Mastering OOP and design patterns

SensioLabs
Créateur de Symfony

Program

Introduction

Introduction

What defines “good” code?

Introduction

- Code that produces the desired effect
- Code secured and stable
- Code easily extendable
- Code easily adaptable
- Code easily testable

Introduction

Why would you need Object Oriented Programming?

Introduction

Object-Oriented programming aims to allow for a better code reusability, modularity, and testing.

This in turn give object oriented code an easier maintainability with greater possibilities of extension or change.

Object Oriented Design Generics

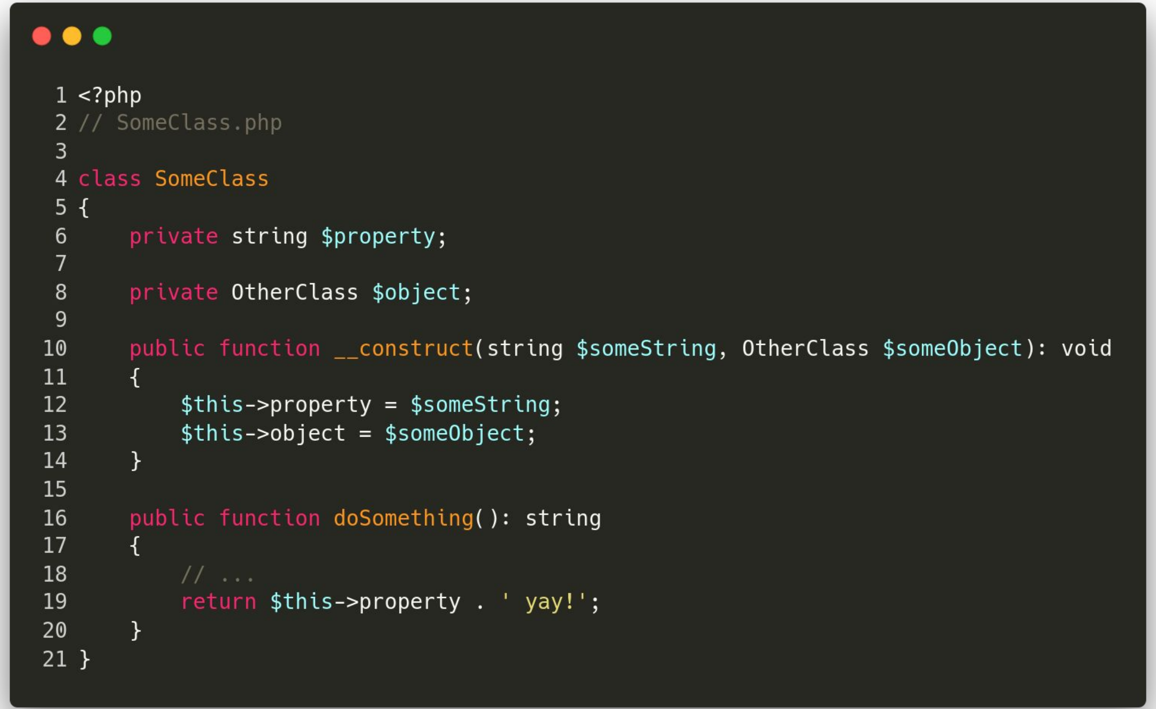
General Principles

- **Objects and classes :** Write classes used as templates to *instantiate* objects to work with. Classes are the blueprint, objects the construction.
- **Encapsulation :** Objects used as black boxes. Public interfaces are known, not internal workings
- **Composition and Inheritance :** Objects can contain other objects, and classes may be organized in a hierarchy or inheritance

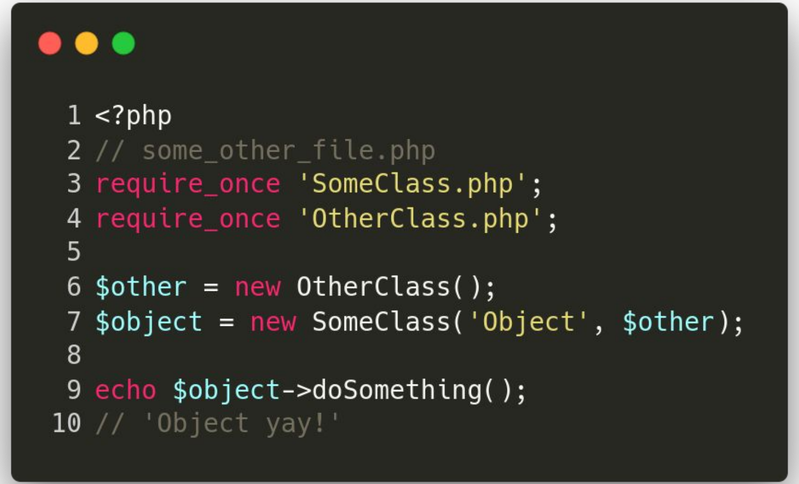
General Principles

- **Polymorphism :** An object can be replaced by any of its descendants typewise. Typing an abstract class allows you to pass any of the children classes
- **Strong typing :** Classes can be used as argument and return types in functions, allowing for safer code

General Principles


```php
<?php
// SomeClass.php

class SomeClass
{
    private string $property;

    private OtherClass $object;

    public function __construct(string $someString, OtherClass $someObject): void
    {
        $this->property = $someString;
        $this->object = $someObject;
    }

    public function doSomething(): string
    {
        // ...
        return $this->property . ' yay!';
    }
}
```

General Principles



```php
<?php
// some_other_file.php
require_once 'SomeClass.php';
require_once 'OtherClass.php';

$other = new OtherClass();
$object = new SomeClass('Object', $other);

echo $object->doSomething();
// 'Object yay!'
```

SensioLabs

SOLID Principles and derivatives

SOLID Principles

SOLID is a mnemonic acronym designing five ideas, or principles, relating to object oriented architecture.

They aim to make object oriented code more understandable, more flexible, and more maintainable.

SOLID Principles

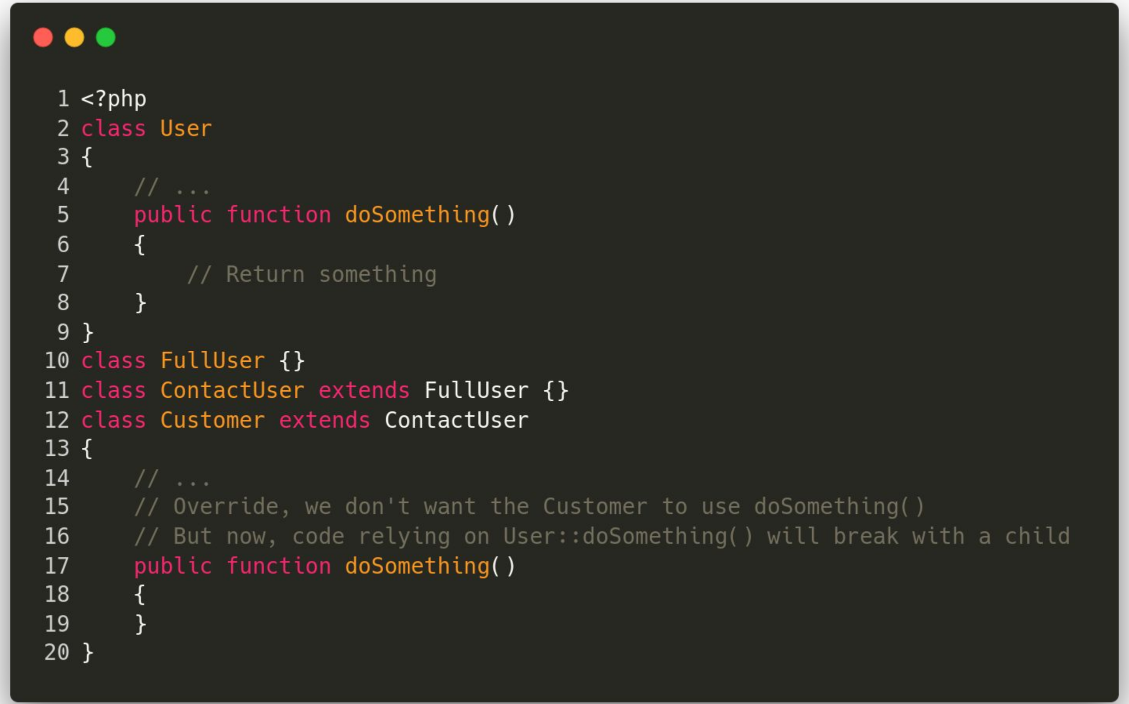
- **Single Responsibility:** One class, one responsibility
- **Open-closed:** Objects can be extended, not modified
- **Liskov substitution:** When typing your data with base classes, you should be able able to use derivatives
- **Interface segregation:** Many specific interfaces over one put-it-all
- **Dependency Inversion:** Depend on abstractions, not concretions

Composition over Inheritance

When overusing inheritance, you can come to have classes needing to override their parents' method, breaking **Liskov Substitution principle** (the child will change the expected behavior).

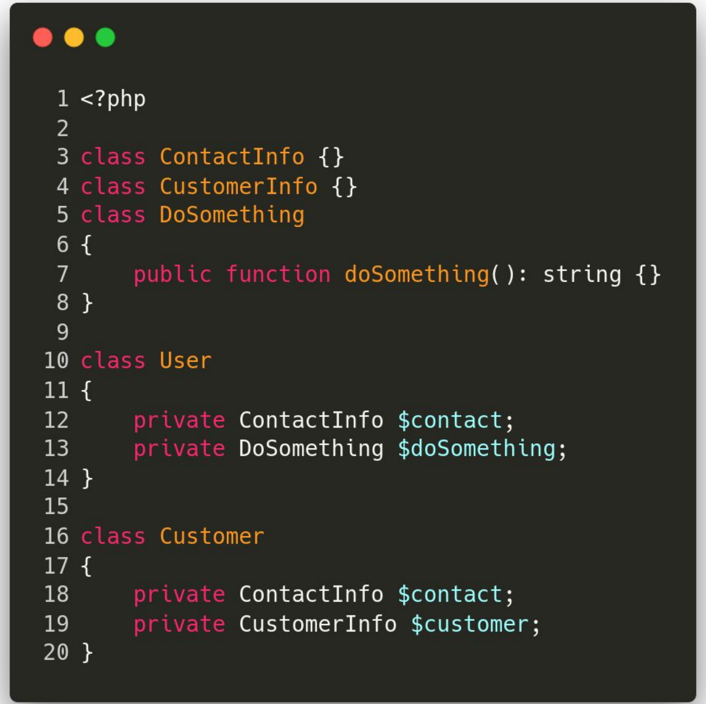
To avoid this, always prefer composition over inheritance. Segregate your interfaces and break big classes into smaller objects.

Composition over Inheritance



```php
<?php
class User
{
    // ...
    public function doSomething()
    {
        // Return something
    }
}
class FullUser {}
class ContactUser extends FullUser {}
class Customer extends ContactUser
{
    // ...
    // Override, we don't want the Customer to use doSomething()
    // But now, code relying on User::doSomething() will break with a child
    public function doSomething()
    {
    }
}
```

Composition over Inheritance



```php
<?php

class ContactInfo {}
class CustomerInfo {}
class DoSomething
{
    public function doSomething(): string {}
}

class User
{
    private ContactInfo $contact;
    private DoSomething $doSomething;
}

class Customer
{
    private ContactInfo $contact;
    private CustomerInfo $customer;
}
```

Inversion of Control

In software engineering, inversion of control (IoC) is a programming principle. IoC inverts the flow of control as compared to traditional control flow. In IoC, custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the framework that calls into the custom, or task-specific, code.

https://en.wikipedia.org/wiki/Inversion_of_control

Inversion of Control

With IoC, instead of having your custom application call for specific parts of a framework, the frameworks itself calls and controls the flow of your application.

Practically speaking, the framework handles and abstracts most of the heavy lifting of running your specific code in the proper context, injecting every time the required dependencies.

Exercise

Find an example of framework using Inversion of Control

Dependency Injection

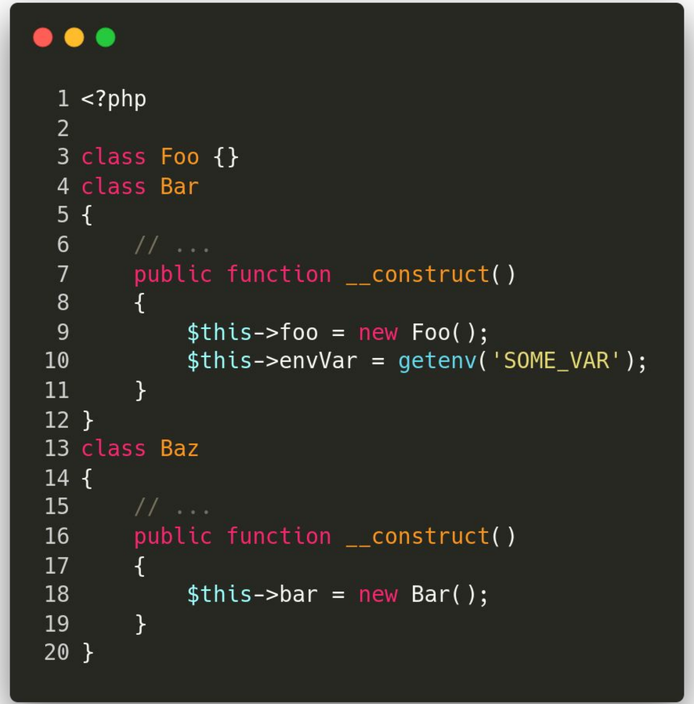
In software engineering, dependency injection is a design pattern in which an object receives other objects that it depends on. A form of inversion of control, dependency injection aims to separate the concerns of constructing objects and using them. The pattern ensures that an object which wants to use a given service should not have to know how to construct those services. Instead, the receiving object (or 'client') is provided with its dependencies by external code (an 'injector'), which it is not aware of. The service is made part of the client's state, available for use. Because the client does not build or find the service itself, it typically only needs to declare the interfaces of the services it uses, rather than their concrete implementations.

https://en.wikipedia.org/wiki/Dependency_injection

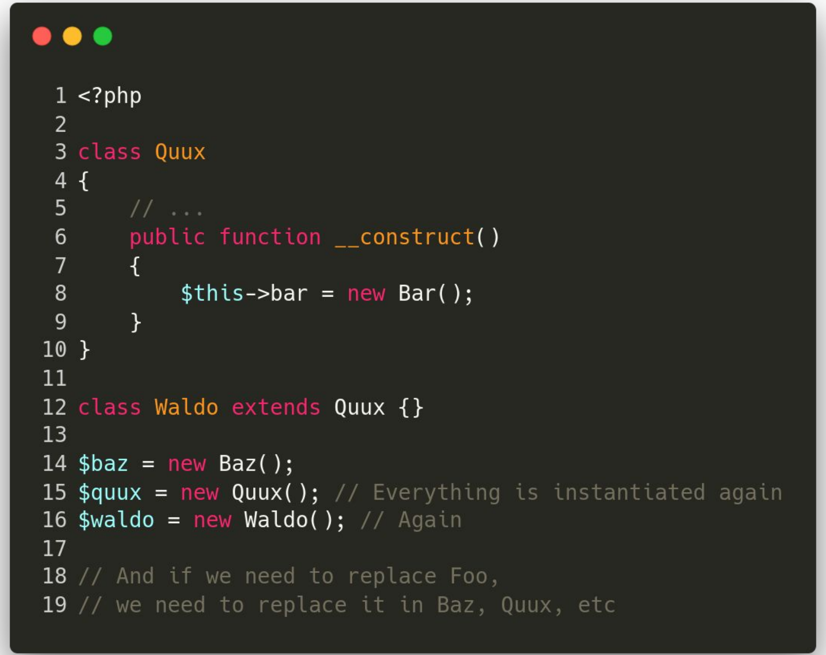
Dependency Injection

- The most visible implementation of the Inversion of Control principle in Symfony.
- In Dependency Injection, dependencies between objects are not statically written in the code but dynamically injected based on a configuration file, scanned metadatas, or both.
- As the client doesn’t instantiates the services it needs, it can rely on interfaces rather than concrete implementations

