

Practical Design Patterns with PHP

SymfonyLive 2019 / Sept. 25th / Berlin / Germany

Hugo Hamon



Hugo

Hamon



 dayuse

SensioLabs

<https://speakerdeck.com/hhamon>

Speaker Deck Browse Upload  Hugo Hamon ▾

Talks by Hugo Hamon



Building an Open-Source Campaign Platform for the President France.
SymfonyCon 2017 / Nov. 17th / Cluj / Roma
Hugo Hamon

Building an Open-Source Campaign Platform for the President France.
PHP-CE 2017 / Nov. 5th / Dessa / Polan
Hugo Hamon



**L'Open-Source.
Au service de la campagne d'Emmanuel Macron**
AFSTY Paris 2017
Lille

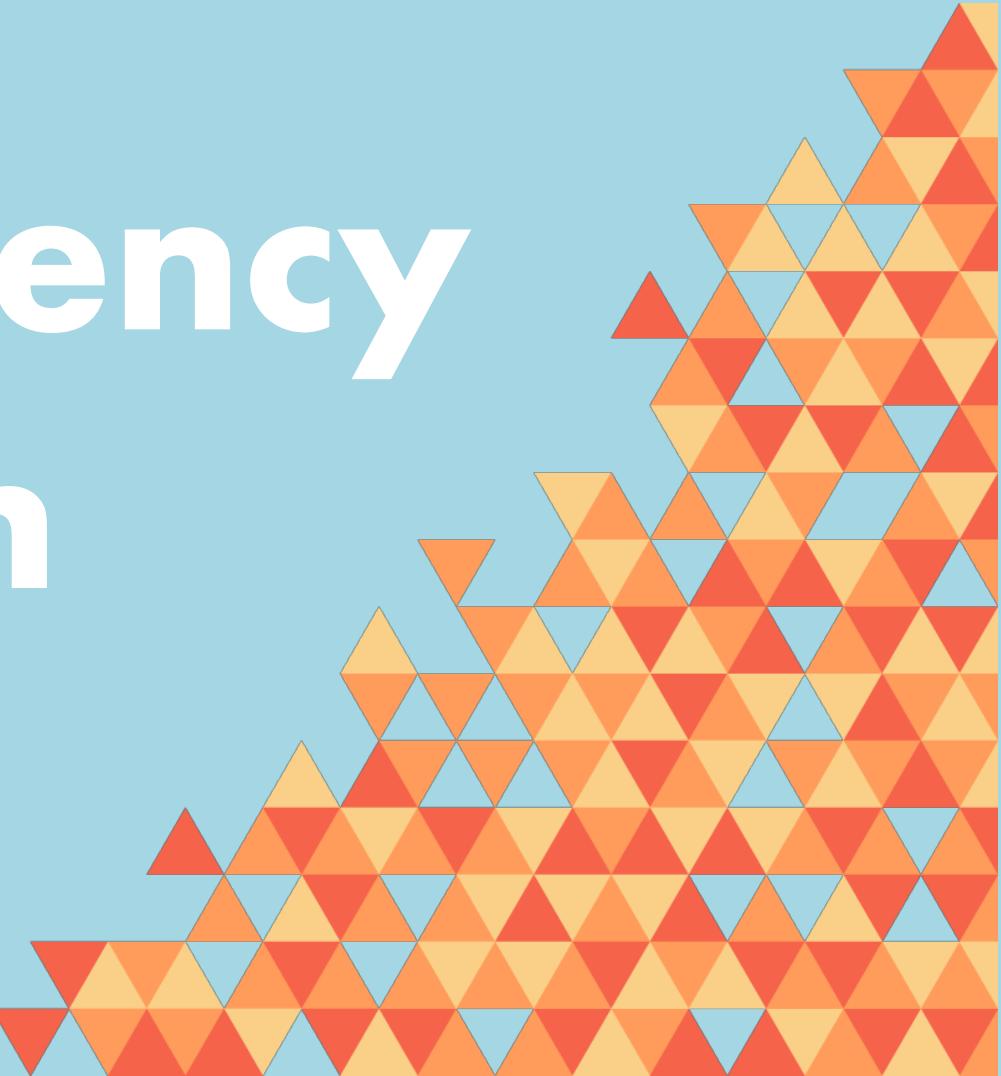
Speaker Details



Hugo Hamon

Hugo Hamon is a PHP and Symfony fan who works with PHP since 2003. After five years of professional PHP web development in web agencies for famous french customers, he now works as a consultant and head of training at SensioLabs. On his free time, Hugo contributes to the Symfony2 project (source code and documentation) and gets involved in the french PHP association as an organization member. Hugo also wrote and contributed to french and english books related to PHP and the Symfony framework.

Dependency Injection



Dependency Injection

Dependency Injection is where components are given their dependencies through their constructors, methods, or directly into fields. Those components do not get their dependencies themselves, or instantiate them directly.

— *picoccontainer.com/injection.html*

```
class ChessGameLoader
{
    private $repository;
    private $cache;
    private $serializer;

    public function __construct()
    {
        $this->repository = new InMemoryChessGameRepository();
        $this->cache = new RedisCache();
        $this->serializer = new Serializer();
    }

    public function load(UuidInterface $id): ?ChessGame
    {
        // ...
    }
}
```

Problems

- **Tight coupling**
- **Concretions instead of abstractions**
- **Not testable**
- **Not flexible**

```
class ChessGameLoader
{
    private $repository;
    private $cache;
    private $serializer;

    public function __construct(
        InMemoryChessGameRepository $repository,
        RedisCache $cache,
        Serializer $serializer
    ) {
        $this->repository = $repository;
        $this->cache = $cache;
        $this->serializer = $serializer;
    }
}
```

Pros

- **Code is unit testable**
- **Dependencies can be mocked**
- **Dependencies can be changed**

Cons

- **Client code is still tightly coupled**
- **Client code doesn't rely on abstractions**

Dependency Injection Container

A dependency injection container is an object that enables to standardize and centralize the way objects are constructed and configured in an application.

— *symfony.com*

Complex Construction

```
$loader = new ChessGameLoader(  
    new InMemoryChessGameRepository(),  
    new RedisCache(new Predis\Client('tcp://...')),  
    new Serializer(new JsonDecoder())  
) ;  
  
$game = $loader->load(Uuid::fromString('1f809a73-...'));
```

parameters:

```
env(REDIS_DSN): 'tcp://10.0.0.1:6379'
```

services:

```
App\Serializer\Serializer:
```

```
    arguments: [ '@App\Serializer\Decoder\JsonDecoder' ]
```

```
Predis\Client:
```

```
    arguments: [ '%env(resolve:REDIS_DSN)%' ]
```

```
App\Cache\RedisCache:
```

```
    arguments: [ '@Predis\Client' ]
```

```
App\ChessGame\InMemoryChessGameRepository: ~
```

```
App\ChessGame\ChessGameLoader:
```

```
    arguments:
```

- '@App\ChessGame\InMemoryChessGameRepository'
- '@App\Cache\RedisCache'
- '@App\Serializer\Serializer'

```
$loader = $container  
->get(ChessGameLoader::class)  
;
```

Composing Objects



Object Composition

In computer science, object composition is a way to combine simple objects or data types into more complex ones.

— wikipedia.com

```
class ChessGameLoader
{
    // ...
    public function __construct(
        InMemoryChessGameRepository $repository,
        RedisCache $cache,
        Serializer $serializer
    ) {
        // ...
    }

    public function load(UuidInterface $id): ?ChessGame
    {
        if ($this->cache->has($key = sprintf('game/%s', $id))) {
            return $this->serializer->deserialize(
                ChessGame::class,
                $this->cache->get($key)
            );
        }

        return $this->repository->byId($id);
    }
}
```



RedisRepository

```
class RedisChessGameRepository implements ChessGameRepository
{
    public function getById(UuidInterface $id): ?ChessGame
    {
        $key = sprintf('game/%s', $id);
        if (!$this->cache->has($key)) {
            return null;
        }

        return $this->serializer->deserialize(
            ChessGame::class,
            $this->cache->get($key)
        );
    }
}
```

```
class ChessGameLoader
{
    // ...
    public function __construct(
        InMemoryChessGameRepository $inMemoryRepository,
        RedisChessGameRepository $redisRepository
    ) {
        // ...
    }

    public function load(UuidInterface $id): ?ChessGame
    {
        if ($game = $this->redisRepository->byId($id)) {
            return $game;
        }

        return $this->inMemoryRepository->byId($id);
    }
}
```

```
class ChainChessGameRepository implements ChessGameRepository
{
    private $repositories = [];

    public function add(ChessGameRepository $repository): void
    {
        $this->repositories[] = $repository;
    }

    public function getById(UuidInterface $id): ?ChessGame
    {
        foreach ($this->repositories as $repository) {
            if ($game = $repository->byId($id)) {
                return $game;
            }
        }

        return null;
    }
}
```

```
class ChessGameLoader
{
    // ...
    public function __construct(
        ChessGameRepository $repository,
        LoggerInterface $logger
    ) {
        // ...
    }

    public function load(UuidInterface $id): ?ChessGame
    {
        $this->logger->log(sprintf('Load game %s', $id));

        return $this->repository->byId($id);
    }
}
```

```
$repository = new ChainChessGameRepository();
repository->add(new RedisChessGameRepository(...));
repository->add(new InMemoryChessGameRepository());  
  
$loader = new ChessGameLoader(
    $repository,
    new NullLogger()
);  
  
$game = $loader->load(Uuid::fromString('1f809a73-...'));
```

SOLID Principles



Single Responsibility

A class should have one, and only one, reason to change.

— *Robert C. Martin*

```
class ChessGameRunner
{
    // ...
    public function startNewGame(ChessGameContext $context): ChessGame
    {
        $game = new ChessGame(
            Uuid::uuid4(),
            $this->loadPlayer($context->getPlayerOne()),
            $this->loadPlayer($context->getPlayerTwo())
        );

        $this->gameRepository->save($game);

        return $game;
    }

    private function loadPlayer(string $player): Player
    {
        return Player::fromUserAccount($this->userRepository->byUsername($player));
    }
}
```

```
class ChessGameRunner
{
    public function startNewGame(ChessGameContext $context): ChessGame
    {
        $game = new ChessGame( ← Object Construction
            Uuid::uuid4(),
            $this->loadPlayer($context->getPlayerOne()),
            $this->loadPlayer($context->getPlayerTwo())
        );
        $this->gameRepository->save($game); ← Persistence
        return $game;
    }
}
```

```
class ChessGameFactory
{
    private $userRepository;

    public function __construct(UserAccountRepository $repository)
    {
        $this->userRepository = $repository;
    }

    public function create(string $player1, string $player2): ChessGame
    {
        return new ChessGame(
            Uuid::uuid4(),
            Player::fromUserAccount($this->userRepository->byUsername($player1)),
            Player::fromUserAccount($this->userRepository->byUsername($player2))
        );
    }
}
```

```
class ChessGameRunner
{
    private $gameRepository;
    private $gameFactory;

    public function __construct(
        ChessGameRepository $repository,
        ChessGameFactory $factory
    ) {
        $this->gameRepository = $repository;
        $this->gameFactory = $factory;
    }
}
```

```
class ChessGameRunner
{
    // ...

    public function startNewGame(ChessGameContext $context): ChessGame
    {
        $game = $this->gameFactory->create(
            $context->getPlayerOne(),
            $context->getPlayerTwo()
        );

        $this->gameRepository->save($game);

        return $game;
    }
}
```

Open Closed Principle

You should be able to extend a classes behavior, without modifying it.

— *Robert C. Martin*

```
class ChessGameFactory
{
    private $userRepository;

    public function __construct(UserAccountRepository $repository)
    {
        $this->userRepository = $repository;
    }

    public function create(string $player1, string $player2): ChessGame
    {
        return new ChessGame(
            Uuid::uuid4(), ←
            Player::fromUserAccount($this->userRepository->byUsername($player1)),
            Player::fromUserAccount($this->userRepository->byUsername($player2))
        );
    }
}
```

```
interface GameIdGenerator
{
    public function generate(): UuidInterface;
}

class FixedGameIdGenerator implements GameIdGenerator
{
    public function generate(): UuidInterface
    {
        return new Uuid::fromString('1f809a73-63d5-40dd-9bc0-f7bc6813a4bc');
    }
}

class RandomGameIdGenerator implements GameIdGenerator
{
    public function generate(): UuidInterface
    {
        return new Uuid::uuid4();
    }
}
```

```
class ChessGameFactory
{
    private $userRepository;
    private $identityGenerator;

    public function __construct(
        UserAccountRepository $repository,
        GameIdGenerator $identityGenerator
    ) {
        $this->userRepository = $repository;
        $this->identityGenerator = $identityGenerator;
    }

    public function create(string $player1, string $player2): ChessGame
    {
        return new ChessGame(
            $this->identityGenerator->generate(),
            $this->loadPlayer($player1),
            $this->loadPlayer($player2)
        );
    }
}
```



Liskov Substitution Principle

Derived classes must be substitutable for their base classes.

— *Robert C. Martin*

```
interface ChessGameRepository
{
    /**
     * @throws ChessGameNotFound
     */
    public function getById(UuidInterface $id): ChessGame;
}
```

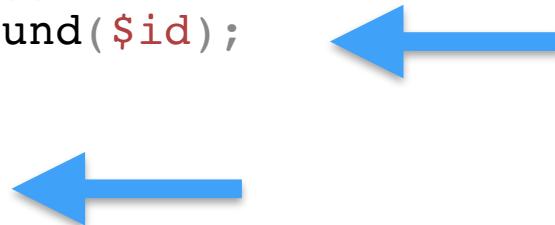
```
class ChessGameRunner
{
    // ...
    public function loadGame(UuidInterface $id): ChessGame
    {
        try {
            return $this->gameRepository->byId($id);
        } catch (ChessGameNotFound $e) {
            throw ChessGameUnavailable::gameNotFound($id, $e);
        }
    }
}
```

```
class InMemoryChessGameRepository implements ChessGameRepository
{
    private $games = [ ];

    // ...

    public function byId(UuidInterface $id): ChessGame
    {
        $id = $uuid->toString();
        if (!isset($this->games[$id])) {
            throw new ChessGameNotFound($id);
        }

        return $this->games[$id];
    }
}
```



```
class DoctrineChessGameRepository implements ChessGameRepository
{
    private $repository;

    public function __construct(ManagerRegistry $registry)
    {
        $this->repository = $registry->getRepository(ChessGame::class);
    }

    public function getById(UuidInterface $id): ChessGame
    {
        if (!$game = $this->repository->find($id->toString())) {
            throw new ChessGameNotFound($id->toString());
        }

        return $game;
    }
}
```



```
$runner = new ChessGameRunner(  
    new InMemoryChessGameRepository(),  
    new ChessGameFactory(...)  
) ;
```

```
$runner = new ChessGameRunner(  
    new DoctrineChessGameRepository(...),  
    new ChessGameFactory(...)  
) ;
```

```
$runner  
    ->loadGame(Uuid::fromString('1f809a73-...'))  
;
```

Interface Segregation Principle

**Make fine grained interfaces
that are client specific.**

— *Robert C. Martin*

```
interface ChessGameRepository
{
    public function byId(UuidInterface $id): ChessGame;
}

interface UserAccountRepository
{
    public function byUsername(string $username): User;
}

interface ChessGameFactory
{
    public function create(string $player1, string $player2): ChessGame;
}
```

```
interface UrlMatcherInterface
{
    public function match(string $pathinfo): array;
}
```

```
interface UrlGeneratorInterface
{
    public function generate(string $name, array $params = []): string;
}
```

```
interface RouterInterface extends UrlMatcherInterface, UrlGeneratorInterface
{
    public function getRouteCollection(): RouteCollection;
}
```

Dependency Inversion Principle

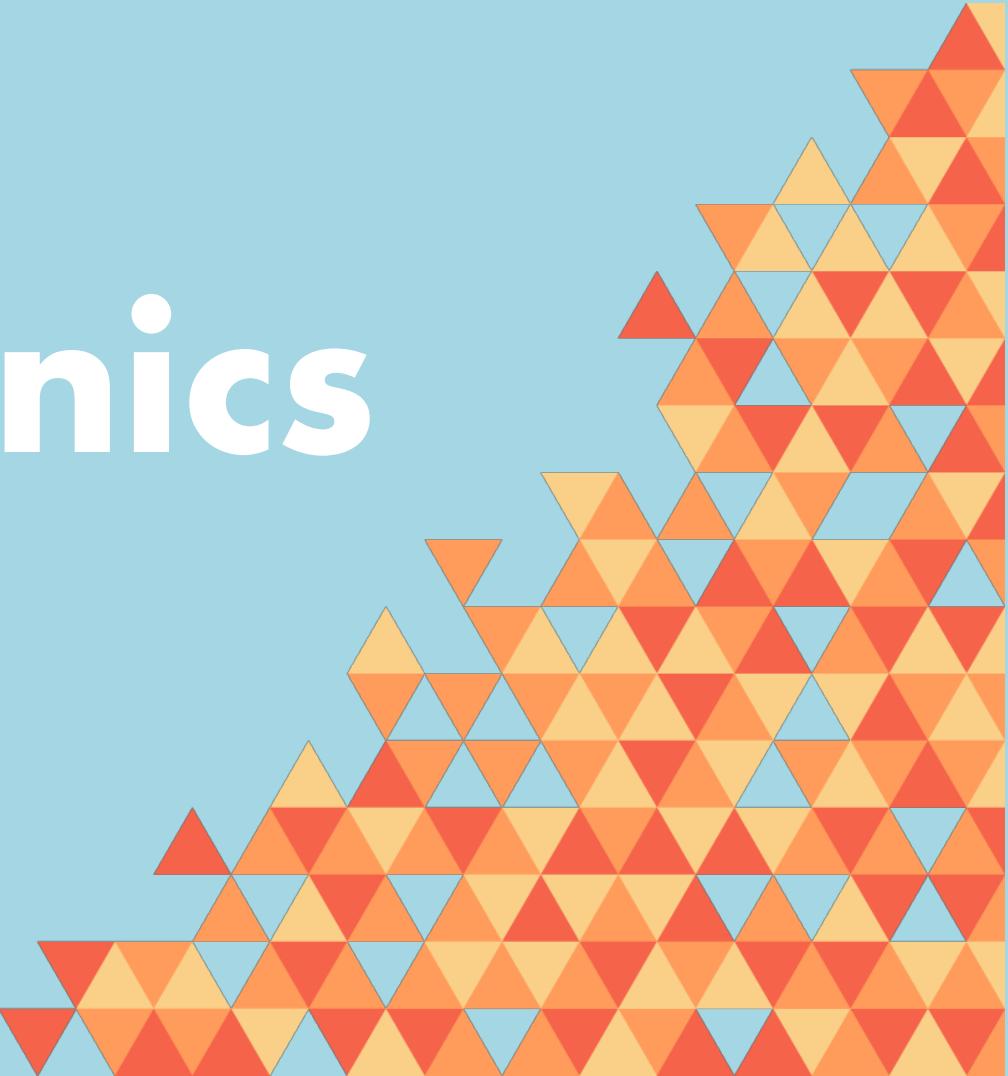
**Depend on abstractions, not
on concretions.**

— *Robert C. Martin*

```
class ChessGameRunner
{
    private $gameRepository;
    private $gameFactory;

    public function __construct(
        ChessGameRepository $repository,
        ChessGameFactory $factory
    ) {
        $this->gameRepository = $repository;
        $this->gameFactory = $factory;
    }
}
```

Object Calisthenics



Object Calisthenics

Calisthenics are gymnastic exercises designed to develop physical health and vigor, usually performed with little or no special apparatus.

