Identifying Design Patterns in the Symfony Framework

Istanbul – Turkey – May 3rd 2014

Hugo HAMON

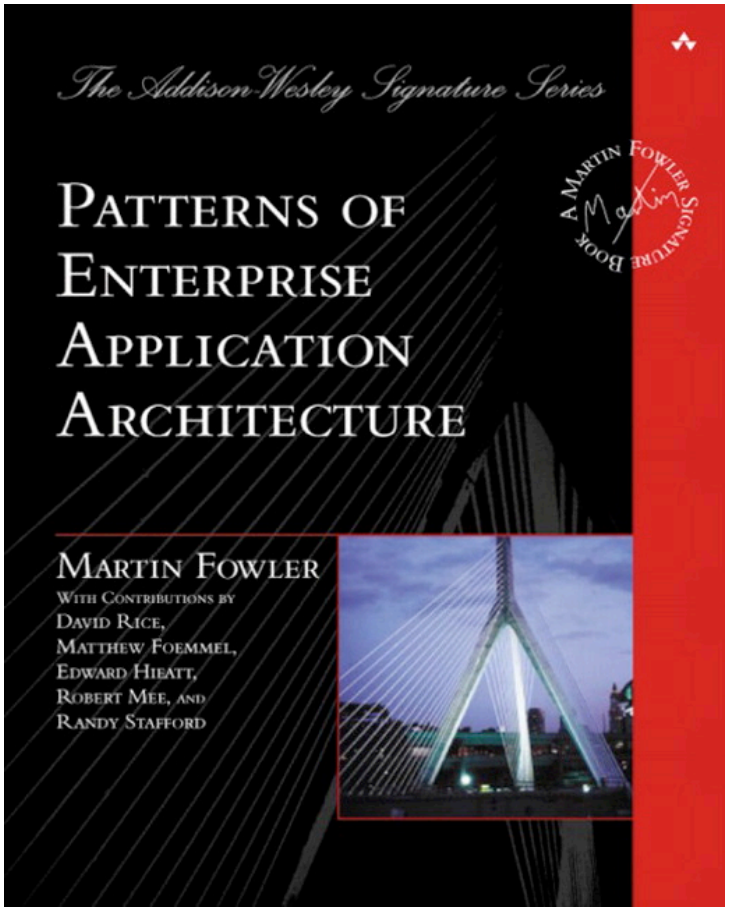
Head of training at SensioLabs

Book author
Speaker at Conferences
Symfony contributor
@hhamon

Introduction to Design Patterns

In software design, a **design pattern** is an abstract **generic solution** to **solve** a particular **common problem**.

Recommended readings

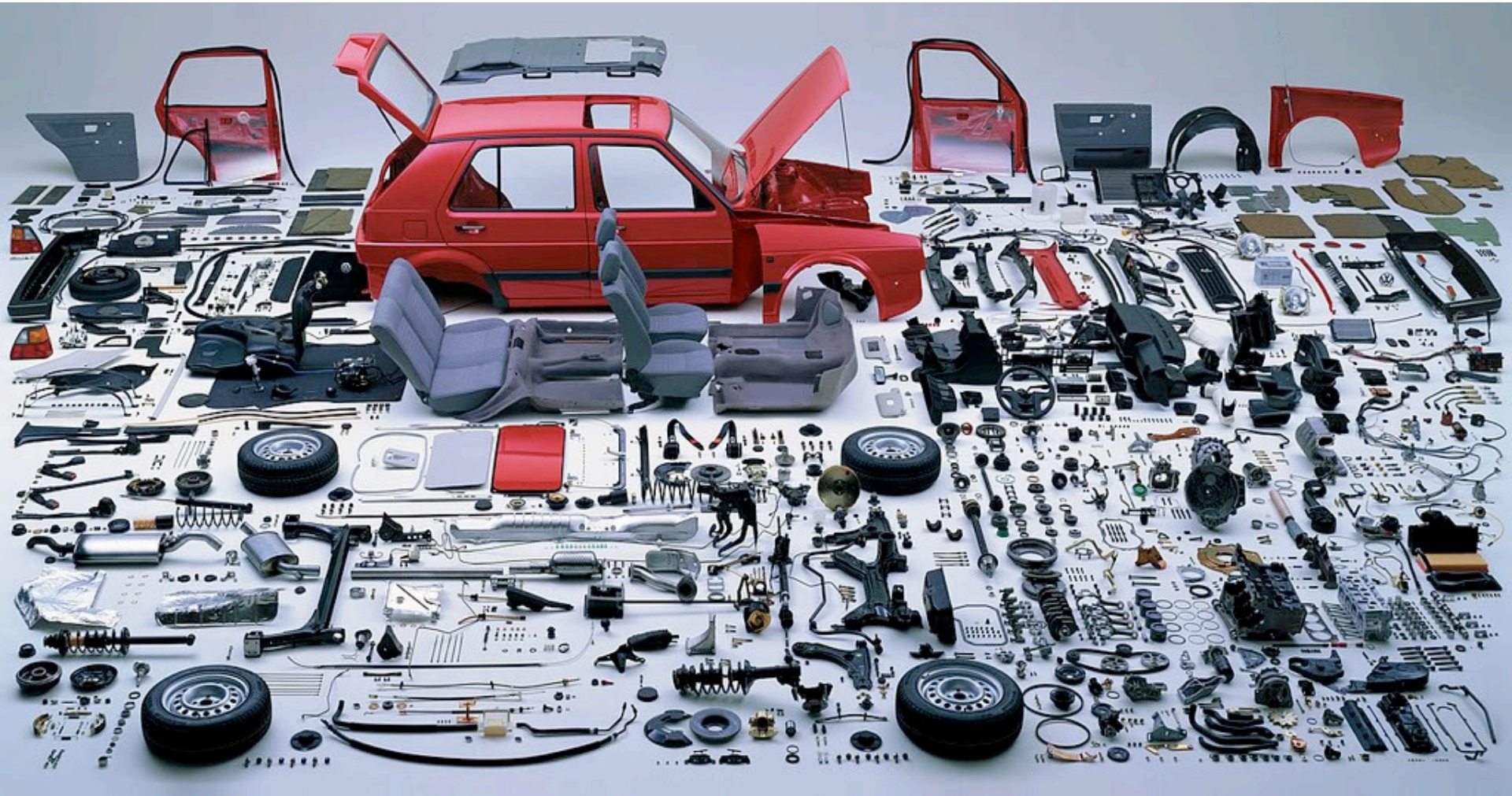


Disclaimer

They aren’t the holy grail!

Communication

Loose Coupling



Unit testability

```
Configuration read from /Users/ctheys/foo/d8/core/phpunit.xml.dist

.............................................................  63 / 557 ( 11%)
.............................................................. 126 / 557 ( 22%)
.............................................................. 189 / 557 ( 33%)
.............................................................. 252 / 557 ( 45%)
.............................................................. 315 / 557 ( 56%)
.............................................................. 378 / 557 ( 67%)
.............................................................. 441 / 557 ( 79%)
.............................................................. 504 / 557 ( 90%)
.....................................................

Time: 15 seconds, Memory: 54.25Mb

OK (557 tests, 1135 assertions)

Generating code coverage report in HTML format ... done
[~/foo/d8/core]
01:30 PM [YesCT]  (8.x)
711 $
```