Dependency Injection



```php
<?php

class Foo {}
class Bar
{
    // ...
    public function __construct()
    {
        $this->foo = new Foo();
        $this->envVar = getenv('SOME_VAR');
    }
}
class Baz
{
    // ...
    public function __construct()
    {
        $this->bar = new Bar();
    }
}
```



```php
<?php

class Quux
{
    // ...
    public function __construct()
    {
        $this->bar = new Bar();
    }
}

class Waldo extends Quux {}

$baz = new Baz();
$quux = new Quux(); // Everything is instantiated again
$waldo = new Waldo(); // Again

// And if we need to replace Foo,
// we need to replace it in Baz, Quux, etc
```

Exercise

1. **Rework the `App\Provider\MovieProvider`:**
   a. Inject dependencies
   b. It should check first if a movie exists in database, or else fetch it from the API
2. **Rework the `App\Controller\MovieController` to inject dependencies**

Objects Calisthenics

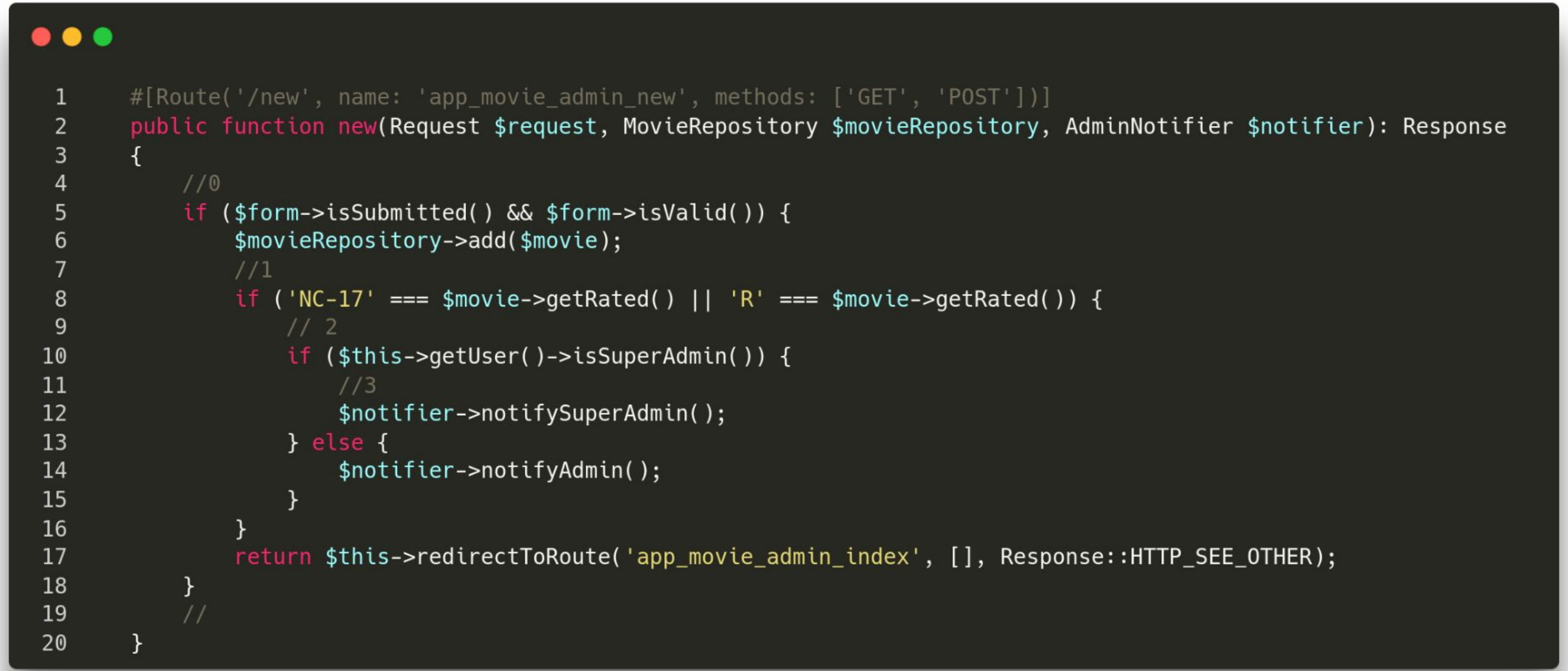
Object Calisthenics

- Invented by Jeff Bay in *The ThoughtWorks Anthology*
- Set of 9 rules focused on maintainability, readability, testability, and comprehensibility
- Named after the workout method invented in the XIXth century
- Not an absolute “follow or don’t be part of the club” list
- Try to implement as much as possible to improve your code

Object Calisthenics

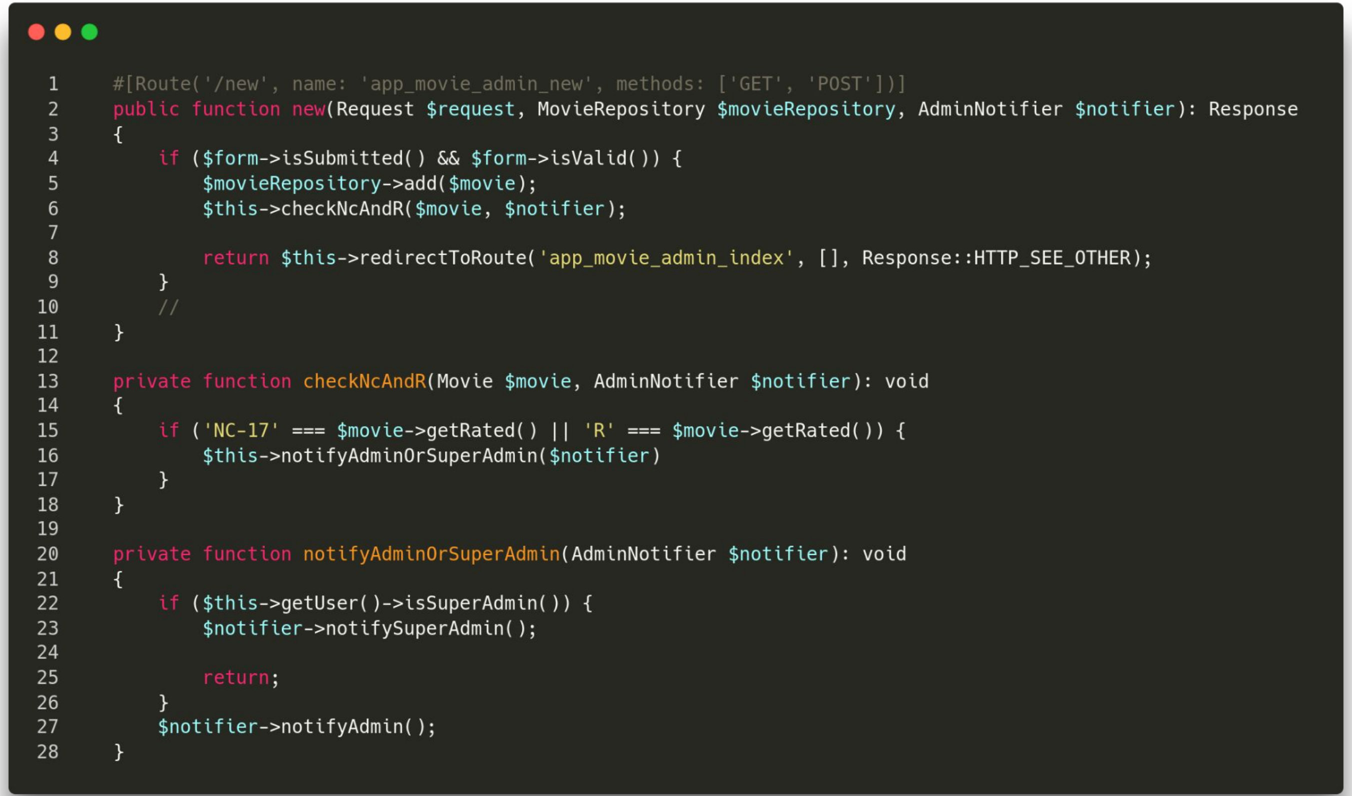
1. Only one level of indentation per method
2. Don’t use the ELSE keyword
3. Wrap all primitives and strings
4. First class collections
5. One dot (arrow) per line
6. Don’t abbreviate
7. Keep all entities small
8. No classes with more than two instance variables
9. Avoid getters and setters

1- One level of indentation per method



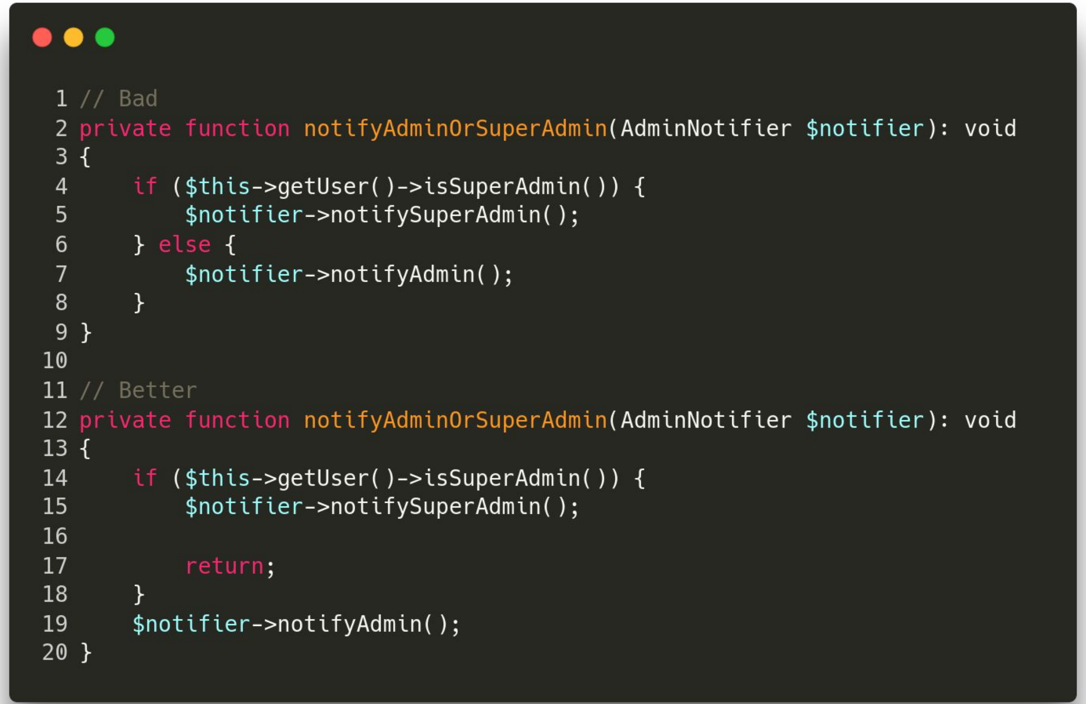
```php
#[Route('/new', name: 'app_movie_admin_new', methods: ['GET', 'POST'])]
public function new(Request $request, MovieRepository $movieRepository, AdminNotifier $notifier): Response
{
    //0
    if ($form->isSubmitted() && $form->isValid()) {
        $movieRepository->add($movie);
        //1
        if ('NC-17' === $movie->getRated() || 'R' === $movie->getRated()) {
            // 2
            if ($this->getUser()->isSuperAdmin()) {
                //3
                $notifier->notifySuperAdmin();
            } else {
                $notifier->notifyAdmin();
            }
        }
        return $this->redirectToRoute('app_movie_admin_index', [], Response::HTTP_SEE_OTHER);
    }
    //
}
```

1- One level of indentation per method



```php
#[Route('/new', name: 'app_movie_admin_new', methods: ['GET', 'POST'])]
public function new(Request $request, MovieRepository $movieRepository, AdminNotifier $notifier): Response
{
    if ($form->isSubmitted() && $form->isValid()) {
        $movieRepository->add($movie);
        $this->checkNcAndR($movie, $notifier);

        return $this->redirectToRoute('app_movie_admin_index', [], Response::HTTP_SEE_OTHER);
    }
    //
}

private function checkNcAndR(Movie $movie, AdminNotifier $notifier): void
{
    if ('NC-17' === $movie->getRated() || 'R' === $movie->getRated()) {
        $this->notifyAdminOrSuperAdmin($notifier)
    }
}

private function notifyAdminOrSuperAdmin(AdminNotifier $notifier): void
{
    if ($this->getUser()->isSuperAdmin()) {
        $notifier->notifySuperAdmin();

        return;
    }
    $notifier->notifyAdmin();
}
```

2- Don’t use the ELSE keyword



```php
// Bad
private function notifyAdminOrSuperAdmin(AdminNotifier $notifier): void
{
    if ($this->getUser()->isSuperAdmin()) {
        $notifier->notifySuperAdmin();
    } else {
        $notifier->notifyAdmin();
    }
}

// Better
private function notifyAdminOrSuperAdmin(AdminNotifier $notifier): void
{
    if ($this->getUser()->isSuperAdmin()) {
        $notifier->notifySuperAdmin();

        return;
    }
    $notifier->notifyAdmin();
}
```

3- Wrap primitive types and strings

- Promotes creating ValueObjects to contain scalars
- Hardly fully applicable to PHP
- Do create ValueObject if the variable has a behavior or needs to be validated

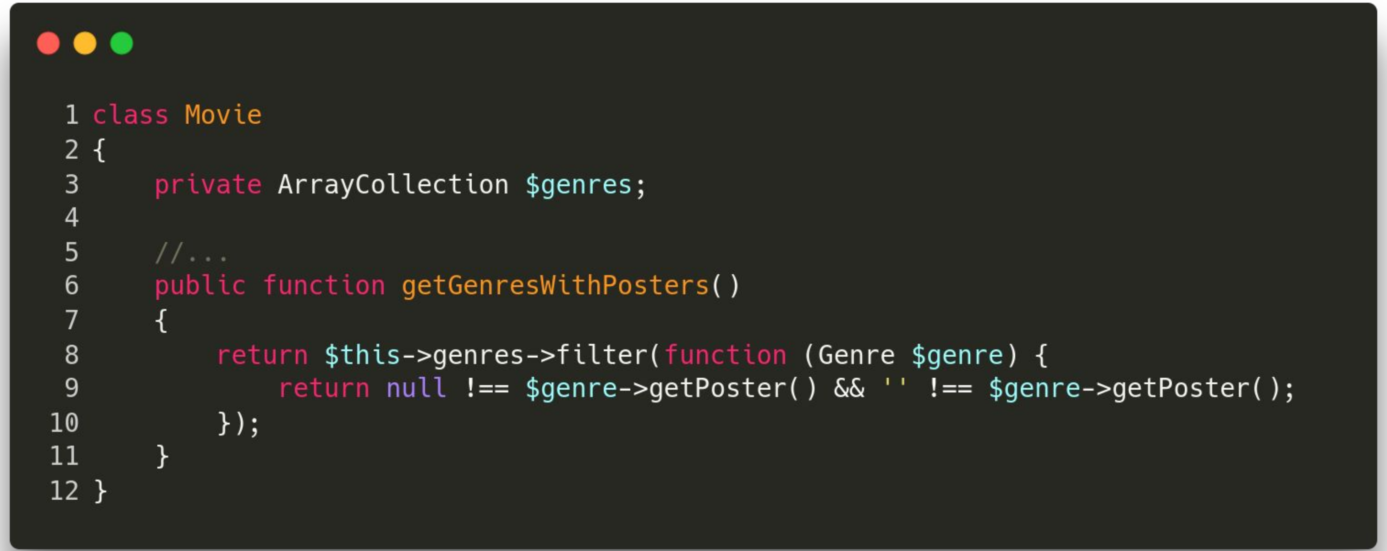
4- First Class collections

- Every collection (Doctrine collection or array) gets its own class
- You can simply extend ArrayCollection for example
- Specific behavior (ex: specific filtering) for this collection gets its natural home