— *dictionary.com*

1. One level of indentation per method
- 2. Don't use the ELSE keyword**
- 3. Wrap primitive types and strings**
4. Two instance operators per line
5. Don't abbreviate
- 6. Make short and focused classes**
7. Keep number of instance properties low
- 8. Treat lists as custom collection objects**
- 9. Avoid public accessors and mutators**

Wrap Primitive Types and Strings

```
class ChessGame
{
    /** @var ChessBoard */
    private $board;

    public function makeMove(
        Pawn $pawn,
        int $originRow, int $originCol,
        int $targetRow, int $targetCol
    ): void {
        $this->ensureValidMove(
            $pawn,
            $originRow, $originCol,
            $targetRow, $targetCol
        );
        $this->board->getSquare($targetRow, $targetCol)->add($pawn);
    }
}
```



Square

```
class Square
{
    private $row;
    private $col;

    public function __construct(int $row, int $col)
    {
        $range = range(1, 8);
        if (!in_array($row, $range, true)) {
            throw new \InvalidArgumentException('Invalid row.');
        }

        if (!in_array($col, $range, true)) {
            throw new \InvalidArgumentException('Invalid col.');
        }

        $this->row = $row;
        $this->col = $col;
    }

    // ...
}
```

```
class ChessGame
{
    /** @var ChessBoard */
    private $board;

    public function makeMove(
        Pawn $pawn,
        Square $origin,
        Square $target
    ) {
        $this->ensureValidMove($pawn, $origin, $target);
        $this
            ->board
            ->getSquare($target->row(), $target->col())
            ->add($pawn);
    }
}
```



```
class Move
{
    private $pawn;
    private $originSquare;
    private $targetSquare;

    public function __construct(
        Pawn $pawn,
        Square $from,
        Square $to
    ) {
        $this->pawn = $pawn;
        $this->originSquare = $from;
        $this->targetSquare = $to;
    }

    // ...
}
```

```
class ChessGame
{
    /** @var ChessBoard */
    private $board;

    public function makeMove(Move $move): void
    {
        $this->ensureValidMove($move);
        $this
            ->board
            ->getSquare($move->getTargetSquare())
            ->add($move->getPawn());
    }
}
```

**One Level of
Indentation per
Method**

```
class ChessGame
{
    private $finished = false;

    public function makeMove(Move $move): void
    {
        0 if (!$this->finished) {
            1 if ($this->isValidMove($move)) {
                2 $this->performMove($move);
            }
        }
    }
}
```

```
class ChessGame
{
    private $finished = false;

    public function makeMove(Move $move): void
    {
        0 if ($this->finished) {
            1 throw new GameAlreadyFinished();
        }

        0 if (! $this->isValidMove($move)) {
            1 throw new InvalidGameMove();
        }

        0 $this->performMove($move);
    }
}
```

```
class ChessGame
{
    private $finished = false;

    public function makeMove(Move $move): void
    {
        $this->ensureGameNotFinished();
        $this->ensureValidMove($move);
        $this->performMove($move);
    }

    private function ensureGameNotFinished(): void
    {
        if ($this->finished) {
            throw new GameAlreadyFinished();
        }
    }

    private function ensureValidMove(Move $move): void
    {
        if (!$this->isValidMove($move)) {
            throw new InvalidGameMove();
        }
    }
}
```

Avoid the ELSE Keyword

```
class ChessGame
{
    // ...
    private $finished = false;

    public function makeMove(Move $move): void
    {
        if (! $this->finished) {
            $this->ensureValidMove($move);
            $this->performMove($move);
        } else {
            throw new GameAlreadyFinished();
        }
    }
}
```

```
class ChessGame
{
    // ...
    private $finished = false;

    public function makeMove(Move $move): void
    {
        if ($this->finished) {
            return;
        }

        $this->ensureValidMove($move);
        $this->performMove($move);
    }
}
```

```
class ChessGame
{
    // ...
    private $finished = false;

    public function makeMove(Move $move): void
    {
        if ($this->finished) {
            throw new GameAlreadyFinished();
        }

        $this->ensureValidMove($move);
        $this->performMove($move);
    }
}
```

```
class ChessGame
{
    // ...
    private $finished = false;

    public function makeMove(Move $move): void
    {
        $this->ensureGameNotFinished();
        $this->ensureValidMove($move);
        $this->performMove($move);
    }
}
```

**Two Instance
Operators
per Line**

```
class ChessGame
{
    /** @var ChessBoard */
    private $board;

    public function makeMove(Move $move): void
    {
        $this->ensureValidMove($move);
        $this
            ->board
            ->getSquare($move->getTargetSquare());
3        ->add($move->getPawn());
    }
}
```

```
class ChessGame
{
    /** @var ChessBoard */
    private $board;

    public function makeMove(Move $move): void
    {
        $this->ensureValidMove($move);
        $this->board->placePawnOnSquare(
            $move->getTargetSquare(),
            $move->getPawn()
        );
    }
}
```

2

Avoid public
accessors
methods

⚠
Beware of
Anemic Models

```
$invoice = new Invoice();
$invoice->setNumber('INV-20180306-66');
$invoice->setIssueDate('2018-03-10');
$invoice->setDueDate('2018-04-10');
$invoice->setDueAmount(1350.90);
$invoice->setDueAmountCurrency('USD');
$invoice->setStatus('issued');

// + all the getter methods
```

```
class Invoice
{
    private $number;
    private $billingEntity;
    private $issueDate;
    private $dueDate;
    private $dueAmount;
    private $remainingDueAmount;

    public function __construct(
        InvoiceId $number,
        BillingEntity $billingEntity,
        Money $dueAmount,
        \DateTimeImmutable $dueDate
    ) {
        $this->number = $number;
        $this->billingEntity = $billingEntity;
        $this->issueDate = new \DateTimeImmutable('today', new \DateTimeZone('UTC'));
        $this->dueDate = $dueDate;
        $this->dueAmount = $dueAmount;
        $this->remainingDueAmount = clone $dueAmount;
    }
}
```

Issue an Invoice

```
$invoice = new Invoice(  
    new InvoiceId('INV-20180306-66'),  
    new BillingEntity('3429234'),  
    new Money(9990, new Currency('EUR')),  
    new \DateTimeImmutable('+30 days')  
) ;
```

```
class Invoice
{
    // ...
    private $overdueAmount;
    private $closingDate;
    private $payments = [];

    public function collectPayment(Payment $payment): void
    {
        $amount = $payment->getAmount();
        $this->remainingDueAmount = $this->remainingDueAmount->subtract($amount);
        $this->overdueAmount = $this->remainingDueAmount->absolute();

        $zero = new Money(0, $this->remainingDueAmount->getCurrency());
        if ($this->remainingDueAmount->lessThanOrEqualTo($zero)) {
            $this->closingDate = new \DateTimeImmutable('now', new \DateTimeZone('UTC'));
        }

        $this->payments[] = new CollectedPayment(
            $payment->getReceptionDate(),
            $amount,
            $payment->getSource() // wire, check, cash, etc.
        );
    }
}
```

```
class Invoice
{
    public function isClosed(): bool
    {
        return $this->closingDate instanceof \DateTimeImmutable;
    }

    public function isPaid(): bool
    {
        $zero = new Money(0, $this->remainingDueAmount->getCurrency());

        return $this->remainingDueAmount->lessThanOrEqual($zero);
    }

    public function isOverpaid(): bool
    {
        $zero = new Money(0, $this->remainingDueAmount->getCurrency());

        return $this->remainingDueAmount->lessThan($zero);
    }
}
```

Collecting Payments

```
$invoice->collectPayment(new Payment(  
    new \DateTimeImmutable('2018-03-04'),  
    new Money(4900, new Currency('EUR')),  
    new WireTransferPayment('450357035')));
```

```
$invoice->collectPayment(new Payment(  
    new \DateTimeImmutable('2018-03-08'),  
    new Money(5100, new Currency('EUR')),  
    new WireTransferPayment('248748484')));
```

Treat lists as
custom collection
objects

```
class Invoice
{
    // ...
    private $payments;

    public function __construct(...)
    {
        // ...
        $this->payments = new ArrayCollection();
    }

    public function collectPayment(Payment $payment): void
    {
        // ...
        $this->payments->add(new CollectedPayment(
            $payment->getReceptionDate(),
            $amount,
            $payment->getSource() // wire, check, cash, etc.
        ));
    }
}
```

Filtering the collection

```
class Invoice
{
    // ...
    public function countPaymentsReceivedAfterDueDate(): int
    {
        return $this
            ->payments
            ->filter(function (CollectedPayment $payment) {
                return $payment->getReceptionDate() > $this->dueDate;
            })
            ->count();
    }
}
```

Custom Collection Type

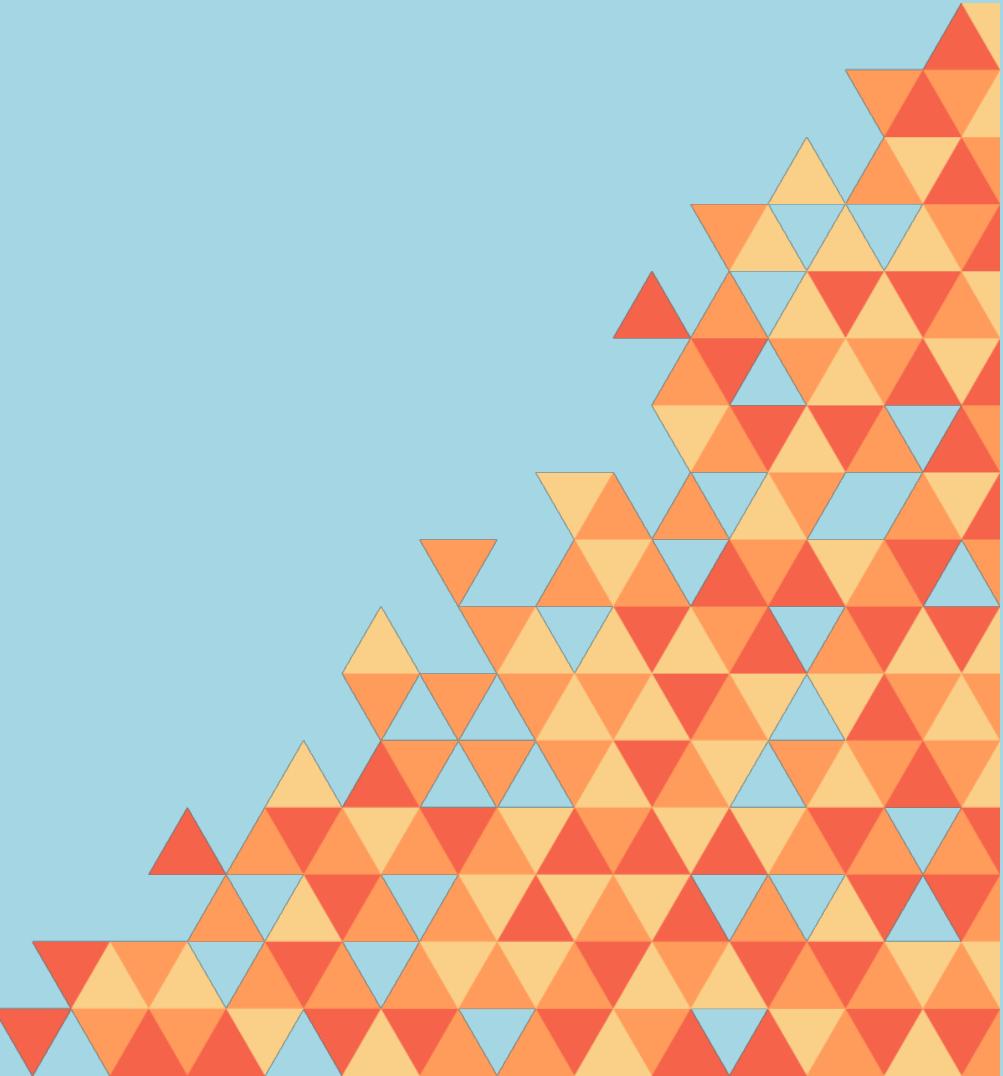
```
class CollectedPaymentCollection extends ArrayCollection
{
    public function receivedAfter(\DateTimeImmutable $origin): self
    {
        $filter = function (CollectedPayment $payment) use ($origin) {
            return $payment->getReceptionDate() > $origin;
        };

        return $this->filter($filter);
    }
}
```

```
class Invoice
{
    // ...
    public function __construct(...)
    {
        // ...
        $this->payments = new CollectedPaymentCollection();
    }

    public function countPaymentsReceivedAfterDueDate(): int
    {
        return $this
            ->payments
            ->receivedAfter($this->dueDate)
            ->count();
    }
}
```

Value Objects



Value Objects

A value object is an object representing an atomic value or concept. The value object is responsible for validating the consistency of its own state.

It's designed to always be in a valid, consistent and immutable state.

Value Object Properties

- **They don't have an identity**
- **They're responsible for validating their state**
- **They are immutable by design**
- **They are always valid by design**
- **Equality is based on their fields**
- **They are interchangeable without side effects**

```
final class Currency
{
    private $code;

    public function __construct(string $code)
    {
        if (!in_array($code, ['EUR', 'USD', 'CAD'], true)) {
            throw new \InvalidArgumentException('Unsupported currency.');
        }

        $this->code = $code;
    }

    public function equals(self $other): bool
    {
        return $this->code === $other->code;
    }
}
```

```
new Currency( 'EUR' );    // OK
new Currency( 'USD' );    // OK
new Currency( 'CAD' );    // OK

new Currency( 'BTC' );    // Error
```

```
final class Money
{
    private $amount;
    private $currency;

    public function __construct(int $amount, Currency $currency)
    {
        $this->amount = $amount;
        $this->currency = $currency;
    }

    // ...
}
```

```
final class Money
{
    // ...

    public function add(self $other): self
    {
        $this->ensureSameCurrency($other->currency);

        return new self($this->amount + $other->amount, $this->currency);
    }

    private function ensureSameCurrency(Currency $other): void
    {
        if (!$this->currency->equals($other)) {
            throw new \RuntimeException('Currency mismatch');
        }
    }
}
```

```
$a = new Money(100, new Currency('EUR')); // 1€  
$b = new Money(500, new Currency('EUR')); // 5€  
  
$c = $a->add($b); // 6€  
  
$c->add(new Money(300, new Currency('USD'))); // Error
```

#1

Introduction to Design Patterns

Design Patterns

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

– Wikipedia

Creational

Creational design patterns are responsible for encapsulating the algorithms for producing and assembling objects.

Patterns

**Abstract Factory
Builder
Factory Method
Prototype
Singleton**

Structural

Structural design patterns organize classes in a way to separate their implementations from their interfaces.

Patterns

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Behavioral

Behavioral design patterns organize objects to make them collaborate together while reducing their coupling.

Patterns

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor



👍 Communication

👍 Code Testability

👍 Maintainability

👍 Loose Coupling

👍 ...



👎 Hard to Teach

👎 Hard to Learn

👎 Hard to Apply

👎 Entry Barrier

👎 ...



**Patterns are not always
the holly grail!!!**

#2

Creational Design Patterns

Singleton



Singleton

The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance.

– GoF

Singleton

- static \$instance : Singleton

+ static getInstance() : Singleton

- __construct()

- __clone()

Characteristics

- Construct method is made private
- Construction logic happens through static method
- Object cloning must be forbidden
- Object deserialization must be forbidden

```
namespace ErrorHandler;

final class ErrorHandler
{
    private static $instance;

    private $errorTypes;

    public static function getInstance(int $errorTypes = E_ALL): self
    {
        if (!self::$instance) {
            self::$instance = new self($errorTypes);
        }

        return self::$instance;
    }

    private function __construct(int $errorTypes = E_ALL)
    {
        $this->errorTypes = $errorTypes;
    }
}
```

```
final class ErrorHandler
{
    // ...

    public function __clone()
    {
        throw new \BadMethodCallException('Not allowed!');
    }

    public function __wakeup()
    {
        throw new \BadMethodCallException('Not allowed!');
    }
}
```

```
final class ErrorHandler
{
    // ...
    public function handleException(\Throwable $e): void
    {
        // handle exception
    }

    public function handleError(
        int $errorLevel,
        string $errorMessage,
        string $errorFile,
        string $errorLine
    ): void {
        // handle error
    }
}
```

```
final class ErrorHandler
{
    // ...
private $registered = false;

public function register(): self
{
    if (!$this->registered) {
        error_reporting($this->errorTypes);
        set_error_handler([$this, 'handleError'], $this->errorTypes);
        set_exception_handler([$this, 'handleException']);
        $this->registered = true;
    }

    return $this;
}
}
```

```
$handler = ErrorHandler::getInstance()->register();

@trigger_error('Trigger A...', E_USER_DEPRECATED);
@trigger_error('Trigger B...', E_USER_DEPRECATED);
@trigger_error('Trigger C...', E_USER_NOTICE);
@trigger_error('Trigger D...', E_USER_WARNING);
@trigger_error('Trigger E...', E_USER_DEPRECATED);
```

Benefits

- **Only one instance is guaranteed**
- **Simplicity**

Downsides

- **Unable to get multiple instances**
- **Parameterized construction is more complex**
- **Global state**
- **Tight coupling**
- **Often misused or abused by developers...**

Prototype



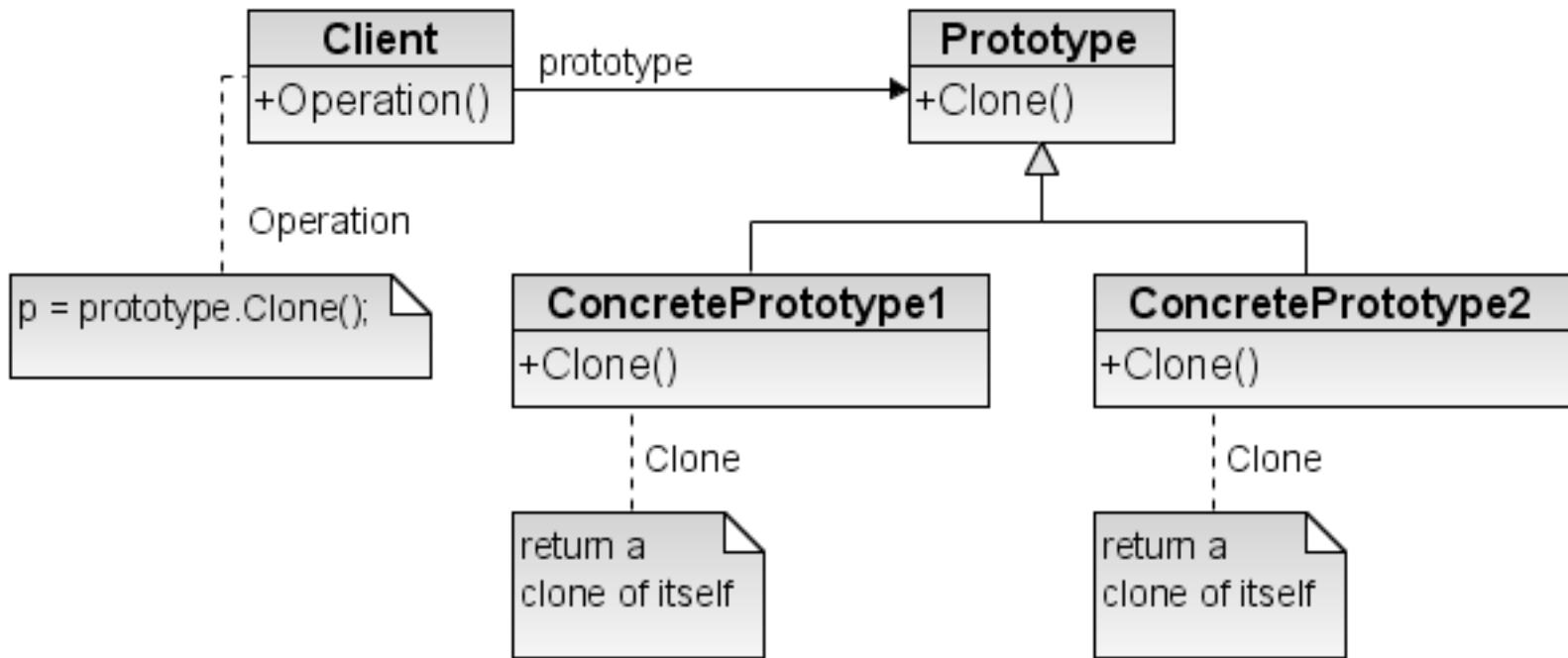
Prototype

The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone. This practise is particularly useful when the construction of a new object is inefficient.

– GoF

Problems to solve

- Avoid using the «**new**» keyword to create an object, especially when construction is complex and heavy.
- Leverage **object cloning** to build and reconfigure new instances of a class.



HttpFoundation

The Request object from the HttpFoundation component provides a mechanism to duplicate itself using object cloning to produce a new fresh and configured instance.

```
class Request
{
    // ...
    public function duplicate(array $query = null, array $request = null, ...)
    {
        $dup = clone $this; ←
        if (null !== $query) {
            $dup->query = new ParameterBag($query);
        }
        if (null !== $request) {
            $dup->request = new ParameterBag($request);
        }
        // ...
        if (null !== $server) {
            $dup->server = new ServerBag($server);
            $dup->headers = new HeaderBag($dup->server->getHeaders());
        }
        $dup->languages = null;
        $dup->Charsets = null;
        // ...

        if (!$dup->get('_format') && $this->get('_format')) {
            $dup->attributes->set('_format', $this->get('_format'));
        }

        if (!$dup->getRequestFormat(null)) {
            $dup->setRequestFormat($this->getRequestFormat(null));
        }

        return $dup;
    }
}
```

```
trait ControllerTrait
{
    // ...
    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); ←

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

Form

The FormBuilder object of the Form component uses object cloning and the Prototype pattern to build a new configured instance of FormConfig.

```
class FormConfigBuilder implements FormConfigBuilderInterface
{
    // ...
    private $locked = false;

    public function getFormConfig()
    {
        if ($this->locked) {
            throw new BadMethodCallException('...');
        }

        // This method should be idempotent, so clone the builder
        $config = clone $this; ←
        $config->locked = true;

        return $config;
    }
}
```

```
class FormBuilder extends FormConfigBuilder
{
    // ...
    public function getFormConfig()
    {
        /** @var $config self */
        $config = parent::getFormConfig(); ←

        $config->children = array();
        $config->unresolvedChildren = array();

        return $config;
    }

    public function getForm()
    {
        // ...
        $form = new Form($this->getFormConfig());
        // ...