Maintenance

Three patterns families

Creational Patterns

Abstract Factory

Builder

Factory Method

Lazy Initialization

Prototype

Singleton

Structural Patterns

Adapter

Bridge

Composite

Decorator

Facade

Flyweight

Proxy

Behavioral Patterns

Chain of Responsability

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

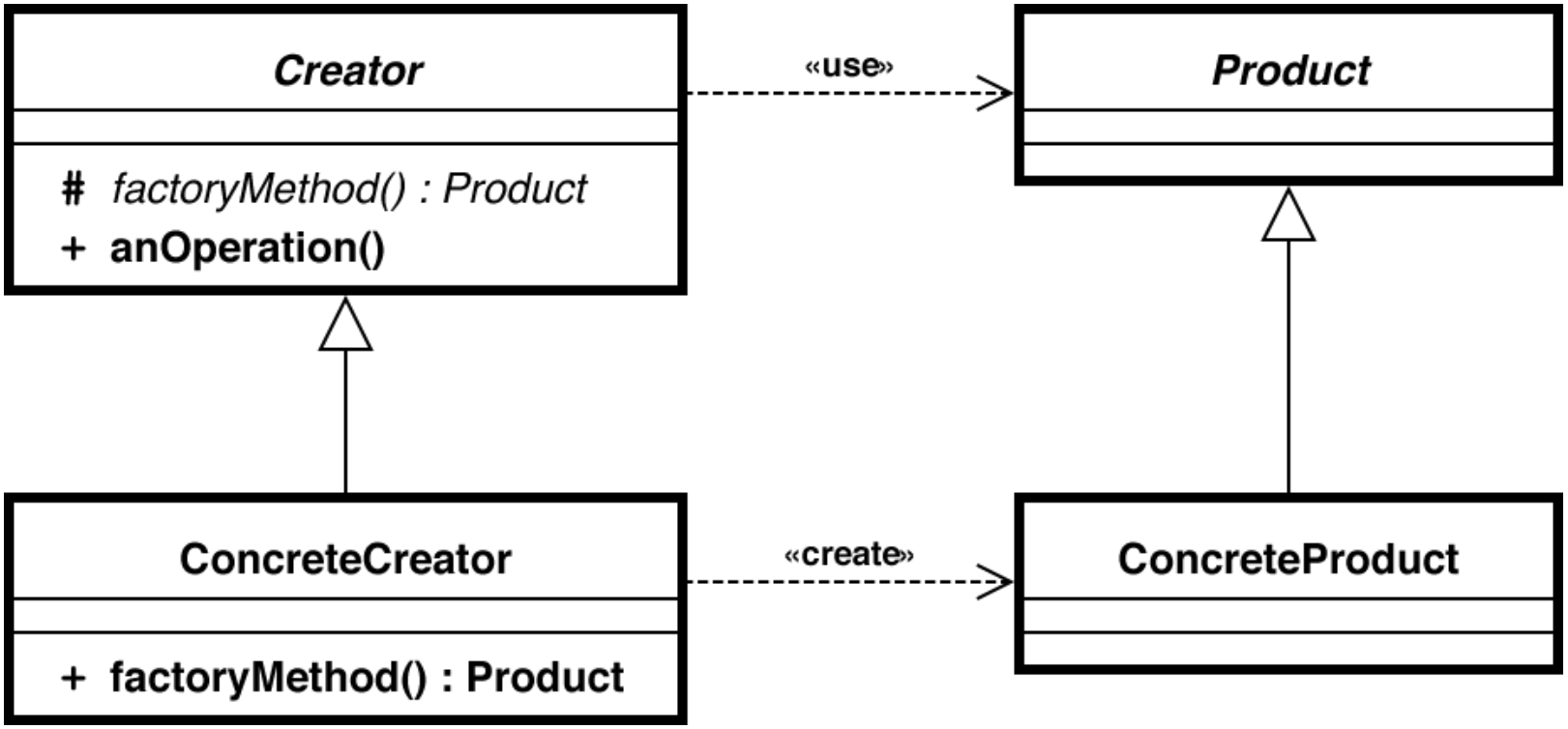
Visitor

Design Patterns applied to Symfony

Creational Patterns

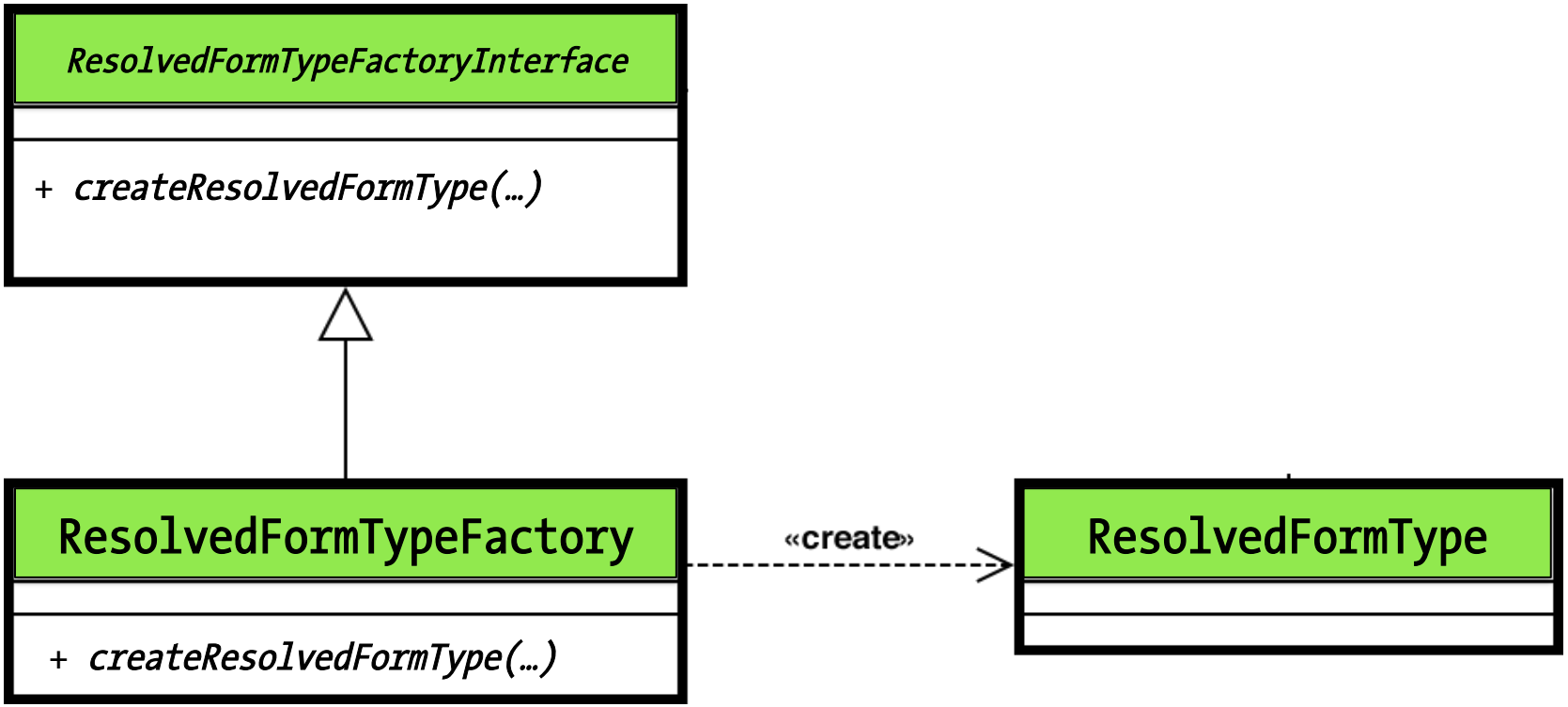
Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate.


Creator
factoryMethod() : Product
+ anOperation()
«use»
Product
ConcreteCreator
+ factoryMethod() : Product
«create»
ConcreteProduct

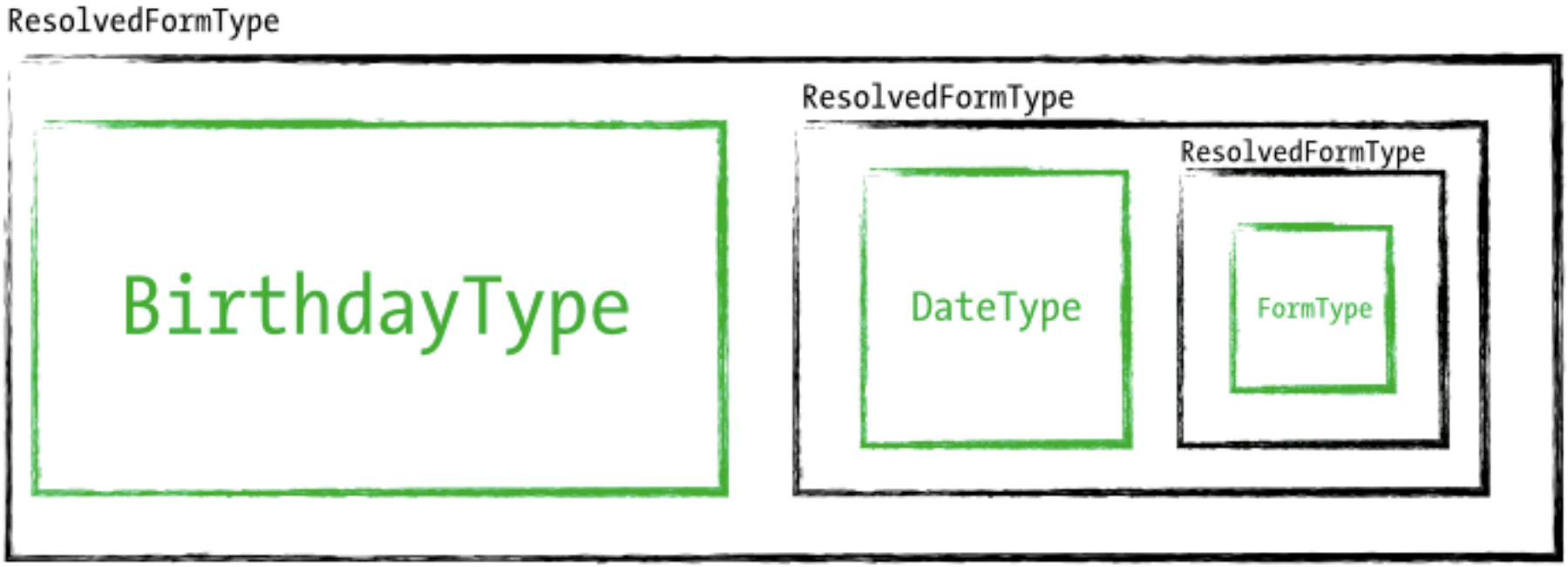
Form Component

Resolving form field types inheritance

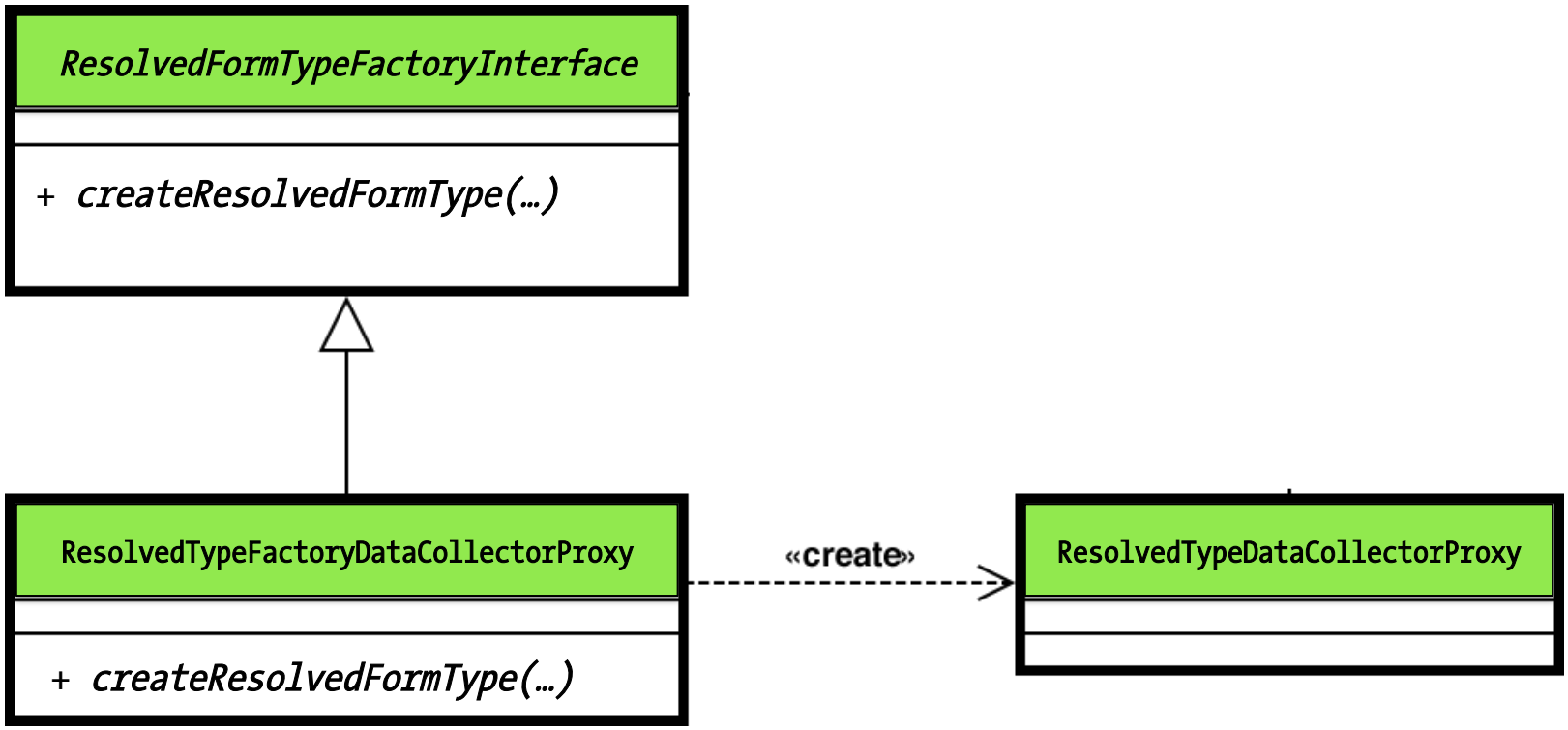


```php
namespace Symfony\Component\Form;

class ResolvedFormTypeFactory
    implements ResolvedFormTypeFactoryInterface
{
    public function createResolvedType(
        FormTypeInterface $type,
        array $typeExtensions,
        ResolvedFormTypeInterface $parent = null
    )
    {
        return new ResolvedFormType(
            $type,
            $typeExtensions,
            $parent
        );
    }
}
```

```php
$f = new ResolvedFormTypeFactory();

$form = $f->createResolvedType(new FormType());
$date = $f->createResolvedType(new DateType(), [], $form);
$bday = $f->createResolvedType(new BirthdayType(), [], $date);
```