4- First Class collections

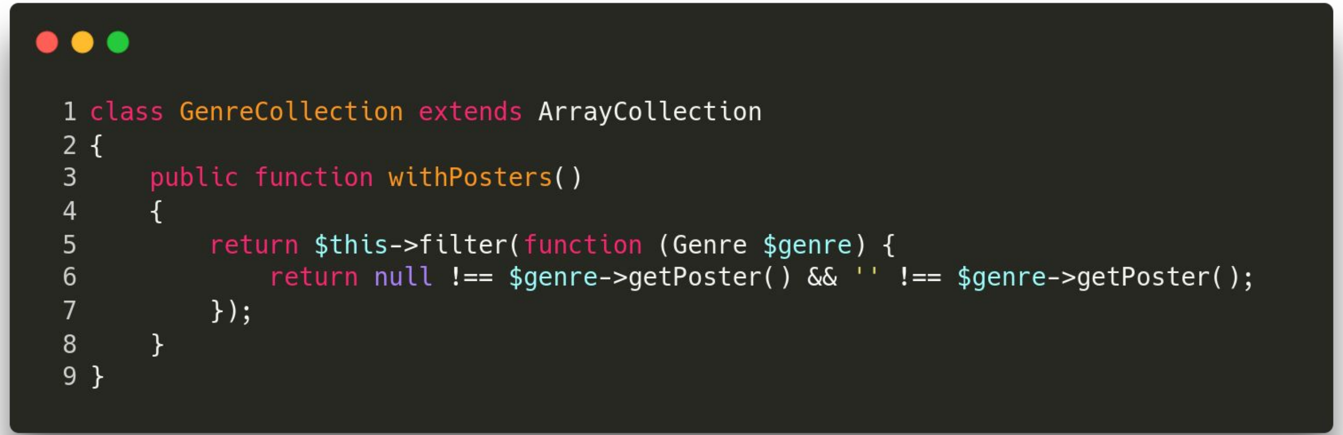
- Every collection (Doctrine collection or array) gets its own class
- You can simply extend ArrayCollection for example
- Specific behavior (ex: specific filtering) for this collection gets its natural home

4- First Class collections



```php
class Movie
{
    private ArrayCollection $genres;

    //...
    public function getGenresWithPosters()
    {
        return $this->genres->filter(function (Genre $genre) {
            return null !== $genre->getPoster() && '' !== $genre->getPoster();
        });
    }
}
```

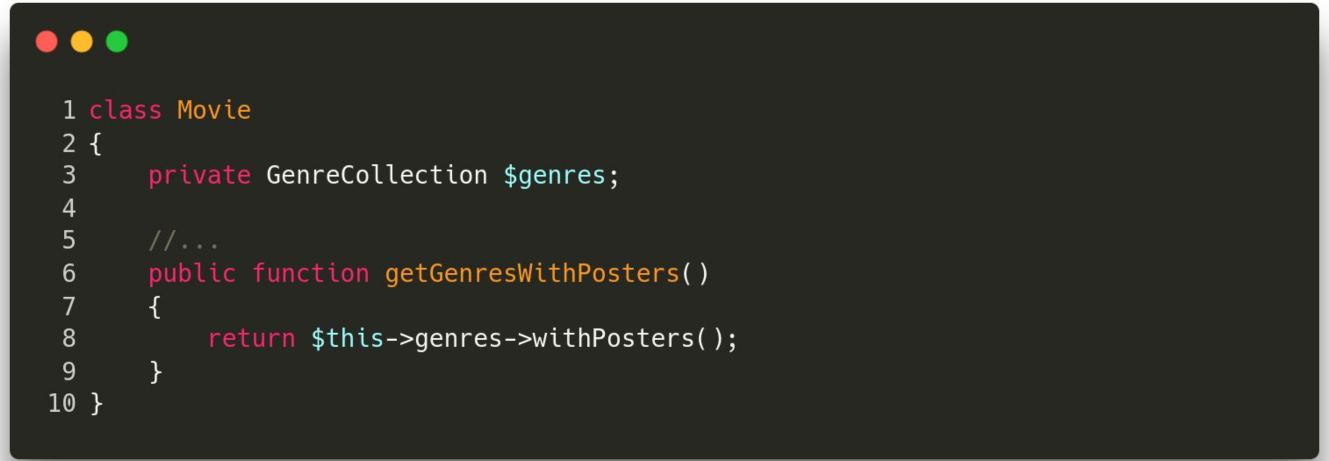
4- First Class collections



```php
class GenreCollection extends ArrayCollection
{
    public function withPosters()
    {
        return $this->filter(function (Genre $genre) {
            return null !== $genre->getPoster() && '' !== $genre->getPoster();
        });
    }
}
```

SensioLabs

4- First Class collections

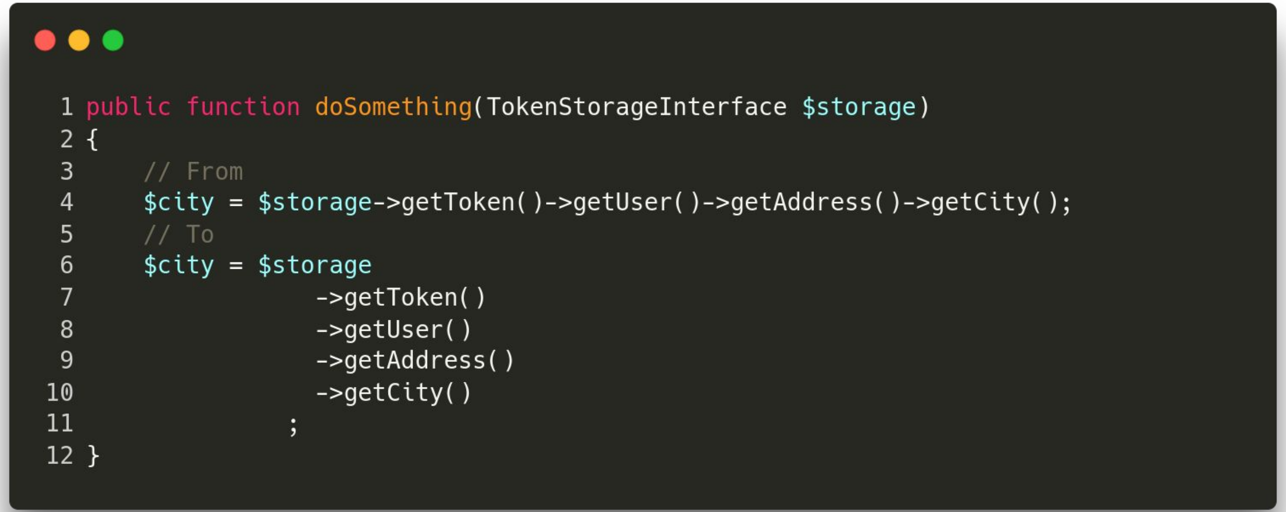


```php
class Movie
{
    private GenreCollection $genres;

    //...
    public function getGenresWithPosters()
    {
        return $this->genres->withPosters();
    }
}
```

5- One dot (arrow) per line

- Basically meant as “don’t chain calls”
- Doesn’t apply to fluent interfaces and Method Chaining Pattern (see later)
- Hardly applicable to languages like PHP
- Still best to limit, at least with styling, to one arrow call per line for readability

5- One dot (arrow) per line



```
public function doSomething(TokenStorageInterface $storage)
{
    // From
    $city = $storage->getToken()->getUser()->getAddress()->getCity();
    // To
    $city = $storage
                ->getToken()
                ->getUser()
                ->getAddress()
                ->getCity()
              ;
}
```

6- Don’t abbreviate

- Rationale: it’s often a code smell, a sign of a larger problem, and you should ask “Why do I want to abbreviate?”
- Writing the same code again and again? May be code duplication
- Name too long? May have too much responsibilities for this class/function

7- Keep all entities small

- 5-10 lines per method
- 100-150 lines per class
- 15 classes per namespace

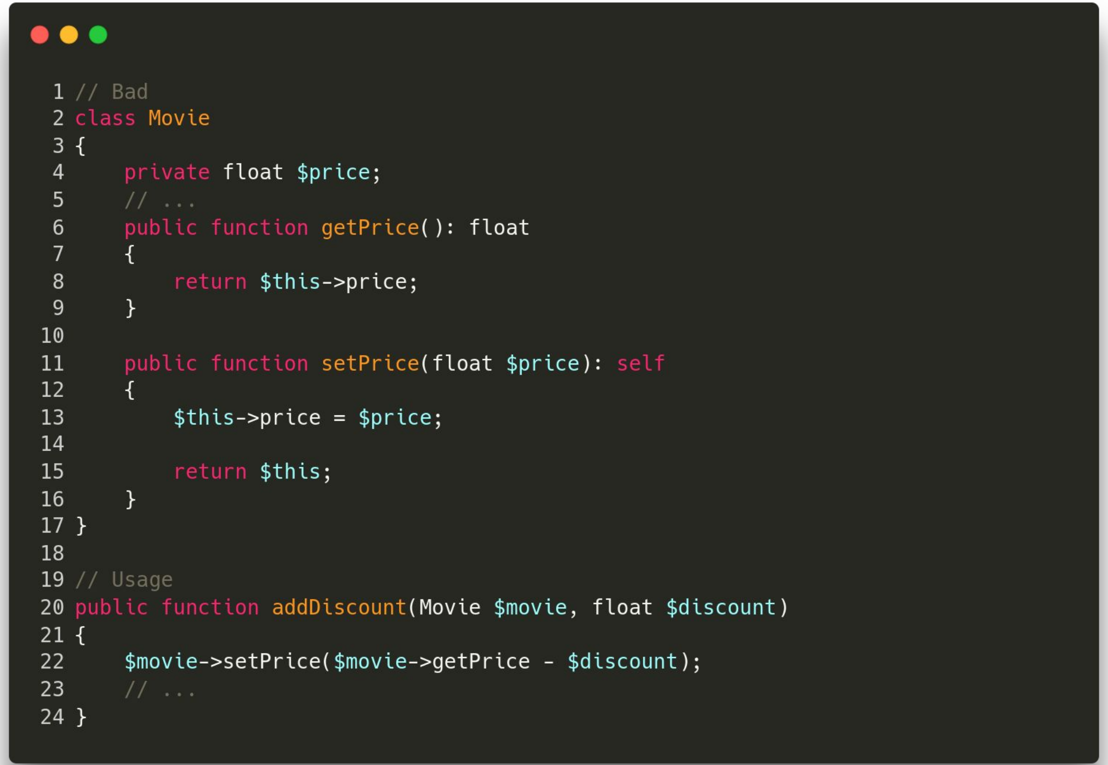
8- No classes with more than two instance variables

- More “Keep number of instance properties low”
- Relies heavily on rule 3 (Wrap primitive types and strings)
- Practically not applicable to Doctrine entities
- Otherwise applicable to ValueObjects, Models, and anything else
- Good rule for utilitarian classes and services
- Too many properties = too many responsibilities

9- Avoid getters and setters

- Alias “Tell, don’t ask”
- Avoid anemic models and create meaningful methods
- To not use getters to make decisions outside the object
- Instead, decision making based solely on a state should happen inside the object

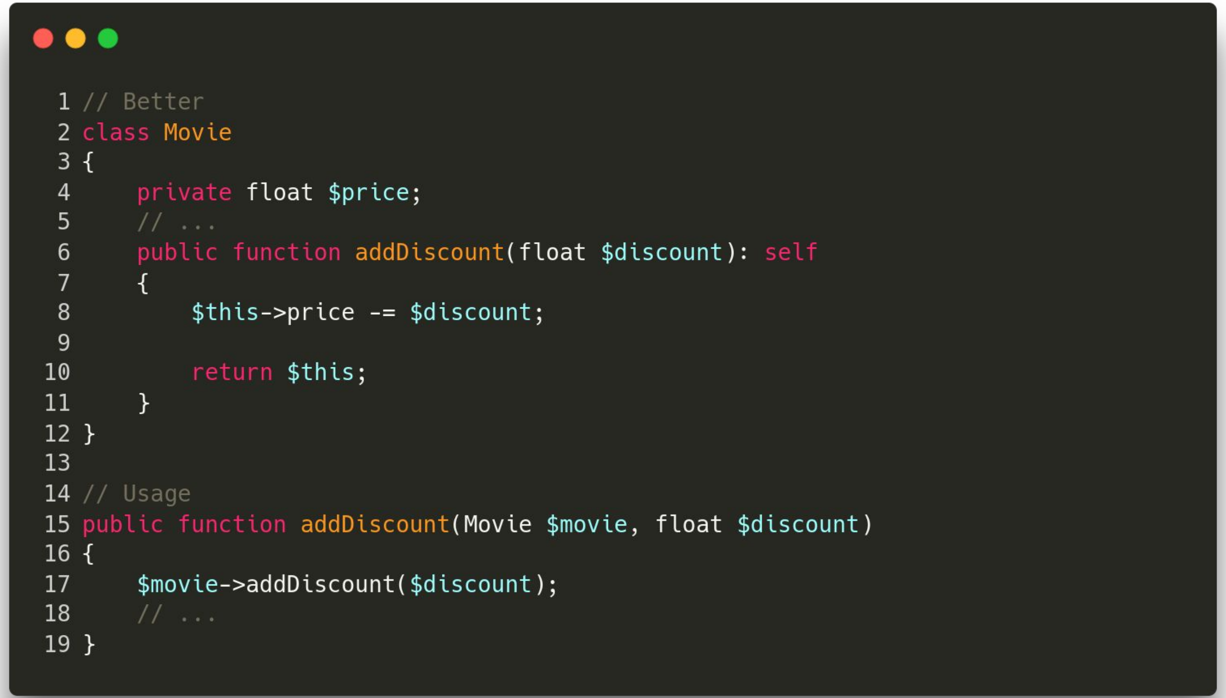
9- Avoid getters and setters



```php
// Bad
class Movie
{
    private float $price;
    // ...
    public function getPrice(): float
    {
        return $this->price;
    }

    public function setPrice(float $price): self
    {
        $this->price = $price;

        return $this;
    }
}

// Usage
public function addDiscount(Movie $movie, float $discount)
{
    $movie->setPrice($movie->getPrice - $discount);
    // ...
}
```