        return $form;
    }
}
```

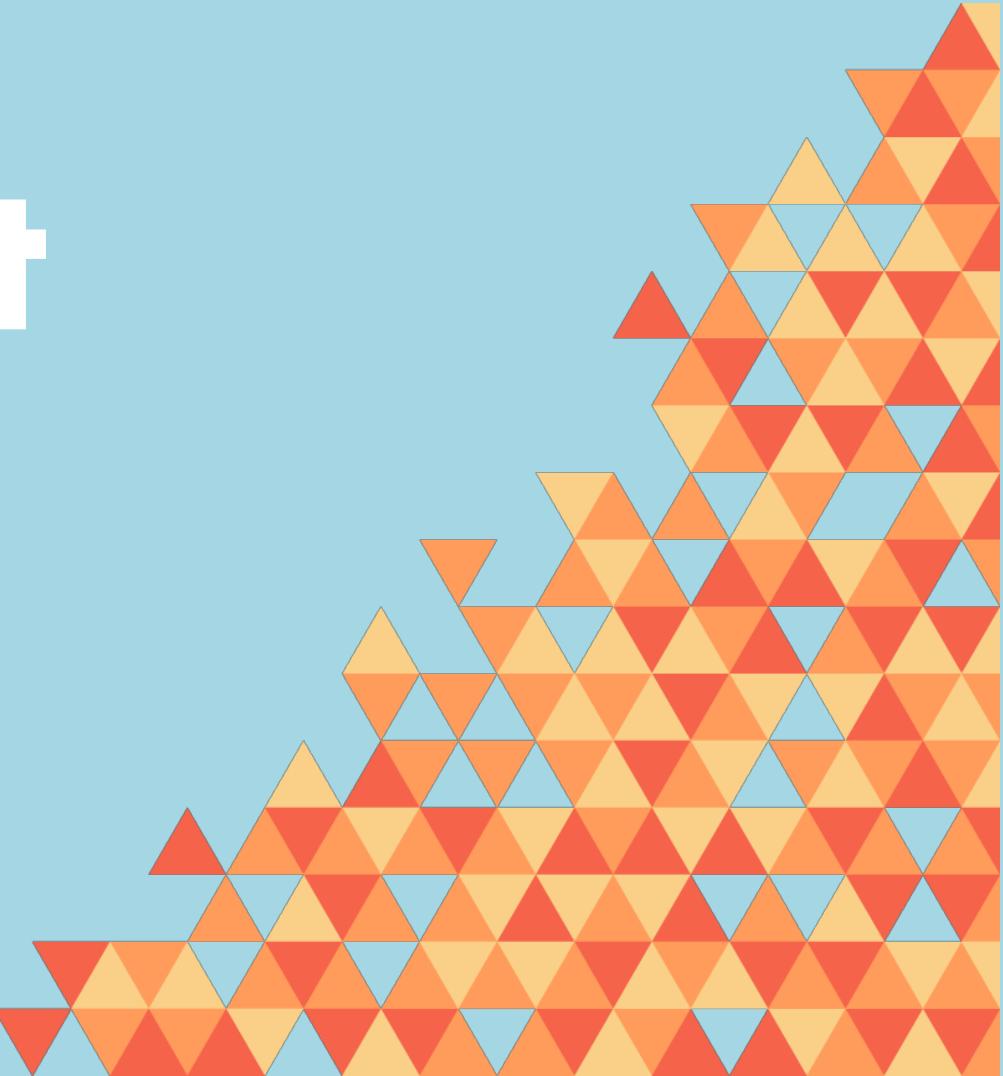
Benefits

- Simple, no need for factories or subclassing
- Reduce repeating initialization code
- Create complex objects faster
- Provide an alternative for subclassing for complex object with many configurations

Disadvantages

- Cloning deep and complex objects graphs composed of many nested objects can be hard

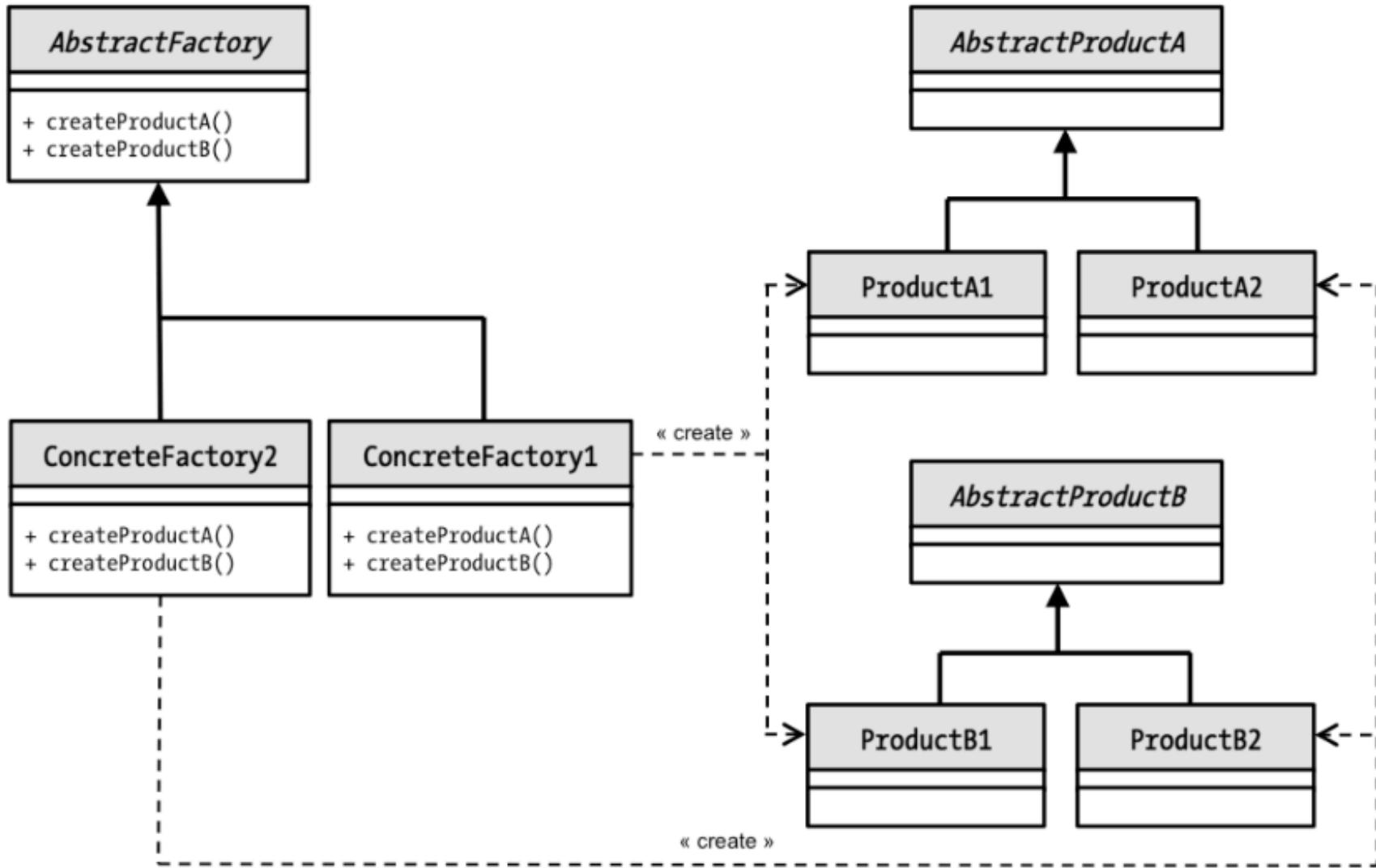
Abstract Factory



Abstract Factory

The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

– Wikipedia





2012 - Symfony 2.3



2015 - Symfony 3.0



2016 - Twig 1.x

Both assessments have similarities

-	Symfony 2.3 / 3.0	Twig
Pricing	€250 (USA, Europe, UAE, Japan, etc.) €200 (Brazil, Tunisia, India, etc.)	€149 (no country restriction)
Eligibility Conditions	<p>Candidate must be at least 18 y.o</p> <p>Candidate must not be Expert Certified</p> <p>Up to 2 sessions max per civil year</p> <p>No active exam registration</p> <p>1 week blank period between 2 exams</p>	<p>Candidate must be at least 18 y.o</p> <p>Candidate must not be Expert Certified</p> <p>No active exam registration</p> <p>1 month blank period between 2 exams</p>
Levels	Expert Symfony Developer Advanced Symfony Developer	Expert Twig Web Designer

First
Implementation
Attempt. . .

```
class PlaceOrderCommandHandler
{
    private $symfonyPricer;
    private $twigPricer;

    // ...

    public function handle(PlaceOrderCommand $command): void
    {
        $order = new Order();
        $order->setQuantity($command->getQuantity());
        $order->setUnitPrice($this->getUnitPrice($command));
        // ...
    }
}
```

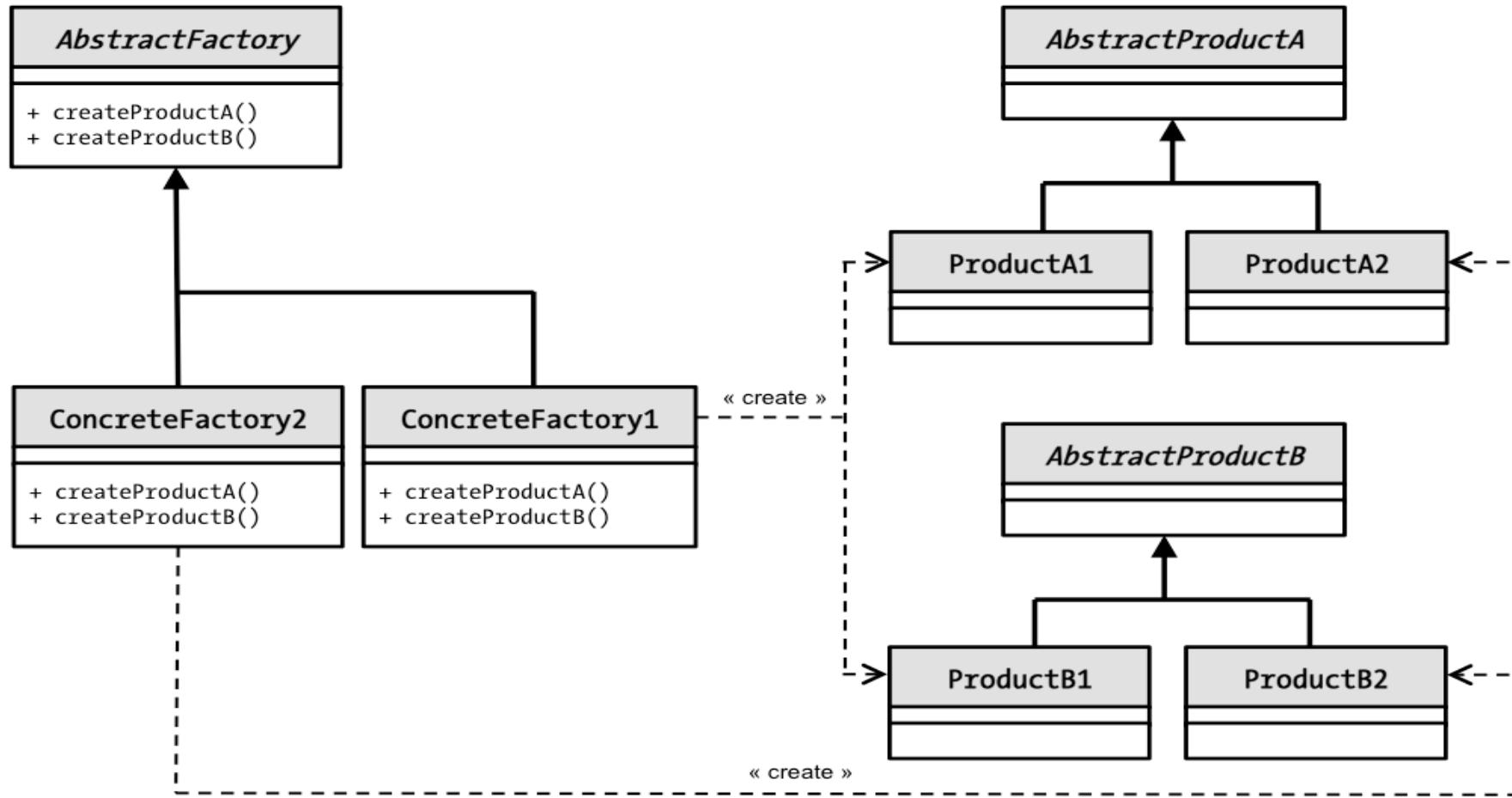
```
class PlaceOrderCommandHandler
{
    // ...

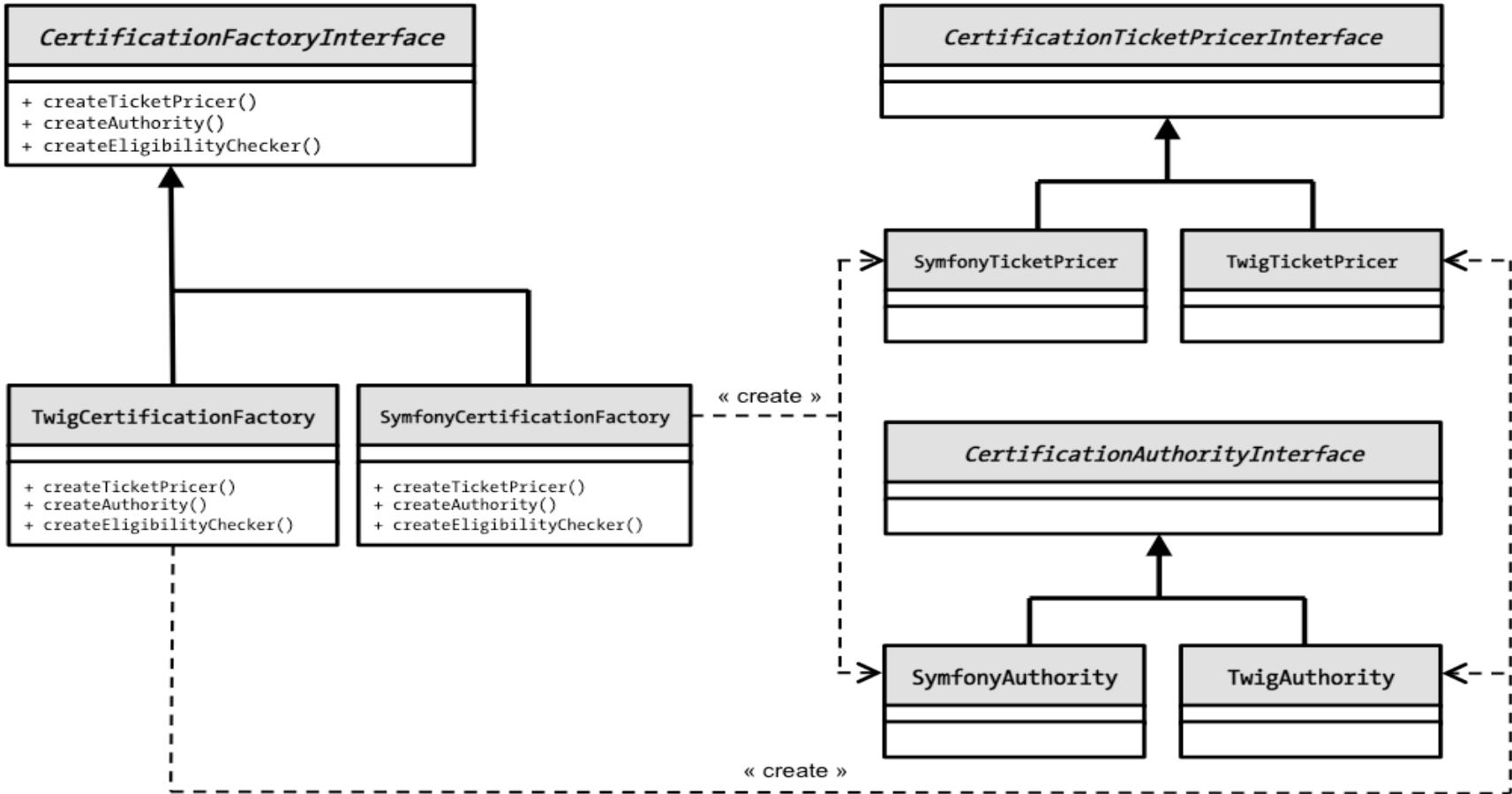
    private function getUnitPrice(PlaceOrderCommand $command): Money
    {
        $country = $command->getCountry();
        switch ($code = $command->getExamSeriesType()) {
            case 'TWXCE':
                return $this->twigPricer->getUnitPrice($country);
            case 'SFXCE':
                return $this->symfonyPricer->getUnitPrice($country);
        }

        throw new UnsupportedAssessmentException($code);
    }
}
```

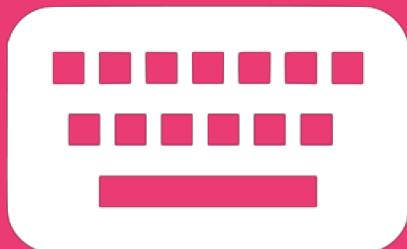


Introducing an
Abstract Factory
Implementation





Defining all main common interfaces



```
namespace Certification;

interface CertificationFactoryInterface
{
    public function createEligibilityChecker()
        : CertificationEligibilityCheckerInterface;

    public function createTicketPricer()
        : CertificationTicketPricerInterface;

    public function createAuthority()
        : CertificationAuthorityInterface;
}
```

```
namespace Certification;

use SebastianBergmann\Money\Money;

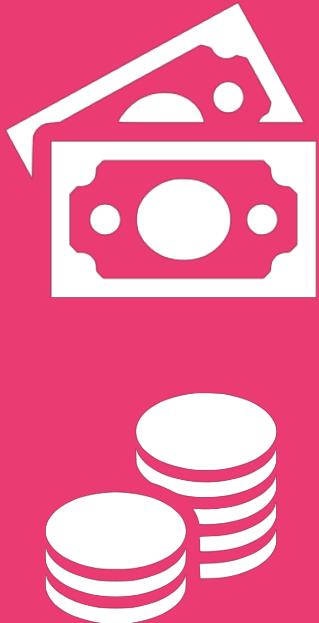
interface CertificationTicketPricerInterface
{
    public function getUnitPrice(string $country): Money;

    public function getTotalPrice(string $country, int $quantity): Money;
}
```

```
namespace Certification;

use Certification\Domain\AssessmentResult;
use Certification\Exception\CandidateNotCertifiedException;
use Certification\Exception\UnsupportedAssessmentException;

interface CertificationAuthorityInterface
{
    /**
     * Returns the candidate's certification level.
     *
     * @throws CandidateNotCertifiedException
     * @throws UnsupportedAssessmentException
     */
    public function getCandidateLevel(AssessmentResult $result): string;
}
```



Varying
Certifications
Tickets Pricing

```
$pricer = new \Certification\Symfony\TicketPricer(  
    Money::EUR('250'),  
    Money::EUR('200'));  
$price = $pricer->getUnitPrice('FR'); // €250  
$price = $pricer->getUnitPrice('TN'); // €200
```

```
$pricer = new \Certification\Twig\TicketPricer(  
    Money::EUR('149'));  
$price = $pricer->getUnitPrice('FR'); // €149  
$price = $pricer->getUnitPrice('TN'); // €149
```

```
namespace Certification;

use SebastianBergmann\Money\Money;

abstract class AbstractTicketPricer
    implements CertificationTicketPricerInterface
{
    /**
     * Returns the total price for the given quantity.
     */
    public function getTotalPrice(string $country, int $quantity): Money
    {
        if ($quantity < 1) {
            throw new \InvalidTicketQuantityException($quantity);
        }

        return $this->getUnitPrice($country)->multiply($quantity);
    }
}
```

```
namespace Certification\Twig;

use SebastianBergmann\Money\Money;
use Certification\AbstractTicketPricer;

class TicketPricer extends AbstractTicketPricer
{
    private $price;

    public function __construct(Money $price)
    {
        $this->price = $price;
    }

    public function getUnitPrice(string $country): Money
    {
        return $this->price;
    }
}
```

```
namespace Certification\Symfony;

use SebastianBergmann\Money\Money;
use Certification\AbstractTicketPricer;

class TicketPricer extends AbstractTicketPricer
{
    private $regularPrice;
    private $discountPrice;

    public function __construct(Money $regularPrice, Money $discountPrice)
    {
        $this->regularPrice = $regularPrice;
        $this->discountPrice = $discountPrice;
    }

    public function getUnitPrice(string $country): Money
    {
        if (in_array($country, ['FR', 'US', 'DE', 'IT', 'CH', 'BE', /* ... */])) {
            return $this->regularPrice;
        }

        return $this->discountPrice;
    }
}
```



Varying Certifications Authorities

```
try {

    $assessment = AssessmentResult::fromCSV('SL038744,SF3CE,35');
    $authority = new \Certification\Symfony\Authority(20, 10);
    $symfonyLevel = $authority->getCandidateLevel($assessment);

    $assessment = AssessmentResult::fromCSV('SL038744,TW1CE,17');
    $authority = new \Certification\Twig\Authority(15);
    $twigLevel = $authority->getCandidateLevel($assessment);

} catch (CandidateNotCertifiedException $e) {
    // ...
} catch (UnsupportedAssessmentException $e) {
    // ...
} catch (\Exception $e) {
    // ...
}
```

```
namespace Certification\Twig;

use Certification\CertificationAuthorityInterface;
use Certification\Domain\AssessmentResult;
use Certification\Exception\CandidateNotCertifiedException;
use Certification\Exception\UnsupportedAssessmentException;

class Authority implements CertificationAuthorityInterface
{
    // ...
    public function getCandidateLevel(AssessmentResult $result): string
    {
        if ('TW1CE' !== $examId = $result->getAssessmentID()) {
            throw new UnsupportedAssessmentException($examId);
        }

        if ($result->getScore() < $this->passingScore) {
            throw new CandidateNotCertifiedException(
                $result->getCandidateID(),
                $examId
            );
        }

        return 'expert';
    }
}
```

```
namespace Certification\Symfony;

use Certification\CertificationAuthorityInterface;
use Certification\Entity\AssessmentResult;
use Certification\Exception\CandidateNotCertifiedException;
use Certification\Exception\UnsupportedAssessmentException;

class Authority implements CertificationAuthorityInterface
{
    private $expertLevelPassingScore;
    private $advancedLevelPassingScore;

    public function __construct(int $expertScore, int $advancedScore)
    {
        $this->expertLevelPassingScore = $expertScore;
        $this->advancedLevelPassingScore = $advancedScore;
    }

    // ...
}
```

```
class Authority implements CertificationAuthorityInterface
{
    public function getCandidateLevel(AssessmentResult $result): string
    {
        $examId = $result->getAssessmentID();
        if (!in_array($examId, [ 'SF2CE' , 'SF3CE' ])) {
            throw new UnsupportedAssessmentException($examId);
        }

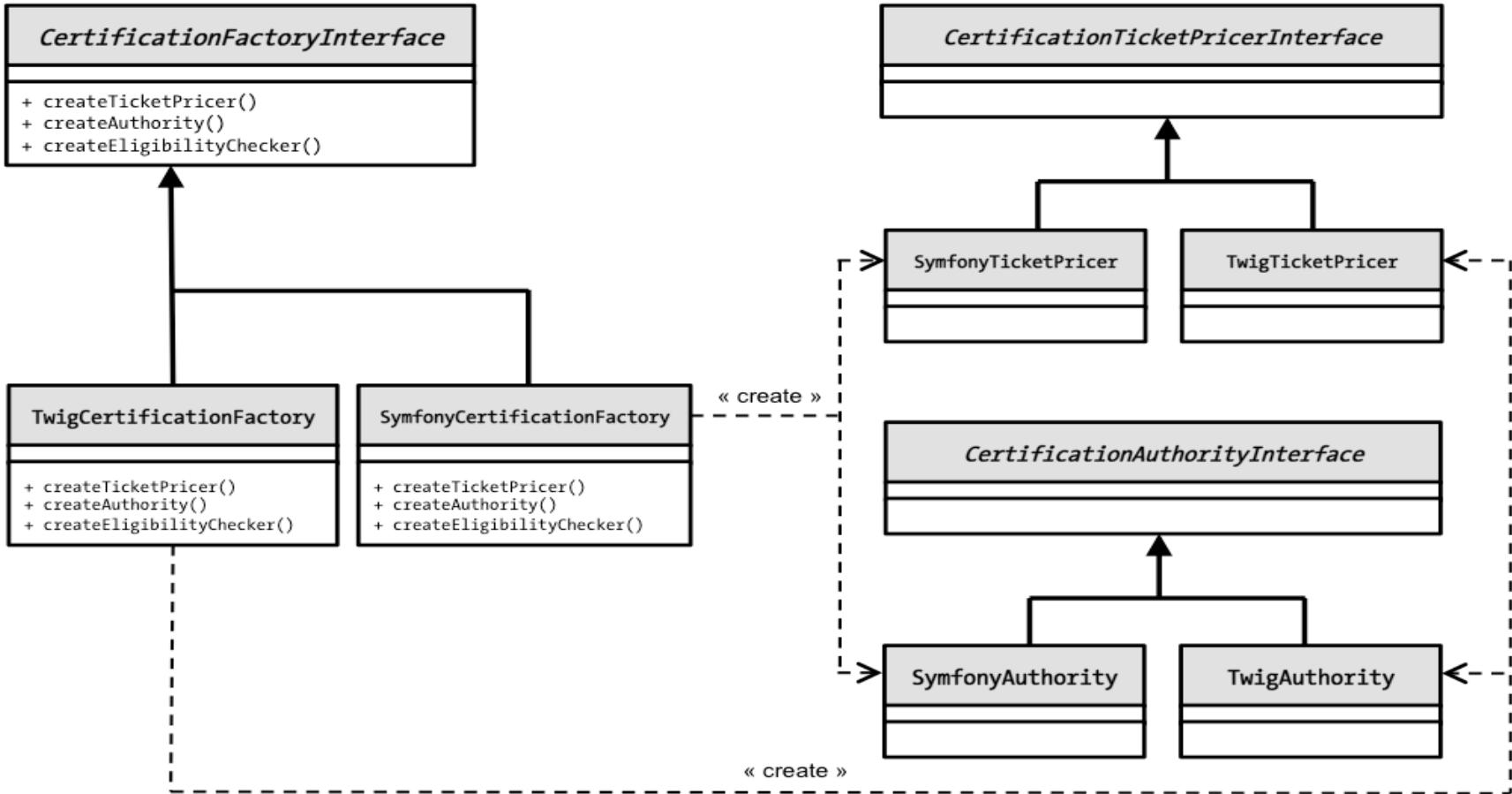
        $score = $result->getScore();
        if ($score >= $this->expertLevelPassingScore) {
            return 'expert';
        }

        if ($score >= $this->advancedLevelPassingScore) {
            return 'advanced';
        }

        throw new CandidateNotCertifiedException(
            $result->getCandidateID(),
            $examId
        );
    }
}
```

Implementing Concrete Factories





CertificationFactoryInterface

