البيانات المطلوبة من ال Complex Data Structure، كما يمكن تنفيذ العديد من المهمات أثناء الحصول على هذه البيانات، وكل هذا حتى لو كانت من classes مختلفة...

قال تعالى في سورة طه: "فَتَعَالَى اللَّهُ الْمَلِكُ الْحَقُّ ۗ وَلَا تَعْجَلْ بِالْقُرْآنِ مِن قَبْلِ أَن يُقْضَىٰ إِلَيْكَ وَحْيُهُ ۖ **وَقُل رَّبِّ زِدْنِي عِلْمًا** (114)"

لما كانت عجلته صلى الله عليه وسلم، على تلقف الوحي ومبادرته إليه، تدل على محبته التامة للعلم وحرصه عليه، أمره الله تعالى أن يسأله زيادة العلم، فإن العلم خير، وكثرة الخير مطلوبة، وهي من الله، والطريق إليها الاجتهاد، والشوق للعلم، وسؤال الله، والاستعانة به، والافتقار إليه في كل وقت.
ويؤخذ من هذه الآية الكريمة، الأدب في تلقي العلم، وأن المستمع للعلم ينبغي له أن يتأنى ويصبر حتى يفرغ المملي والمعلم من كلامه المتصل بعضه ببعض، فإذا فرغ منه سأل إن كان عنده سؤال، ولا يبادر بالسؤال وقطع كلام ملقي العلم، فإنه سبب للحرمان، وكذلك المسئول، ينبغي له أن يستملي سؤال السائل، ويعرف المقصود منه قبل الجواب، فإن ذلك سبب لإصابة الصواب.

تفسير السعدي - سورة طه - الآية 114

الخاتمة

عند وصولك إلى هذه الشريحة، فهذا يعني أنك أتممت قراءة الكثير والعديد من المواضيع المهمة والشيقة، لكن، لا تظن أن هذه هي النهاية!، بل يجب أن تكون هذه هي البداية لمواضيع أعمق وأكبر، وليكن الكتاب صاحبك دوما!

إن أهم ما يجب عليك إدراكه أن ال Design Pattern هي مجرد وسيلة أو آلية تفكير لحل مشاكل معينة، آلية التفكير هذه ليست عقيدة يجب أن تسير عليها في حذافيرها في كل وقت، فمثلا قد يكون إنشاء Simple Class مع interface بسيطة أفضل من استخدام ال Design pattern في السيناريو المباشر، والقصد من هذا أن تعقيد الشيفرة البرمجية دون داع لذلك أمر سيء، بالإضافة إلى ذلك، تجاهل ال Design Pattern عند التعرض للمشاكل التي قد وجدت لحلها، سيكون أمرا سيئا أيضا!، لأجل ذلك، فوظيفتك هي النظر إلى المشكلة ثم البحث عن الحل المناسب، ومقارنة حسنات وسيئات كل فكرة قبل الشروع بتنفيذها...، وتأكد أن هذه بعض ال Patterns الموجودة وليست كلها، لكن، من هذه اللحظة ومن هذا الموقع...، يمكنك الانطلاق والتعرف على المزيد أو اختراع Pattern جديد ونشره...والله ولي التوفيق.

وآخر دعوانا أن الحمد لله رب العالمين