SensioLabs

9- Avoid getters and setters



```php
// Better
class Movie
{
    private float $price;
    // ...
    public function addDiscount(float $discount): self
    {
        $this->price -= $discount;

        return $this;
    }
}

// Usage
public function addDiscount(Movie $movie, float $discount)
{
    $movie->addDiscount($discount);
    // ...
}
```

SensioLabs

Introduction to Design Patterns

Introduction to Design Patterns

- Re-usable form of solutions to commonly occurring design problem depending on context
- Templates of best practices used to solve problems, not absolute recipes
- **Absolutely not a finished design directly translatable into source code**
- **Absolutely tools to be used or not used carefully**

Types of Design Patterns

Creational Design Patterns:

- Provide the capability to create objects based on a criteria in a controlled way

Types of Design Patterns

Structural Design Patterns:

- Provide means to organize classes in large structures and provide new functionalities

Types of Design Patterns

Behavioral Design Patterns:

- Provide common means of communication between objects

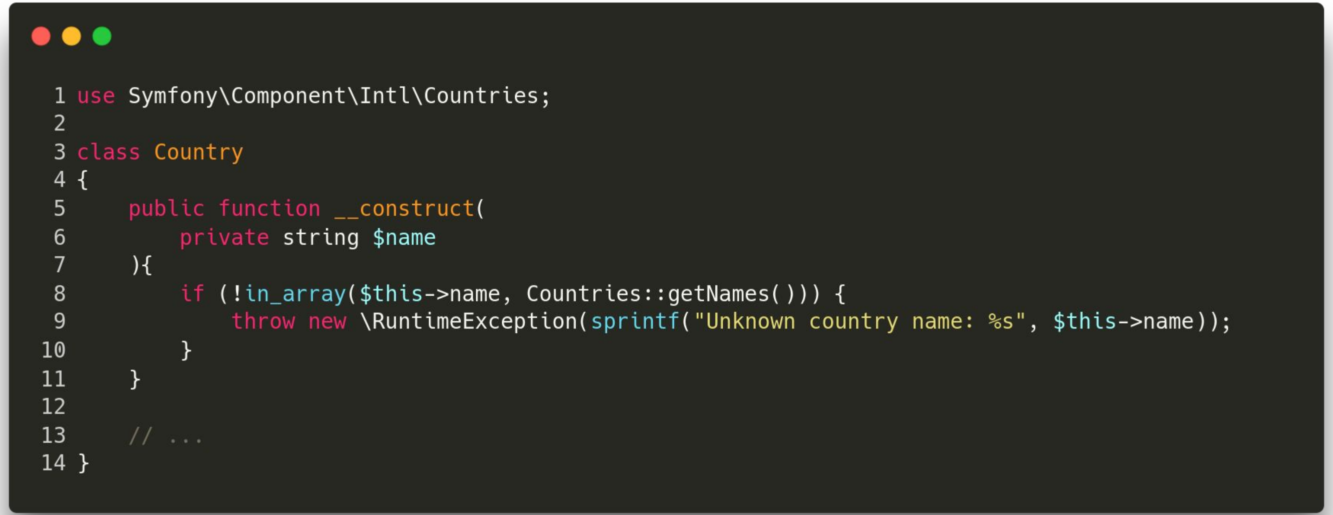
Value Objects

Interlude: Value Objects

Value Objects

- Perfect example Calisthenic 3
- Don’t have an identity
- Validate their one data
- Immutable and always valid by design
- Interchangeable without side effect, their equality is based on underlying data
- Perfect for passing data in most design patterns

Value Objects



```php
use Symfony\Component\Intl\Countries;

class Country
{
    public function __construct(
        private string $name
    ){
        if (!in_array($this->name, Countries::getNames())) {
            throw new \RuntimeException(sprintf("Unknown country name: %s", $this->name));
        }
    }

    // ...
}
```

SensioLabs

Creational Design Patterns

Creational Design Patterns

Creational design patterns deal with object creation.

They are based around two different concepts:

- Encapsulating knowledge about classes used by the system
- Hiding and centralizing knowledge about how these objects are created

Creational Design Patterns

Some Creational design patterns:

- Factory
- Abstract Factory
- Builder
- Method
- Prototype
- Singleton

Singleton

Singleton

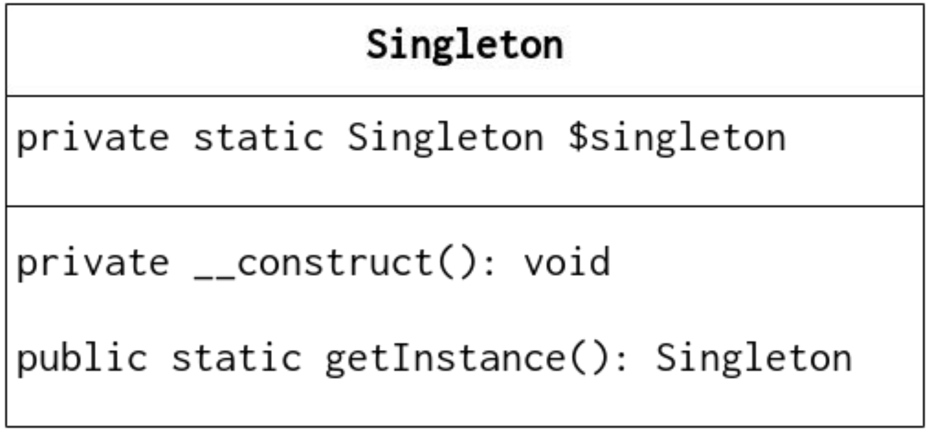
The Singleton design pattern aims to restrict the instantiation of a given class to one single instance.

This is useful when there can be only one object of the same type during a given execution (like a database connection), and/or when an object needs to coordinate actions across the system

Exercise

Create a class `App\Singleton\Singleton` that can be instantiated only once

Singleton



```
Singleton
---------
private static Singleton $singleton
---------
private __construct(): void

public static getInstance(): Singleton
```

Prototype

Prototype

The Prototype design pattern is used to create a new object by copying or cloning an existing template, or prototype.

It is particularly useful when the creation and initialization of a new instance would be expensive and/or complex.

Prototype

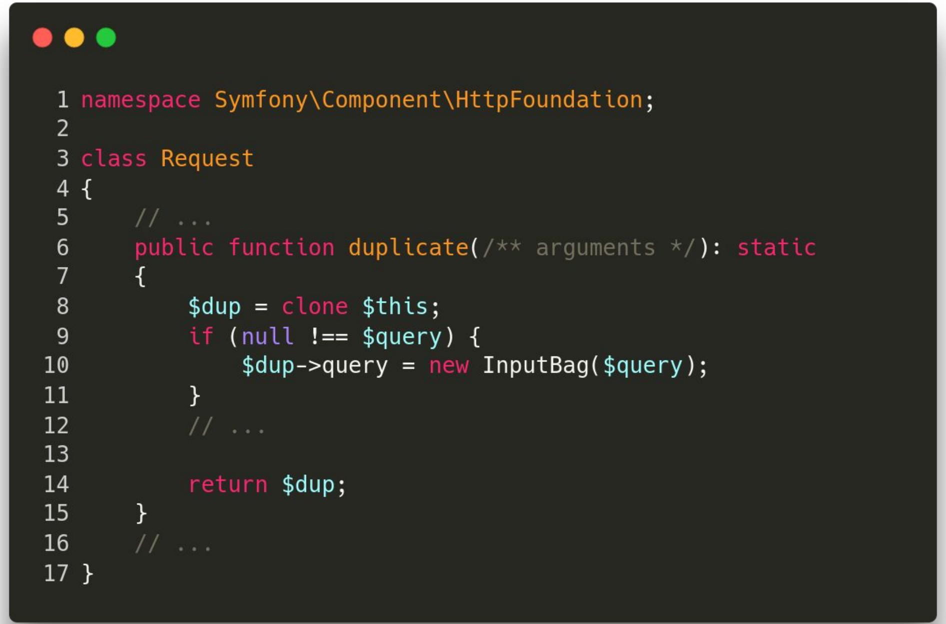
Benefits:

- Eliminates the need for factories or subclassing
- Reduces initialization code duplication
- Create objects faster

Disadvantages:

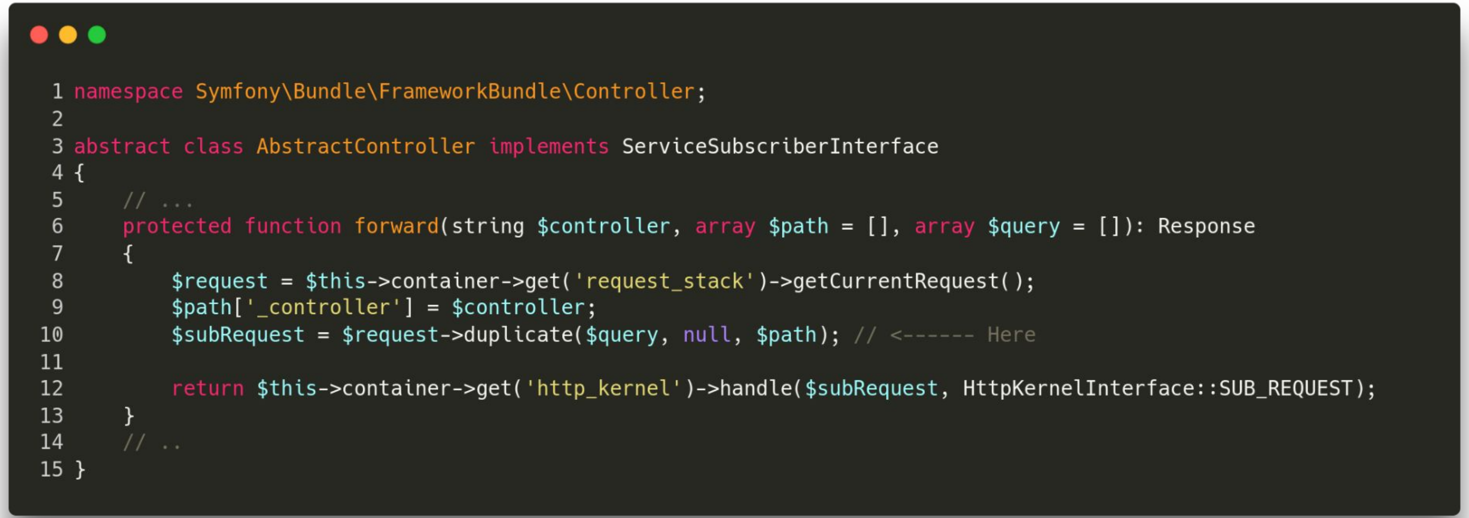
- Cloning deep nested and complex objects can be hard and costly

Prototype



```php
namespace Symfony\Component\HttpFoundation;

class Request
{
    // ...
    public function duplicate(/** arguments */): static
    {
        $dup = clone $this;
        if (null !== $query) {
            $dup->query = new InputBag($query);
        }
        // ...

        return $dup;
    }
    // ...
}
```

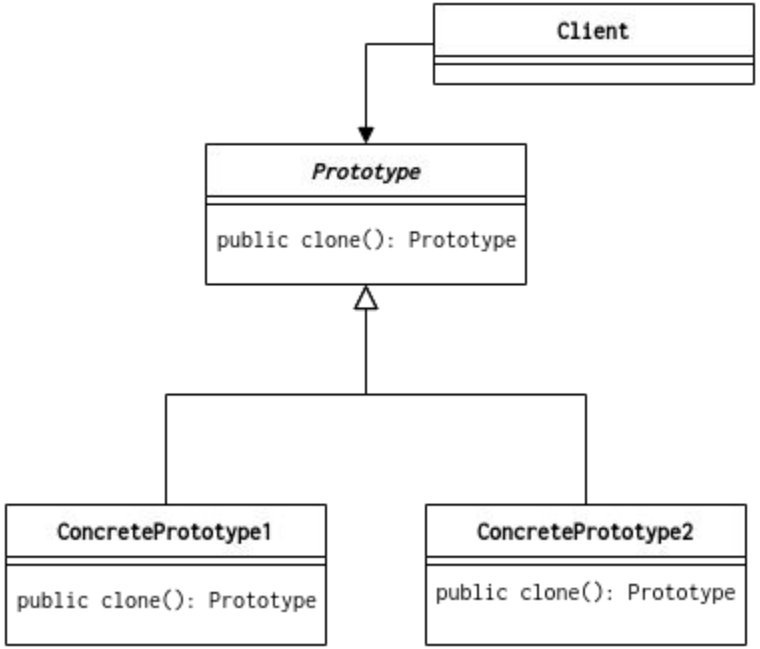
Prototype



```php
namespace Symfony\Bundle\FrameworkBundle\Controller;

abstract class AbstractController implements ServiceSubscriberInterface
{
    // ...
    protected function forward(string $controller, array $path = [], array $query = []): Response
    {
        $request = $this->container->get('request_stack')->getCurrentRequest();
        $path['_controller'] = $controller;
        $subRequest = $request->duplicate($query, null, $path); // <------ Here

        return $this->container->get('http_kernel')->handle($subRequest, HttpKernelInterface::SUB_REQUEST);
    }
    // ..
}
```

SensioLabs

Prototype


Client
Prototype
public clone(): Prototype
ConcretePrototype1
public clone(): Prototype
ConcretePrototype2
public clone(): Prototype

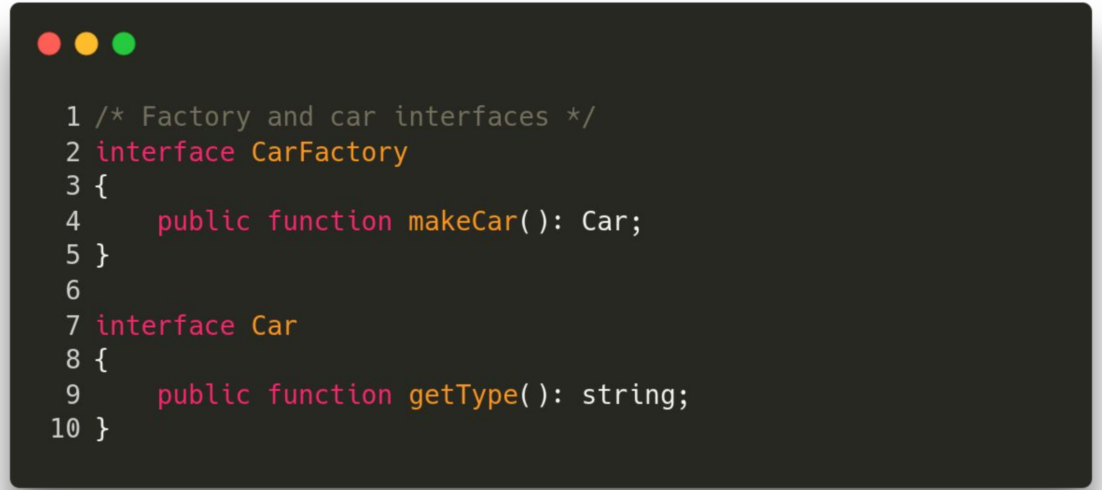
Factory Method

Factory Method

The Factory Method design pattern deals with the problem of object creation without specifying the actual class of the object that will be created.

