SensioLabs

Symfony

Hacking & Extending Symfony2

SF2C3

Events Managenent

```
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);
    }
}
```

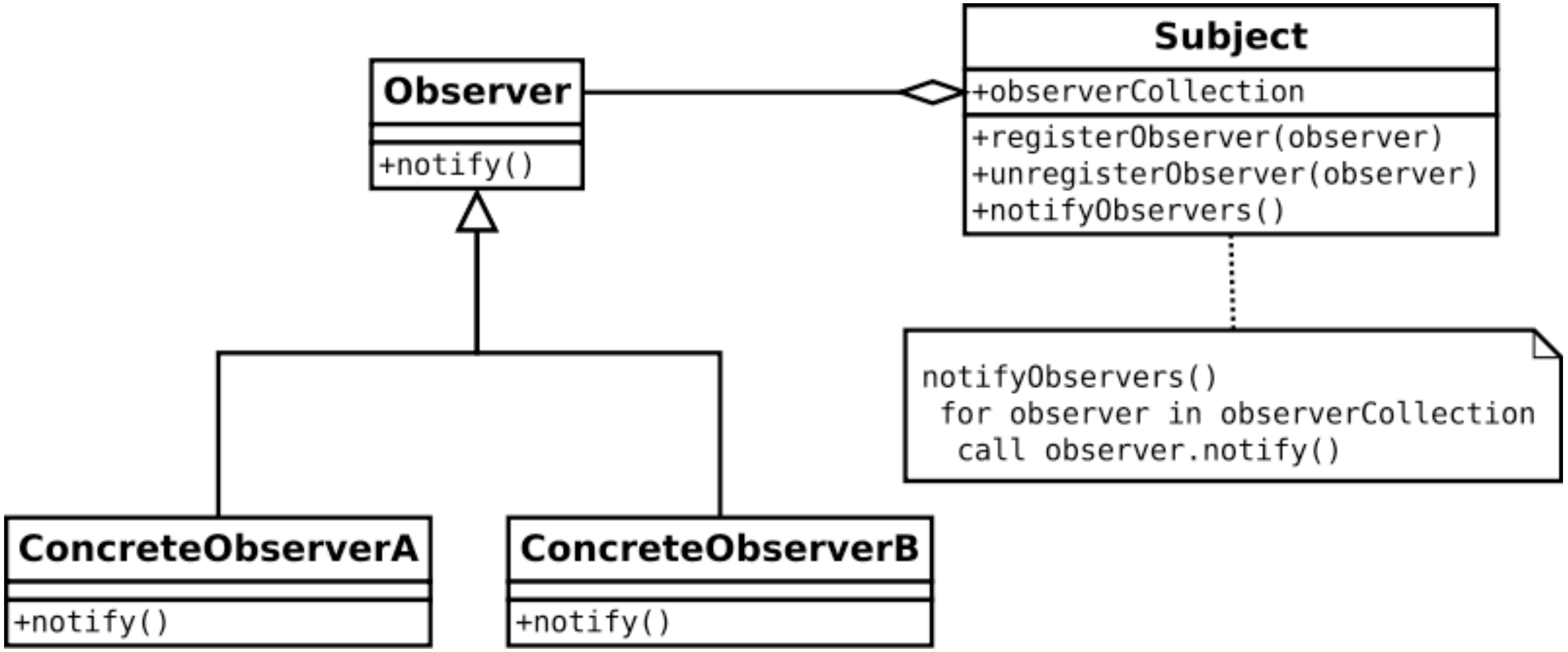
Too much coupling and responsabilities

Main drawbacks

- Tight coupling between classes
- OrderService class has too much responsibilities
- Evolutivity and extension are limited
- Maintaining the code is difficult

The Observer Design Pattern

A subject, the observable, emits a signal to a list of modules known as observers.


Observer
+notify()
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
notifyObservers()
 for observer in observerCollection
  call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

```php
interface ObserverInterface
{
    function notify(
        ObservableInterface $subject,
        OrderInterface $order
    );
}

interface ObservableInterface
{
    function attach(ObserverInterface $observer);
    function notifyObservers(OrderInterface $order);
}
```

Decoupling classes with observers

```
class LoggerHandler implements ObserverInterface
{
    public $logger;

    public function notify(
        ObservableInterface $subject,
        OrderInterface $order,
    )
    {
        $reference = $order->getReference();
        $this->logger->log('New order #'. $reference);
    }
}
```

Decoupling classes with observers

```php
class CustomerNotifier implements ObserverInterface
{
    public $mailer;

    public function notify(
        ObservableInterface $subject,
        OrderInterface $order,
    )
    {
        $mail = new Email();
        $mail->recipient = $order->getCustomer()->getEmail();
        $mail->subject = 'Your order!';
        $mail->message = 'Thanks for ordering...';
        $this->mailer->send($mail);
    }
}
```

