Fasada + Singleton

Tą lekcję uznajmy za lekcję bonusową. To nie będzie czyste i piękne rozwiązanie architektoniczne. Ale skoro mówiliśmy tutaj o fasadzie i singletonie, to warto byłoby pokazać wam, jak oba wzorce można połączyć... I w jaki sposób mógłby robić to Laravel.

Pamiętaj jednak, że tworzy to wysoki coupling w Twoim systemie. Nie chcesz tego.

Czasem jednak takie rozwiązanie może być uzasadnione, choćby z czystej ciekawości.

Zacznijmy od drobnej feraktoryzacji. Klasę Studio zamieńmy na “HardwareStudio”. Stwórzmy też interfejs, który realizować będzie zarówno fasada jak i konkretne implementacje. Nazwijmy ją po prostu Studio.

```
interface Studio
{
    public function on(): void;
    public function off(): void;
    public function setLightPower(int $power): void;
}

class HardwareStudio implements Studio { /* Implementation */
```

Teraz kolejnym krokiem, będzie użycie singletona do tego by użycie kodu

```
StudioFacade::on();
```

faktycznie dostarczyło nam takiej implementacji, jaką chcemy.

I w tym miejscu trzeba zrobić mały stop. Uzależniam swoją fasadę od zahardcodowanej standardowo klasy. W przypadku frameworków, mógłbym tutaj skorzystać np. z konfiguracji, w której tą klasę bym trzymał.

Ten odcinek jest jedynie ciekawostką. To co robimy teraz jest architektonicznie... średnie. W porywach do złego. Nie mniej jednak w przypadku faktycznego wdrożenia czegoś podobnego, zamiast podawania konkretnej klasy, powinniśmy pobrać ją z jakiegoś service containera czy czegoś w tym stylu.

Sama klasa fasady będzie pewnego rodzaju połączeniem, singletona, adaptera czy mostu. Z resztą, zaprogramujmy ją.

```
class StudioFacade
{
    private static Studio $studioHandler;

    private function __construct()
    {
    }

    public static function getInstance(): Studio
    {
        if(!isset(self::$studioHandler)) {
            // In real life this should be get from configura
            self::$studioHandler = new HardwareStudio(
                [
                    new LightPower('Listwa 220V'),
                    new LightPower('Kontrowe 220V'),
                    new WixLed()
                ],
                new CurtainRemote(),
                new WeatherStation()
            );
        }

        return self::$studioHandler;
    }

    public static function on(): void
    {
        self::getInstance()->on();
    }
```

```
    public static function off(): void
    {
        self::getInstance()->off();
    }

    public static function setLightPower(int $power): void
    {
        self::getInstance()->setLightPower($power);
    }
}
```

Zauważ co się tutaj dzieje.

Po pierwsze korzystamy z prywatnego konstruktora, by naszej fasady nikt nie mógł wykorzystać w “niewłaściwy” sposób.

Po drugie każda metoda zachowuje się trochę tak jak most, czy adapter. Posiada wszystkie metody które może wywołać, przy czym te metody są statyczne.

Jest tutaj jednak bardzo poważny problem, bo dane i klasy które przekazujemy przy instancji klasy to dane zahardcodowane. Nie powinniśmy robić tego w ten sposób. W normalnej sytuacji moglibyśmy skorzystać z bazy danych, pliku konfiguracyjnego itd.

Co nam to dało? Naszą klasę faktycznie możemy teraz wywołać tak:

```
StudioFacade::on();
StudioFacade::setLightPower(10);
```

Elegancko, nie? No - może dla kogoś kto ma do napisania tylko te 2 linijki. Dla całego systemu, to co się tutaj stało naprawdę nie jest za dobre. Fasada studia stała się boskim obiektem, możemy korzystać z niej gdzie chcemy, ale nie za bardzo mamy nad nią kontrolę.

Dlatego właśnie uważam, że podejście tego typu nie jest najlepsze i Laravel, mimo że jest moim ulubionym frameworkiem - ma swoje poważne wady i problemy.

No ale nasz cel nie jest jeszcze do końca osiągnięty. Co jeśli chcielibyśmy móc tą klasę przetestować? Czyli sprawdzić, czy jeśli studio zostało włączone, to

faktycznie się to wydarzyło?

Do tego użyjemy kolejnej metody - fake, która jednocześnie zmieni obiekt singletona na klasę anonimową.

```
class StudioFacade
{
    /** Tutaj poprzednia implementacja */

    public static function fake(): void
    {
        self::$studioHandler = new class implements Studio {
            public bool $on = false;
            public int $power = 0;

            public function on(): void
            {
                echo "Fake on\n";
                $this->on = true;

            }

            public function off(): void
            {
                echo "Fake off\n";
                $this->on = false;
            }

            public function setLightPower(int $power): void
            {
                echo "Fake setLightPower\n";
                $this->power = $power;
            }
        };
    }

    public static function assertLightOn(bool $on): void
    {
        if((self::$studioHandler->on ?? null) === null) {
```

```
            throw new Exception('You need to call fake method
        }

        assert(
            self::$studioHandler->on === $on,
            'Asserting that current '.
            (self::$studioHandler->on ? 'on' : 'off').
            ' light is not '.
            ($on ? 'on' : 'off')
        );
    }

    public static function assertLightPower(int $power): void
    {
        if((self::$studioHandler->power ?? null) === null) {
            throw new Exception('You need to call fake method
        }

        assert(
            self::$studioHandler->power === $power,
            'Asserting that current power '. (self::$studioHa
            ' has '. ($power) . ' volume'
        );
    }
}
```

W klasie poza metodą fake pojawiły się również dwa asserty. I teraz, to co możemy zrobić, to napisać coś w rodzaju testu:

```
StudioFacade::fake();
StudioFacade::on();
StudioFacade::setLightPower(10);

StudioFacade::assertLightOn(true);
StudioFacade::assertLightPower(30);
```

Który da nam jako rezultat:

```
Fake on
Fake setLightPower

bool(true)
PHP Fatal error:  Uncaught AssertionError: Asserting that cur
```

Ciekawe? Oczywiście jest tutaj dużo więcej możliwości, takich jak na przykład inne ustawienia studia, czyli możliwość zmiany instancji klasy rzeczywistej, jak Laravel robi to np. za pomocą disk(), by zmienić określony sterownik.

Mimo jednak że finalnie lądujemy z rozwiązaniem które jest zarówno testowalne, jak i całkiem ładne w “użyciu” to pod spodem kryje się mały potworek.

Dlatego właśnie zawsze musisz uważać i myśleć co i jak wdrażasz. Czy narzut w stylu “potrzebujemy określonych instancji” jest duży? To oczywiście zależy, natomiast jak widzisz, za rozwiązaniami, które na zewnątrz mogą wyglądać elegancko, czasem idzie niezły potworek.