```
+ createTicketPricer()  
+ createAuthority()  
+ createEligibilityChecker()
```



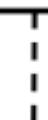
TwigCertificationFactory

```
+ createTicketPricer()  
+ createAuthority()  
+ createEligibilityChecker()
```

SymfonyCertificationFactory

```
+ createTicketPricer()  
+ createAuthority()  
+ createEligibilityChecker()
```

« create »



```
namespace Certification\Symfony;

use SebastianBergmann\Money\Money;
use Certification\CertificationAuthorityInterface;
use Certification\CertificationFactoryInterface;
// ...

class CertificationFactory implements CertificationFactoryInterface
{
    // ...

    public function createTicketPricer(): CertificationTicketPricerInterface
    {
        return new TicketPricer(Money::EUR('250'), Money::EUR('200'));
    }

    public function createAuthority(): CertificationAuthorityInterface
    {
        return new Authority(20, 10);
    }
}
```

```
namespace Certification\Twig;

use SebastianBergmann\Money\Money;
use Certification\CertificationAuthorityInterface;
use Certification\CertificationFactoryInterface;
// ...

class CertificationFactory implements CertificationFactoryInterface
{
    // ...

    public function createTicketPricer(): CertificationTicketPricerInterface
    {
        return new TicketPricer(Money::EUR('149'));
    }

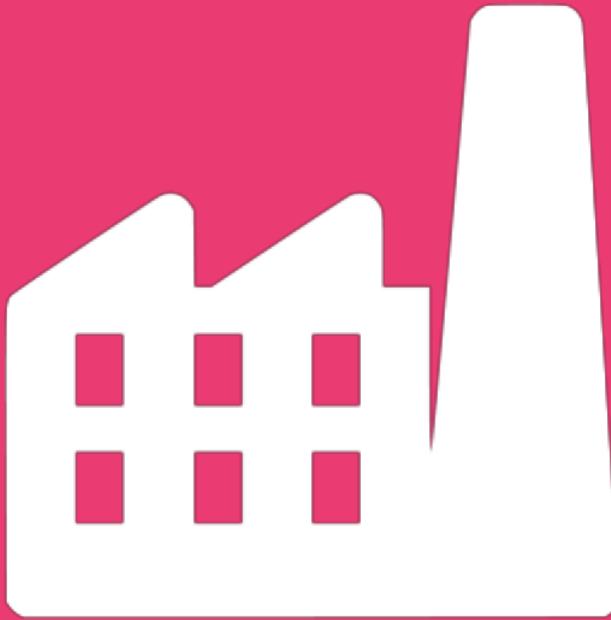
    public function createAuthority(): CertificationAuthorityInterface
    {
        return new Authority(20);
    }
}
```

```
$examSeries = 'SFXCE';

// Choose the right factory
switch ($assessmentFamily) {
    case 'TWXCE':
        $factory = new TwigCertificationFactory(...);
        break;
    case 'SFXCE':
        $factory = new SymfonyCertificationFactory(...);
        break;
    default:
        throw new UnsupportedAssessmentSeriesException($examSeries);
}

// Request and use the factory services
$pricer = $factory->createTicketPricer();
$checker = $factory->createEligibilityChecker();
$authority = $factory->createAuthority();
```

The Giga Factory



```
namespace Certification;

use Certification\Exception\UnsupportedAssessmentException;
//...

class CertificationFactoryFactory implements CertificationFactoryInterface
{
    private $factories = [];

    public function getFactory(string $assessment): CertificationFactoryInterface
    {
        if ($this->factories[$assessment] ?? false) {
            return $this->factories[$assessment];
        }

        if ('TW1CE' === $assessment) {
            return $this->createTwigCertificationFactory($assessment);
        }

        if (\in_array($assessment, ['SF2CE', 'SF3CE'], true)) {
            return $this->createSymfonyCertificationFactory($assessment);
        }

        throw new UnsupportedAssessmentException($assessment);
    }
}
```

```
class CertificationFactoryFactory implements CertificationFactoryInterface
{
    // ...
    private function createTwigCertificationFactory(string $assessment)
        : CertificationFactoryInterface
    {
        $this->factories[$assessment] = $f = new TwigCertificationFactory(...);

        return $f;
    }

    private function createSymfonyCertificationFactory(string $assessment)
        : CertificationFactoryInterface
    {
        $this->factories[$assessment] = $f = new SymfonyCertificationFactory(...);

        return $f;
    }
}
```

```
class CertificationFactoryFactory implements CertificationFactoryInterface
{
    // ...
    public function createCertificationTicketPricer(string $assessment)
        : CertificationTicketPricerInterface
    {
        return $this->getFactory($assessment)->createTicketPricer();
    }

    public function createCertificationAuthority(string $assessment)
        : CertificationAuthorityInterface
    {
        return $this->getFactory($assessment)->createCertificationAuthority();
    }

    public function createCertificationEligibilityChecker(string $assessment)
        : CertificationEligibilityCheckerInterface
    {
        return $this->getFactory($assessment)->createEligibilityChecker();
    }
}
```

```
class PlaceOrderCommandHandler
{
    private $factory;

    function __construct(CertificationFactoryFactoryInterface $factory)
    {
        $this->factory = $factory;
    }

    // ...

    private function getUnitPrice(PlaceOrderCommand $command): void
    {
        return $this
            ->factory
            ->getCertificationTicketPricer($command->getExamSeriesType())
            ->getUnitPrice($command->getCountry());
    }
}
```

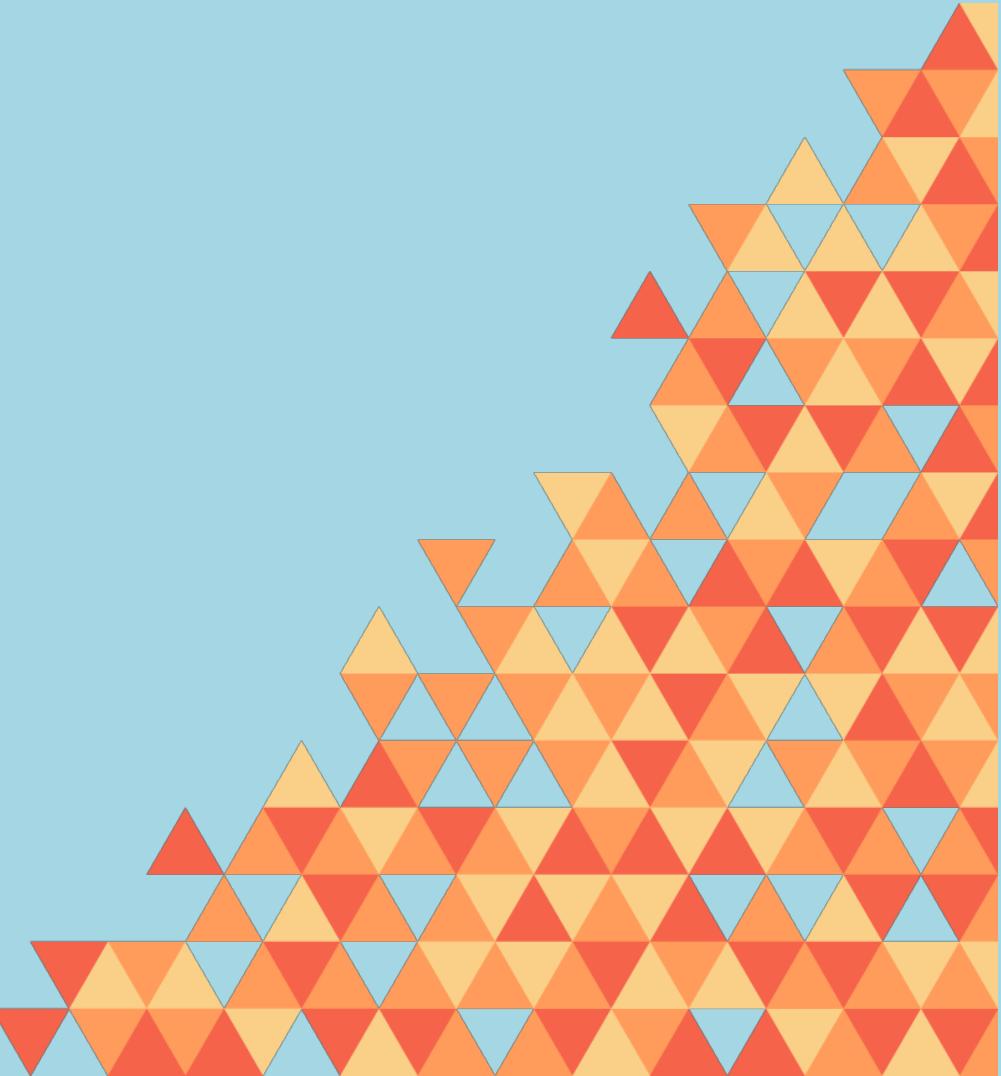
Benefits

- **Each factory produces one specific concrete type**
- **Easy to replace a concrete factory by another**
- **Adaptability to the run-time environment**
- **Objects construction is centralized**

Disadvantages

- **Lots of classes and interfaces are involved**
- **Client code doesn't know the exact concrete type**
- **Hard to implement**

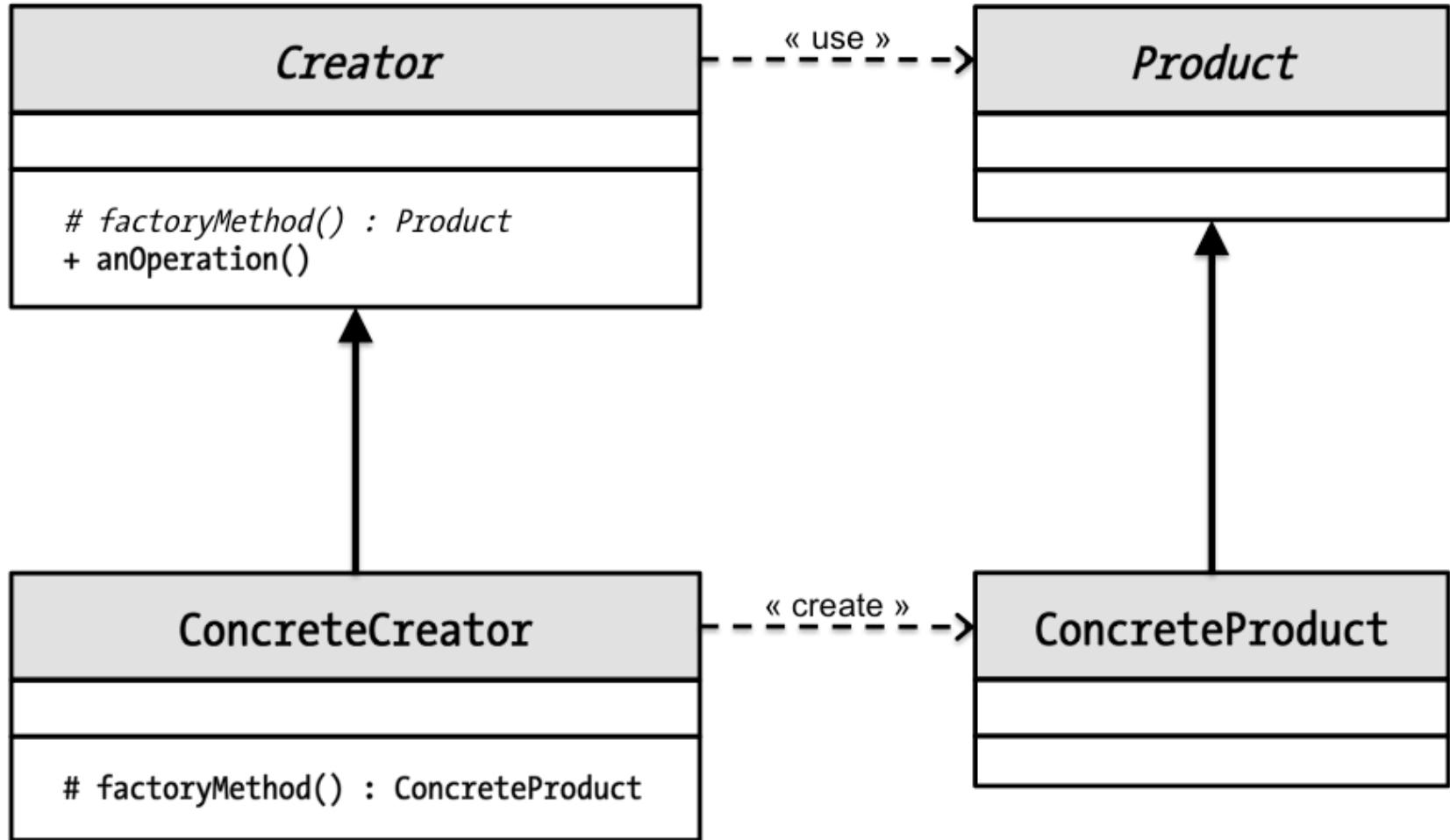
Factory Method

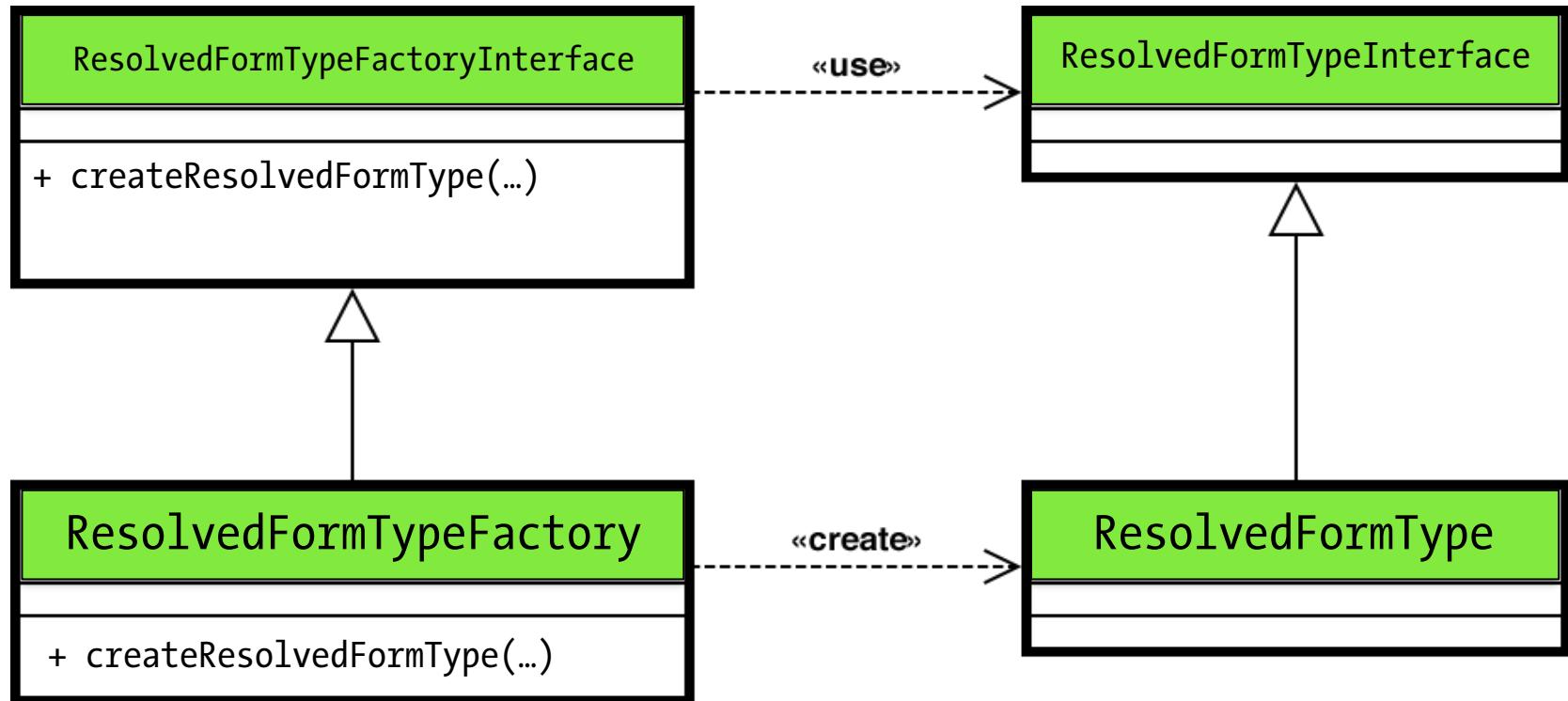


Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses.

– GoF





```
namespace Symfony\Component\Form;

interface ResolvedFormTypeFactoryInterface
{
    /**
     * @param FormTypeInterface $type
     * @param FormTypeExtensionInterface[] $typeExtensions
     * @param ResolvedFormTypeInterface|null $parent
     *
     * @return ResolvedFormTypeInterface
     */
    public function createResolvedType(
        FormTypeInterface $type,
        array $typeExtensions,
        ResolvedFormTypeInterface $parent = null
    );
}
```

```
namespace Symfony\Component\Form;

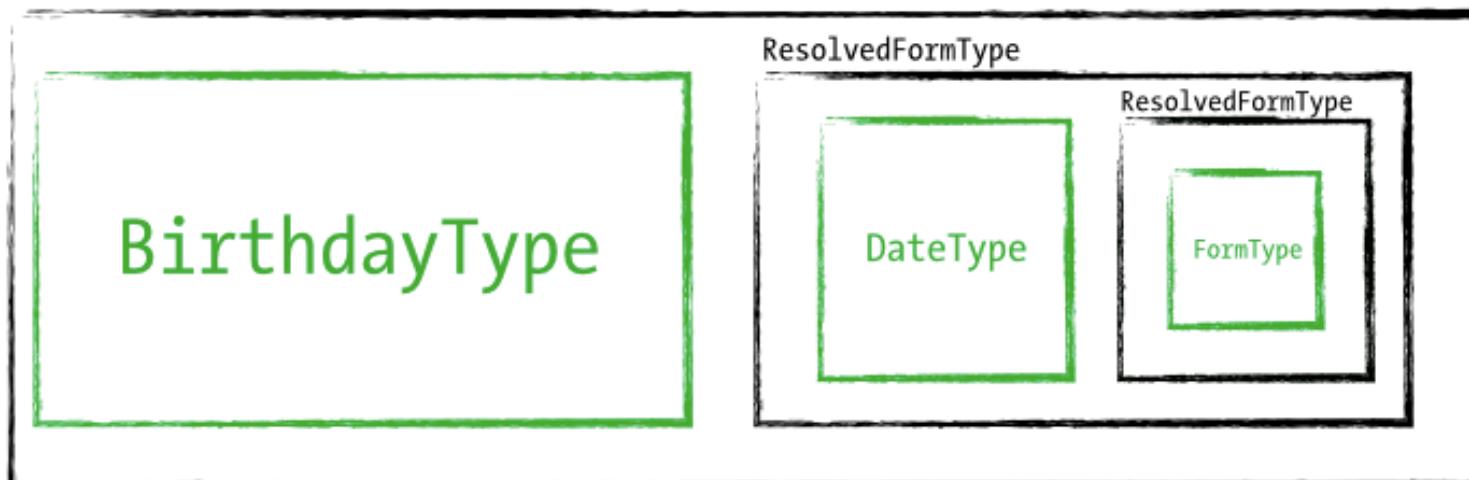
class ResolvedFormTypeFactory implements
ResolvedFormTypeFactoryInterface
{
    public function createResolvedType(
        FormTypeInterface $type,
        array $typeExtensions,
        ResolvedFormTypeInterface $parent = null
    ) {
        return new ResolvedFormType($type, $typeExtensions, $parent);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