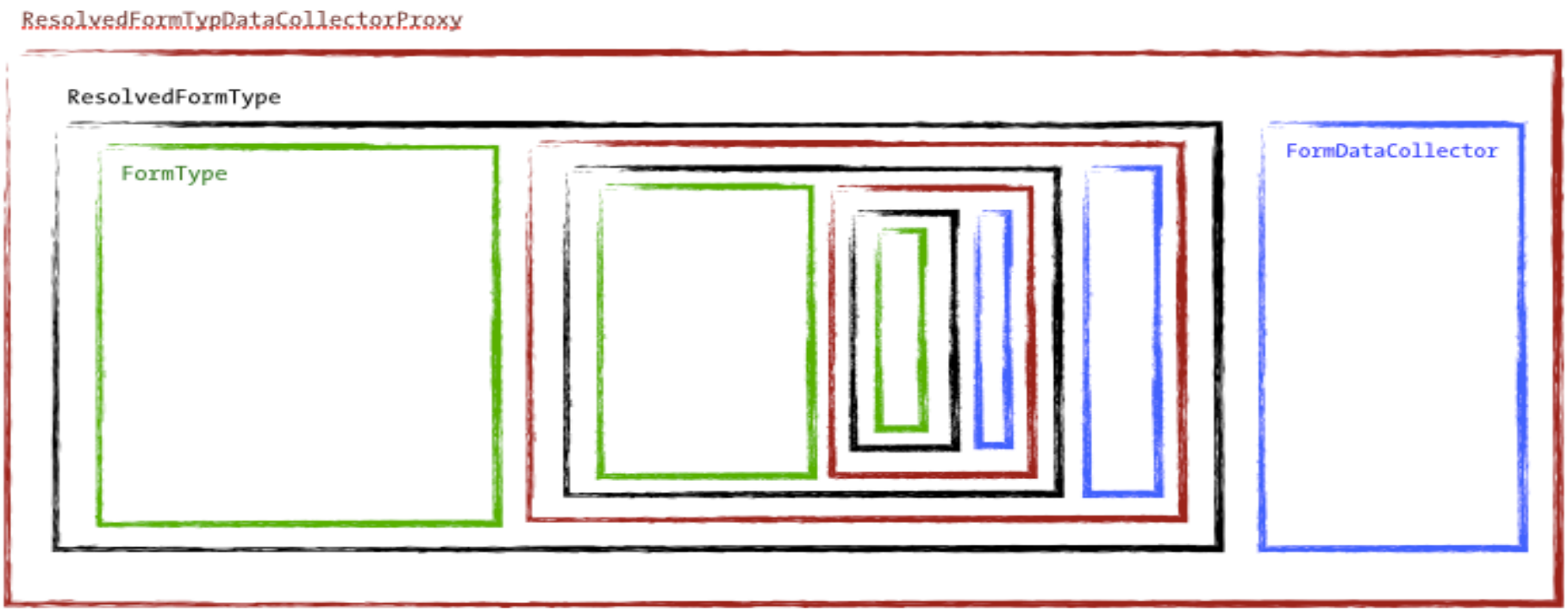


Collecting Resolved Types States

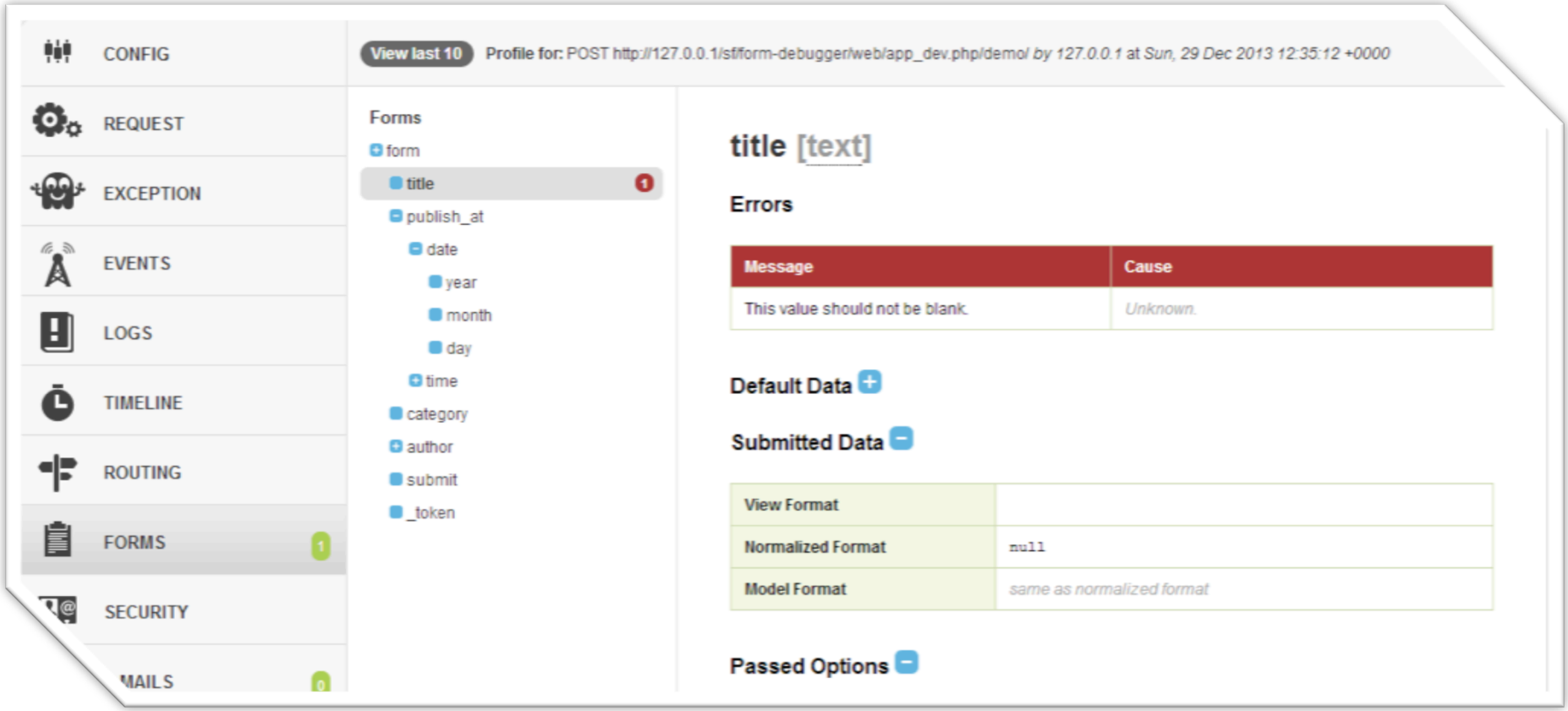


```php
class ResolvedTypeFactoryDataCollectorProxy implements ResolvedFormTypeFactoryInterface
{
    private $proxiedFactory;
    private $dataCollector;

    public function __construct(
        ResolvedFormTypeFactoryInterface $proxiedFactory,
        FormDataCollectorInterface $dataCollector
    )
    {
        $this->proxiedFactory = $proxiedFactory;
        $this->dataCollector = $dataCollector;
    }

    public function createResolvedType(
        FormTypeInterface $type,
        array $typeExtensions,
        ResolvedFormTypeInterface $parent = null
    )
    {
        return new ResolvedTypeDataCollectorProxy(
            $this->proxiedFactory->createResolvedType($type, $typeExtensions, $parent),
            $this->dataCollector
        );
    }
}
```

```php
$factory = new ResolvedTypeDataCollectorProxyFactory(
    new ResolvedFormTypeFactory(),
    new FormDataCollector(…)
);

$form = $f->createResolvedType(new FormType());
$date = $f->createResolvedType(new DateType(), [], $form);
$bday = $f->createResolvedType(new BirthdayType(), [], $date);
```



The Form Profiler Debug Panel



Lazy Initialization

The lazy initialization pattern is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is really needed.

Why using it?

Optimizing performance
Saving memory consumption
Opening connections when really needed
Getting information on-demand

Dependency Injection Component

The Service Container

The service container allows you to standardize and centralize the way objects are constructed in your application.

Lazy Initialization

+ Factory Method

= Service Container

```php
class Container implements ContainerInterface
{
    private $services  = [];
    private $methodMap = [];

    public function get($id)
    {
        // Re-use shared service instance if it exists.
        if (isset($this->services[$id])) {
            return $this->services[$id];
        }

        // Lazy instantiate the service
        $method = $this->methodMap[$id];

        return call_user_func(array($this, $method));
    }
}
```

```php
class appDevDebugProjectContainer extends Container
{
    protected function getLoggerService()
    {
        // Instanciate the requested class.
        $obj = new \Symfony\Bridge\Monolog\Logger('app');

        $this->services['logger'] = $obj;

        // Initialize the instance before usage.
        $obj->pushHandler($this->get('monolog.handler.chromephp'));
        $obj->pushHandler($this->get('monolog.handler.firephp'));
        $obj->pushHandler($this->get('monolog.handler.main'));
        $obj->pushHandler($this->get('monolog.handler.debug'));

        return $obj;
    }
}
```

```php
// Logger is created on demand
$logger1 = $container->get('logger');

// Logger is simply fetched from array map
$logger2 = $container->get('logger');

// Two variables reference same instance
var_dump($logger1 === $logger2);
```

Structural Patterns

Adapter

The adapter pattern allows the interface of an existing class to be used from another interface.

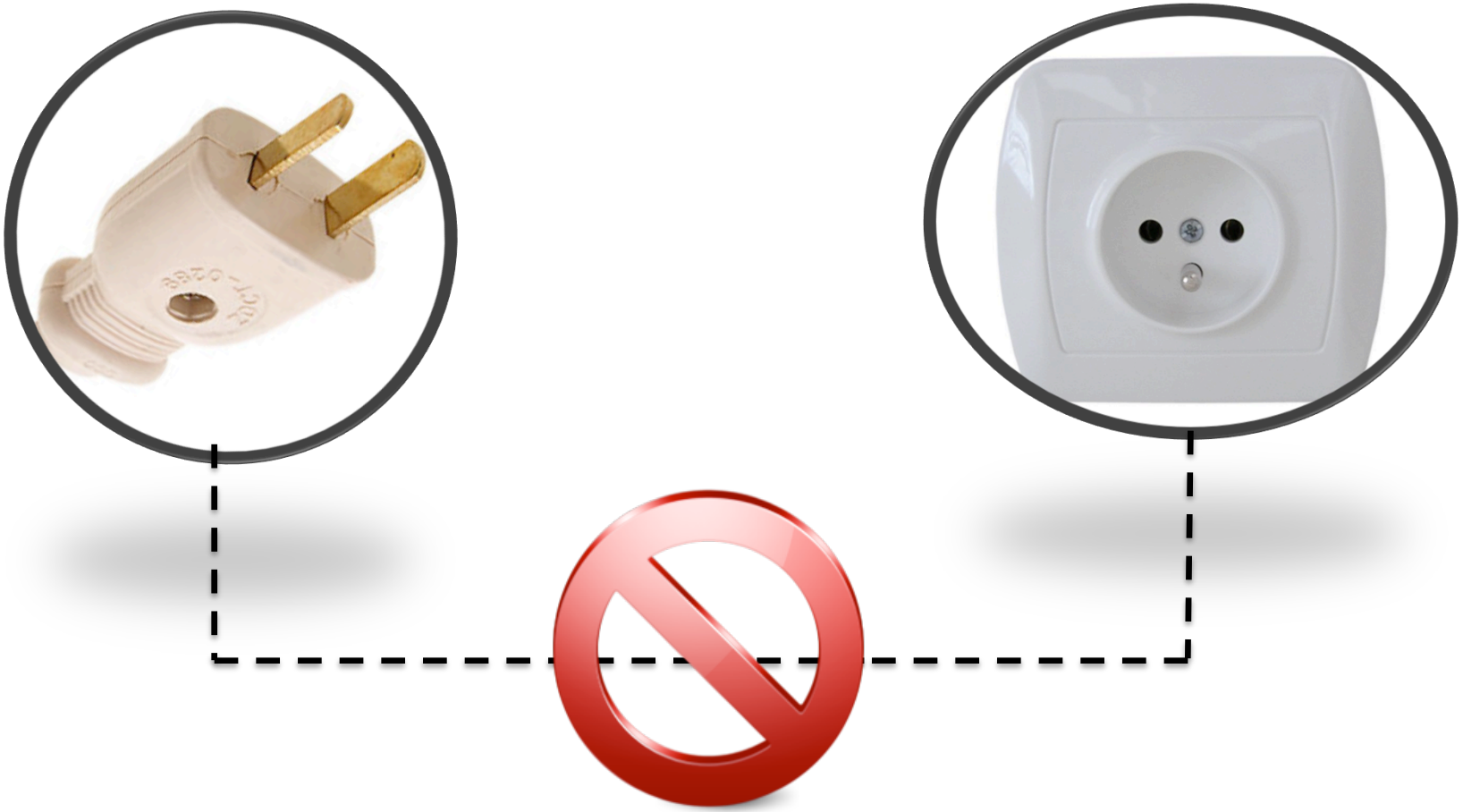
Why using it?

Make classes work with others without changing their code.

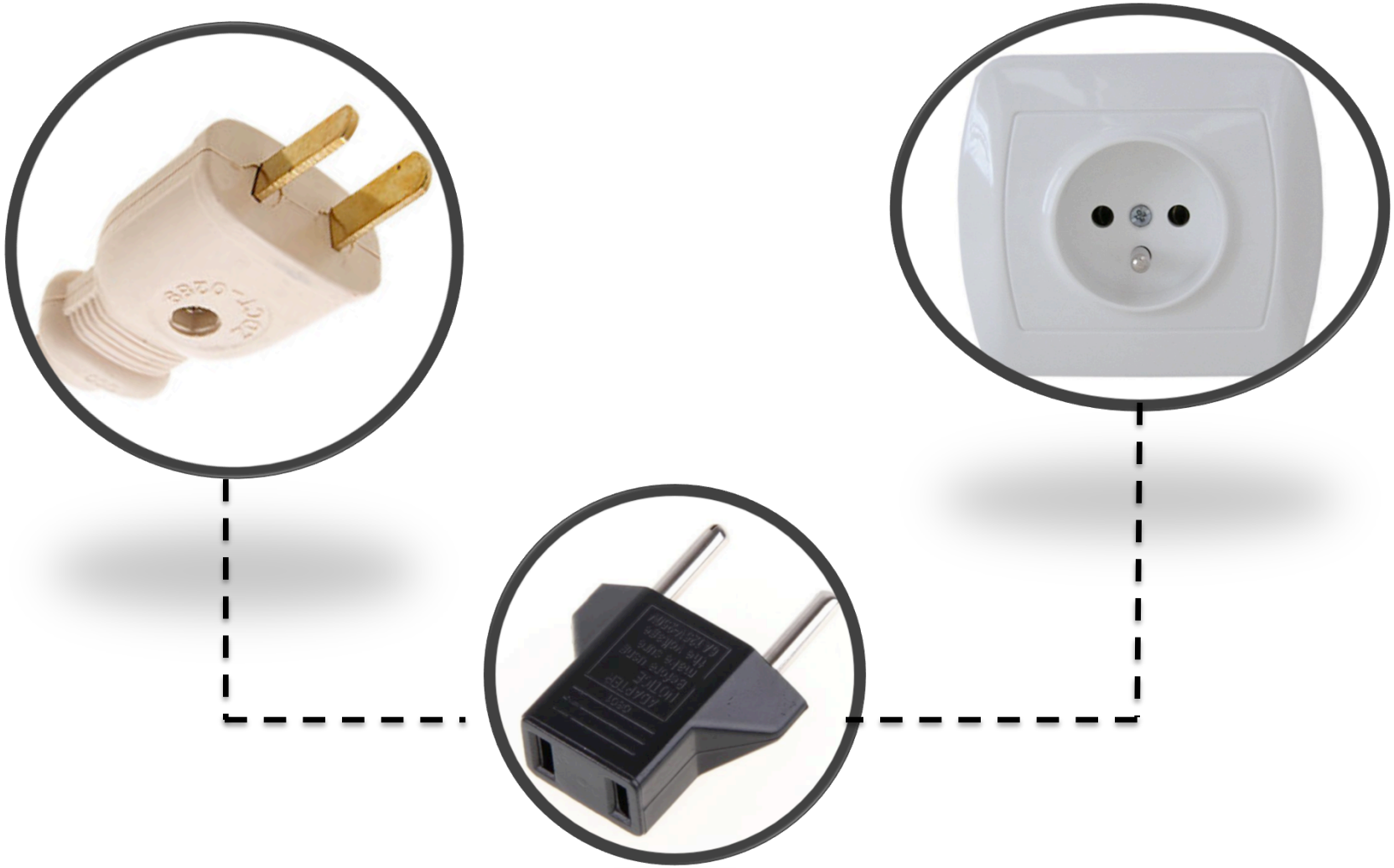
Examples

Adapting several database vendors
Adapting a new version of a REST API
Offering a backward compatibility layer