Linking observers to the observable subject

```php
class OrderService implements ObservableInterface
{
    // ...
    private $observers;

    public function attach(ObserverInterface $observer)
    {
        $this->observers[] = $observer;
    }

    public function notifyObservers(OrderInterface $order)
    {
        foreach ($this->observers as $observer) {
            $observer->notify($this, $order);
        }
    }
}
```

Notifying the attached observers

```
class Order
{
    public function confirm()
    {
        $this->status = 'confirmed';
        $this->save();
    }
}
```

Notifying the attached observers

```
class OrderService implements ObservableInterface
{
    public function confirmOrder(Order $order)
    {
        $order->confirm();
        $this->notifyObservers($order);
    }
}
```

Notifying the attached observers

```php
$service = new OrderService();
$service->attach(new LoggerNotifier($logger));
$service->attach(new CustomerNotifier($mailer));
$service->attach(new SalesNotifier($mailer));

$order = new Order();
$order->customer = $customer;
$order->amount = 150;

$service->confirmOrder($order);
```

Main advantages

- Objects are less coupled together
- Easy to attach new « responsabilities »
- Easy to remove a « responsibility »
- Easy to maintain and make evolve

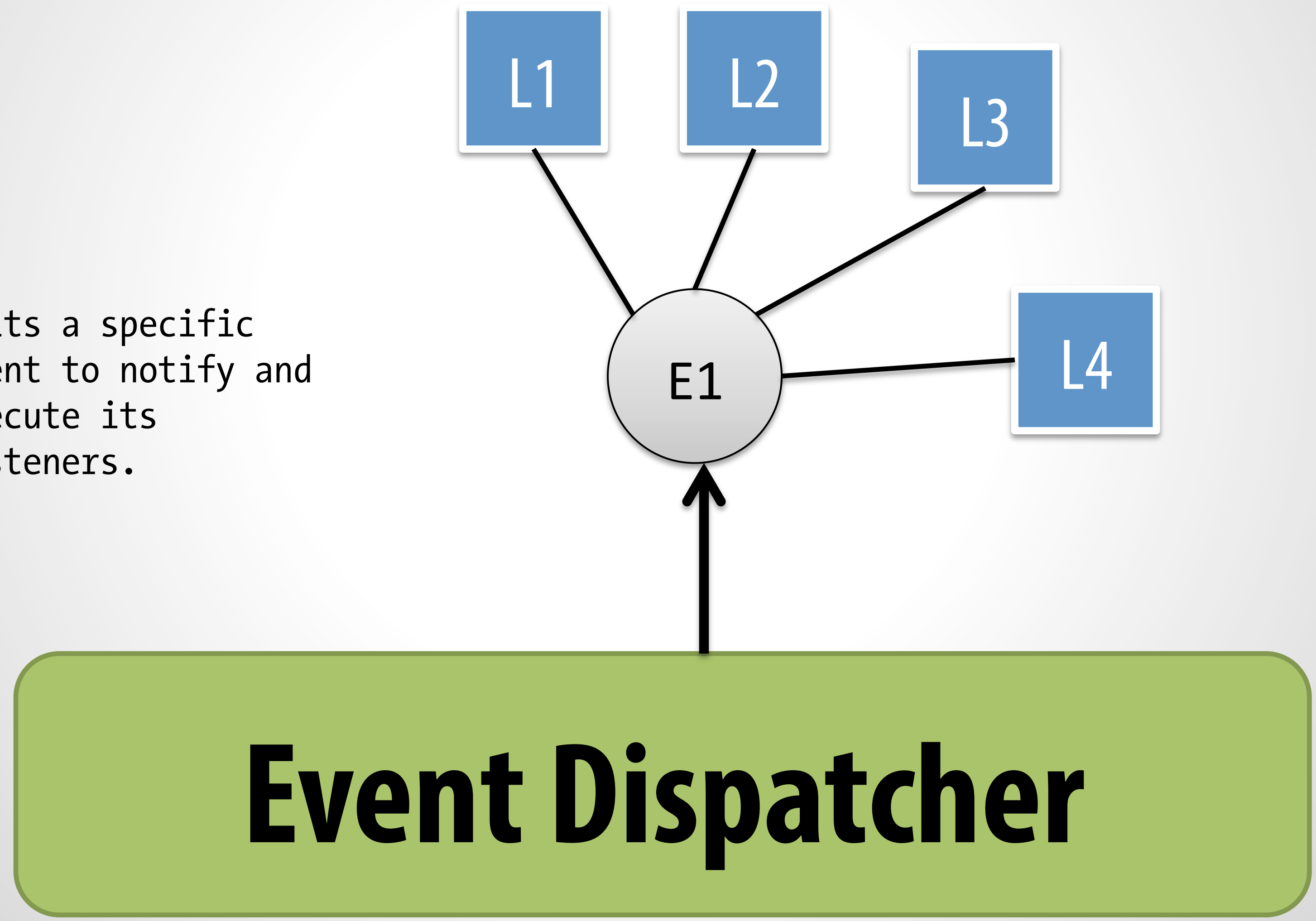
The EventDispatcher component

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

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

$dp = new EventDispatcher();

$dp->addListener('event.name', function ($event) {
    // do whatever you want...
});

$dp->addListener('event.name', function ($event) {
    // do whatever you want...
});

$dp->dispatch('event.name', new Event());
```

The Event Dispatcher in Action

Emits a specific event to notify and execute its listeners.



The EventDispatcher 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);
}
```

Getting the EventDispatcher Service

```
$container->get('event_dispatcher');
```

The Event

The dispatched event is an object, which carries all the needed data to retrieve into each listener.