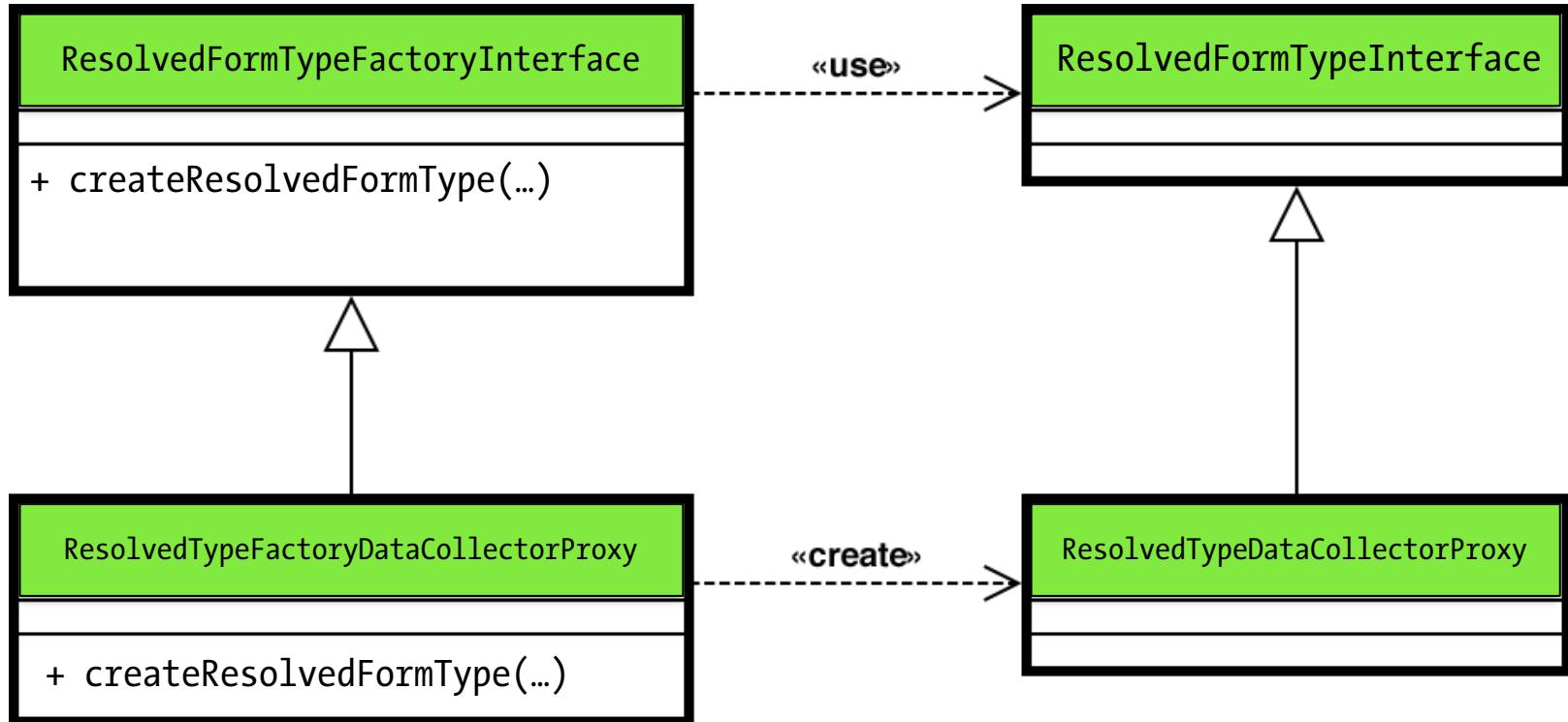


```
$f = new ResolvedFormTypeFactory();

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

ResolvedFormType





```
namespace Symfony\Component\Form\Extension\DataCollector\Proxy;

use Symfony\Component\Form\Extension\DataCollector\FormDataCollectorInterface;
use Symfony\Component\Form\FormTypeInterface;
use Symfony\Component\Form\ResolvedFormTypeFactoryInterface;
use Symfony\Component\Form\ResolvedFormTypeInterface;

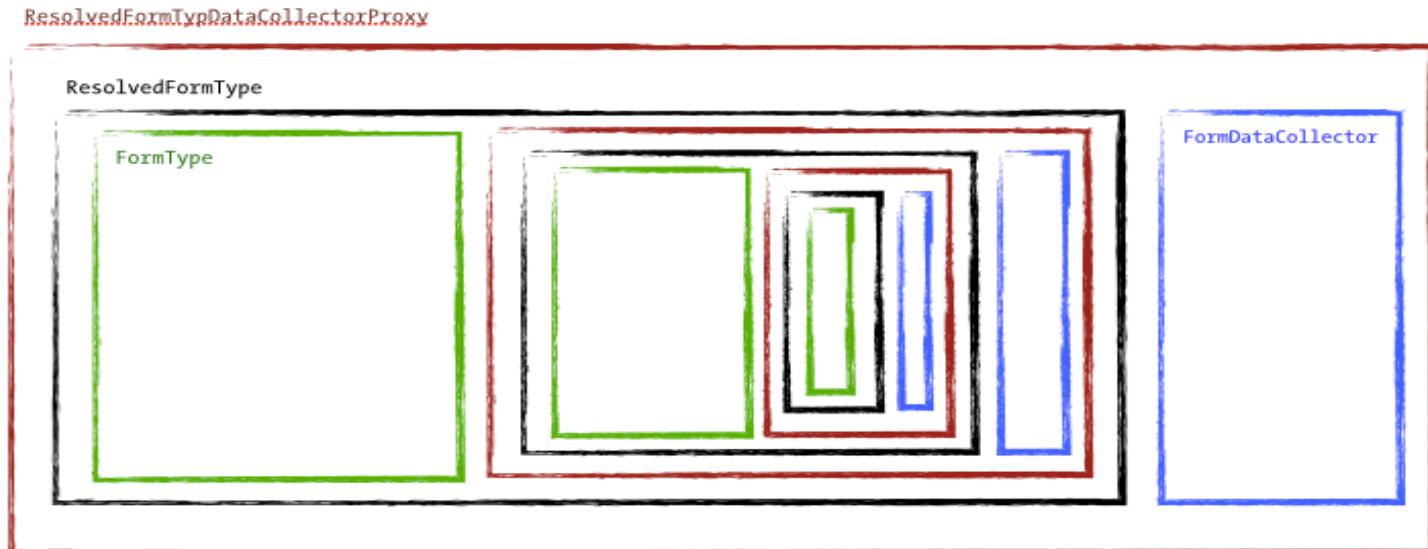
class ResolvedTypeFactoryDataCollectorProxy implements ResolvedFormTypeFactoryInterface
{
    private $proxiedFactory;
    private $dataCollector;

    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
        );
    }
}
```



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



```
class FormRegistry implements FormRegistryInterface
{
    /**
     * @var ResolvedFormTypeFactoryInterface
     */
    private $resolvedTypeFactory;

    private function resolveType(FormTypeInterface $type)
    {
        // ...
        try {
            // ...
            return $this->resolvedTypeFactory->createResolvedType(
                $type,
                $typeExtensions,
                $parentType ? $this->getType($parentType) : null
            );
        } finally {
            unset($this->checkedTypes[$fqcn]);
        }
    }
}
```

```
// Prod environment
$factory = new ResolvedFormTypeFactory();

// Dev environment
$factory = new ResolvedTypeFactoryDataCollectorProxy(
    new ResolvedFormTypeFactory(),
    new FormDataCollector(...))

);

// Factory injection
$registry = new FormRegistry(..., $factory);
$type = $registry->getType(EmailType::class);
```

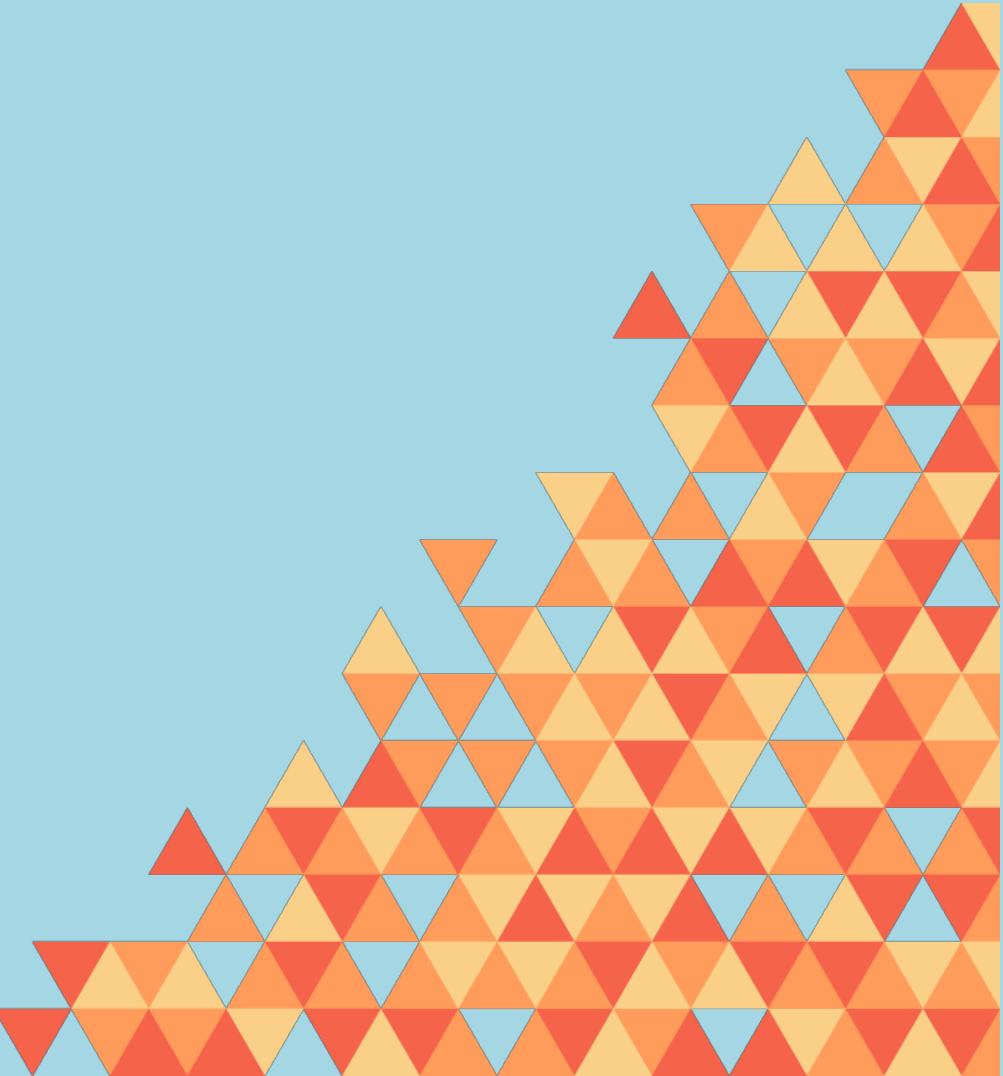
Benefits

- **Each factory produces one specific concrete type**
- **Easy to replace a concrete factory by another**
- **Adaptability to the run-time environment**
- **Objects construction is centralized**

Disadvantages

- **Lots of classes and interfaces are involved**
- **Client code doesn't know the exact concrete type**
- **Hard to implement**

Builder



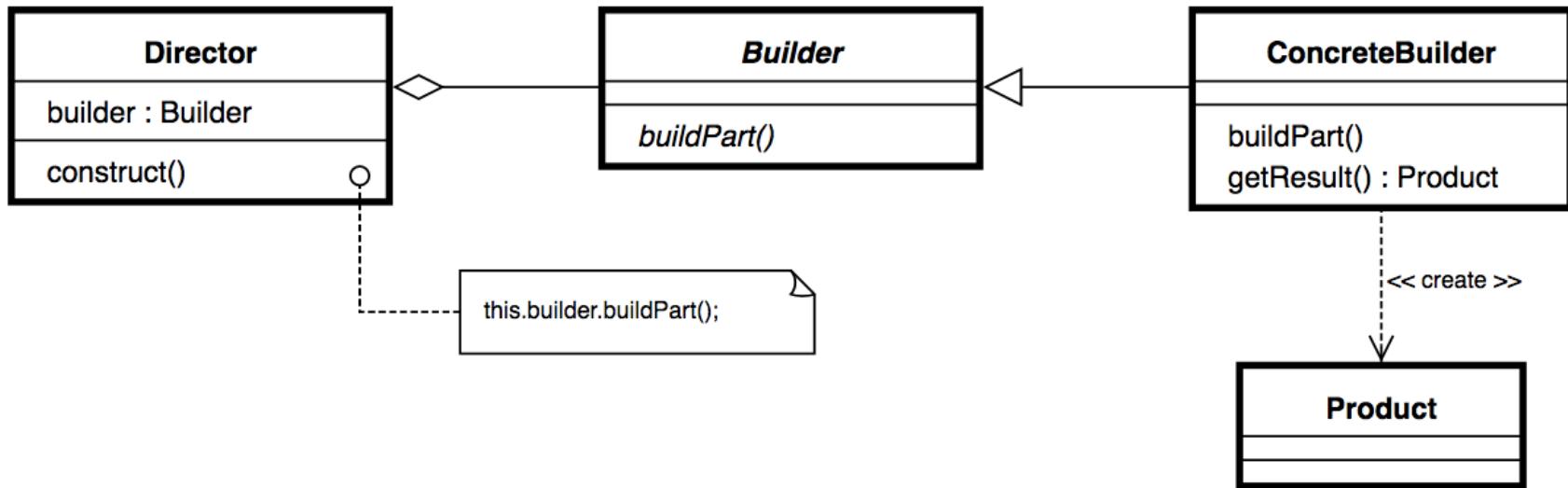
Builder

**The Builder design pattern
separates the construction of
a complex object from its
representation.**

– Wikipedia

Problems

- **Avoiding constructors that have too many optional parameters.**
- **Simplifying the process of creating a complex object.**
- **Abstract the steps order to assemble a complex object.**



Doctrine

Doctrine comes with a QueryBuilder object in order to provide a simpler way to produce a Query instance from a Repository.

```
class UserRepository extends EntityRepository
{
    public function byEmailAddress(string $email): ?User
    {
        $query = $this
            ->createQueryBuilder('u, p')
            ->leftJoin('u.profile', 'p')
            ->where('LOWER(u.emailAddress) = :email')
            ->andWhere('u.active = :active')
            ->setParameter('email', mb_strtolower($email))
            ->setParameter('active', 1)
            ->getQuery()
        ;

        return $query->getOneOrNullResult();
    }
}
```

```
class QueryBuilder
{
    // ...
    public function where($predicates)
    {
        if ( ! ($func_num_args() == 1 && $predicates instanceof Expr\Composite)) {
            $predicates = new Expr\Andx(func_get_args());
        }

        return $this->add('where', $predicates);
    }

    public function setMaxResults($maxResults)
    {
        $this->_maxResults = $maxResults;

        return $this;
    }
}
```

```
class QueryBuilder
{
    // ...
    public function getQuery()
    {
        $parameters = clone $this->parameters;
        $query      = $this->_em->createQuery($this->getDQL())
            ->setParameters($parameters)
            ->setFirstResult($this->_firstResult)
            ->setMaxResults($this->_maxResults);

        if ($this->lifetime) {
            $query->setLifetime($this->lifetime);
        }

        if ($this->cacheMode) {
            $query->setCacheMode($this->cacheMode);
        }

        if ($this->cacheable) {
            $query->setCacheable($this->cacheable);
        }

        if ($this->cacheRegion) {
            $query->setCacheRegion($this->cacheRegion);
        }

        return $query;
    }
}
```



Form

The Symfony Form component provides a FormBuilder object, which simplifies the construction and the initialization of a Form instance.

```
class RegistrationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('emailAddress', EmailType::class)
            ->add('firstName', TextType::class)
            ->add('lastName', TextType::class)
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
            ])
            ->add('submit', SubmitType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Registration::class,
        ]);
    }
}
```



```
interface FormBuilderInterface extends FormConfigBuilderInterface
{
    public function add($child, $type = null, array $options = []);
    public function create($name, $type = null, array $options = []);
    public function get($name);
    public function remove($name);
    public function has($name);
    public function all();
    public function getForm();
}
```

```
interface FormConfigBuilderInterface extends FormConfigInterface
{
    public function addEventListener($eventName, $listener, $priority = 0);
    public function addEventSubscriber(EventSubscriberInterface $subscriber);
    public function addViewTransformer(DataTransformerInterface $viewTransformer, $forcePrepend = false);
    public function resetViewTransformers();
    public function addModelTransformer(DataTransformerInterface $modelTransformer, $forceAppend = false);
    public function resetModelTransformers();
    public function setAttribute($name, $value);
    public function setAttributes(array $attributes);
    public function setDataMapper(DataMapperInterface $dataMapper = null);
    public function setDisabled($disabled);
    public function setEmptyData($emptyData);
    public function setErrorBubbling($errorBubbling);
    public function setRequired($required);
    public function setPropertyPath($propertyPath);
    public function setMapped($mapped);
    public function setByReference($byReference);
    public function setInheritData($inheritData);
    public function setCompound($compound);
    public function setType(ResolvedFormTypeInterface $type);
    public function setData($data);
    public function setDataLocked($locked);
    public function setFormFactory(FormFactoryInterface $formFactory);
    public function setAction($action);
    public function setMethod($method);
    public function setRequestHandler(RequestHandlerInterface $requestHandler);
    public function setAutoInitialize($initialize);
    public function getFormConfig();
}
```

```
class FormBuilder extends FormConfigBuilder implements \IteratorAggregate,  
FormBuilderInterface  
{  
    // ...  
    public function getForm()  
    {  
        if ($this->locked) {  
            throw new BadMethodCallException('...');  
        }  
  
        $this->resolveChildren();  
  
        $form = new Form($this->getFormConfig());  
          
        foreach ($this->children as $child) {  
            // Automatic initialization is only supported on root forms  
            $form->add($child->setAutoInitialize(false)->getForm());  
        }  
  
        if ($this->getAutoInitialize()) {  
            // Automatically initialize the form if it is configured so  
            $form->initialize();  
        }  
  
        return $form;  
    }  
}
```

Form

The Symfony Form component provides a FormFactoryBuilder object, which simplifies the construction and the initialization of a FormFactory instance.

```
$factory = (new FormFactoryBuilder())
    ->addExtension(new CoreExtension(...))
    ->addExtension(new CsrfExtension(...))
    ->addExtension(new ValidatorExtension(...))
    ->addType(new CustomFormType())
    ->addType(new OtherFormType())
    ->addTypeExtension(new EmojiRemoverTypeExtension())
    ->addTypeGuesser(new CustomTypeGuesser(...))
    ->getFormFactory()
;

;
```

```
class FormBuilderFactory implements FormBuilderFactoryInterface
{
    private $resolvedTypeFactory;
    private $extensions = array();
    private $types = array();
    private $typeExtensions = array();
    private $typeGuessers = array();

    // ...

    public function addExtension(FormExtensionInterface $extension)
    {
        $this->extensions[] = $extension;
        return $this;
    }

    public function addType(FormTypeInterface $type)
    {
        $this->types[] = $type;
        return $this;
    }

    public function addTypeExtension(FormTypeExtensionInterface $typeExtension)
    {
        $this->typeExtensions[$typeExtension->getExtendedType()][] = $typeExtension;
        return $this;
    }

    public function addTypeGuesser(FormTypeGuesserInterface $typeGuesser)
    {
        $this->typeGuessers[] = $typeGuesser;
        return $this;
    }
}
```

```
class FormBuilderFactory implements FormBuilderFactoryInterface
{
    // ...

    public function getFormFactory()
    {
        $extensions = $this->extensions;

        if (count($this->types) > 0 || count($this->typeExtensions) > 0 || count($this->typeGuessers) > 0) {
            if (count($this->typeGuessers) > 1) {
                $typeGuesser = new FormTypeGuesserChain($this->typeGuessers);
            } else {
                $typeGuesser = isset($this->typeGuessers[0]) ? $this->typeGuessers[0] : null;
            }

            $extensions[] = new PreloadedExtension($this->types, $this->typeExtensions, $typeGuesser);
        }

        return new FormFactory(new FormRegistry(
            $extensions,
            $this->resolvedTypeFactory ?: new ResolvedFormTypeFactory()
        ));
    }
}
```

Validator

The Symfony Validator component provides a ConstraintViolationBuilder object, which simplifies the construction of a new ViolationConstraint instance.

```
class ExecutionContext implements ExecutionContextInterface
{
    private $root;
    private $translator;
    private $translationDomain;
    private $violations;
    private $value;
    private $propertyPath = '';
    private $constraint;

    // ...

    public function buildViolation($message, array $parameters = [])
    {
        return new ConstraintViolationBuilder(
            $this->violations,
            $this->constraint,
            $message,
            $parameters,
            $this->root,
            $this->propertyPath,
            $this->value,
            $this->translator,
            $this->translationDomain
        );
    }
}
```

```
class ConstraintViolationBuilder implements ConstraintViolationBuilderInterface
{
    // ...

    public function atPath($path)
    {
        $this->propertyPath = PropertyPath::append($this->propertyPath, $path);
        return $this;
    }

    public function setParameter($key, $value)
    {
        $this->parameters[$key] = $value;
        return $this;
    }

    public function setInvalidValue($invalidValue)
    {
        $this->invalidValue = $invalidValue;
        return $this;
    }

    public function setPlural($number)
    {
        $this->plural = $number;
        return $this;
    }
}
```

```
class ConstraintViolationBuilder implements ConstraintViolationBuilderInterface
{
    // ...
    public function addViolation()
    {
        if (null === $this->plural) {
            $translatedMessage = $this->translator->trans(
                $this->message,
                $this->parameters,
                $this->translationDomain
            );
        } else {
            try {
                $translatedMessage = $this->translator->transChoice(
                    $this->message,
                    $this->plural,
                    $this->parameters,
                    $this->translationDomain
                );
            } catch (\InvalidArgumentException $e) {
                $translatedMessage = $this->translator->trans(
                    $this->message,
                    $this->parameters,
                    $this->translationDomain
                );
            }
        }

        $this->violations->add(new ConstraintViolation(
            $translatedMessage,
            $this->message,
            $this->parameters,
            $this->root,
            $this->propertyPath,
            $this->invalidValue,
            $this->plural,
            $this->code,
            $this->constraint,
            $this->cause
        ));
    }
}
```

Translate the error message.

Construct the violation object and add it to the list.