Combining heterogenous systems



Adapting one to the other



Security Component

Adapting the new CSRF API

New CSRF token management system since Symfony 2.4

Now done by the Security Component instead of the Form Component

Keeping a backward compatibility layer with the old API until it's removed in Symfony 3.0

The old CSRF Management API

```
namespace Symfony\Component\Form\Extension\Csrf\CsrfProvider;

interface CsrfProviderInterface
{
    public function generateCsrfToken($intention);

    public function isCsrfTokenValid($intention, $token);
}
```

The old CSRF Management API (< 2.4)

```php
namespace Symfony\Component\Form\Extension\Csrf\CsrfProvider;

class DefaultCsrfProvider implements CsrfProviderInterface
{
    // ...

    public function generateCsrfToken($intention)
    {
        return sha1($this->secret.$intention.$this->getSessionId());
    }

    public function isCsrfTokenValid($intention, $token)
    {
        return $token === $this->generateCsrfToken($intention);
    }
}
```

The old CSRF Management API (< 2.4)

```
$provider = new DefaultCsrfProvider('SecretCode');

$csrfToken = $provider
    ->generateCsrfToken('intention')
;

$csrfValid = $provider
    ->isCsrfTokenValid('intention', $token)
;
```

The new CSRF Management API (>= 2.4)

```php
namespace Symfony\Component\Security\Csrf;

interface CsrfTokenManagerInterface
{
    public function getToken($tokenId);

    public function refreshToken($tokenId);

    public function removeToken($tokenId);

    public function isTokenValid(CsrfToken $token);
}
```

Combining the old and new APIs for BC

```
class TwigRenderer extends FormRenderer implements TwigRendererInterface
{
    private $engine;

    public function __construct(
        TwigRendererEngineInterface $engine,
        $csrfTokenManager = null
    )
    {
        if ($csrfTokenManager instanceof CsrfProviderInterface) {
            $csrfTokenManager = new CsrfProviderAdapter($csrfTokenManager);
        }

        parent::__construct($engine, $csrfTokenManager);
        $this->engine = $engine;
    }
}
```

The CSRF Provider Adapter

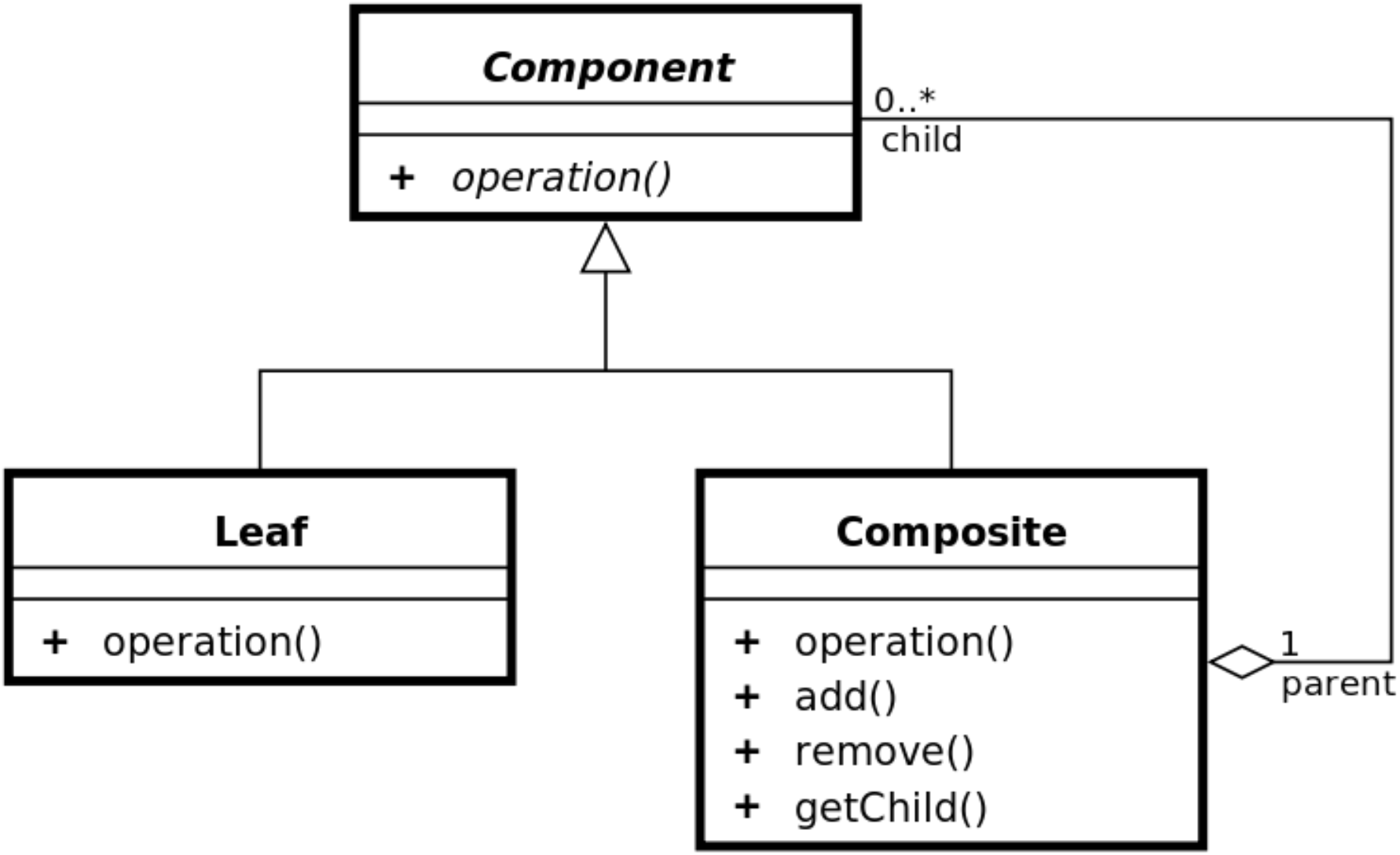
```
class CsrfProviderAdapter implements CsrfTokenManagerInterface
{
    private $csrfProvider;

    public function __construct(CsrfProviderInterface $csrfProvider)
    {
        $this->csrfProvider = $csrfProvider;
    }

   public function refreshToken($tokenId)
    {
        throw new BadMethodCallException('Not supported');
    }

    public function removeToken($tokenId)
    {
        throw new BadMethodCallException('Not supported');
    }
}
```

The CSRF Provider Adapter

```
class CsrfProviderAdapter implements CsrfTokenManagerInterface
{
    public function getToken($tokenId)
    {
        $token = $this->csrfProvider->generateCsrfToken($tokenId);
        return new CsrfToken($tokenId, $token);
    }

    public function isTokenValid(CsrfToken $token)
    {
        return $this->csrfProvider->isCsrfTokenValid(
            $token->getId(),
            $token->getValue()
        );
    }
}
```

Composite

Composite lets clients treat individual objects and compositions of objects uniformly.


Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

Why using it?

Representing trees of objects uniformely

Examples

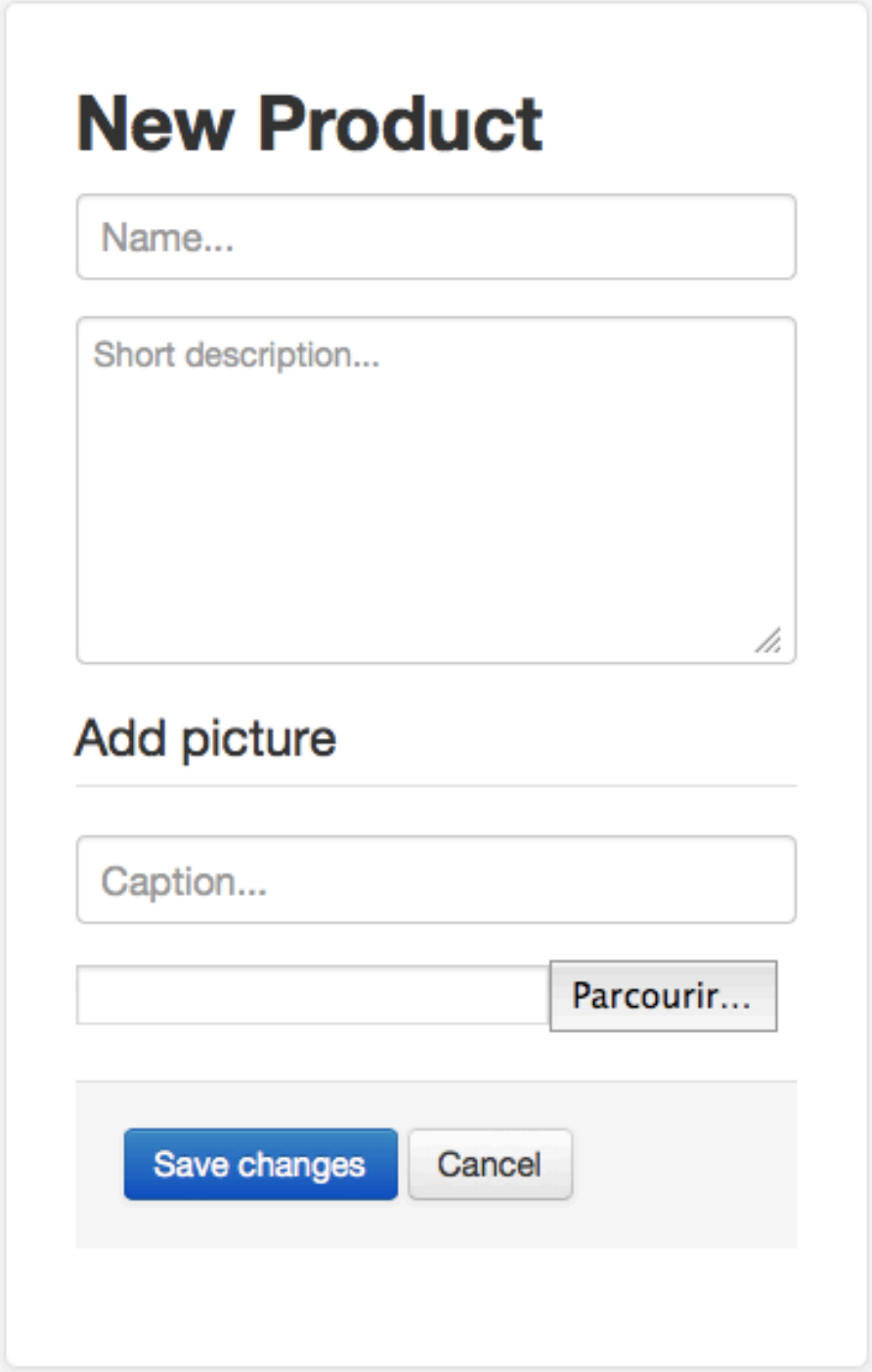
Representing a binary tree
Representing a multi level navigation bar
Parsing an XML/HTML document
Designing & validating nested forms
…