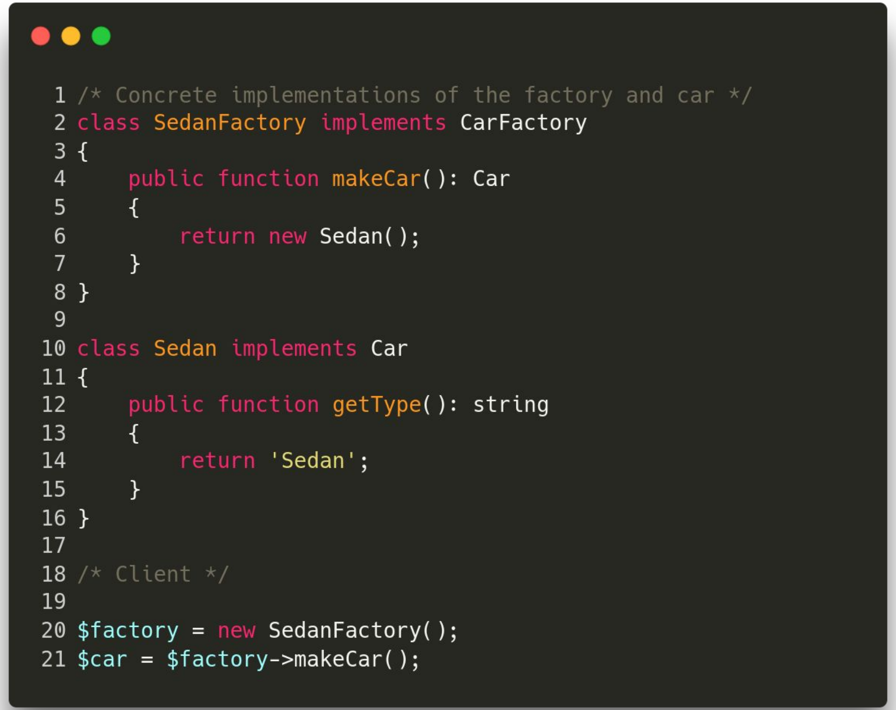
This is usually done by calling a factory method which is either specified in an interface and implemented in child classes or defined in a base class and overridden by child classes.

Factory Method

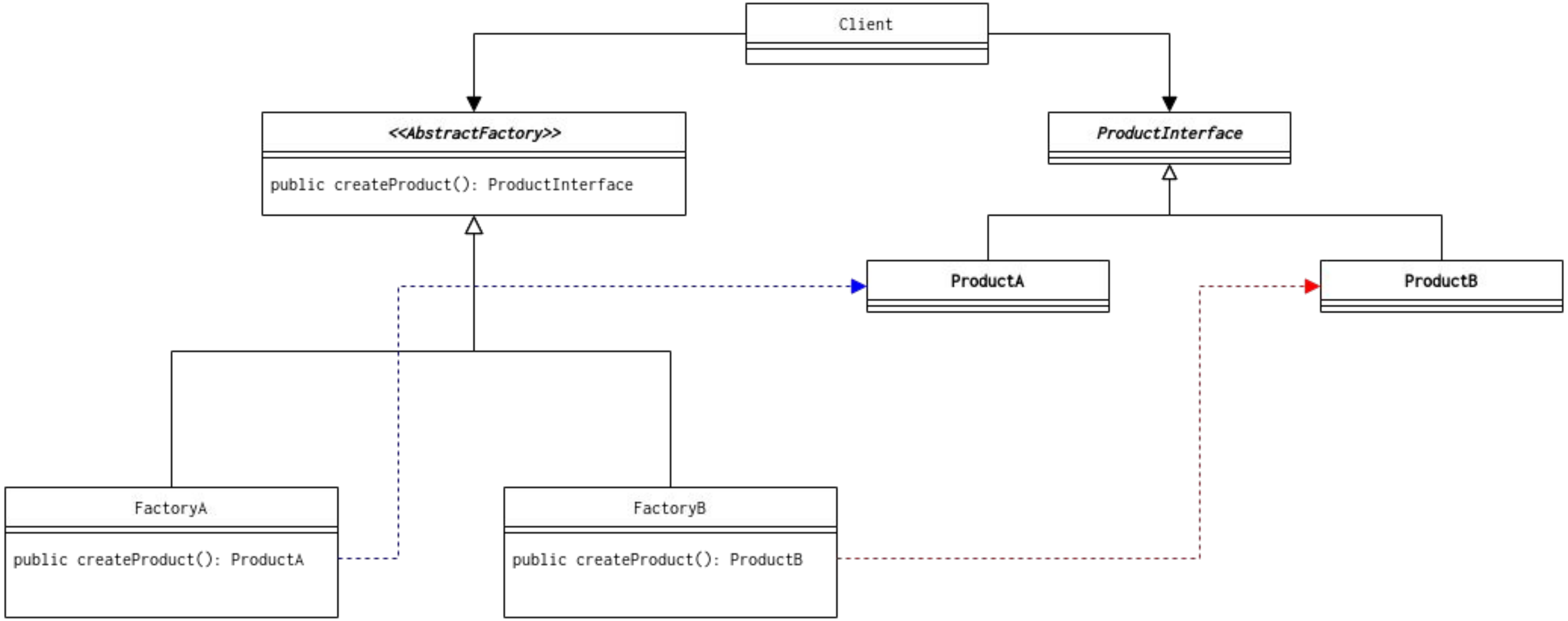


```php
/* Factory and car interfaces */
interface CarFactory
{
    public function makeCar(): Car;
}

interface Car
{
    public function getType(): string;
}
```

Factory Method


```php
/* Concrete implementations of the factory and car */
class SedanFactory implements CarFactory
{
    public function makeCar(): Car
    {
        return new Sedan();
    }
}

class Sedan implements Car
{
    public function getType(): string
    {
        return 'Sedan';
    }
}

/* Client */

$factory = new SedanFactory();
$car = $factory->makeCar();
```

Factory Method


Client
<<AbstractFactory>>
public createProduct(): ProductInterface
ProductInterface
ProductA
ProductB
FactoryA
public createProduct(): ProductA
FactoryB
public createProduct(): ProductB

Factory Method

Benefits:

- Each factory produces one specific concrete type
- Each factory can be replaced easily by another
- Adaptability to the runtime environment
- Object construction is centralized

Disadvantages:

- Lots of classes and interfaces
- Client code doesn’t know the exact type
- Hard to implement

Exercise

Check the `App\Notifier\` directory:

- Create a `Factory` folder
- Create a `NotificationFactoryInterface`
- Create factories for every notification type
- Check your type hints ;)

Abstract Factory

Abstract Factory

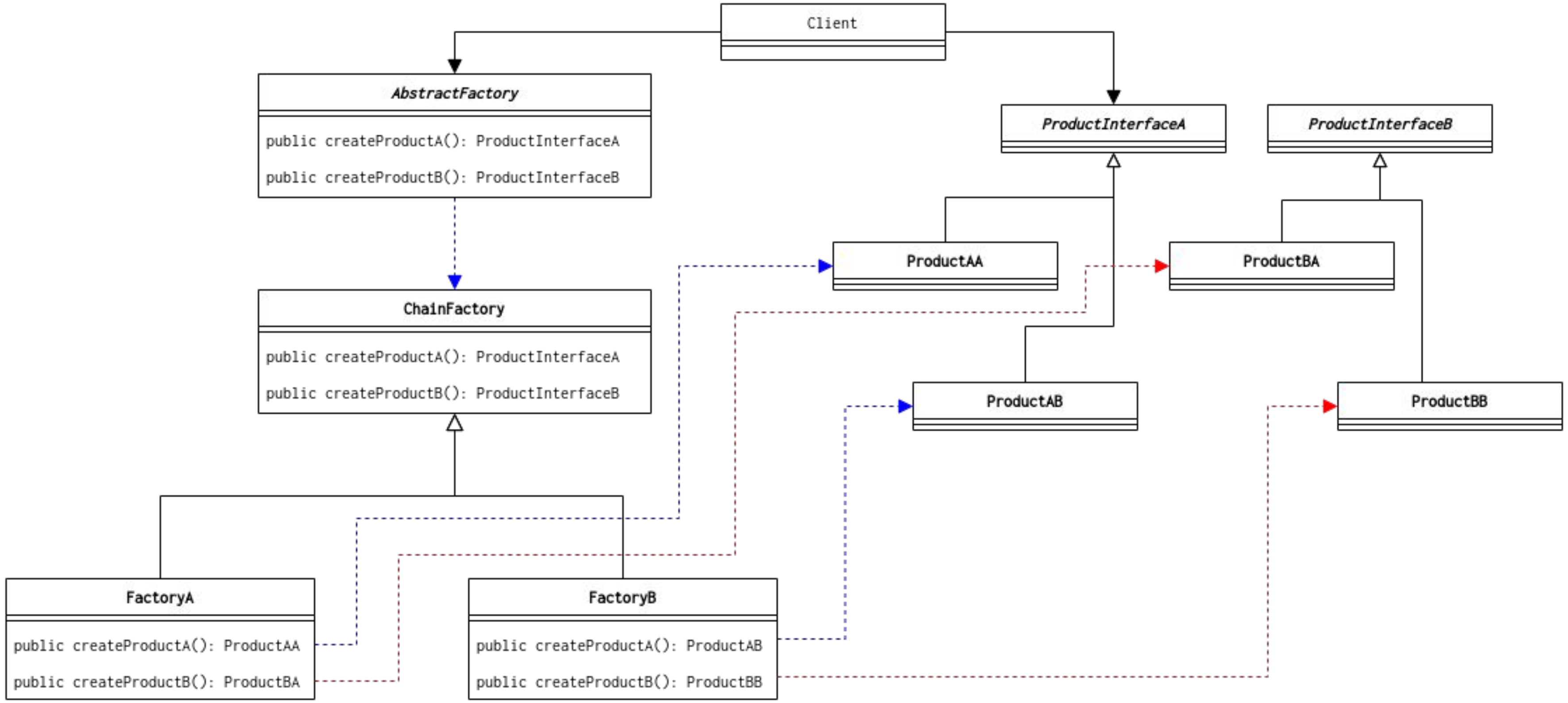
The Abstract Factory design pattern provides a way to encapsulate a group of individual factories pertaining to a certain functionality

Abstract Factory

Difference with Factory Method:

The factory class can create multiple types of objects (and contain multiple factories itself).

Factory Method


Client
AbstractFactory
public createProductA(): ProductInterfaceA
public createProductB(): ProductInterfaceB
ChainFactory
public createProductA(): ProductInterfaceA
public createProductB(): ProductInterfaceB
FactoryA
public createProductA(): ProductAA
public createProductB(): ProductBA
FactoryB
public createProductA(): ProductAB
public createProductB(): ProductBB
ProductInterfaceA
ProductInterfaceB
ProductAA
ProductBA
ProductAB
ProductBB

Abstract Factory

Benefits:

- Each factory produces one specific concrete type
- Each factory can be replaced easily by another
- Adaptability to the runtime environment
- Object construction is centralized

Disadvantages:

- Lots of classes and interfaces
- Client code doesn’t know the exact type
- Hard to implement

Exercise

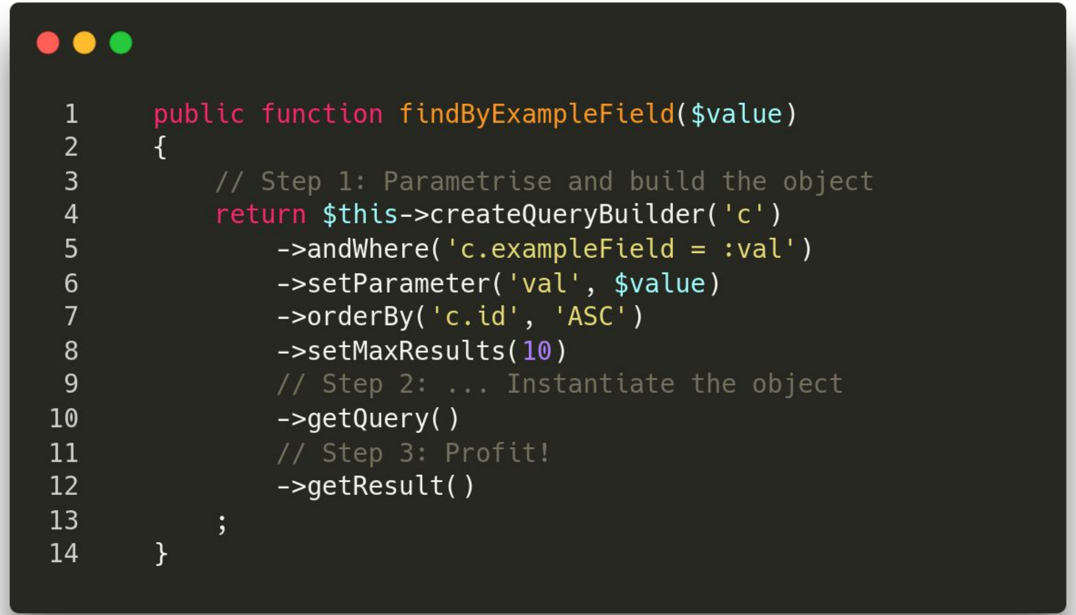
Create a `ChainNotificationFactory` for our factories

Builder

Builder

The Builder design pattern aims to separate the construction of a complex object from its final representation.

Builder



```php
public function findByExampleField($value)
{
    // Step 1: Parametrise and build the object
    return $this->createQueryBuilder('c')
        ->andWhere('c.exampleField = :val')
        ->setParameter('val', $value)
        ->orderBy('c.id', 'ASC')
        ->setMaxResults(10)
        // Step 2: ... Instantiate the object
        ->getQuery()
        // Step 3: Profit!
        ->getResult()
    ;
}
```

SensioLabs

Builder

Benefits:

- Avoid constructors with many optional parameters
- Variable number of build steps
- Leverage fluent interfaces
- Ideal for high level of encapsulation & consistency

Disadvantages:

- Duplicated code in builder and builded classes
- Often very verbose

One more?

Dependency Injection

Yes, it’s a Creational Design Pattern

Structural Design Patterns

Structural Design Patterns

Structural design patterns aim at easing the communication between objects by identifying the simplest way to realize relationships.

They help define the most efficient structures and architectures for a given context to facilitate communication and maintainability across an application.

Structural Design Patterns

Some Structural design patterns:

- Adapter
- Composite
- Decorator
- Marker
- Facade
- Proxy

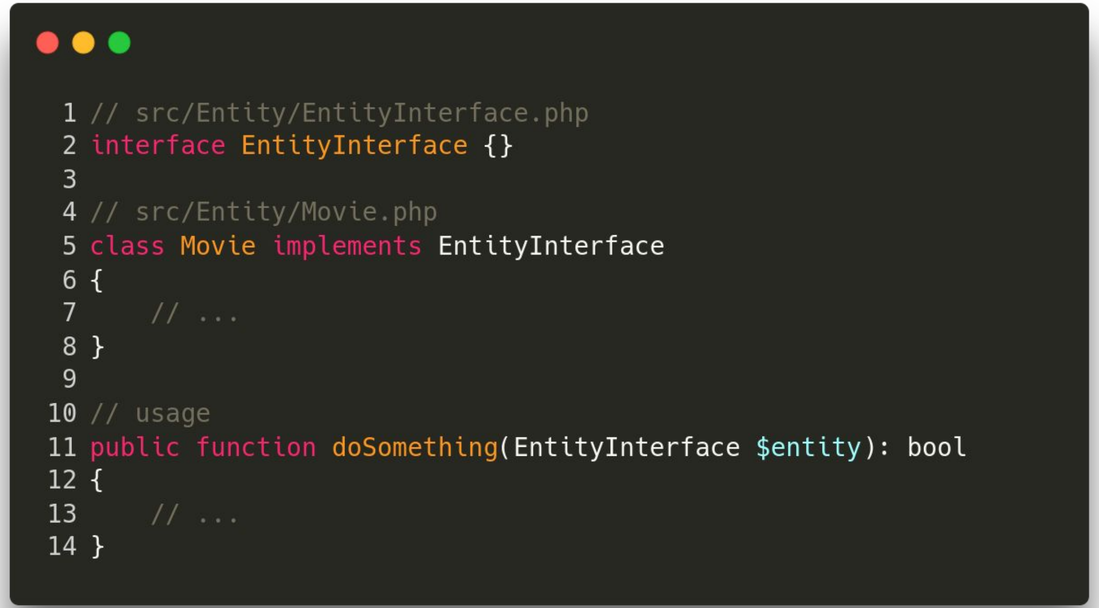
Marker

Marker

The Marker design pattern provides a simple way to attach metadata to a class in order to identify it in a quick and efficient way.