```
class UniqueEntityValidator extends ConstraintValidator
{
    // ...
    public function validate($entity, Constraint $constraint)
    {
        // ...
        $value = $this->formatWithIdentifiers($em, $class, $invalidValue);

        $this->context->buildViolation($constraint->message)
            ->atPath($errorPath)
            ->setParameter('{{ value }}', $value)
            ->setInvalidValue($invalidValue)
            ->setCode(UniqueEntity::NOT_UNIQUE_ERROR)
            ->setCause($result)
            ->addViolation();
    }
}
```

Benefits

- **Avoid constructor with many optional arguments**
- **No need to know the exact order of build steps**
- **Leverage fluent interfaces**
- **Ideal for high level of encapsulation & consistency**
- **Different builder implementations can be offered**

Disadvantages

- **Duplicated code in builder and builded object classes**
- **Sometimes very verbose**

Differences with Abstract Factory

- *Abstract Factory* emphasizes a **family of product objects** (either simple or complex). *Builder* focuses on constructing a **complex object** step by step.
- *Abstract Factory* focuses on **what** is made. *Builder* focus on **how** it is made.
- *Abstract Factory* focuses on defining **many different types of factories** to build **many products**, and it is not a one builder for just one product. *Builder* focus on building a **one complex** but **one single product**.
- *Abstract Factory* **defers** the choice of what concrete type of object to make **until run time**. *Builder* hides the logic/operation of **how to compile** that complex object.
- In *Abstract Factory*, **every method call** creates and returns different objects. In *Builder*, **only the last method call** returns the object, while other calls partially build the object

#3

Structural Design Patterns

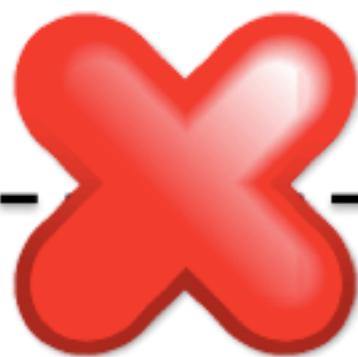
Adapter



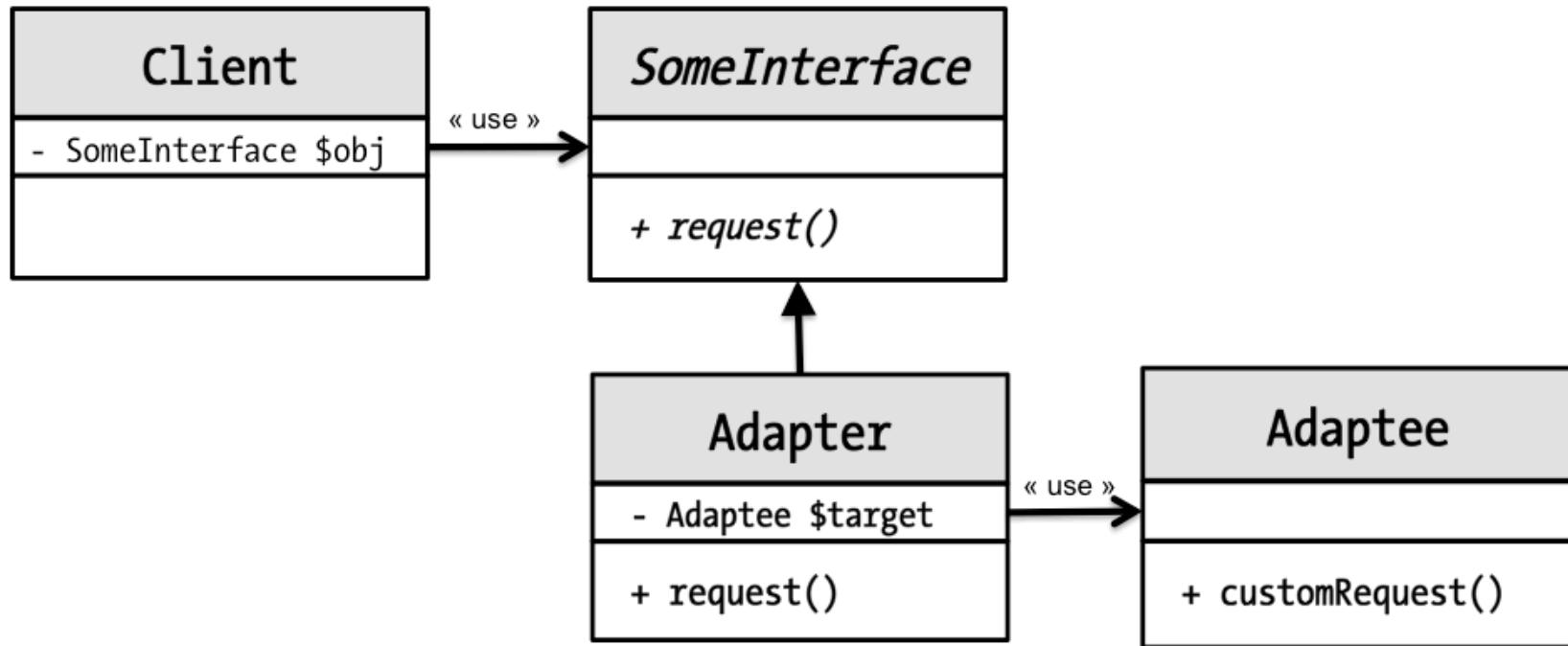
Adapter

**The Adapter pattern makes
two incompatible objects
work together without
changing their interfaces.**

– GoF







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 Symfony CSRF API

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

interface CsrfProviderInterface
{
    public function generateCsrfToken($intention);

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

The old Symfony CSRF API

```
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 Symfony CSRF API

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

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

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

The new Symfony CSRF API

```
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 both API 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()
        );
    }
}
```

Benefits

- **Easy to implement**
- **Leverage object composition**
- **Do not break existing interfaces**
- **Ideal to maintain BC layers**
- **Ideal to isolate legacy code**

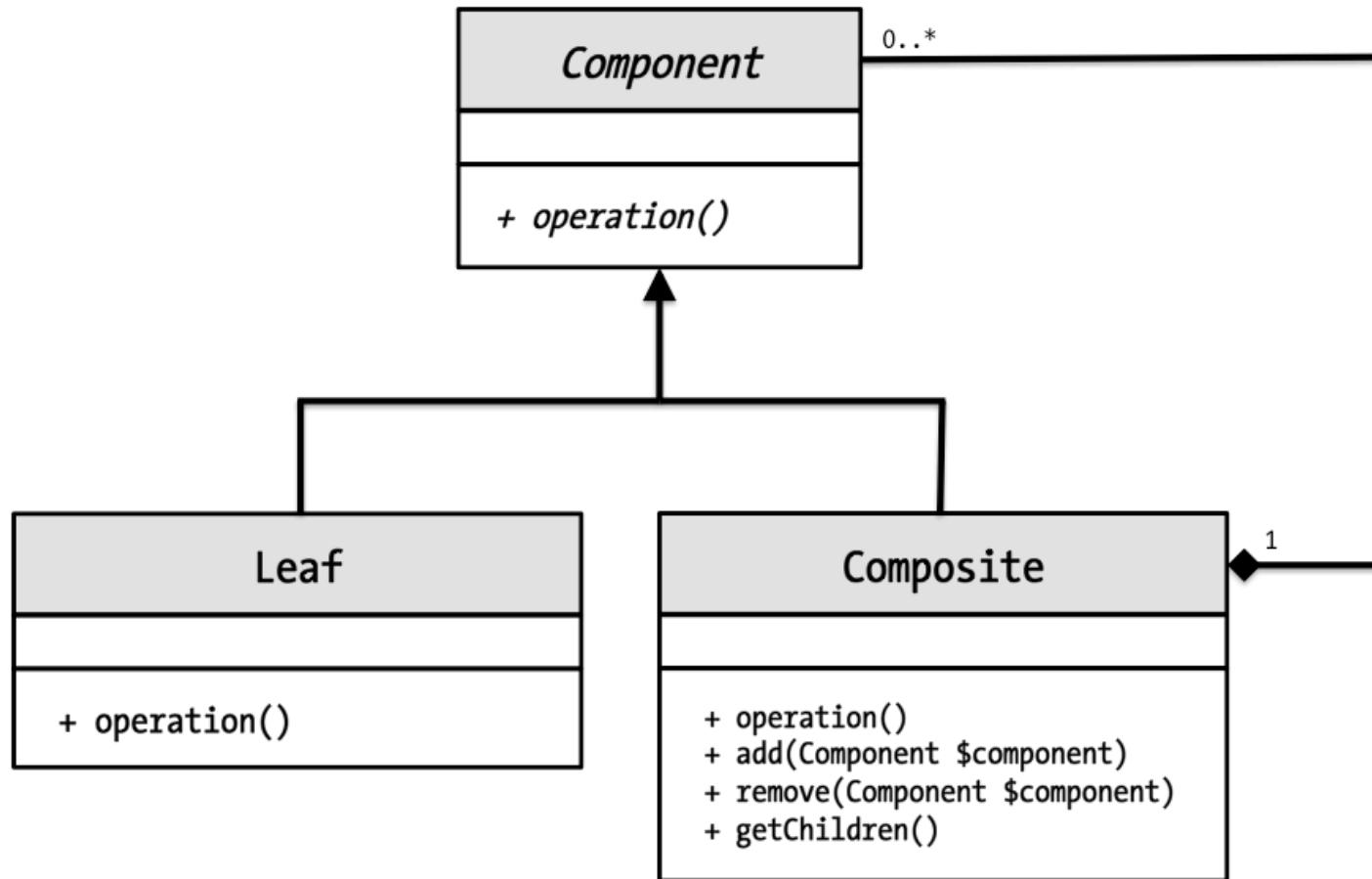
Composite



Composite

The Composite pattern lets clients treat single objects compositions of objects uniformly with a common interface.

– GoF



Usage examples

- Representing a binary tree
- Modelling a multi nested level navigation bar
- Parsing an XML / HTML document
- Submitting & validating nested Web forms
- Iterating over a filesystem
- ...

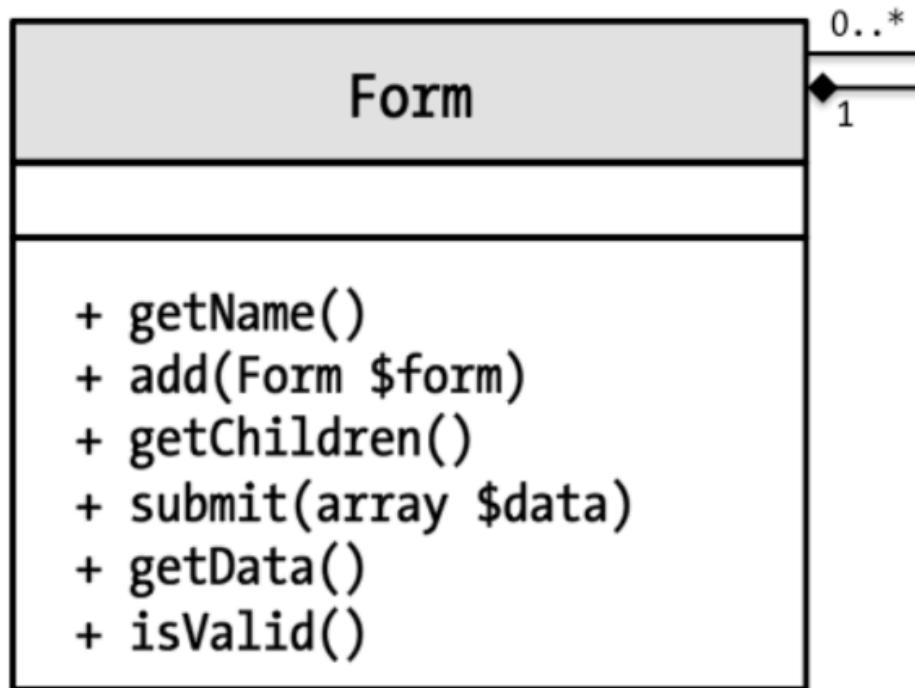
```
$nestedComposite = new ConcreteComposite();
$nestedComposite->add(new ConcreteLeaf());
$nestedComposite->add(new ConcreteLeaf());

$composite = new ConcreteComposite();
$composite->add(new ConcreteLeaf());
$composite->add(new ConcreteLeaf());
$composite->add($nestedComposite);
$composite->operation();

$leaf = new ConcreteLeaf();
$leaf->operation();
```

Symfony Forms

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

```
namespace Symfony\Component\Form;

class Form implements FormInterface
{
    private $name;

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

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

```
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);
```

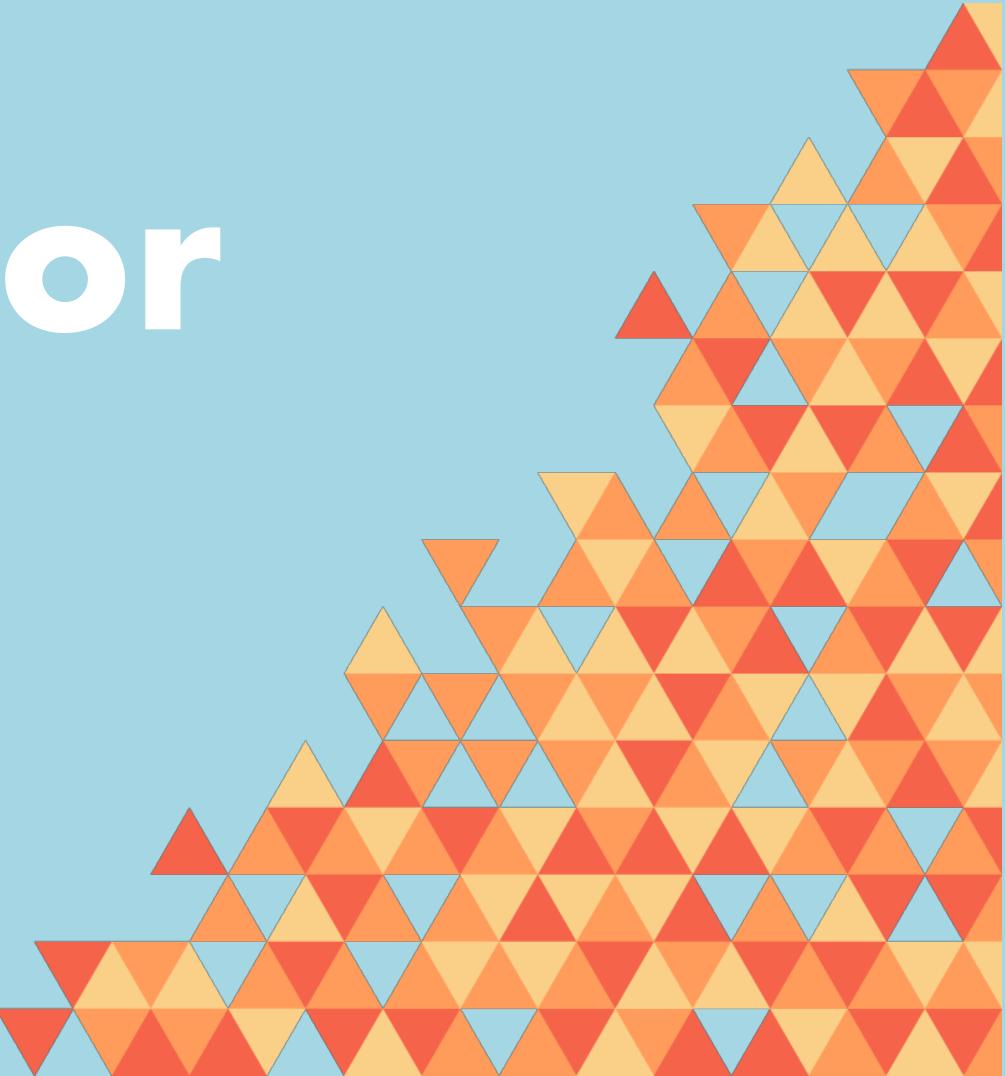
Submitting the form data

```
$form->submit([
    'name' => 'Apple Macbook Air 11',
    'description' => 'The thinnest laptop',
    'picture' => [
        '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);
            }
        }
    }
}
```

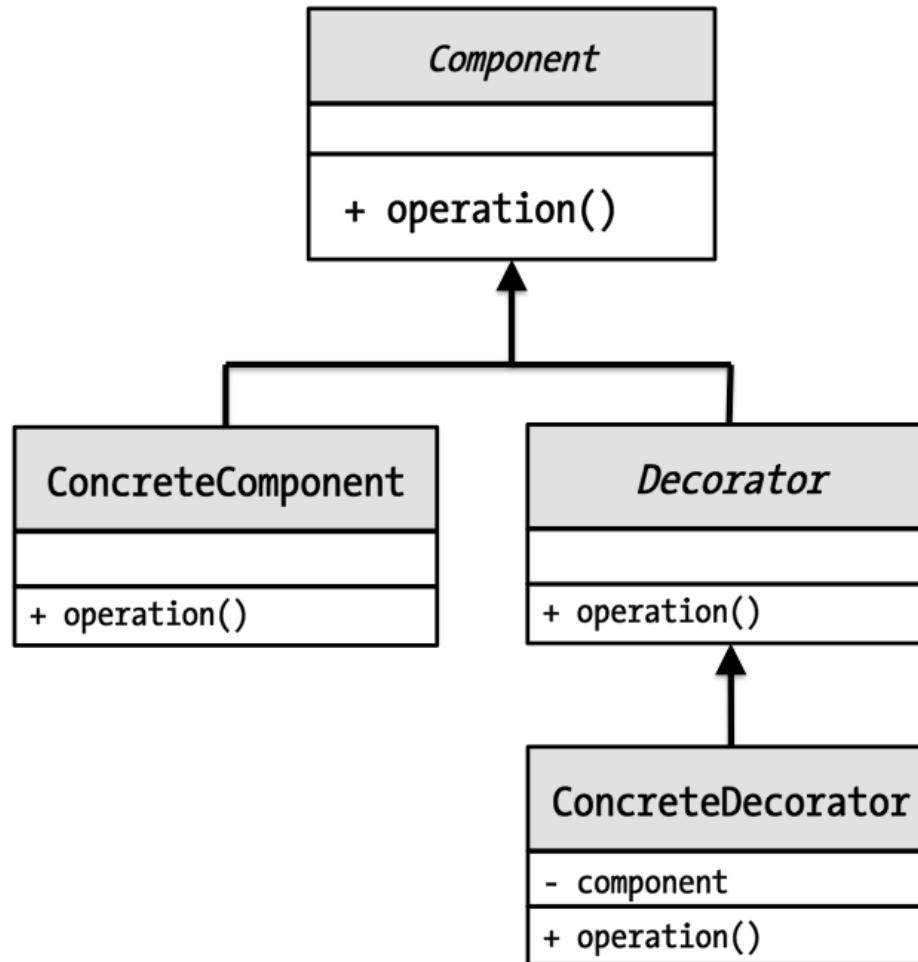
Decorator



Decorator

**The Decorator pattern allows
to add new responsibilities to
an object without changing its
class.**

– GoF



Why using it?

Extending objects without bloating the code

Making code reusable and composable

Avoiding vertical inheritance

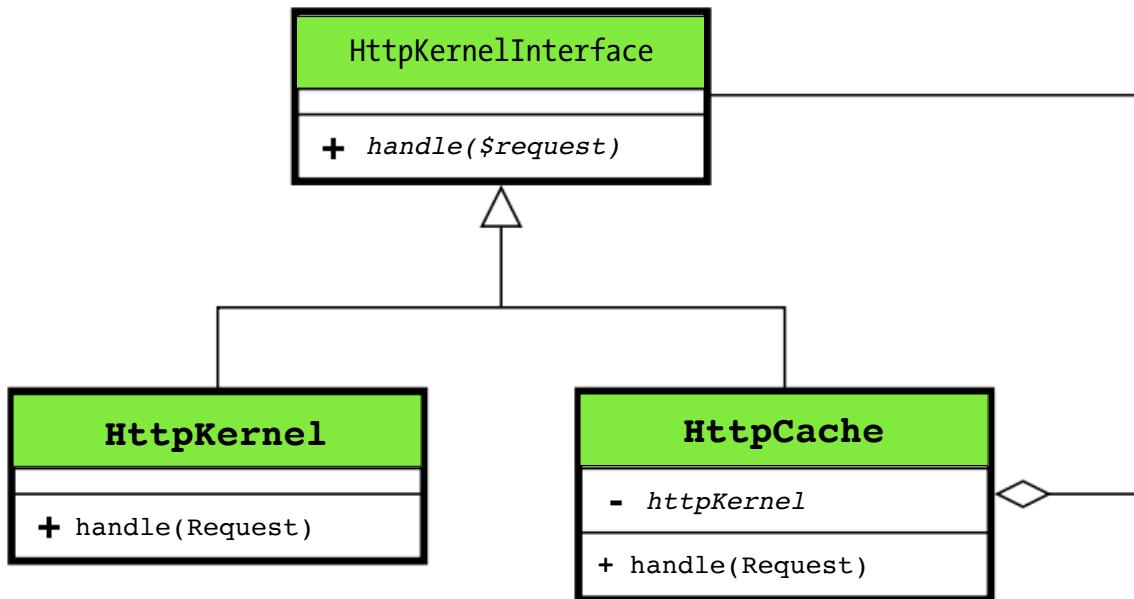
HttpKernel

The **HttpKernel component** comes with the famous **HttpKernelInterface** interface. This interface is implemented by the **HttpKernel**, **Kernel**, and **HttpCache** classes as well.

Adding an HTTP caching layer

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

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



```
// 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()
;
```

```
class HttpCache implements HttpKernelInterface, TerminableInterface
{
    private $kernel;
    // ...

    public function __construct(HttpKernelInterface $kernel, ...)
    {
        $this->kernel = $kernel;
        // ...
    }

    public function handle(Request $request, ...)
    {
        // ...
    }
}
```

```
class HttpCache implements HttpKernelInterface, TerminableInterface
{
    protected function forward(Request $request, $catch = false, Response $entry = null)
    {
        // ...

        // make sure HttpCache is a trusted proxy
        if (!in_array('127.0.0.1', $trustedProxies = Request::getTrustedProxies())) {
            $trustedProxies[] = '127.0.0.1';
            Request::setTrustedProxies($trustedProxies, Request::HEADER_X_FORWARDED_ALL);
        }

        // always a "master" request (as the real master request can be in cache)
        $response = $this->kernel->handle($request, ...);

        // ...

        return $response;
    }
}
```

DependencyInjection

The `DependencyInjection` component provides a way to define service definition decorators in order to easily decorate services.