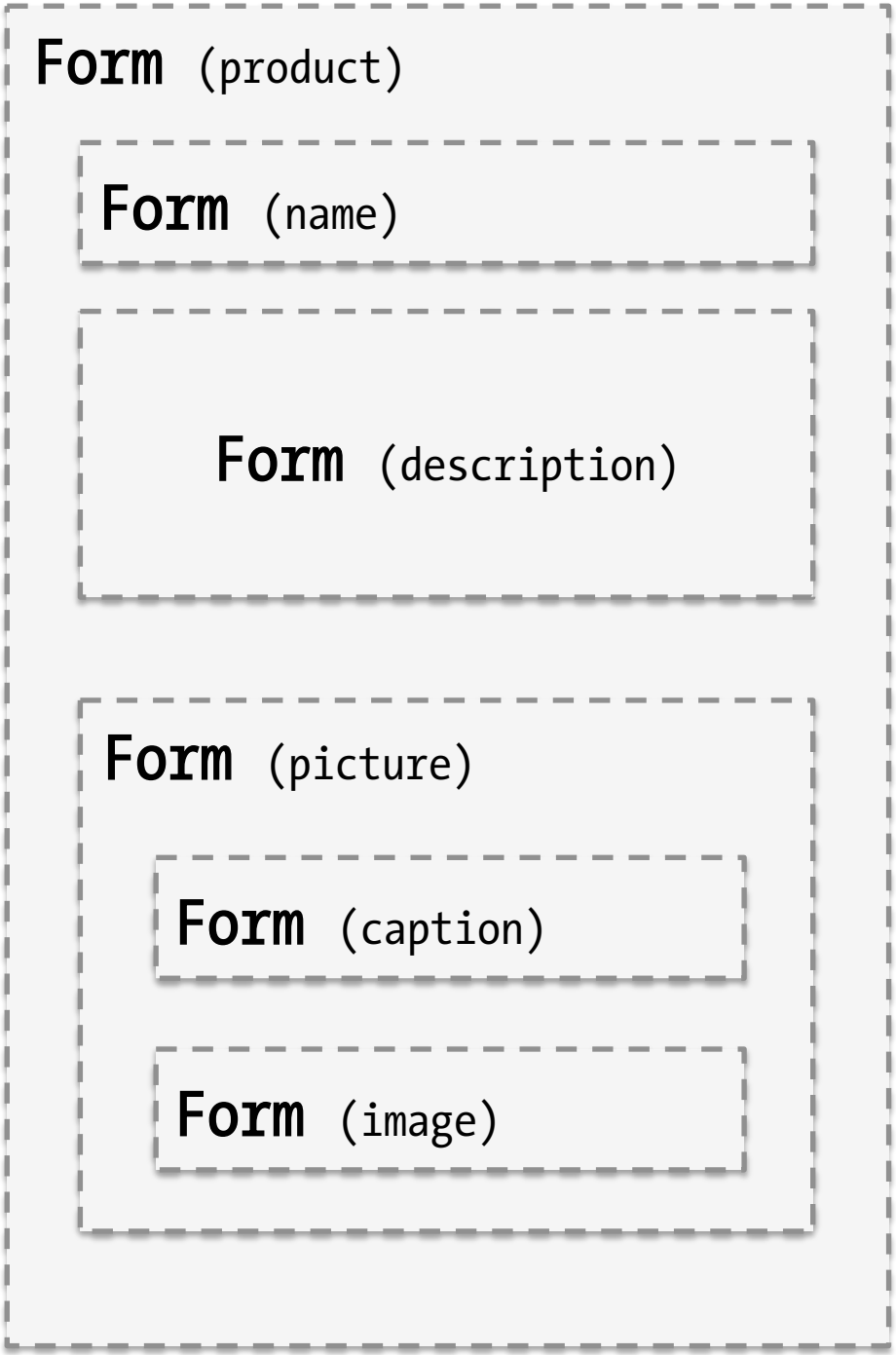
Form Component

Everything is a Form

Each element that composes a Symfony Form is an instance of the Form class.

Each **Form** instance keeps a reference to its parent **Form** instance and a collection of its children references.


New Product
Name...
Short description...
Add picture
Caption...
Parcourir...
Save changes
Cancel


Form (product)
Form (name)
Form (description)
Form (picture)
Form (caption)
Form (image)

The (simplified) Form class

```php
namespace Symfony\Component\Form;

class Form implements FormInterface
{
    private $name;

    public function __construct($name = null)
    {
        $this->name = $name;
    }

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

```php
namespace Symfony\Component\Form;

class Form implements FormInterface
{
    private $parent;
    private $children;

    public function add(FormInterface $child)
    {
        $this->children[$child->getName()] = $child;
        $child->setParent($this);

        return $this;
    }
}
```

Building the Form tree

```
$picture = new Form('picture');
$picture->add(new Form('caption'));
$picture->add(new Form('image'));

$form = new Form('product');
$form->add(new Form('name'));
$form->add(new Form('description'));
$form->add($picture);
```

Submitting the form data

```
$form->submit(array(
    'name' => 'Apple Macbook Air 11',
    'description' => 'The thinest laptop',
    'picture' => array(
        'caption' => 'The new Macbook Air.',
    ),
));
```

Submitting the form data

```
class Form implements FormInterface
{
    public function submit(array $data)
    {
        $this->data = $data;
        foreach ($this->children as $child) {
            if (isset($data[$child->getName()])) {
                $childData = $data[$child->getName()];
                $child->submit($childData);
            }
        }
    }
}
```

Decorator

Adding responsibilities to objects without subclassing their classes.

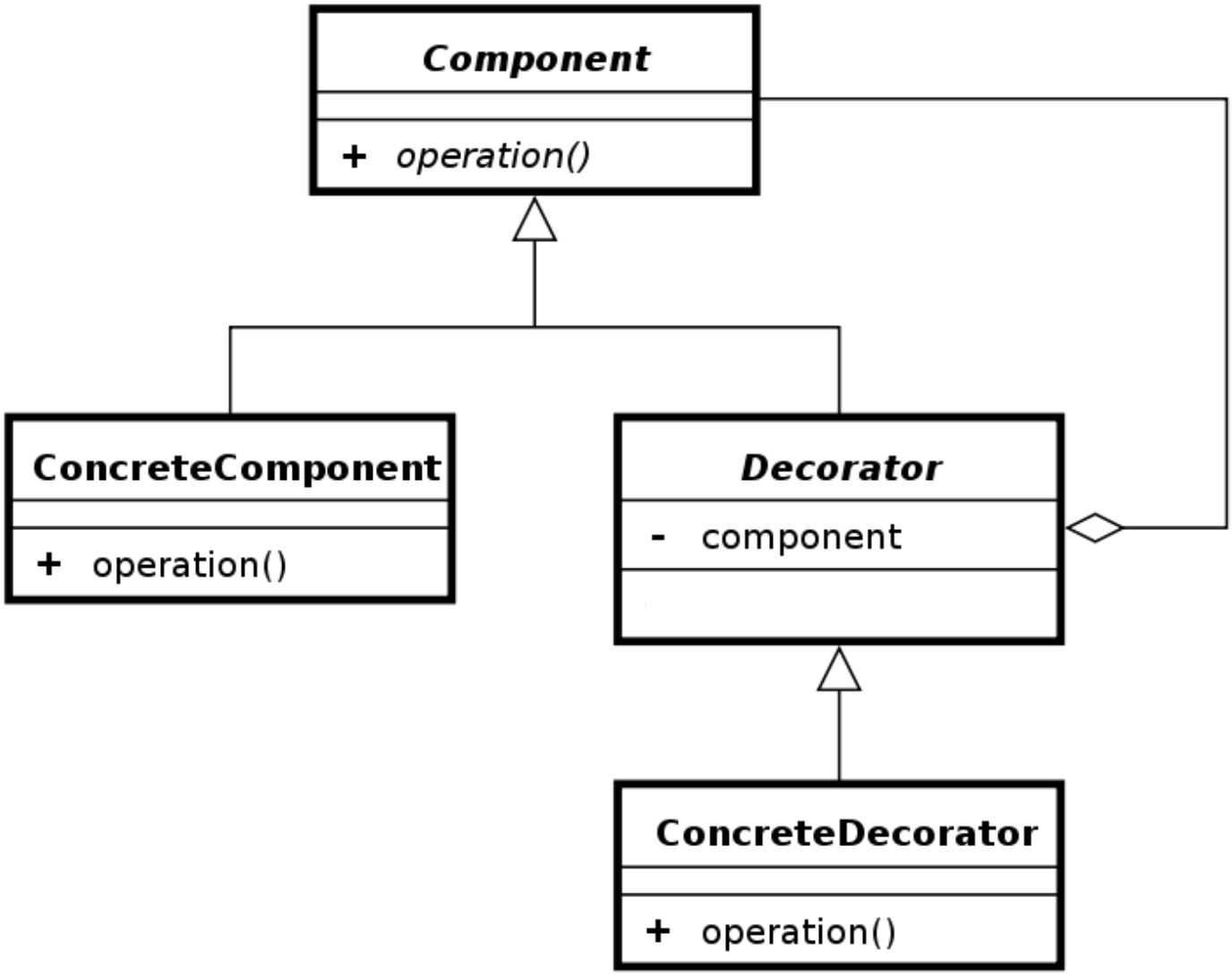
Why using it?

Extending objects without bloating the code
Making code reusable and composable
Avoiding vertical inheritance

Examples

Adding some caching capabilities
Adding some logging capabilities
Applying discount strategies to an order
Decorating/wrapping a string content

. . .

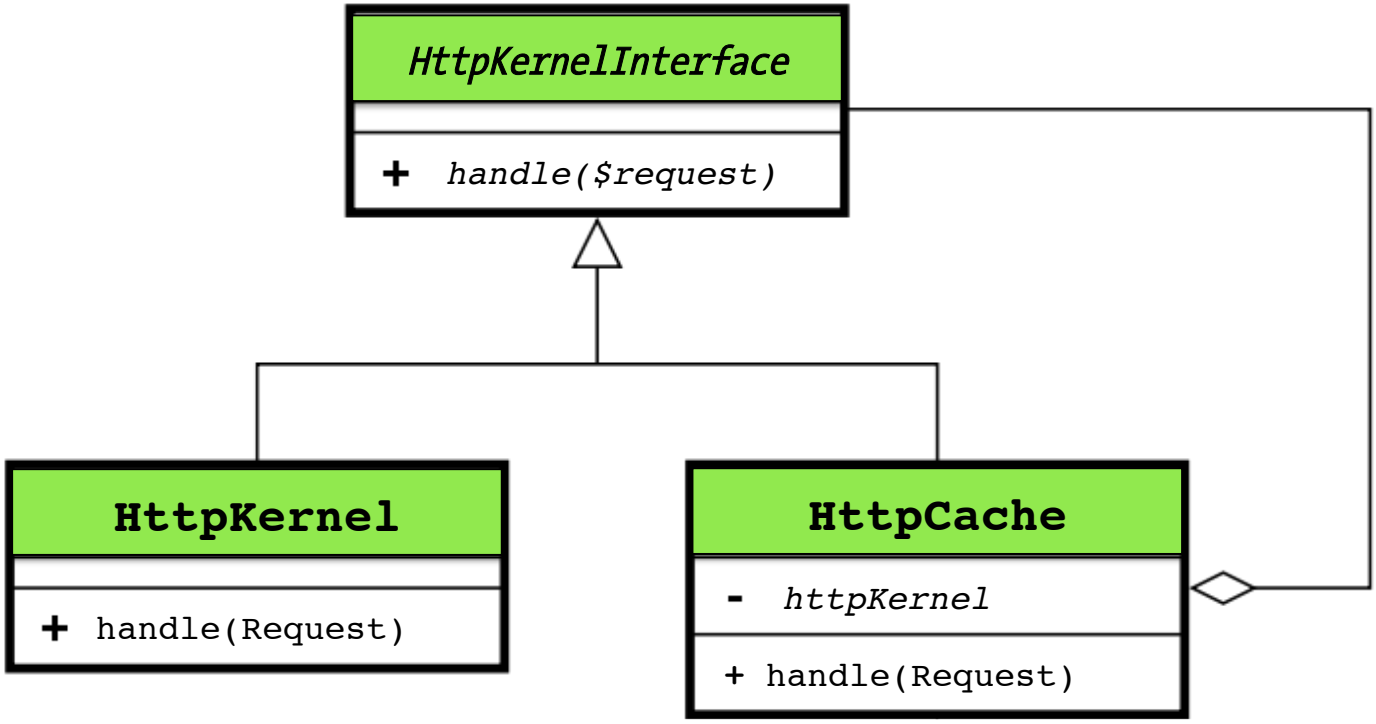

Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
ConcreteDecorator
+ operation()

HttpKernel Component

Adding an HTTP Caching Layer

The default implementation of the HttpKernel class doesn't support caching capabilities.

Symfony provides an HttpCache class to decorate an instance of HttpKernel in order to emulate an HTTP reverse proxy cache.


HttpKernelInterface
+ handle($request)
HttpKernel
+ handle(Request)
HttpCache
- httpKernel
+ handle(Request)