The Event Class

```php
namespace Symfony\Component\EventDispatcher;

class Event
{
    public function isPropagationStopped();
    public function stopPropagation();

    public function setDispatcher(EventDispatcher $dispatcher);
    public function getDispatcher();

    public function getName();
    public function setName($name);
}
```

The Listener

The listener can be any valid PHP « callable » like a function name, an instance method, a static method and even a lambda function or a closure.

Well known events to listen to in Symfony2

The 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

The Forms 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

The Security User Events

Event name	Meaning
`security.interactive_login`	Triggered when the user manages to authenticate.
`security.switch_user`	Triggered when a user switches to another user's account if he has the permission to do it.

The Security Authentication Events

Event name	Meaning
`security.authentication.success`	When authentication is succesful
`security.authentication.failure`	When authentication fails

Registering a new single event listener

```xml
<?xml version="1.0" ?>
<container ...>
    <services>
        <!-- ... -->
        <service id="data_collector.router" ...>
            <tag name="kernel.event_listener"
                 event="kernel.controller"
                 method="onKernelController"
                 priority="256"/>
        </service>
    </services>
</container>
```

The Generic Event Object

The GenericEvent class

While it’s recommended to implement specific events classes, Symfony introduces a generic event class called GenericEvent.

The main advantages

- It encapsulates a subject object
- It can embed a list of extra parameters
- It implements the ArrayAccess interface

The GenericEvent API

```php
namespace Symfony\Component\EventDispatcher;

class GenericEvent extends Event
{
    public function getSubject();

    public function getArguments();
    public function getArgument($key);
    public function hasArgument($key);

    public function setArguments(array $args = array());
    public function setArgument($key, $value);
}
```

The GenericEvent API

```php
use Symfony\Component\EventDispatcher\GenericEvent;
use Model\Article;

$subject = new Article();
$subject->setText('Some **markdown**!');

$event = new GenericEvent($subject);
$event->setArgument('author', 'hhamon');

$dispatcher->dispatch('article.save', $event);
```

```
class ArticleListener
{
    private $parser;

    public function __construct(Markdown $parser)
    {
        $this->parser = $parser;
    }

    public function onArticleSave(GenericEvent $event)
    {
        $article = $event->getSubject();

        $html = $this->parser->getHtml($article->getText())
        $article->setHtml($html);

        if (!empty($event['author'])) {
            $article->setAuthor($event['author']);
        }
    }
}
```

Event Subscribers

Event subscribers

Another way to listen to events is via an event subscriber. An event subscriber is a PHP class that's able to tell the dispatcher exactly which events it should subscribe to.

The EventSubscriber interface

```php
namespace Symfony\Component\EventDispatcher;

interface EventSubscriberInterface
{
    /**
     * Returns an array of event names this subscriber wants to listen to.
     *
     * For instance:
     *
     *  * array('eventName' => 'methodName')
     *  * array('eventName' => array('methodName', $priority))
     *  * array('eventName' => array(
     *         array('methodName1', $priority),
     *         array('methodName2'),
     *    )
     *
     * @return array The event names to listen to
     */
    public static function getSubscribedEvents();
}
```

Implementing the EventSubscriber interface

```
namespace SensioLabs\ArticleBundle\Listener;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class ArticleListener implements EventSubscriberInterface
{
    // ...
    public static function getSubscribedEvents()
    {
        return array(
            'article.save'  => array(
                array('onArticleInsert', 10),
                array('onArticleUpdate', 5),
            ),
            'article.delete' => 'onArticleDelete',
        );
    }
}
```

The EventSubscriber interface

```
class ArticleListener implements EventSubscriberInterface
{
    // ...
    public function onArticleInsert(ArticleEvent $event)
    {
        // ...
    }

    public function onArticleUpdate(ArticleEvent $event)
    {
        // ...
    }

    public function onArticleDelete(ArticleEvent $event)
    {
        // ...
    }
}
```

Registering a new single event subscriber

```
<?xml version="1.0" ?>
<container ...>
    <services>
        <!-- ... -->
        <service id="data_collector.router" ...>
            <tag name="kernel.event_subscriber"/>
        </service>
    </services>
</container>
```

Training Department

SensioLabs Training

92-98 Boulevard Victor Hugo

92 115 Clichy Cedex

FRANCE

Phone: +33 140 998 211
Email: training@sensiolabs.com

symfony.com - trainings.sensiolabs.com