```
# config/services.yaml
services:
    App\Mailer: ~

    App\DecoratingMailer:
        # overrides the App\Mailer service
        # but that service is still available as
        # App\DecoratingMailer.inner
        decorates: App\Mailer

        # pass the old service as an argument
        arguments: [ '@App\DecoratingMailer.inner' ]

    # private, because usually you do not need
    # to fetch App\DecoratingMailer directly
    public:      false
```

Generated code in the container

```
$this->services[App\Mailer::class] = new  
App\DecoratingMailer(  
    new App\Mailer(  
        ...  
    )  
)  
;
```

Benefits

- Easy way to extend an object's capabilities
- No need to change the existing code
- Leverage SRP and OCP principles

Disadvantages

- Object construction becomes more complex
- Does not work well for objects with a large public API
- Difficulty to access the real concrete object

Flyweight



Flyweight

The Flyweight pattern is used to reduce the memory and resource usage for complex models containing many hundreds, thousands or hundreds of thousands of similar objects.

– GoF

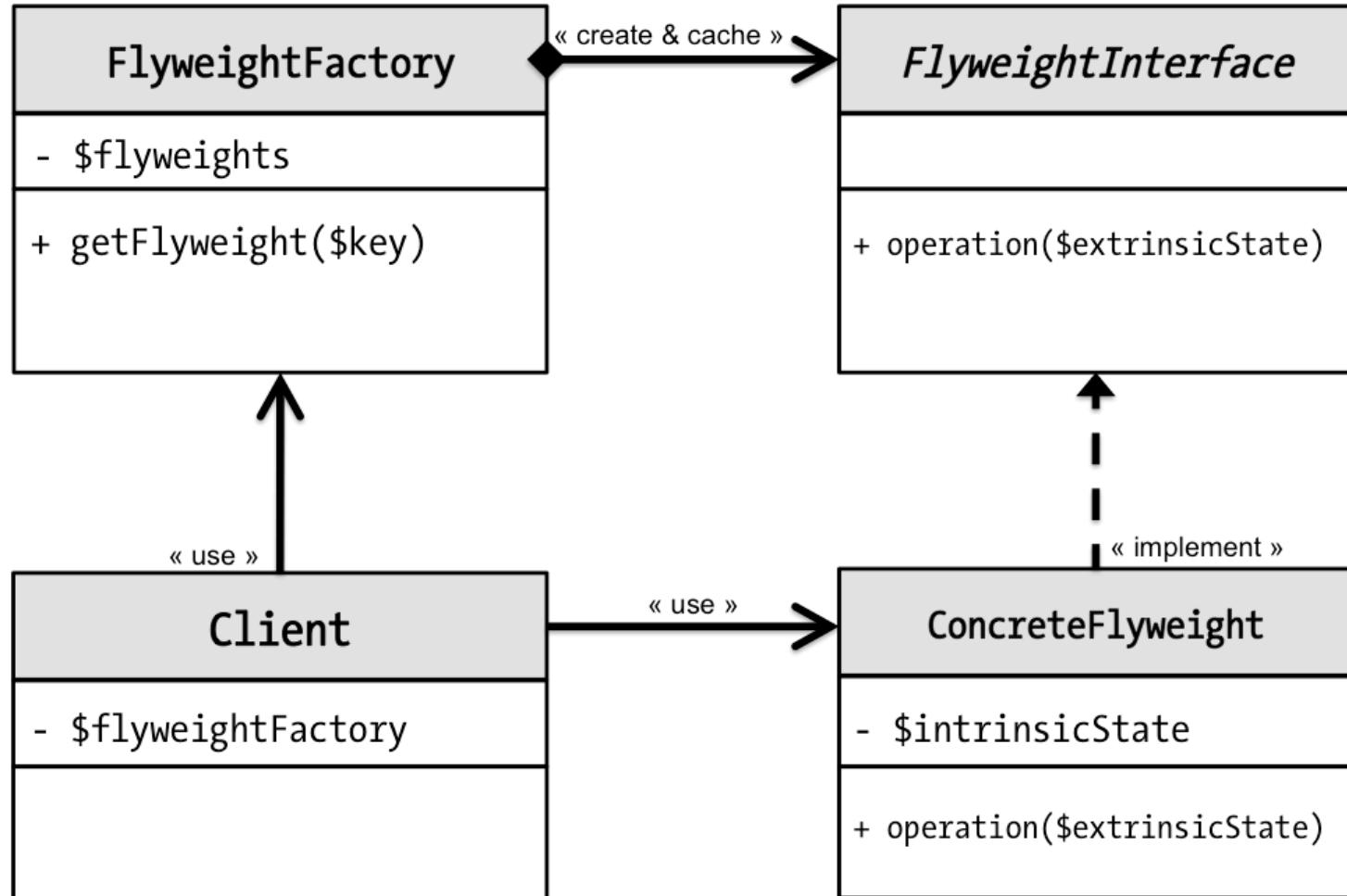
Main challenges of Flyweight

Sharing and reusing instances

Creating objects on-demand with a factory

Keeping memory usage as low as possible

Handling huge amount of similar objects



Intrinsic state

The **intrinsic state** (aka **Flyweight**) is defined as a simple **immutable value object** that encapsulates the common shared properties of all distinct entities.

Extrinsic state

The **extrinsic state** refers to the attributes that distinguish objects from each other. This state is **always extracted** and kept outside of the flyweight object. It can be kept in a separate entity or passed as an argument of the flyweight instance methods.

Symfony Forms

In the Symfony Form framework, form types instances are in fact designed as Flyweight objects. The same form type instance can be reused several times in the same form or several different forms.

```
namespace Symfony\Component\Form\Extension\Core\Type;

use Symfony\Component\Form\AbstractType;

class EmailType extends AbstractType
{
    public function getParent()
    {
        return __NAMESPACE__.\TextType';
    }

    public function getBlockPrefix()
    {
        return 'email';
    }
}
```

```
class RegistrationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('emailAddress', EmailType::class)
            ->add('firstName', TextType::class)
            ->add('lastName', TextType::class)
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
            ])
            ->add('submit', SubmitType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Registration::class,
        ]);
    }
}
```



Extrinsic State

```
class FormFactory implements FormFactoryInterface
{
    /** @var FormRegistry */
    private $registry;

    // ...

    public function createNamedBuilder($name, $type, $data = null, array $options = array())
    {
        if (null !== $data && !array_key_exists('data', $options)) {
            $options['data'] = $data;
        }

        if (!is_string($type)) {
            throw new UnexpectedTypeException($type, 'string');
        }

        $type = $this->registry->getType($type);
        $builder = $type->createBuilder($this, $name, $options);

        // Explicitly call buildForm() in order to be able to override either
        // createBuilder() or buildForm() in the resolved form type
        $type->buildForm($builder, $builder->getOptions());

        return $builder;
    }
}
```

← **Intrinsic State**

← **Extrinsic State**

```
class FormRegistry implements FormRegistryInterface
{
    /** @var FormTypeInterface[] */
    private $types = [];

    // ...
    public function getType($name)
    {
        if (!isset($this->types[$name])) {
            $type = null;
            foreach ($this->extensions as $extension) {
                if ($extension->hasType($name)) {
                    $type = $extension->getType($name);
                    break;
                }
            }

            if (!$type) {
                // Support fully-qualified class names
                if (!class_exists($name) || !is_subclass_of($name, 'Symfony\Component\Form\FormTypeInterface')) {
                    throw new InvalidArgumentException(...);
                }
                $type = new $name();
            }
            $this->types[$name] = $this->resolveType($type);
        }
        return $this->types[$name];
    }
}
```

Load form type from registered extension

Lazy load custom form type instance

Served already resolved form type

```
namespace Symfony\Component\Form\Extension\Core;  
  
//...  
class CoreExtension extends AbstractExtension  
{  
    // ...  
  
    protected function loadTypes()  
    {  
        return array(  
            new Type\FormType($this->propertyAccessor),  
            ...,  
            new Type\EmailType(),  
            ...,  
            new Type\RepeatedType(),  
            ...,  
            new Type\TextType(),  
            ...,  
        );  
    }  
}
```

```
abstract class AbstractExtension implements FormExtensionInterface
{
    /** @var FormTypeInterface[] */
    private $types;

    // ...

    public function getType($name)
    {
        if (null === $this->types) {
            $this->initTypes();
        }

        if (!isset($this->types[$name])) {
            throw new InvalidArgumentException(...);
        }

        return $this->types[$name];
    }
}
```

Benefits

- **Easy to implement**
- **Leverage value objects**
- **Reduce memory usage**
- **Great for handling large numbers of objects**

Downsides

- **Need a factory**

#4

Behavioral Design Patterns

Iterator



Iterator

The Iterator pattern provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

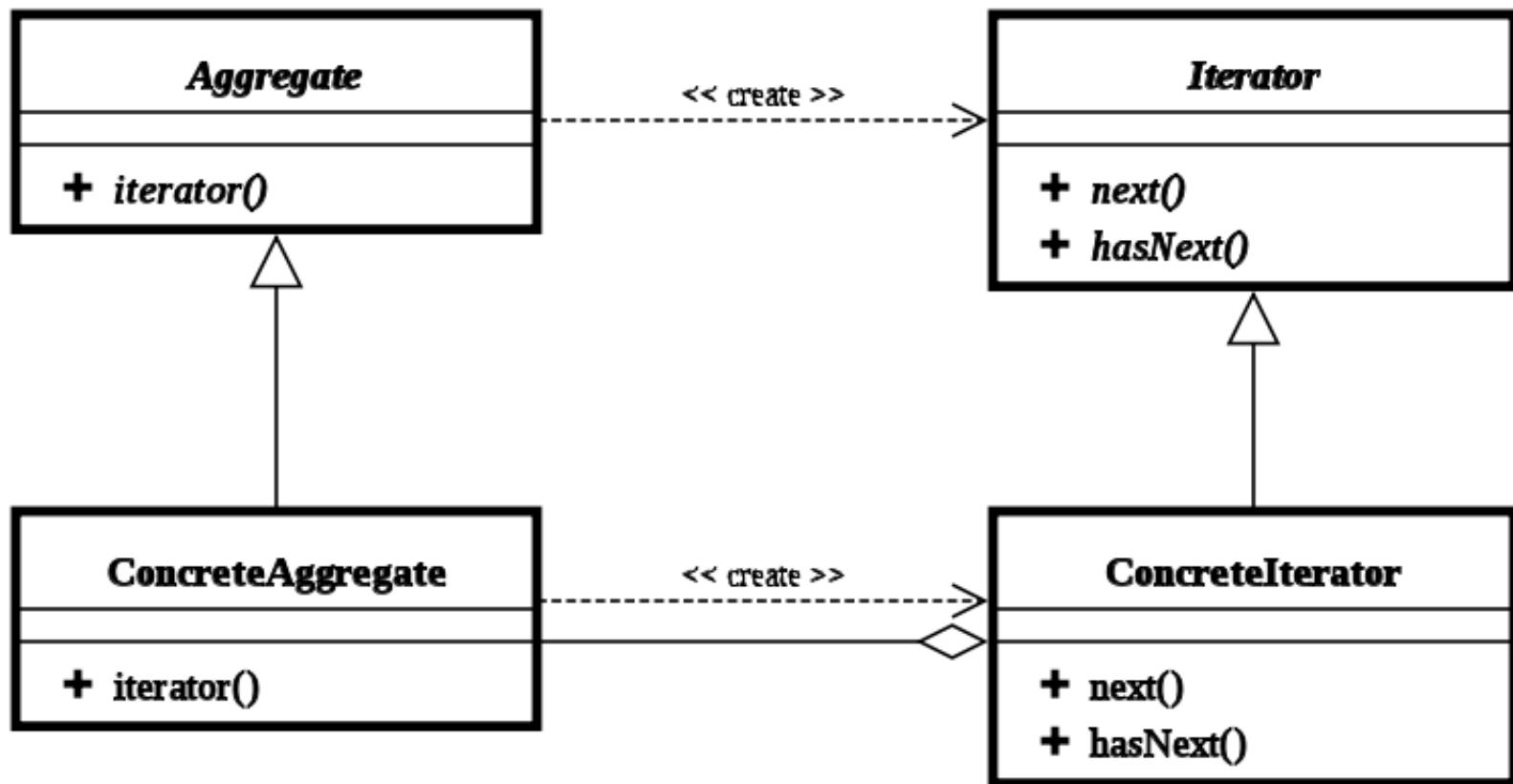
– GoF

Main goals of Iterator

- Accessing and traversing an aggregate object without exposing its representation (data structures).
- Adding new traversal operations on the aggregate should not force it to change its interface.

When using it?

- Modeling generic or custom objects collections
- Performing a set of operations on an aggregate
- Filtering or reducing a collection of objects
- Easing recursive operations on an aggregate
- Sorting items in a collection
- Lazy loading data from a datastore



Routing

In the Symfony Routing component, the RouteCollection class is an implementation of a simple iterator allowing it to be traversed.

```
class RouteCollection implements \IteratorAggregate
{
    /** @var Route[ ] */
    private $routes = [ ];
    private $resources = array();

    // ...

    public function getIterator()
    {
        return new \ArrayIterator($this->routes);
    }
}
```



```
$routes = new RouteCollection();
$routes->add('foo', new Route('/foo'));
$routes->add('bar', new Route('/bar'));
$routes->add('baz', new Route('/baz'));

foreach ($routes as $name => $route) {
    echo sprintf(
        'Route "%s" maps "%s" ' ,
        $name,
        $route->getPath()
    );
}
```

Finder

The Symfony Finder component provides several iterators to traverse a filesystem. Concrete iterators help filtering and reducing the list of files based on custom search criteria (size, date, name, etc.).

```
$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

```
class Finder implements \IteratorAggregate, \Countable
{
    // ...

    public function getIterator()
    {
        if (0 === count($this->dirs) && 0 === count($this->iterators)) {
            throw new \LogicException('You must call one of in() or append() first.');
        }

        if (1 === count($this->dirs) && 0 === count($this->iterators)) {
            return $this->searchInDirectory($this->dirs[0]);
        }

        $iterator = new \AppendIterator();
        foreach ($this->dirs as $dir) {
            $iterator->append($this->searchInDirectory($dir));
        }

        foreach ($this->iterators as $it) {
            $iterator->append($it);
        }

        return $iterator;
    }
}
```



```
class Finder implements \IteratorAggregate, \Countable
{
    // ...
    private function searchInDirectory(string $dir): \Iterator
    {
        // ...
        $iterator = new Iterator\RecursiveDirectoryIterator($dir, $flags, $this->ignoreUnreadableDirs);

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

        $iterator = new \RecursiveIteratorIterator($iterator, \RecursiveIteratorIterator::SELF_FIRST);
        if ($minDepth > 0 || $maxDepth < PHP_INT_MAX) {
            $iterator = new Iterator\DepthRangeFilterIterator($iterator, $minDepth, $maxDepth);
        }

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

        if ($this->names || $this->notNames) {
            $iterator = new Iterator\FilenameFilterIterator($iterator, $this->names, $this->notNames);
        }
        // ...

        return $iterator;
    }
}
```



Iterator of iterators

Sorting a list of files

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

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

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

Benefits

- **Powerful system**
- **Plenty of iterators in the Standard PHP Library**
- **Easy to combine with other patterns**

Downsides

- **Hard to learn and master**

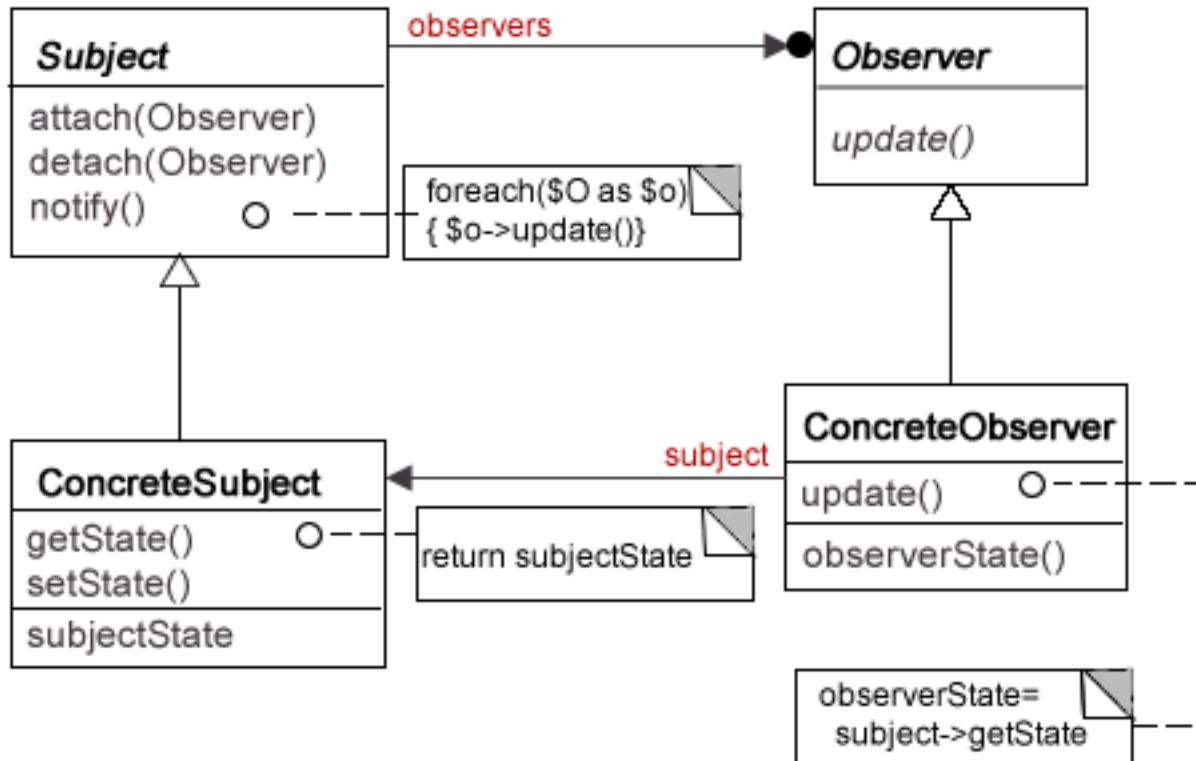
Observer



Observer

The Observer pattern allows an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.

– GoF



Main goals of Observer

Reducing communication coupling

Enforcing single responsibility

Leveraging extensibility

Leveraging unit testability

Identifying Tight Coupling

```
final class ErrorHandler
{
    // ...
    private $minErrorLevel;
    private $logger;
    private $deprecationErrorCollector;
    private $emailNotifier;

    public function __construct(
        LoggerInterface $logger,
        UserDeprecationErrorCollector $deprecationErrorCollector,
        EmailNotifier $emailNotifier,
        // ...
        int $minErrorLevel = E_ALL
    ) {
        $this->minErrorLevel = $minErrorLevel;
        $this->logger = $logger;
        $this->deprecationErrorCollector = $deprecationErrorCollector;
        $this->emailNotifier = $emailNotifier;
    }
}
```

```
final class ErrorHandler
{
    // ...
    public function handleError(
        int $errorLevel,
        string $errorMessage,
        string $errorFile,
        string $errorLine
    ): void {
        if ($errorLevel <= $this->minErrorLevel) {
            return;
        }

        if (E_USER_DEPRECATED === $errorLevel) {
            $this->deprecationErrorCollector->record($errorMessage, $errorFile, $errorFile);
        }

        $this->logger->log(LogLevel::ERROR, $errorMessage, [
            'file' => $errorFile,
            'line' => $errorLine,
            'level' => $errorLevel,
        ]);

        if (in_array($errorLevel, [E_USER_ERROR, E_COMPILE_ERROR, E_CORE_ERROR, E_PARSE], true)) {
            $this->emailNotifier->sendErrorAlert('admin@foobar.tld', new Error($errorLevel,
$errorMessage, $errorFile, $errorLine));
        }
    }

    // ...
}
```

```
$handler = new ErrorHandler(
    new Monolog\Logger('app', [
        new StreamHandler('/tmp/app.log'),
    ]),
    new UserDeprecationErrorCollector('/tmp/deprecated.log'),
    new EmailNotifier(new Mailer(...), /* ... */),
    // ...
);
$handler->register();

// Generate error
$fh = fopen('/foo/bar/invalid.txt', 'r+');
```

Problems?

Too many responsibilities

Tight coupling with concrete classes

Difficulty to unit test

Difficulty to support new error processors

Code is not open for future change

Decoupling with Observer Pattern

Solution?

- Decouple from concrete processors
- Depend on abstractions instead
- Isolate each error processing task

Abstract error processors

```
namespace ErrorHandler;

interface ErrorProcessor
{
    public function process(FlattenException $e): void;
}
```

```
final class ErrorHandler
{
    /** @var ErrorProcessor[] */
    private $processors;

    // ...

    private function __construct(int $minErrorLevel = E_ALL)
    {
        $this->minErrorLevel = $minErrorLevel;
        $this->processors = new ErrorProcessorList();
    }

    public function addProcessor(ErrorProcessor $processor, int $priority = 0): self
    {
        if (!$this->processors->contains($processor)) {
            $this->processors->insert($processor, $priority);
        }

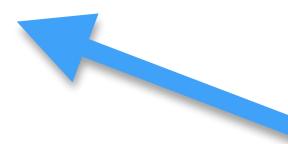
        return $this;
    }
}
```

```
final class ErrorHandler
{
    // ...
    public function handleException(\Throwable $e): void
    {
        $this->exceptions[] = $e;
        $flattenException = FlattenException::fromThrowable($e);

        foreach ($this->processors as $processor) {
            $processor->process($flattenException);
        }
    }

    public function handleError(
        int $errorLevel, string $errorMessage,
        string $errorFile, string $errorLine
    ): void {
        if ($errorLevel <= $this->minErrorLevel) {
            return;
        }

        $this->handleException(new \ErrorException(
            $errorMessage, 0, $errorLevel, $errorFile, $errorLine
        ));
    }
}
```



```
class FileErrorLogger implements ErrorProcessor
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function process(FlattenException $e): void
    {
        $this->logger->log(
            LogLevel::ERROR,
            $e->getMessage(),
            [ 'exception' => $e ]
        );
    }
}
```

```
class UserDeprecatedErrorCollector implements ErrorProcessor
{
    private $errors = [];

    public function process(FlattenException $e): void
    {
        $wrappedException = $e->getException();

        if (! $wrappedException instanceof \ErrorException) {
            return;
        }

        if (\E_USER_DEPRECATED !== $e->getSeverity()) {
            return;
        }

        $this->errors[ ] = [
            'message' => $e->getMessage(),
            'code' => $e->getCode(),
            'file' => $e->getFile(),
            'line' => $e->getLine(),
        ];
    }
}
```

```
class EmailNotifier implements ErrorProcessor
{
    private $mailer;

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

    public function process(FlattenException $e): void
    {
        $this->mailer->send(...);
    }
}
```

```
$logger = new Logger('app', [
    new StreamHandler('/tmp/error.log'),
]);
$deprecationsCollector = new UserDeprecatedErrorCollector();
$errorHandler = ErrorHandler::getInstance()
    ->addProcessor($deprecationsCollector)
    ->addProcessor(new FileErrorLogger($logger), 10)
    ->addProcessor(new EmailNotifier(new Mailer(...), 'admin@foo.tld'))
    ->register()
;
@trigger_error('Trigger A...', E_USER_DEPRECATED);
@trigger_error('Trigger B...', E_USER_DEPRECATED);
@trigger_error('Trigger C...', E_USER_NOTICE);
@trigger_error('Trigger D...', E_USER_WARNING);
@trigger_error('Trigger E...', E_USER_DEPRECATED);

print_r($deprecationsCollector->getErrors());
```

Push or Pull?