```php
// index.php

$dispatcher = new EventDispatcher();
$resolver = new ControllerResolver();
$store = new Store(__DIR__.'/http_cache');

$httpKernel = new HttpKernel($dispatcher, $resolver);
$httpKernel = new HttpCache($httpKernel, $store);

$httpKernel
    ->handle(Request::createFromGlobals())
    ->send()
;
```

```php
class HttpCache implements HttpKernelInterface
{
    public function handle(Request $request)
    {
        // ...
        if (!$request->isMethodSafe()) {
            $response = $this->invalidate($request, $catch);
        } elseif ($request->headers->has('expect')) {
            $response = $this->pass($request, $catch);
        } else {
            // Get response from the Store or fetch it with http kernel
            $response = $this->lookup($request, $catch);
        }

        // ...

       return $response;
    }
}
```

```php
class HttpCache implements HttpKernelInterface
{
    protected $httpKernel;

    protected function forward(Request $request, ...)
    {
        // ...
        $request->server->set('REMOTE_ADDR', '127.0.0.1');
        $response = $this->httpKernel->handle($request);

        // ...
        $this->processResponseBody($request, $response);

        return $response;
    }
}
```

Pros and cons

+ Easy way to extend an object capabilities
+ No need to change the underlying code

- Object construction becomes more complex
- Difficulty to test the concrete object type

Behavioral Patterns

Iterator

The iterator pattern allows to traverse a container and access its elements.

Why using it?

Easing iterations over collections of objects
Filtering records in a collection
…

Examples

Reading the content of directory recursively
Sorting items in a collection
Applying filters on items of a collection
Lazy fetch data from a datastore

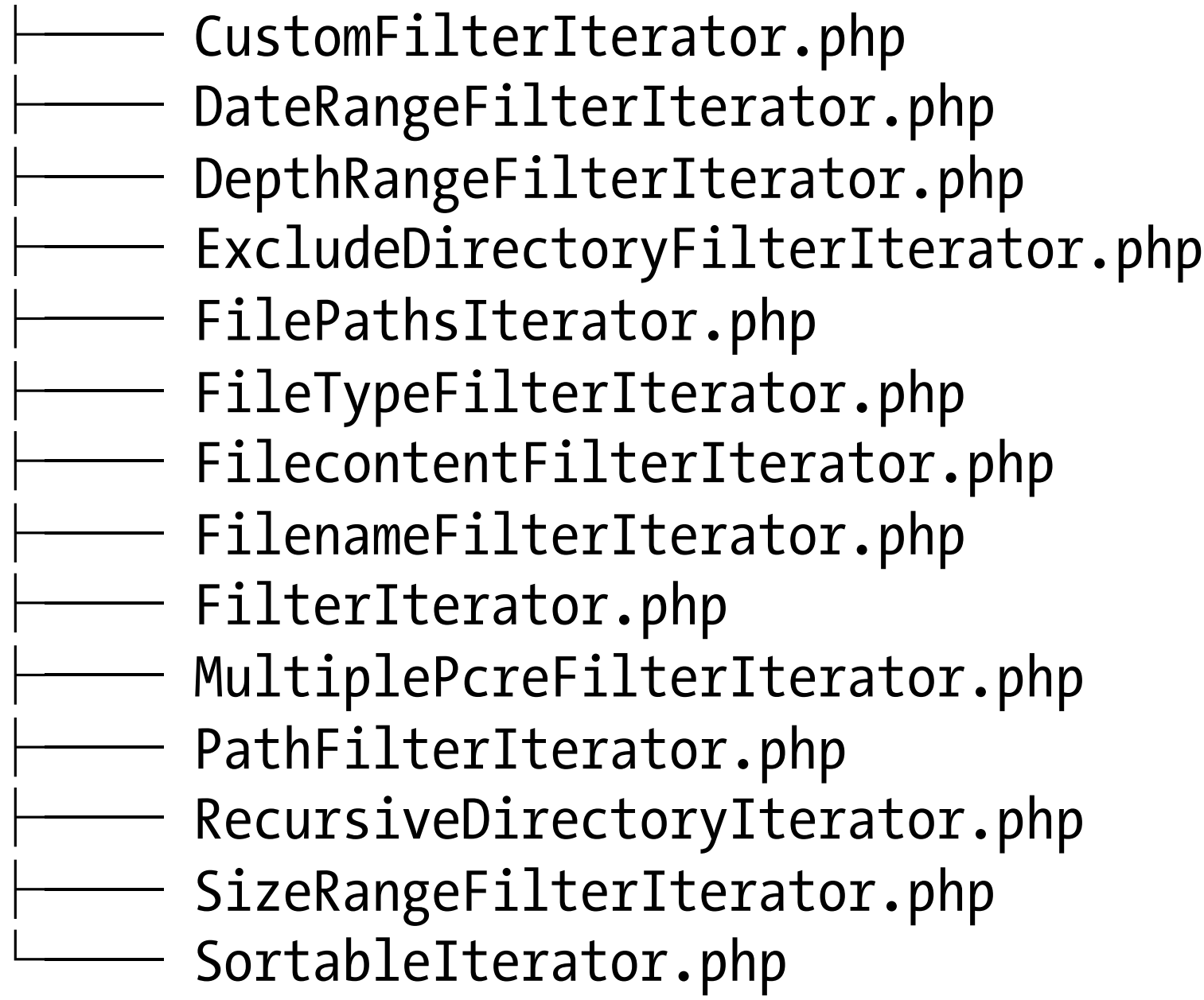
...

```
interface Iterator
{
    public function current();
    public function next();
    public function rewind();
    public function valid();
    public function key();
}
```

Finder Component

The Finder

```
$iterator = Finder::create()
    ->files()
    ->name('*.php')
    ->depth(0)
    ->size('>= 1K')
    ->in(__DIR__);

foreach ($iterator as $file) {
    print $file->getRealpath()."\n";
}
```


CustomFilterIterator.php
DateRangeFilterIterator.php
DepthRangeFilterIterator.php
ExcludeDirectoryFilterIterator.php
FilePathsIterator.php
FileTypeFilterIterator.php
FilecontentFilterIterator.php
FilenameFilterIterator.php
FilterIterator.php
MultiplePcreFilterIterator.php
PathFilterIterator.php
RecursiveDirectoryIterator.php
SizeRangeFilterIterator.php
SortableIterator.php

Sorting a list of files

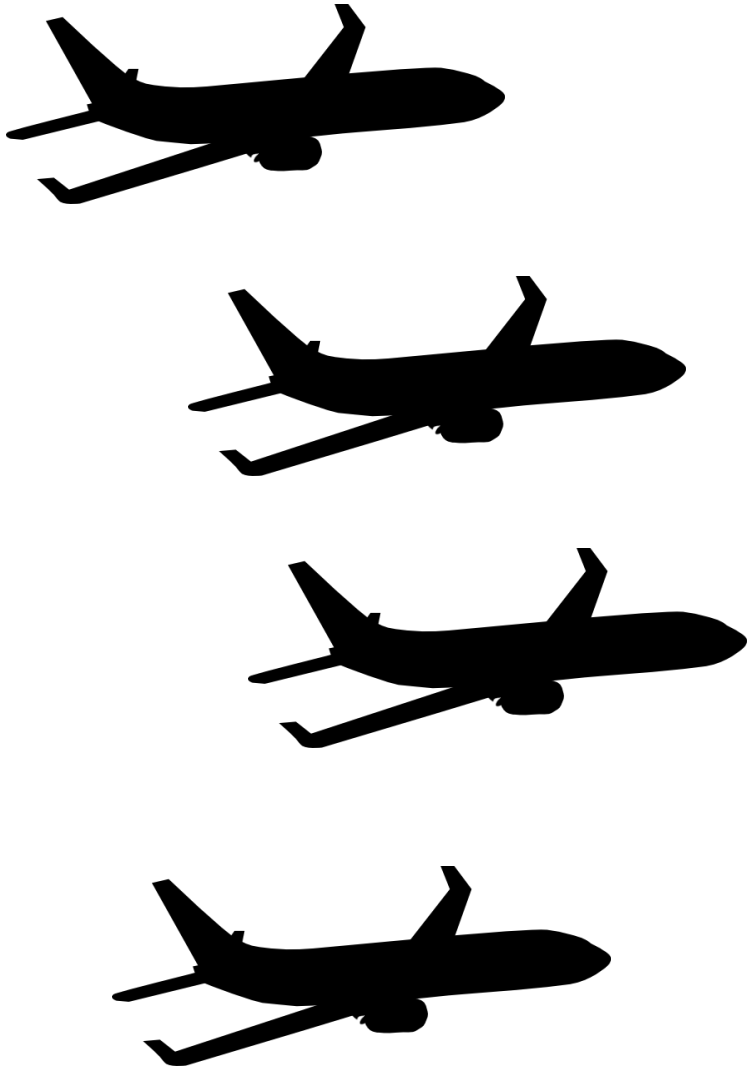
```
use Symfony\Component\Finder\Iterator\SortableIterator;
use Symfony\Component\Finder\Iterator\RecursiveDirectoryIterator;

$dirIterator = new \RecursiveIteratorIterator(
    new RecursiveDirectoryIterator(
        __DIR__,
        \RecursiveDirectoryIterator::SKIP_DOTS
    )
);
$dirIterator->setMaxDepth(0);

$iterator = new SortableIterator(
    $dirIterator,
    SortableIterator::SORT_BY_NAME
);
```

Finder Adapter

```
class PhpAdapter extends AbstractAdapter
{
    public function searchInDirectory($dir)
    {
        $iterator = new \RecursiveIteratorIterator(...);

        if ($this->minDepth > 0 || $this->maxDepth < PHP_INT_MAX) {
            $iterator = new DepthRangeFilterIterator($iterator, $this->minDepth, $this->maxDepth);
        }

        if ($this->mode) {
            $iterator = new FileTypeFilterIterator($iterator, $this->mode);
        }

        if ($this->exclude) {
            $iterator = new ExcludeDirectoryFilterIterator($iterator, $this->exclude);
        }

        // ... Other iterators are added

        return $iterator;
    }
}
```

Mediator

The mediator pattern defines an object that encapsulates how a set of objects interact.



Why using it?

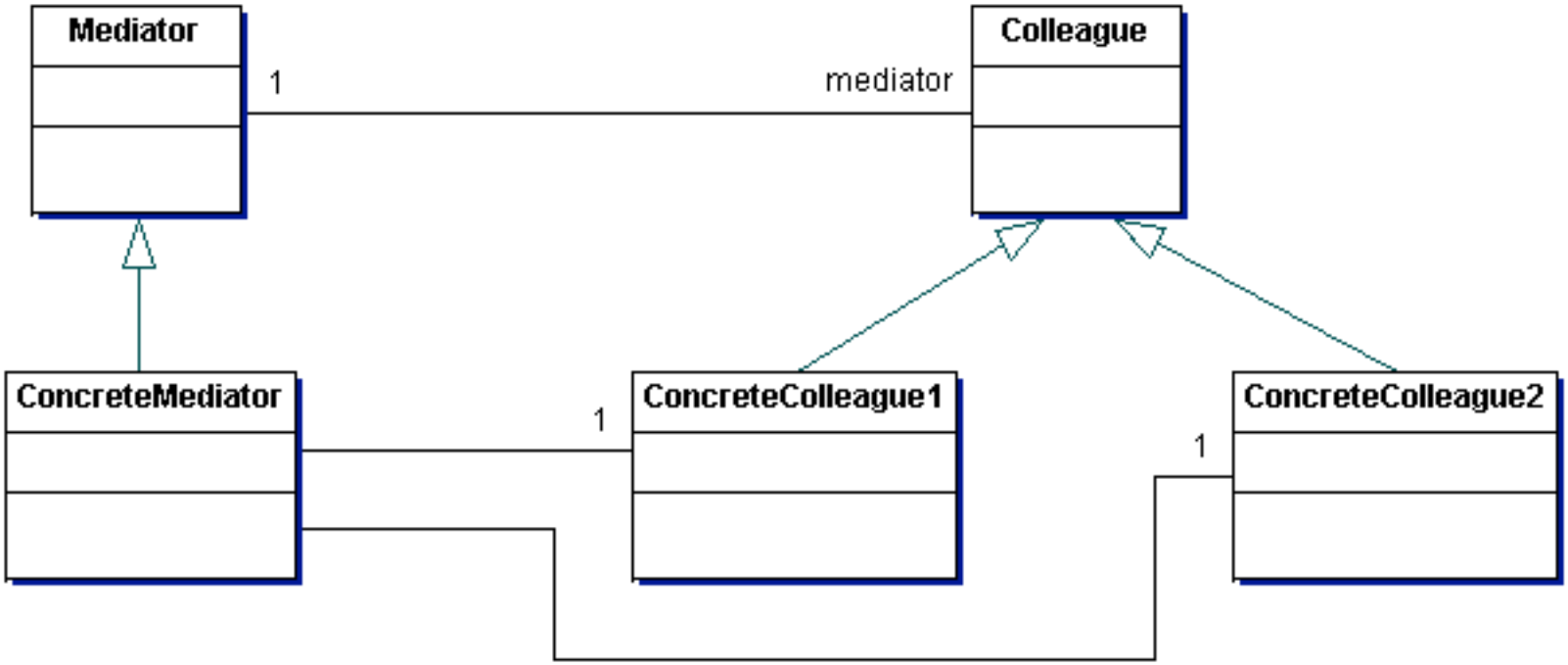
Decoupling large pieces of code
Hooking a third party algorithm to an existing one
Easing unit testability of objects
Filtering user input data
…

Examples

Dispatching events when an object's state changes

Hooking new responsabilities to a model object

Filtering some input data


Mediator
1
mediator
Colleague
ConcreteMediator
1
ConcreteColleague1
1
ConcreteColleague2