To use it, simply create an empty identifier interface to be applied to a certain kind of object. Classes needing this kind of objects will have a quick way of testing the existence of said interface.

Marker



```php
// src/Entity/EntityInterface.php
interface EntityInterface {}

// src/Entity/Movie.php
class Movie implements EntityInterface
{
    // ...
}

// usage
public function doSomething(EntityInterface $entity): bool
{
    // ...
}
```

SensioLabs

Marker

Benefits:

- Quick way to identify all objects belonging to the same “family”
- Tests and type hints easy

Disadvantages:

- You cannot “unimplement” the interface in a child class if you need to

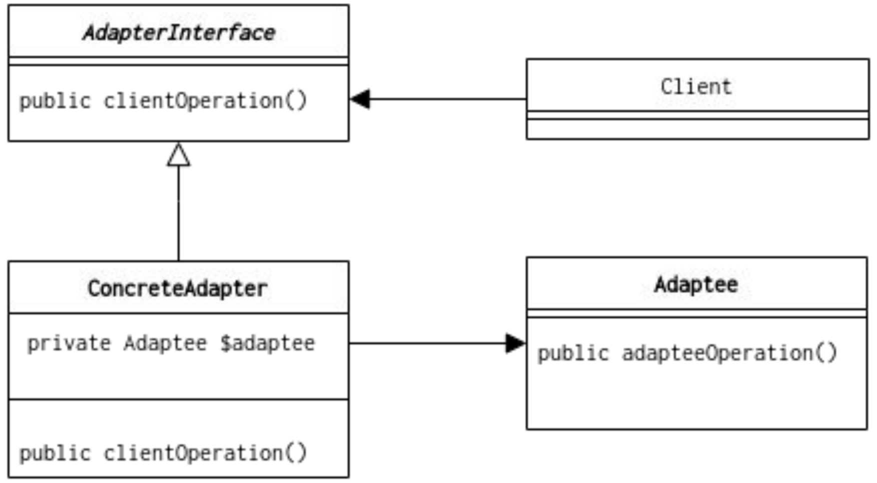
Adapter

Adapter

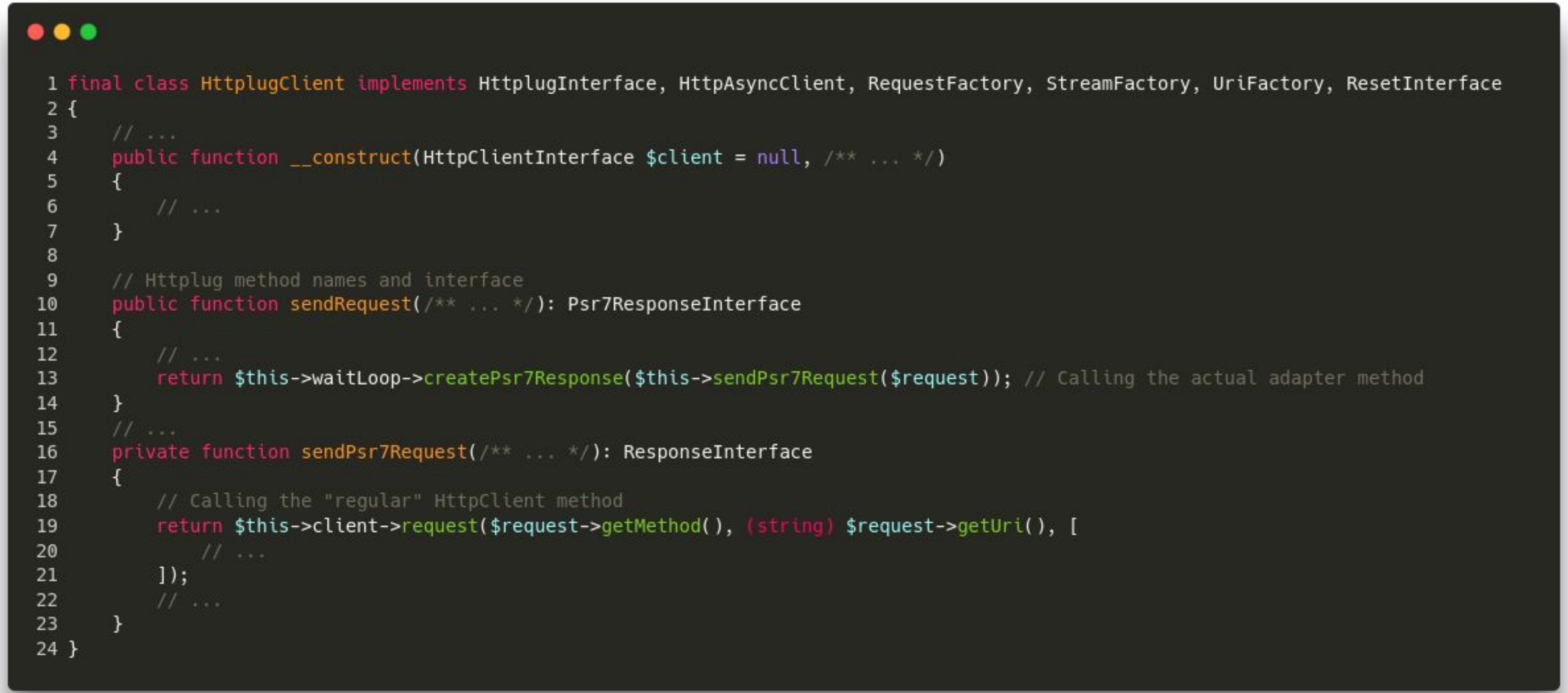
The Adapter pattern aims to allow two objects with completely different and incompatible APIs to work together.

The Adapter patterns consists in a class working as an interface between two incompatible objects to allow communication.

Adapter


AdapterInterface
public clientOperation()
Client
ConcreteAdapter
private Adaptee $adaptee
public clientOperation()
Adaptee
public adapteeOperation()

Adapter



```php
final class HttplugClient implements HttplugInterface, HttpAsyncClient, RequestFactory, StreamFactory, UriFactory, ResetInterface
{
    // ...
    public function __construct(HttpClientInterface $client = null, /** ... */)
    {
        // ...
    }

    // Httplug method names and interface
    public function sendRequest(/** ... */): Psr7ResponseInterface
    {
        // ...
        return $this->waitLoop->createPsr7Response($this->sendPsr7Request($request)); // Calling the actual adapter method
    }
    // ...
    private function sendPsr7Request(/** ... */): ResponseInterface
    {
        // Calling the "regular" HttpClient method
        return $this->client->request($request->getMethod(), (string) $request->getUri(), [
            // ...
        ]);
        // ...
    }
}
```

SensioLabs

Adapter

Benefits:

- Easy way to adapt an existing class or library to your code without breaking interfaces
- Ideal to ensure backward compatibility
- Ideal to reuse subclasses lacking some new functionnalities

Disadvantages:

- Code complexity can rise enormously with adapters

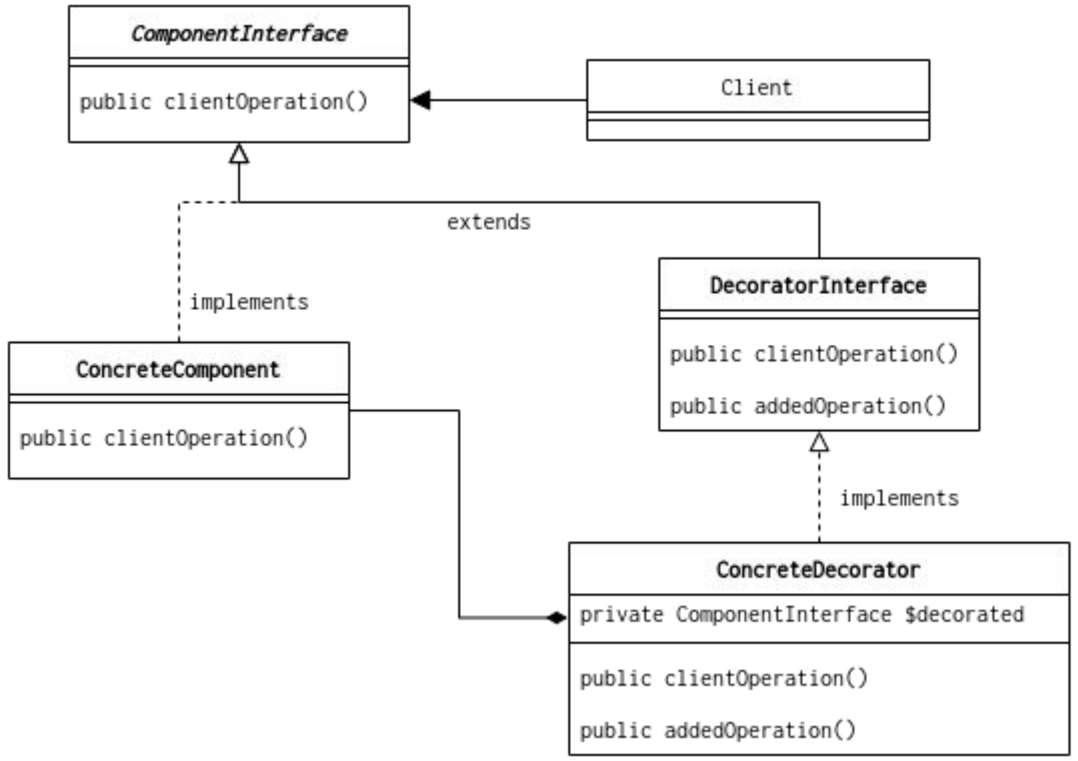
Decorator

Decorator

The Decorator pattern allows you to add functionalities to an object without changing the behavior of other objects of the same class.

By creating decorator wrappers, you can add new functionalities to a particular object without changing its base class or breaking compatibility.

Decorator

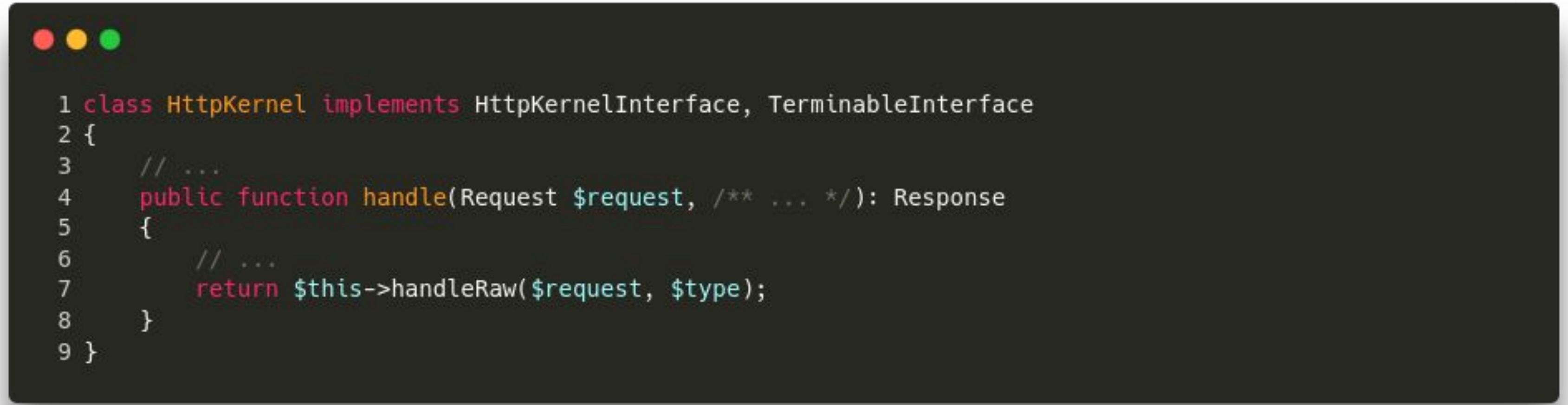

ComponentInterface
public clientOperation()
Client
extends
implements
DecoratorInterface
public clientOperation()
public addedOperation()
ConcreteComponent
public clientOperation()
implements
ConcreteDecorator
private ComponentInterface $decorated
public clientOperation()
public addedOperation()

Decorator

Example: The Symfony Kernel

- The `HttpKernel` implements the `HttpKernelInterface` to deal with requests
- The Kernel (located in `src/Kernel.php`) extends the abstract `Kernel` from HttpKernel component, which implements the `KernelInterface`
- `KernelInterface` extends `HttpKernelInterface`, adding it functionalities
- The Kernel wraps a `HttpKernel` instance via the Container

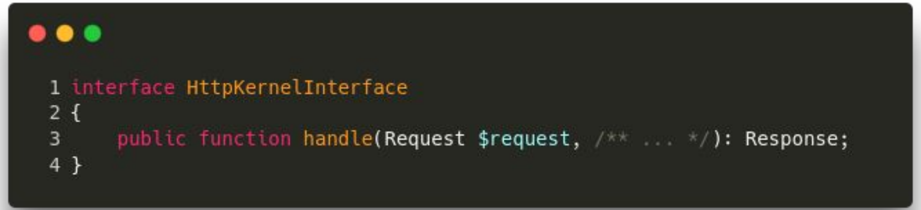
Decorator



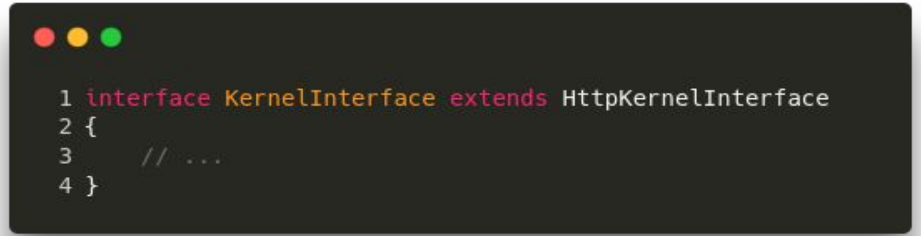
```php
class HttpKernel implements HttpKernelInterface, TerminableInterface
{
    // ...
    public function handle(Request $request, /** ... */): Response
    {
        // ...
        return $this->handleRaw($request, $type);
    }
}
```

