للأصناف الأساسية بحاوي الخدمات مما يوفر كل فوائد (static proxies) "الساكنة "وسطاء ساكنات Laravelتمثل واجهات المقتضبة والمُعبّرة مع الحفاظ على قابلية الاختبار ومرونة أكبر من الدالات الساكنة التقليدية. كل واجهات (syntax)الصِّيغ Laravel الساكنة مُعرَّفةٌ في مجال الأسماءIlluminate\Support\Facades

كيفية الاستخدام

1. تأكد من إمكانية توفير واجهة أبسط من التي يوفرها النظام الفرعي الموجود حاليًا، فإن كانت الواجهة ستجعل شيفرة العميل مستقلة من فئات النظام الفرعي فهي أفضل إذًا من واجهة النظام الموجودة.

2.

جديدة، ينبغي لهذه الواجهة أن تعيد توجيه الاستدعاءات من (facade)صرّح عن هذه الواجهة واستخدمها في فئة واجهة شيفرة العميل إلى الكائنات المناسبة في النظام الفرعي، ويجب أن تكون الواجهة مسؤولة عن بدء النظام الفرعي وإدارة دورة حياته ما لم تكن شيفرة العميل تقوم بهذا فعلًا.

3. للحصول على أقصى فائدة من النمط، اجعل شيفرة العميل تتواصل مع النظام الفرعي من خلال الواجهة فقط، وهكذا تكون شيفرة العميل محصنة من أي تغيير يحدث في شيفرة النظام الفرعي. فمثلًا حين يحصل نظام فرعي على ترقية إلى إصدار أعلى، فلا تحتاج إلى تغيير شيء سوى الشيفرة التي في الواجهة.

4. إن أصبحت الواجهة كبيرة جدًا، فابحث إمكانية استخراج جزء من سلوكها إلى فئة واجهة جديدة منقحة.

المزايا

يمكنك عزل شيفرتك من تعقيد النظام الفرعي.

العيوب

يرتبط بكل الفئات في التطبيق (God Object)يمكن أن تتحول الواجهة إلى كائن إلهي.

أمثلة الاستخدام: يشيع استخدام نمط الواجهة في التطبيقات المكتوبة بلغة جافا، وهو مفيد خاصة عند العمل مع مكتبات وواجهات برمجة تطبيقات معقدة. إليك بعض أمثلة الواجهة في مكتبات جافا:

تستخدم مكتبة javax.faces.context.FacesContext فئات LifeCycle و ViewHandler و NavigationHandler ، لكن أغلب العملاء لا تدرك هذا.

تستخدم مكتبة javax.faces.context.ExternalContext فئات ServletContext و HttpSession و HttpServletRequest و HttpServletResponse وغيرها.

يمكن ملاحظة نمط الواجهة في فئة لديها واجهة بسيطة لكن تفوض أغلب العمل إلى فئات أخرى، وعادة ما تدير الواجهات دورة حياة كاملة للكائنات التي تستخدمها.

أمثلة الاستخدام: يشيع استخدام نمط الواجهة في التطبيقات المكتوبة بلغة PHP ، وهو مفيد خاصة عند العمل مع مكتبات وواجهات برمجة تطبيقات معقدة.

<?php

```
namespace RefactoringGuru\Facade\Conceptual;

/**
 * نمط الواجهة يوفر واجهة بسيطة للمنطق المعقد الذي يكون في نظام فرعي أو أكثر
 * وتفوض  الواجهة طلبات العميل إلى الكائنات المناسبة داخل النظام الفرعي، كما تكون
 * مسؤولة عن إدارة دورات الحياة الخاصة بها.
 * كل هذا يحمي العميل من التعقيد غير المرغوب فيه للنظام الفرعي.
 */
class Facade
{
    protected $subsystem1;

    protected $subsystem2;

    /**
     * بناءً على احتياجات برنامجك، تستطيع تزويد الواجهة بكائنات من النظام الفرعي أو
     * تجبرها على إنشائها بنفسها.
     */
    public function __construct(
        Subsystem1 $subsystem1 = null,
        Subsystem2 $subsystem2 = null
    ) {
        $this->subsystem1 = $subsystem1 ?: new Subsystem1;
        $this->subsystem2 = $subsystem2 ?: new Subsystem2;
    }

    /**
     * أساليب الواجهة هي اختصارات سهلة للوظائف المعقدة للنظم الفرعية، لكن العملاء لا
```

```
     * يحصلون إلا على جزء بسيط من قدرات النظام الفرعي.
     */
    public function operation(): string
    {
        $result = "Facade initializes subsystems:\n";
        $result .= $this->subsystem1->operation1();
        $result .= $this->subsystem2->operation1();
        $result .= "Facade orders subsystems to perform the action:\n";
        $result .= $this->subsystem1->operationN();
        $result .= $this->subsystem2->operationZ();

        return $result;
    }
}

/**
 * يستطيع النظام الفرعي قبول الطلبات من الواجهة أو العميل مباشرة، وفي كل الحالات
 * فإن الواجهة بالنسبة للنظام الفرعي ما هي إلا عميل آخر، وليست جزءًا منه.
 */
class Subsystem1
{
    public function operation1(): string
    {
        return "Subsystem1: Ready!\n";
    }

    // ...

    public function operationN(): string
```

```
    {
        return "Subsystem1: Go!\n";
    }
}

/**
 * تستطيع بعض الواجهات أن تعمل مع عدة نظم فرعية في نفس الوقت.
 */
class Subsystem2
{
    public function operation1(): string
    {
        return "Subsystem2: Get ready!\n";
    }

    // ...

    public function operationZ(): string
    {
        return "Subsystem2: Fire!\n";
    }
}

/**
 * تعمل شيفرة العميل مع أنظمة معقدة من خلال واجهة بسيطة يوفرها نمط الواجهة.
 * وحين تدير واجهة دورة حياة نظام فرعي فإن العميل قد لا يعرف بوجود النظام الفرعي
 * أصلًا، وهذا يحافظ على مستوى التعقيد تحت السيطرة.
 */
function clientCode(Facade $facade)
```

```
{
    // ...

    echo $facade->operation();

    // ...
}

/**
 * قد تكون بعض كائنات النظام الفرعي منشأة بالفعل داخل شيفرة العميل، وفي تلك
 * الحالة، يفضل أن تُبدأ الواجهة بتلك الكائنات بدلًا من ترك الواجهة تنشئ كائنات جديدة.
 */
$subsystem1 = new Subsystem1;
$subsystem2 = new Subsystem2;
$facade = new Facade($subsystem1, $subsystem2);
clientCode($facade);
```