```php
class OrderService
{
    public function confirmOrder(Order $order)
    {
        $order->status = 'confirmed';
        $order->save();

        if ($this->logger) {
            $this->logger->log('New order...');
        }

        $mail = new Email();
        $mail->recipient = $order->getCustomer()->getEmail();
        $mail->subject = 'Your order!';
        $mail->message = 'Thanks for ordering...';
        $this->mailer->send($mail);

        $mail = new Email();
        $mail->recipient = 'sales@acme.com';
        $mail->subject = 'New order to ship!';
        $mail->message = '...';
        $this->mailer->send($mail);
    }
}
```

What's wrong with this code?

Too many responsabilities

Not easy extensible

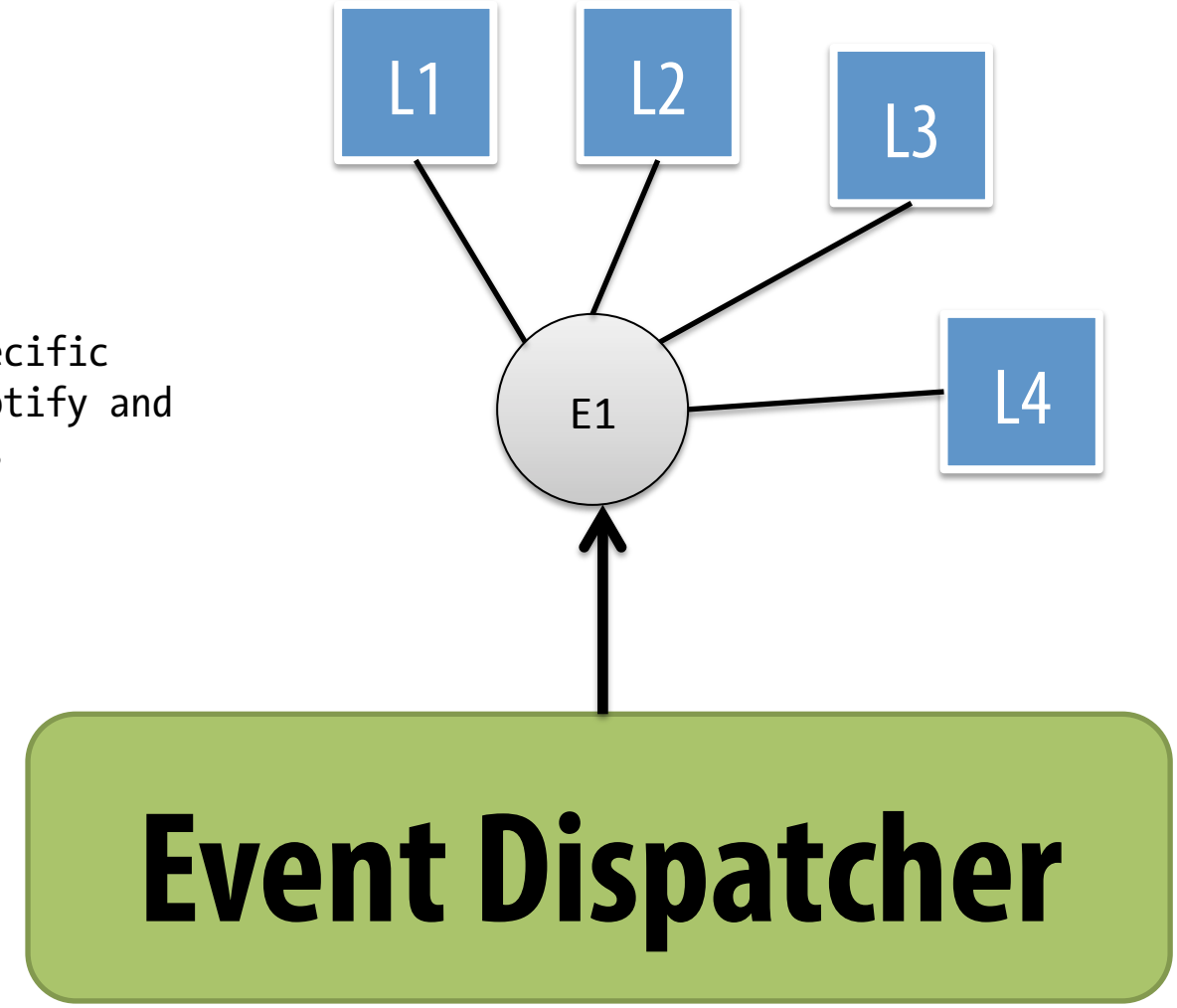
Difficulty to unit test

Bloated code

EventDispatcher Component

The event dispatcher manages connections between a subject and its attached observers.

Emits a specific event to notify and execute its listeners.



Benefits of the Mediator

Extremly simple and easy to use

Decouple objects from each other

Make code truly extensible

Easy to add responsabilities to an object

Main usages

Plugin / hook

Filtering data

Decoupling code

The Event Dispatcher Class

```
namespace Symfony\Component\EventDispatcher;

class EventDispatcher
{
    function dispatch($eventName, Event $event = null);

    function getListeners($eventName);
    function hasListeners($eventName);

    function addListener($eventName, $listener, $priority = 0);
    function removeListener($eventName, $listener);

    function addSubscriber(EventSubscriberInterface $subscriber);
    function removeSubscriber(EventSubscriberInterface $subscriber);
}
```

The Event Class

```
class Event
{
    function isPropagationStopped();
    function stopPropagation();
}
```

Dispatching Events

```php
$dp = new EventDispatcher();

$dp->dispatch('order.paid', new Event());
$dp->dispatch('order.cancelled', new Event());
$dp->dispatch('order.refunded', new Event());
```

Connecting listeners to events

```php
$listener1 = new CustomerListener($mailer);
$listener2 = new SalesListener($mailer);
$listener3 = new StockListener($stockHandler);

$dp = new EventDispatcher();

$dp->addListener('order.paid', [ $listener1, 'onOrderPaid' ]);
$dp->addListener('order.paid', [ $listener2, 'onOrderPaid' ]);
$dp->addListener('order.paid', [ $listener3, 'onOrderPaid' ], 100);

$dp->dispatch('order.paid', new Event());
```

Implementing listener classes

```
class CustomerListener
{
    private $mailer;

    public function __construct(\Swift_Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function onOrderPaid(Event $event)
    {
        $mail = $this->mailer->createMessage(...);
        $this->mailer->send($mail);
    }
}
```

Creating custom event objects

```php
use Symfony\Component\EventDispatcher\Event;

class OrderEvent extends Event
{
    private $order;

    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    public function getOrder()
    {
        return $this->order;
    }
}
```

Accessing data from custom events

```
class CustomerListener
{
    // ...

    public function onOrderPaid(OrderEvent $event)
    {
        $order = $event->getOrder();
        $customer = $order->getCustomer();

        $mail = $this->mailer->createMessage(...);
        $this->mailer->send($mail);
    }
}
```

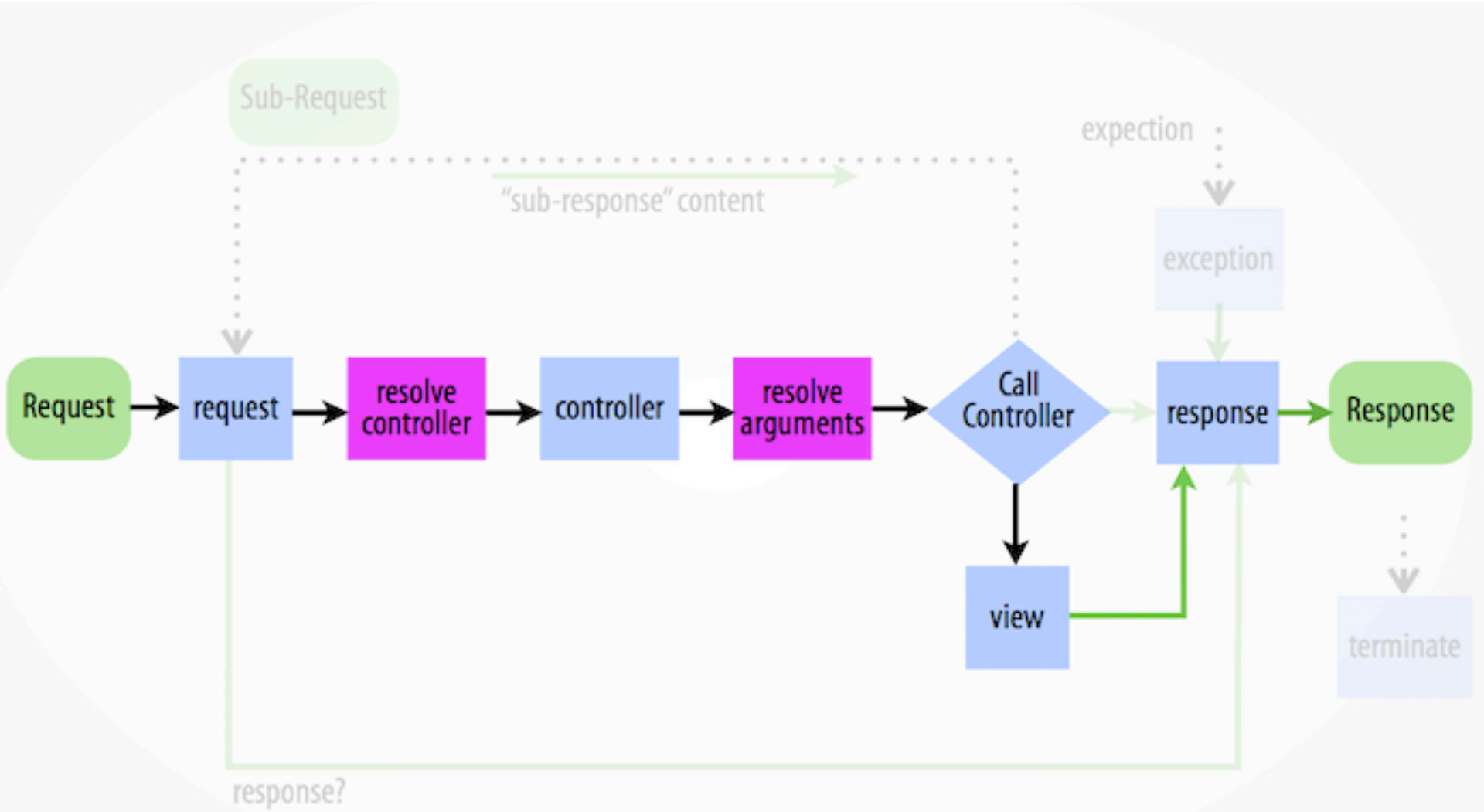
Decoupling the code with events

```php
class OrderService
{
    private $dispatcher;

    public function __construct(EventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function confirmOrder(Order $order)
    {
        $order->status = 'confirmed';
        $order->save();