SensioLabs

Decorator



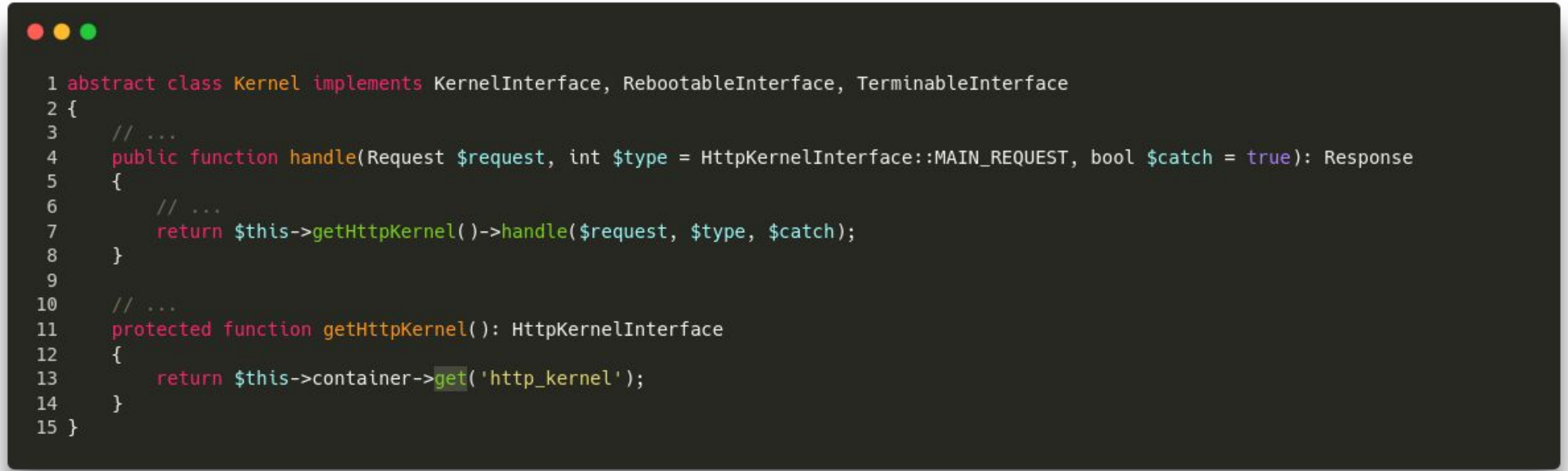
```
interface HttpKernelInterface
{
    public function handle(Request $request, /** ... */): Response;
}
```



```
interface KernelInterface extends HttpKernelInterface
{
    // ...
}
```

SensioLabs

Decorator



```php
abstract class Kernel implements KernelInterface, RebootableInterface, TerminableInterface
{
    // ...
    public function handle(Request $request, int $type = HttpKernelInterface::MAIN_REQUEST, bool $catch = true): Response
    {
        // ...
        return $this->getHttpKernel()->handle($request, $type, $catch);
    }

    // ...
    protected function getHttpKernel(): HttpKernelInterface
    {
        return $this->container->get('http_kernel');
    }
}
```

SensioLabs

Decorator

Benefits:

- Easy way to extend or restrict and object’s responsibilities
- No creation of subclass
- Leverages Single Responsibility Principle and Open/Closed Principle

Disadvantages:

- Does not work well for objects with a large API
- Initial configuration code can become pretty complex and/or ugly

Behavioral Design Patterns

Behavioral Design Patterns

Behavioral patterns are concerned with the assignment of responsibilities between objects.

They identify common communication patterns among objects, increasing flexibility by doing so and allowing for looser coupling.

Behavioral Design Patterns

Some Behavioral design patterns:

- Chain of Responsibility
- Iterator
- Mediator
- Memento
- Observer
- Strategy
- Visitor

Chain of Responsibility

Chain of Responsibility

Chain of Responsibility let you pass a request to a chain of handlers. Each handler will in turn decides if it handles the request or it passes along to the next handler.

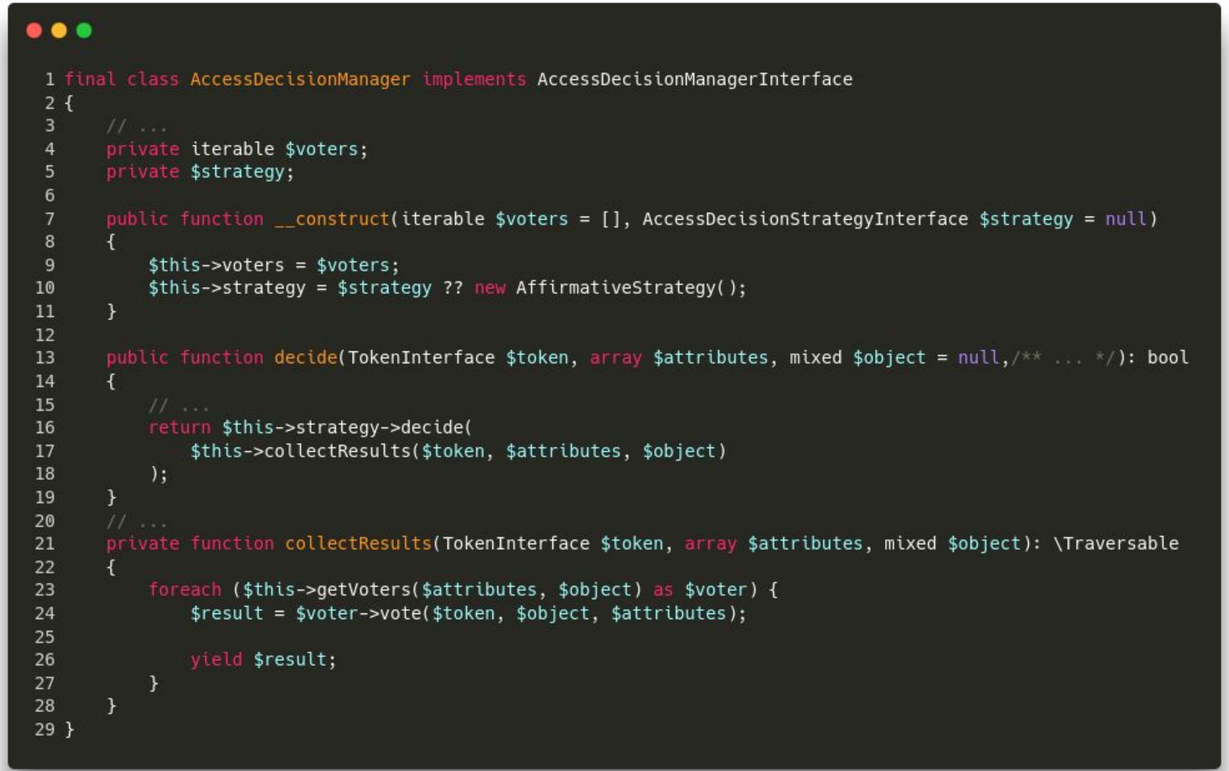
It allows to avoid coupling the sender of the request to one particular receiver. This way, only the more adapted receiver handles the request.

Chain of Responsibility

Example: the Security Access Decision Manager

- The `AccessDecisionManager` receives an iterable containing objects extending the `Voter` class
- Each voter will be called in turn.
- The voter currently called will first check if it supports or not the security request based on a attribute and a subject
- If not, it will pass to the next
- If it does, it will evaluate the legitimacy of the request

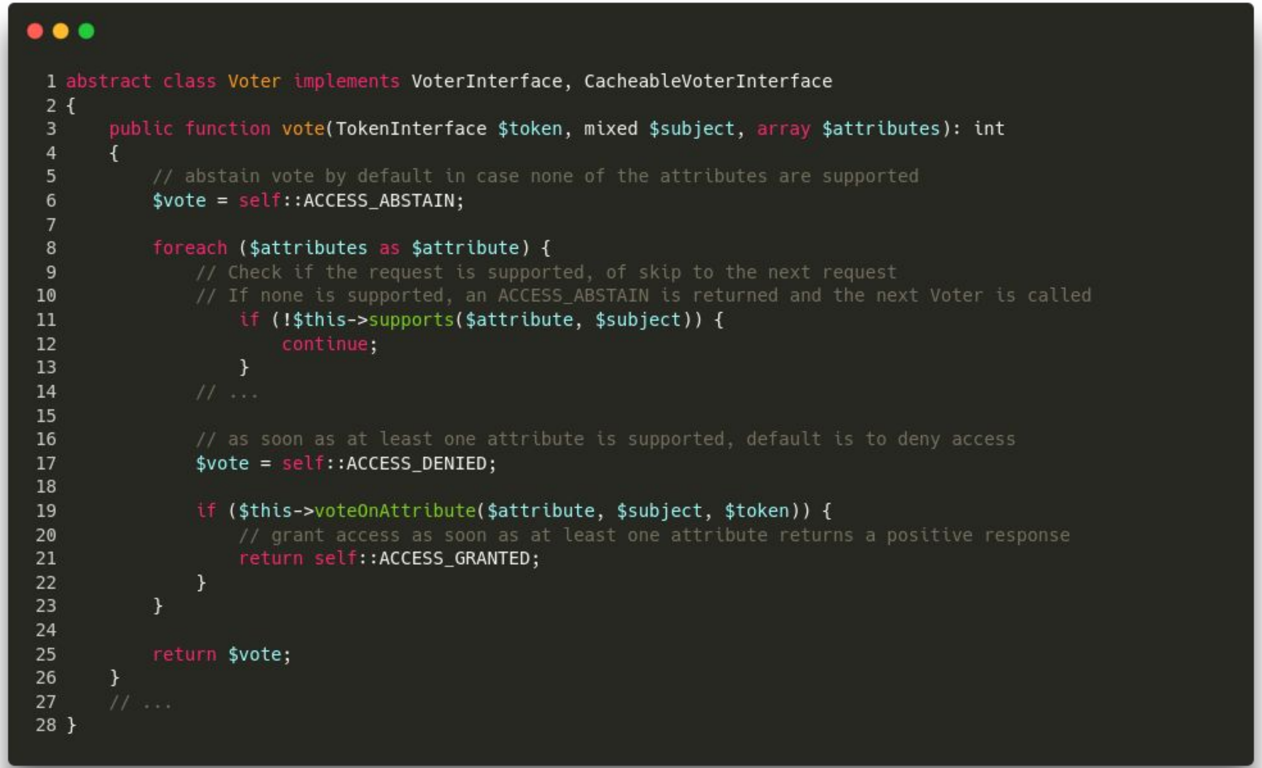
Chain of Responsibility



```php
final class AccessDecisionManager implements AccessDecisionManagerInterface
{
    // ...
    private iterable $voters;
    private $strategy;

    public function __construct(iterable $voters = [], AccessDecisionStrategyInterface $strategy = null)
    {
        $this->voters = $voters;
        $this->strategy = $strategy ?? new AffirmativeStrategy();
    }

    public function decide(TokenInterface $token, array $attributes, mixed $object = null,/** ... */): bool
    {
        // ...
        return $this->strategy->decide(
            $this->collectResults($token, $attributes, $object)
        );
    }
    // ...
    private function collectResults(TokenInterface $token, array $attributes, mixed $object): \Traversable
    {
        foreach ($this->getVoters($attributes, $object) as $voter) {
            $result = $voter->vote($token, $object, $attributes);

            yield $result;
        }
    }
}
```

SensioLabs

Chain of Responsibility



```php
abstract class Voter implements VoterInterface, CacheableVoterInterface
{
    public function vote(TokenInterface $token, mixed $subject, array $attributes): int
    {
        // abstain vote by default in case none of the attributes are supported
        $vote = self::ACCESS_ABSTAIN;

        foreach ($attributes as $attribute) {
            // Check if the request is supported, of skip to the next request
            // If none is supported, an ACCESS_ABSTAIN is returned and the next Voter is called
                if (!$this->supports($attribute, $subject)) {
                    continue;
                }
            // ...

            // as soon as at least one attribute is supported, default is to deny access
            $vote = self::ACCESS_DENIED;

            if ($this->voteOnAttribute($attribute, $subject, $token)) {
                // grant access as soon as at least one attribute returns a positive response
                return self::ACCESS_GRANTED;
            }
        }

        return $vote;
    }
    // ...
}
```

SensioLabs

Chain of Responsibility

Benefits:

- Flexibility, the chain of responsibility can handle multiple types of requests
- Leveraging Open/Closed Principle and Single Responsibility Principle
- A handler can be added or removed without breaking the code

Disadvantages:

- Some requests may end up unhandled
- Can lead to code duplication unless you have some boilerplate code in an abstract handler, or a Template Method

Mediator

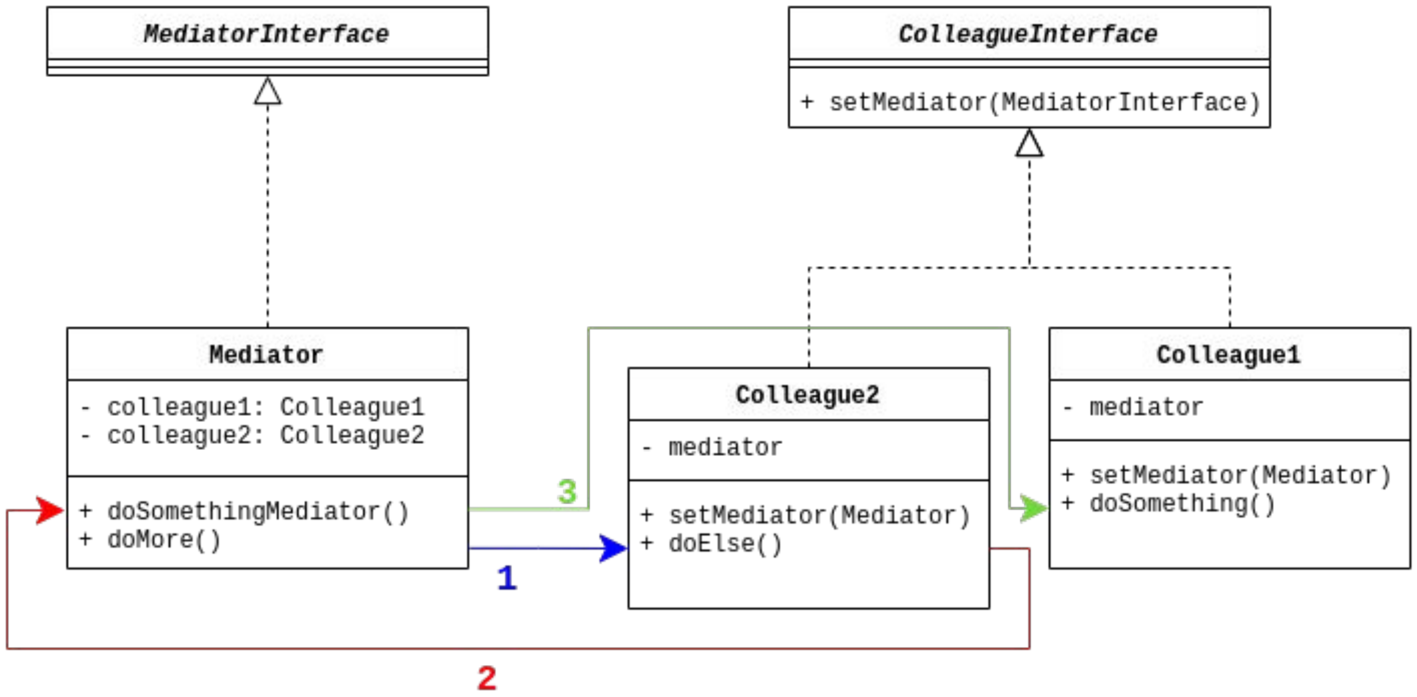
Mediator

The Mediator pattern aims to decouple objects working together.

By creating a centralizing object dealing with the communication between two services, the Mediator pattern forbids objects from referring each others directly.