        $event = new OrderEvent($order);
        $this->dispatcher->dispatch('order.paid', $event);
    }
}
```

Kernel Events

Event name	Meaning
`kernel.request`	Filters the incoming HTTP request
`kernel.controller`	Initializes the controller before it's executed
`kernel.view`	Generates a template view
`kernel.response`	Prepares the HTTP response nefore it's sent
`kernel.exception`	Handles all caught exceptions
`kernel.terminate`	Terminates the kernel

HttpKernel Events Workflow



Form Events

Event name	Meaning
`form.pre_bind`	Changes submitted data before they're bound to the form
`form.bind`	Changes data into the normalized representation
`form.post_bind`	Changes the data after they are bound to the form
`form.pre_set_data`	Changes the original form data
`form.post_set_data`	Changes data after they were mapped to the form

Security Events

Event name	Meaning
`security.interactive_login`	User is successfully authenticated.
`security.switch_user`	User switched to another user account.
`security.authentication.success`	User is authenticated by a provider
`security.authentication.failure`	User cannobt be authenticated by a provider

Strategy

The strategy pattern encapsulates algorithms of the same nature into dedicated classes to make them interchangeable.

HttpKernel Component

The Profiler stores the collected data into a storage engine.

The persistence algorithms must be interchangeable so that we can use any types of data storage (filesystem, database, nosql stores…).

Designing the profiler storage strategy

```
namespace Symfony\Component\HttpKernel\Profiler;

interface ProfilerStorageInterface
{
    function find($ip, ...);
    function read($token);
    function write(Profile $profile);
    function purge();
}
```

The default profiler storage

```php
namespace Symfony\Component\HttpKernel\Profiler;

class FileProfilerStorage implements ProfilerStorageInterface
{
    public function read($token)
    {
        if (!$token || !file_exists($file = $this->getFilename($token))) {
            return;
        }

        return $this->createProfileFromData($token, unserialize(file_get_contents($file)));
    }

    public function write(Profile $profile)
    {
        $file = $this->getFilename($profile->getToken());
        $data = array(...);
        // ...
        return false !== file_put_contents($file, serialize($data));
    }
}
```

The PDO profiler storage

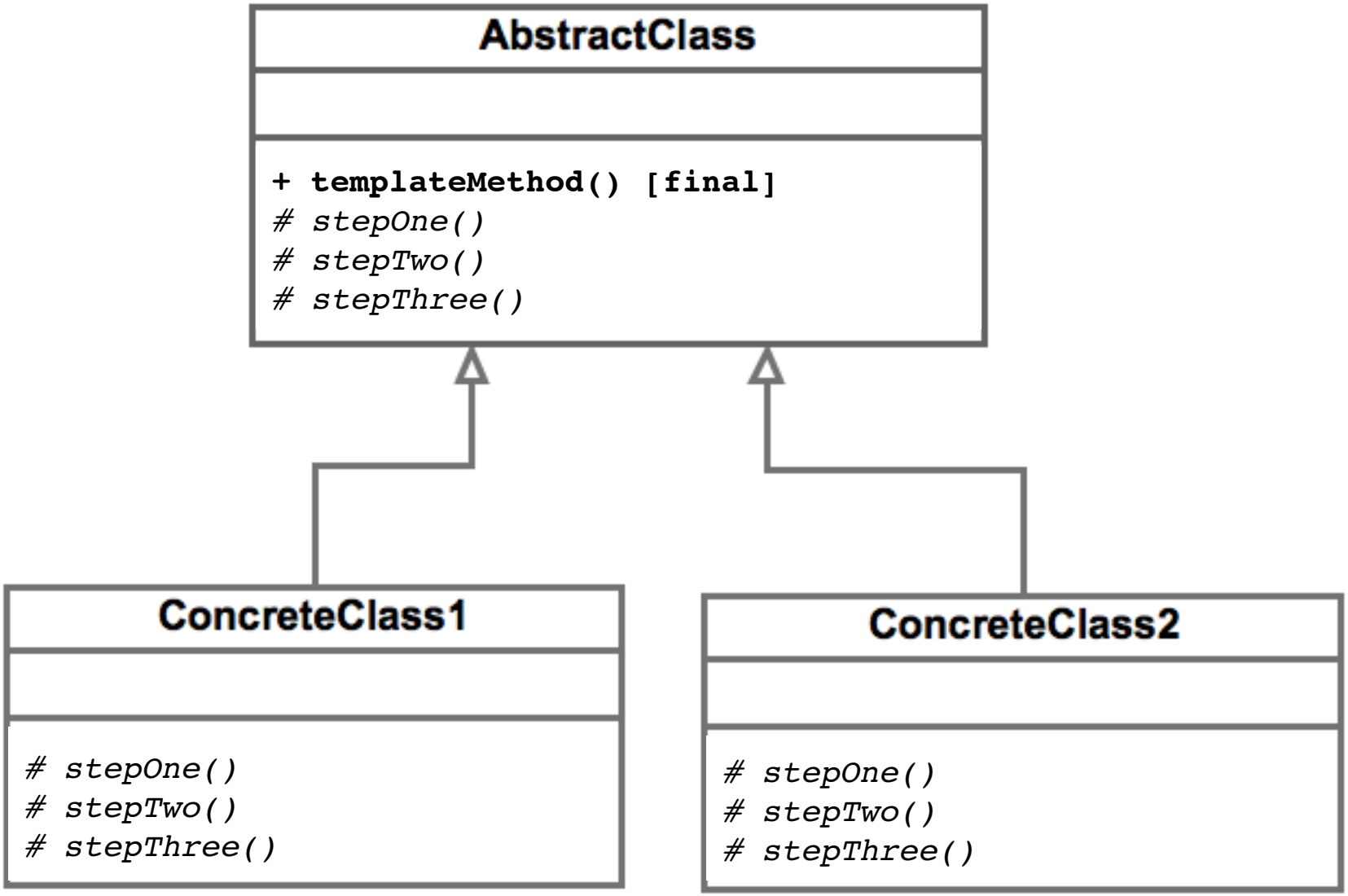
```php
namespace Symfony\Component\HttpKernel\Profiler;

abstract class PdoProfilerStorage implements ProfilerStorageInterface
{
    public function read($token)
    {
        $db = $this->initDb();
        $args = array(':token' => $token);
        $data = $this->fetch($db, 'SELECT ...', $args);
        $this->close($db);
        if (isset($data[0]['data'])) {
            return $this->createProfileFromData($token, $data[0]);
        }
    }
}
```

Using the Profiler

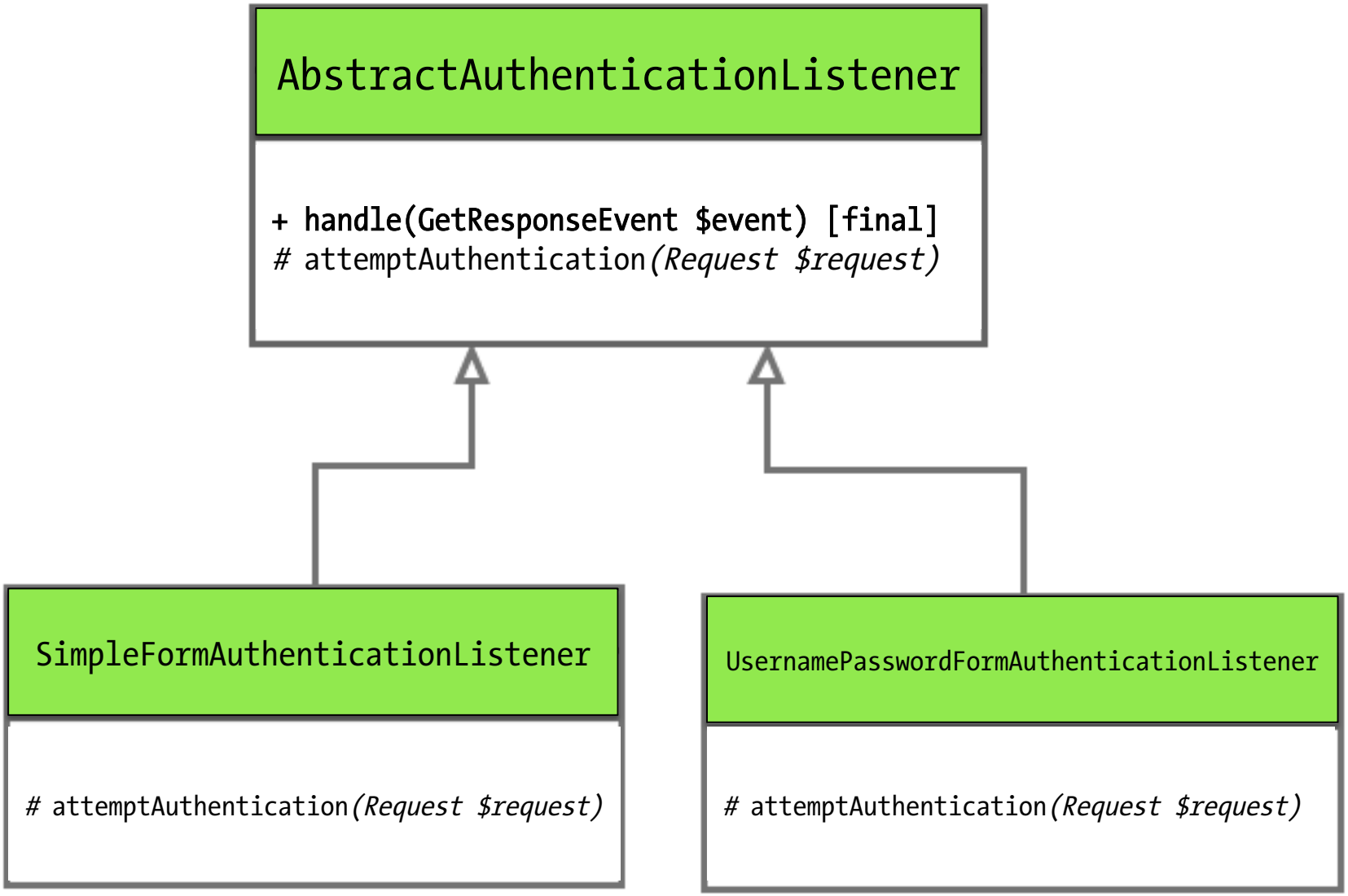
```
// Profiled data to be saved
$profile = new Profile('abcdef');

// Filesystem storage
$fsStorage = new FileProfilerStorage('/tmp/profiler');
$profiler = new Profiler($fsStorage);
$profiler->saveProfile($profile);

// Mysql database storage
$dsn = 'mysql:host=localhost';
$dbStorage = new MySQLProfilerStorage($dsn);
$profiler = new Profiler($dbStorage);
$profiler->saveProfile($profile);
```

Template Method

Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.


AbstractClass
+ templateMethod() [final]
# stepOne()
# stepTwo()
# stepThree()
ConcreteClass1
stepOne()
stepTwo()
stepThree()
ConcreteClass2
stepOne()
stepTwo()
stepThree()

Security Component


AbstractAuthenticationListener
+ handle(GetResponseEvent $event) [final]
# attemptAuthentication(Request $request)
SimpleFormAuthenticationListener
attemptAuthentication(Request $request)
UsernamePasswordFormAuthenticationListener
attemptAuthentication(Request $request)

```php
abstract class AbstractAuthenticationListener implements ListenerInterface
{
    final public function handle(GetResponseEvent $event)
    {
        // …
        try {
            // …
            if (null === $returnValue = $this->attemptAuthentication($request)) {
                return;
            }
            // …
        } catch (AuthenticationException $e) {
            $response = $this->onFailure($event, $request, $e);
        }
        $event->setResponse($response);
    }

    abstract protected function attemptAuthentication(Request $request);
}
```

The Simple Form Authentication Listener

```php
namespace Symfony\Component\Security\Http\Firewall;

class SimpleFormAuthenticationListener extends AbstractAuthenticationListener
{
    protected function attemptAuthentication(Request $request)
    {
        // ...
        $token = $this->simpleAuthenticator->createToken(
            $request,
            trim($request->get('_username')),
            $request->get('_password')
        );

        return $this->authenticationManager->authenticate($token);
    }
}
```

Custom Authentication Listener

```
namespace Acme\SsoBundle\Security\Core\Authentication;

class SsoAuthenticationListener extends AbstractAuthenticationListener
{
    protected function attemptAuthentication(Request $request)
    {
        if (!$ssoToken = $request->query->get('ssotoken')) {
            return;
        }

        $token = new SsoToken($ssoToken);

        return $this->authenticationManager->authenticate($token);
    }
}
```

Conclusion…


QUESTIONS