This allows for more reusability and varying interactions

Mediator



SensioLabs

Mediator

Benefits:

- Loose coupling and improved reusability
- Single Responsibility: objects deal only with their responsibility, the Mediator deals with communication
- Easing testability

Disadvantages:

- Over time, the mediator can become bloated and evolve in a God Object

Observer

Observer

The Observer design pattern lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

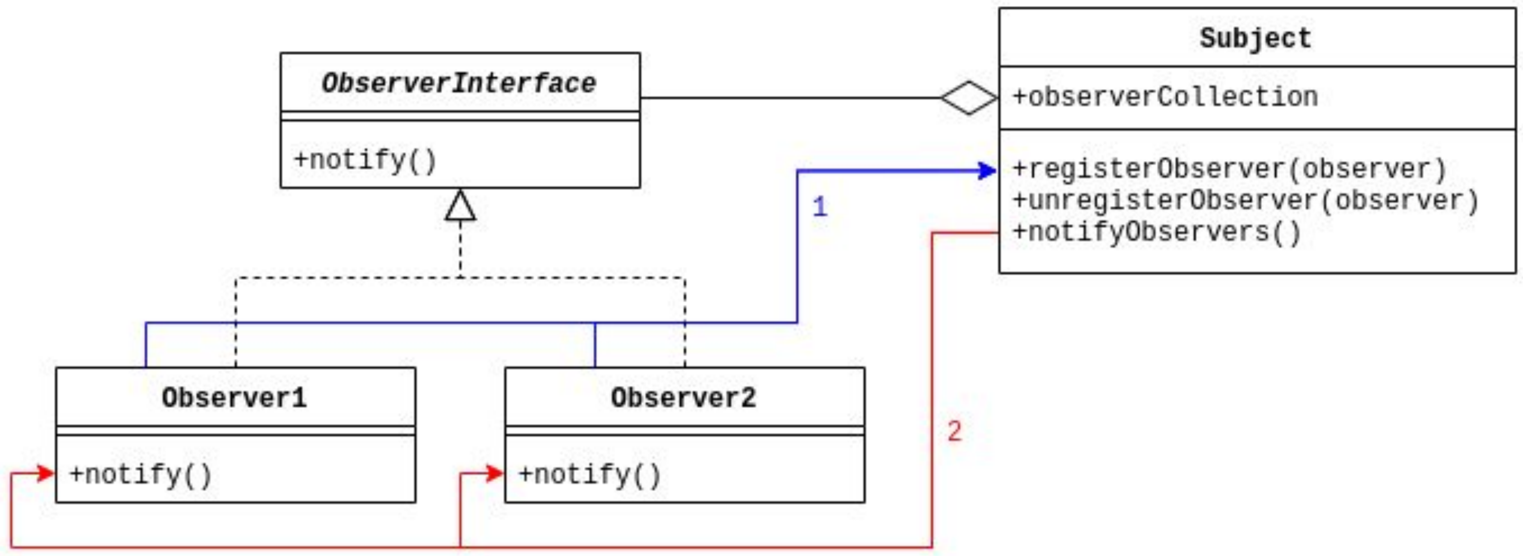
A Publisher registers a list of Subscribers listening for a specific event. The Published then notifies the Subscribers when said event is dispatched.

Observer

Observer is pretty similar to the Mediator design. The differences:

- The object themselves notify their observers.
- Mediator allows communication between a broader range of objects

Observer


ObserverInterface
+notify()
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
1
2
Observer1
+notify()
Observer2
+notify()

Observer

Benefits:

- Easy to implement
- Pluggable behavior lets you add subscribers without breaking code

Disadvantages:

- Mediator generally leads to less coupling

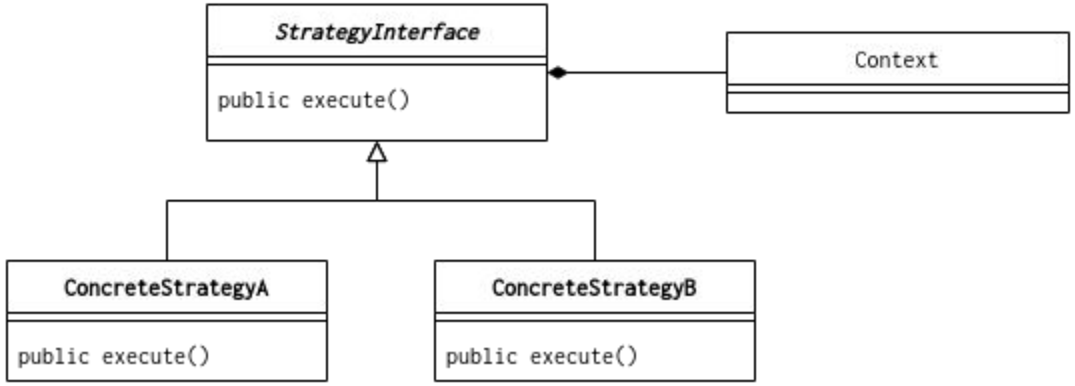
Strategy

Strategy

The Strategy design pattern lets you build a family of algorithms, each one in a separate class, from which to choose at runtime.

It allows for modularity, and adding new strategies without breaking previous implementations.

Strategy


StrategyInterface
public execute()
Context
ConcreteStrategyA
public execute()
ConcreteStrategyB
public execute()

Strategy

Benefits:

- Easy to implement
- Allow the behavior to change at runtime depending on the client request
- Easily compatible with structural patterns

Disadvantages:

- Easily overused - if you have only two algorithm to choose from, not very useful

Exercise

Change the `App\Notifier\MovieOrderNotifier`:

- Its constructor will receive a second argument `iterable $factories` containing every factory created earlier
- Make the notifier choose the correct factory according to the `sendNotification` `$channel` argument

Template Method

Template Method

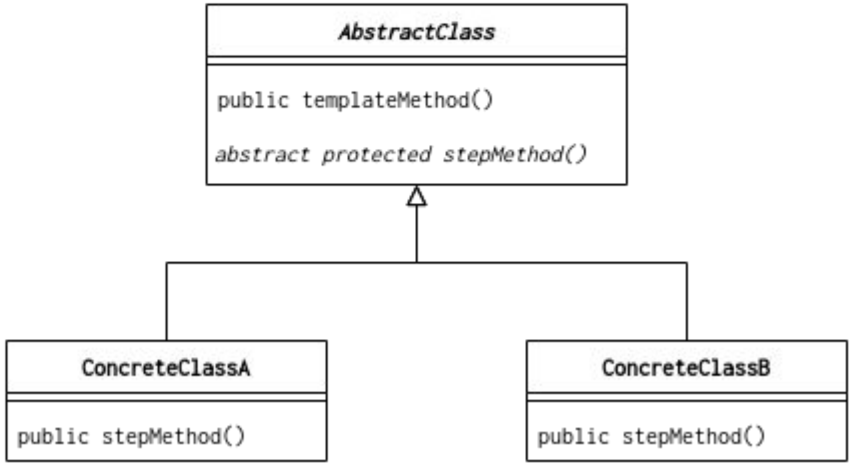
The Template Method pattern lets you define the skeleton of an algorithm in a superclass and allow subclasses to redefine certain steps without changing the structure.

Template Method

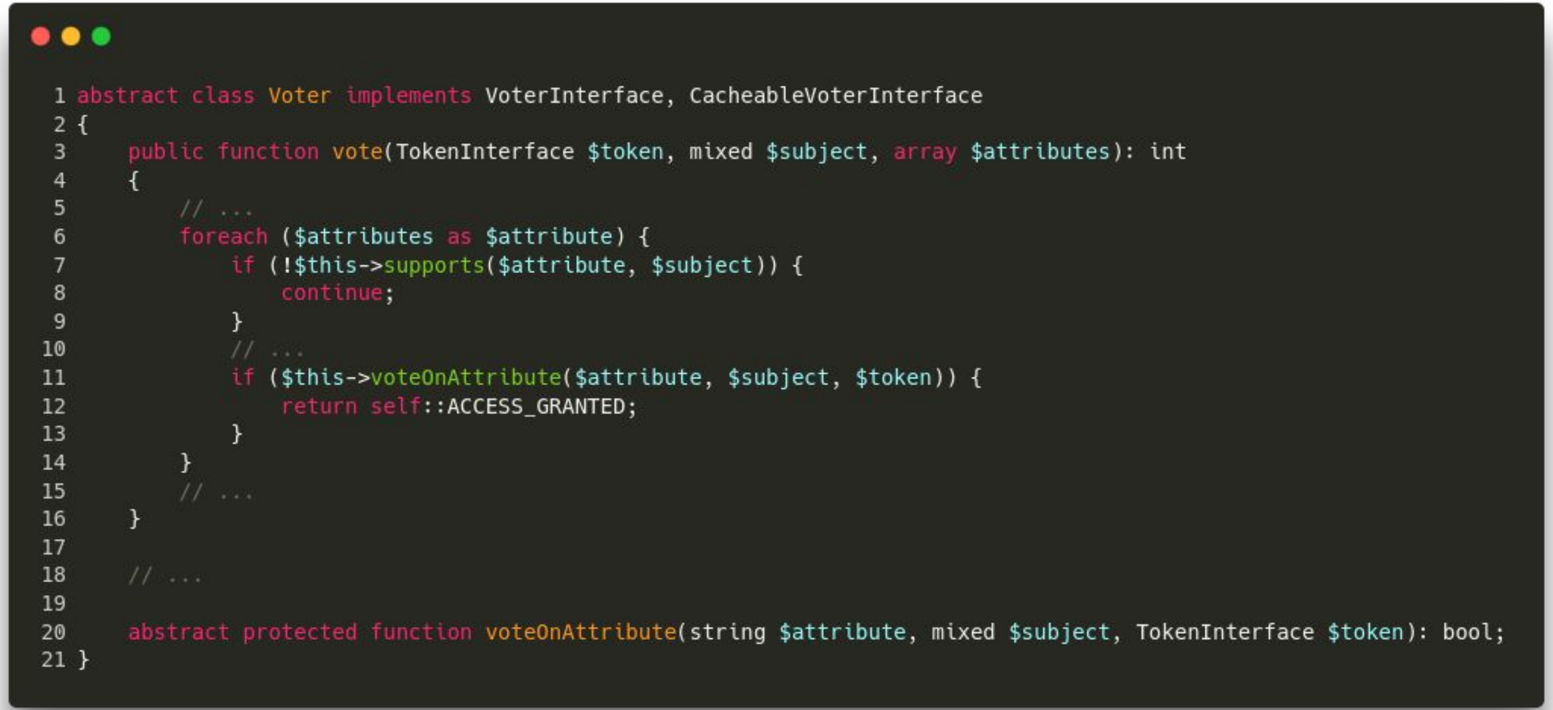
The Template Method design pattern aims to encapsulate an algorithm and prevent it from being completely overridden by subclasses.

It does encourage subclasses to change algorithm steps they wish to customize to their needs, by leveraging the Hollywood Principle: “Don’t call us, we’ll call you!”

Template Method



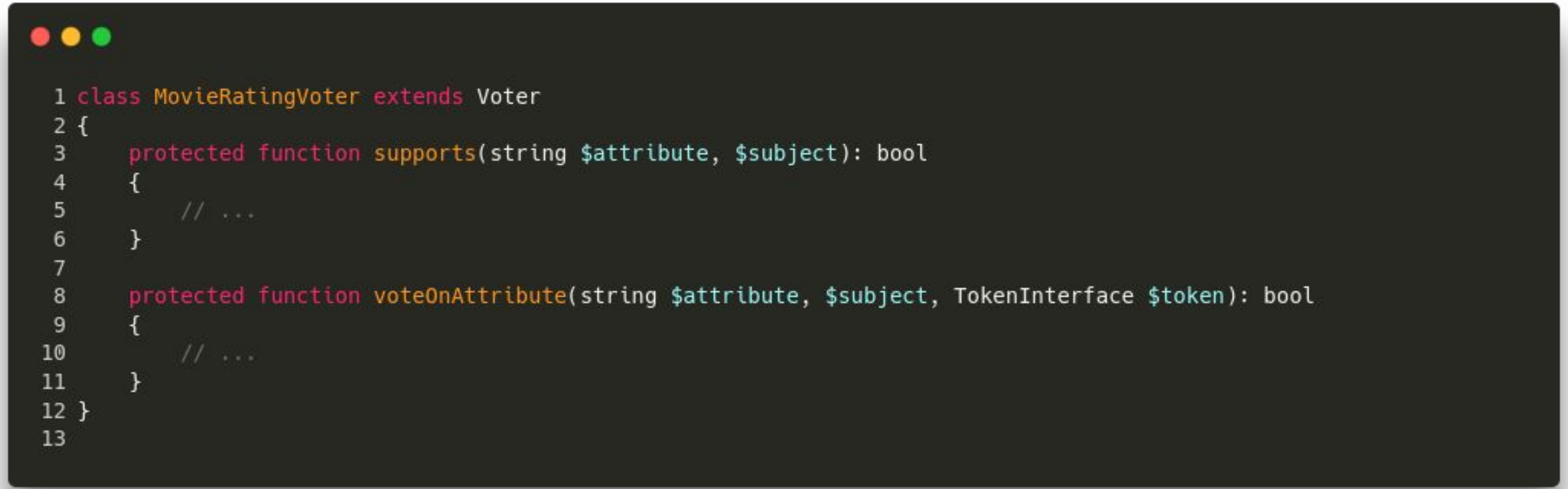
Template Method



```php
abstract class Voter implements VoterInterface, CacheableVoterInterface
{
    public function vote(TokenInterface $token, mixed $subject, array $attributes): int
    {
        // ...
        foreach ($attributes as $attribute) {
            if (!$this->supports($attribute, $subject)) {
                continue;
            }
            // ...
            if ($this->voteOnAttribute($attribute, $subject, $token)) {
                return self::ACCESS_GRANTED;
            }
        }
        // ...
    }

    // ...

    abstract protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool;
}
```

SensioLabs

Template Method



```php
class MovieRatingVoter extends Voter
{
    protected function supports(string $attribute, $subject): bool
    {
        // ...
    }

    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
    {
        // ...
    }
}
```

Template Method

Benefits:

- Easy to implement
- Ensures the algorithm is fully executed by providing default implementations if need be
- Help eliminate duplicated code

Disadvantages:

- Potentially breaks the Liskov Substitution Principle by suppressing a default step
- Client code may be limited by the provided skeleton
- May become harder to maintain when adding steps

Exercise

Exercise

1. Add a complete Model folder with a ValueObject representation of the `Movie` and `Genre` entities
2. Change the `MovieTransformer` to have the following methods:
   a. `public function toArray(Model): array`
   b. `public function fromArray(array): Model`
   c. `public function toEntity(Model): Entity`
   d. `public function fromEntity(Entity): Model`
3. Adapt the `MovieProvider` to use and return `Movie` Model

Conclusion

Keep your code organized

Conclusion

Design pattern and Calisthenics may help you, but they are merely tools, to be chosen and used ***carefully***.

Don’t use Design Patterns

Conclusion

Keep asking yourself if you really need a design pattern.
Sometimes the easiest way is the best.

KISS

Keep It Simple, Stupid!

Thank you!

@sensiolabs
sensiolabs.com

SensioLabs
Créateur de Symfony

Resources

Thanks Titouan Galopin for the original slides - @titouangalopin

https://en.wikipedia.org/wiki/SOLID

https://williamdurand.fr/2013/06/03/object-calisthenics/

https://en.wikipedia.org/wiki/Software_design_pattern

https://refactoring.guru/design-patterns

@sensiolabs
sensiolabs.com

SensioLabs
Créateur de Symfony