Push Approach

The **Push** approach is when the subject passes values other than itself to the notified observers, so that they can use them accordingly.

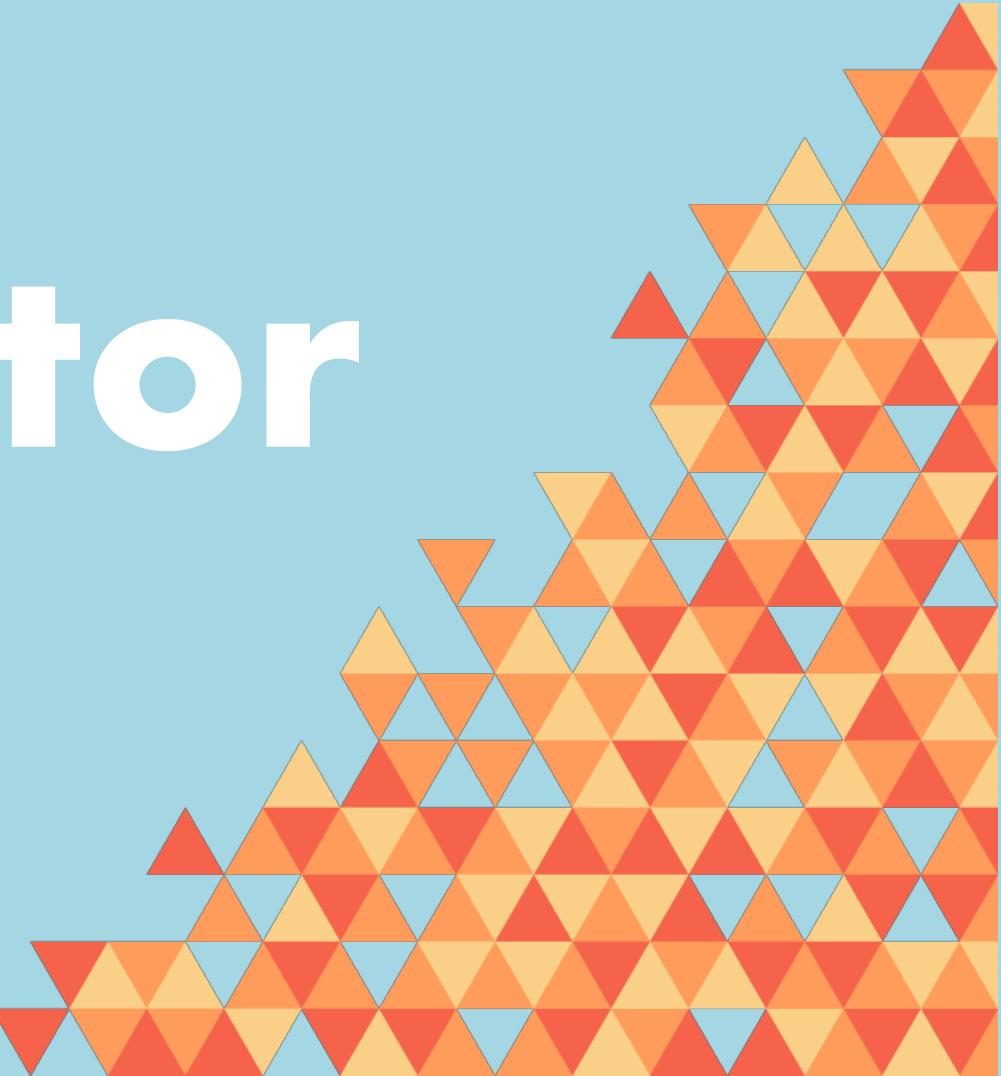
Pull Approach

The **Pull** approach is when the subject passes itself to the notified observers, so that they can pull contextual data out of it.

Dual Approach

The **push + pull** dual approach is when observers keep a reference to the subject in their constructor so that they can pull contextual data out of it. They also receive other data from the subject when they get notified.

Mediator



Mediator

The Mediator pattern reduces coupling between classes that communicate with each other. Instead of classes communicating directly, and thus requiring knowledge of their implementation, the classes send messages via a mediator object.

– GoF

Main goals of Mediator

Reducing coupling between objects

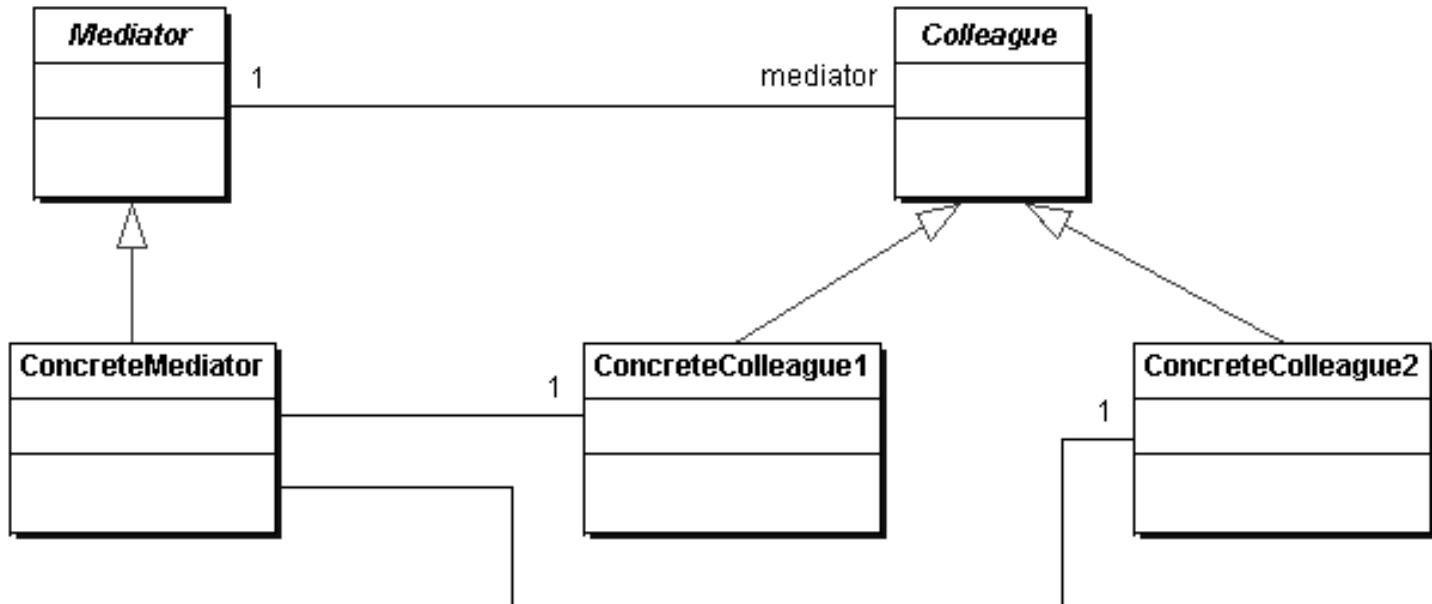
Easing communications between objects

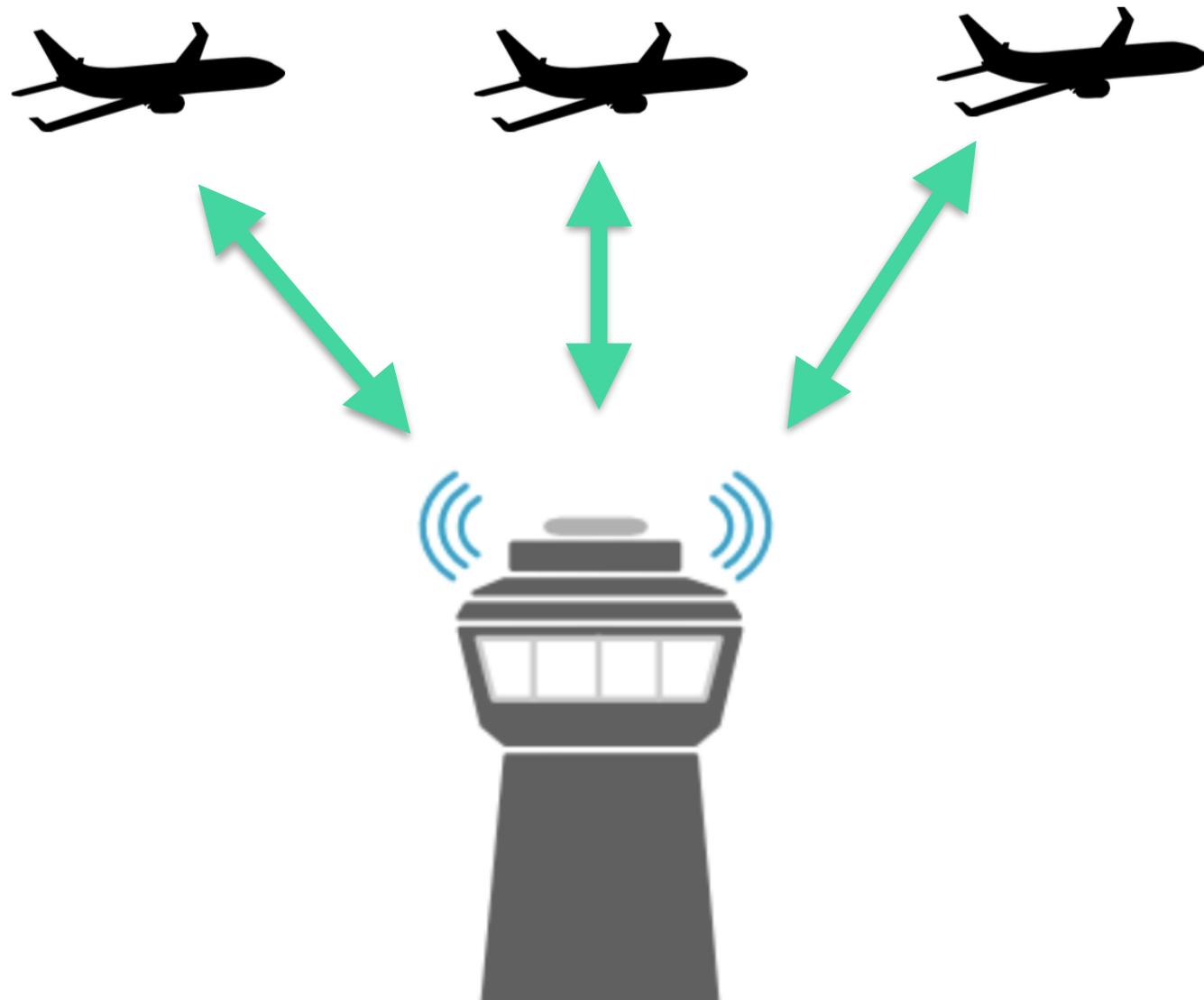
Leveraging objects' extensibility at run-time

Empowering the SRP and OCP principles

When / why using it?

- Decoupling large pieces of code
- Easing objects' unit testability
- Filtering users' input data in a form
- Hooking «plugins» on an object
- ...





Symfony Devs



hhamon

Jump to...

All Unreads

All Threads

Channels



afsy

blackfire

french

general

jobs

random

support

twig

university

Direct Messages



slackbot

hhamon (you)

dragoonis

leannapelham

nicolasgrekas

vudaltsov

xabbuh

xavierlacot



Message #general

#general

5,458 | 11 | General disc



Search

Today



javiereguiluz 5:23 PM

yes! because we love contributors ... and performance 😊

✓ 1 🎉 1



andrewmy 5:23 PM

So much for refreshing ;D



BackEndTea 5:23 PM

~~clicking the same link does seem to refresh it~~ Some pages aren't seems (edited)



magnusnordlander 5:23 PM

yeah, shift-reload does too

✉ 1



javiereguiluz 5:24 PM

yes ... results are cached for 600 seconds in some pages ... but we improve it in the future. Let's just enjoy the first iteration of this feature! (edited)



magnusnordlander 5:24 PM

yeah, it seems to vary 😊



dkarlovi 6:19 PM

Which branch counts?



ciaranmcnulty 6:23 PM

what do you do if they don't have a Connect profile?



Justin 6:35 PM

What if it's not a code contribution?



EventDispatcher

The Symfony EventDispatcher component is an implementation of the Mediator pattern that helps developers hook extensions to a piece of code without changing its class.

```
class EventDispatcher implements EventDispatcherInterface
{
    private $listeners = [ ];

    // ...

    public function addListener(
        string $eventName,
        callable $listener,
        int $priority = 0
    ) {
        $this->listeners[$eventName][$priority][] = $listener;
    }
}
```

```
class EventDispatcher implements EventDispatcherInterface
{
    // ...

    public function dispatch($eventName, Event $event = null)
    {
        $event = $event ?: new Event();
        if ($listeners = $this->getListeners($eventName)) {
            $this->doDispatch($listeners, $eventName, $event);
        }

        return $event;
    }

    protected function doDispatch($listeners, $eventName, Event $event)
    {
        foreach ($listeners as $listener) {
            if ($event->isPropagationStopped()) {
                break;
            }
            \call_user_func($listener, $event, $eventName, $this);
        }
    }
}
```

Registering colleagues

```
$listener1 = new CustomerListener($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->addListener('order.refunded', [ $listener3, 'onOrderRefunded' ]);
```

```
class OrderService
{
    private $dispatcher;
    private $repository;

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

    public function collectPayment(Payment $payment): void
    {
        $order = $this->repository->byReference($payment->getReference());
        $order->collectPayment($payment);
        $this->repository->save($order);

        if ($order->isFullyPaid()) {
            $this->dispatcher->dispatch('order.paid', new OrderEvent($order));
        }
        // ...
    }
}
```



```
class CustomerListener  
{  
    // ...  
  
    public function onOrderPaid(OrderEvent $event): void  
    {  
        $order = $event->getOrder();  
        $customer = $order->getCustomer();  
  
        $mail = $this->mailer->createMessage(...);  
        $this->mailer->send($mail);  
    }  
}
```



Benefits

- **Easy to implement (few classes & interfaces)**
- **Mediator manages all communications**
- **Colleagues are only aware of the Mediator**

Downsides

- **May be harder to debug**

Memento



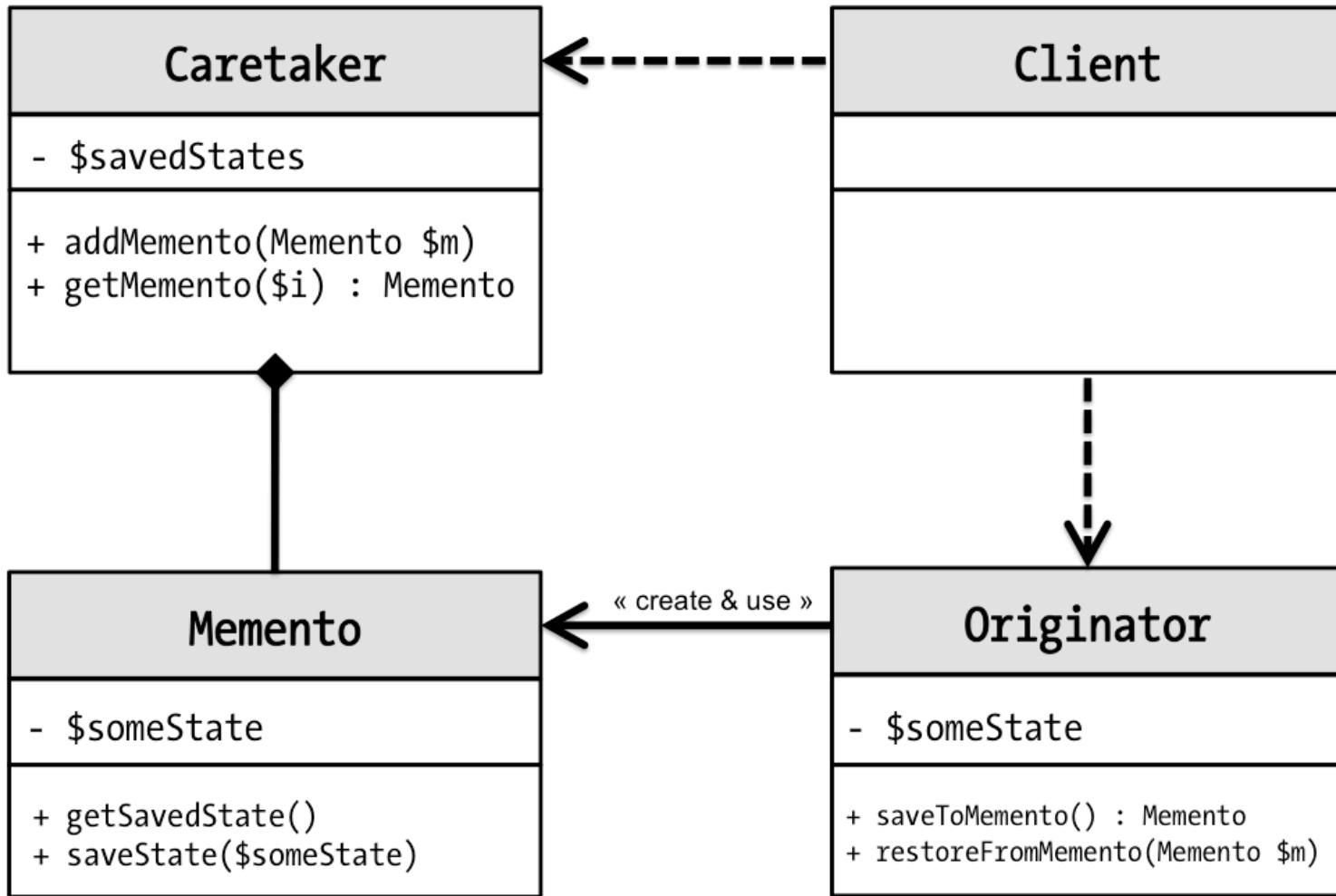
Memento

The Memento pattern captures the current state of an object and stores it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.

– GoF

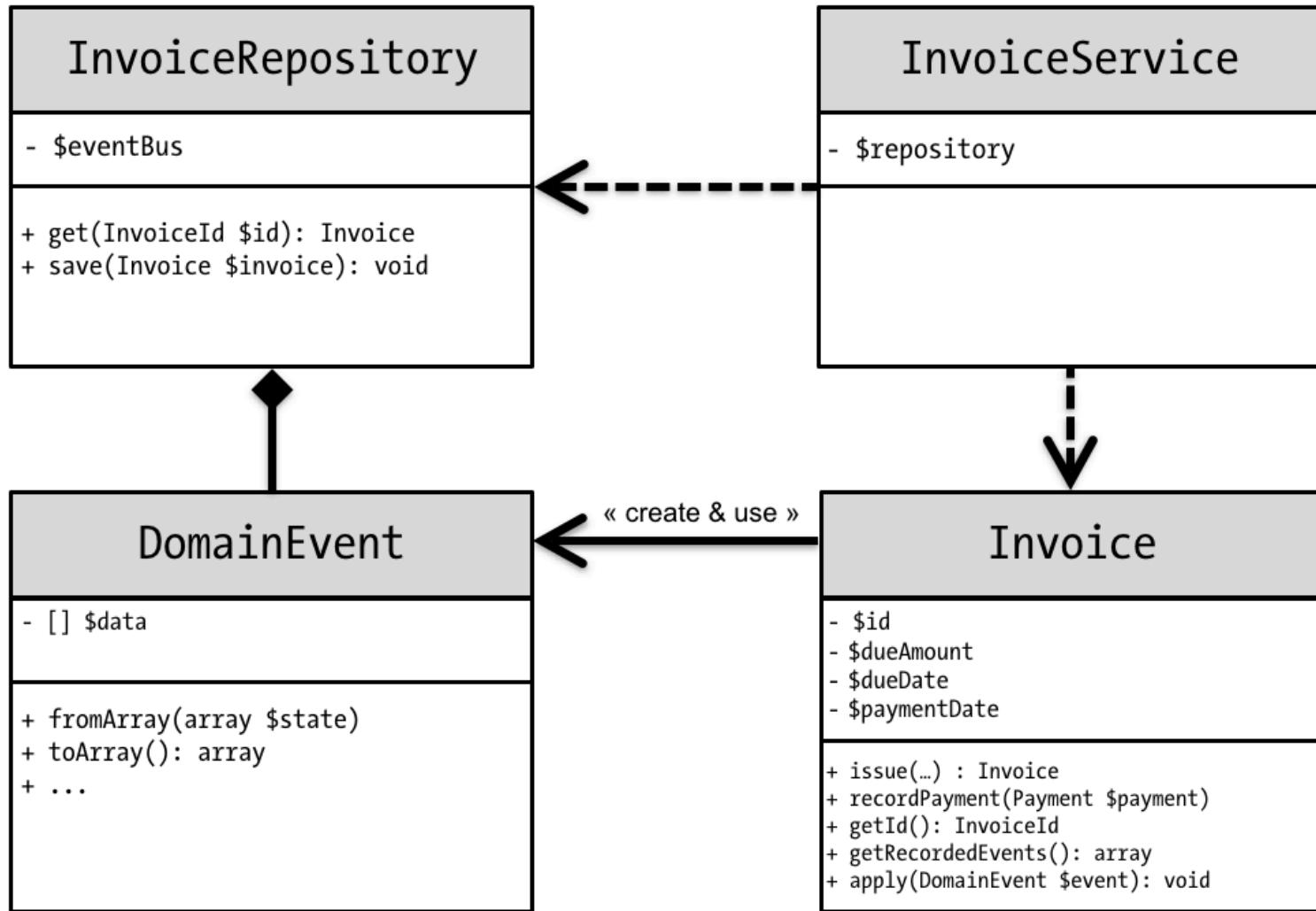
Main goals of Memento

- Extract and save an object's state outside of it
- Restore the object's state from its saved state
- Restore without breaking encapsulation

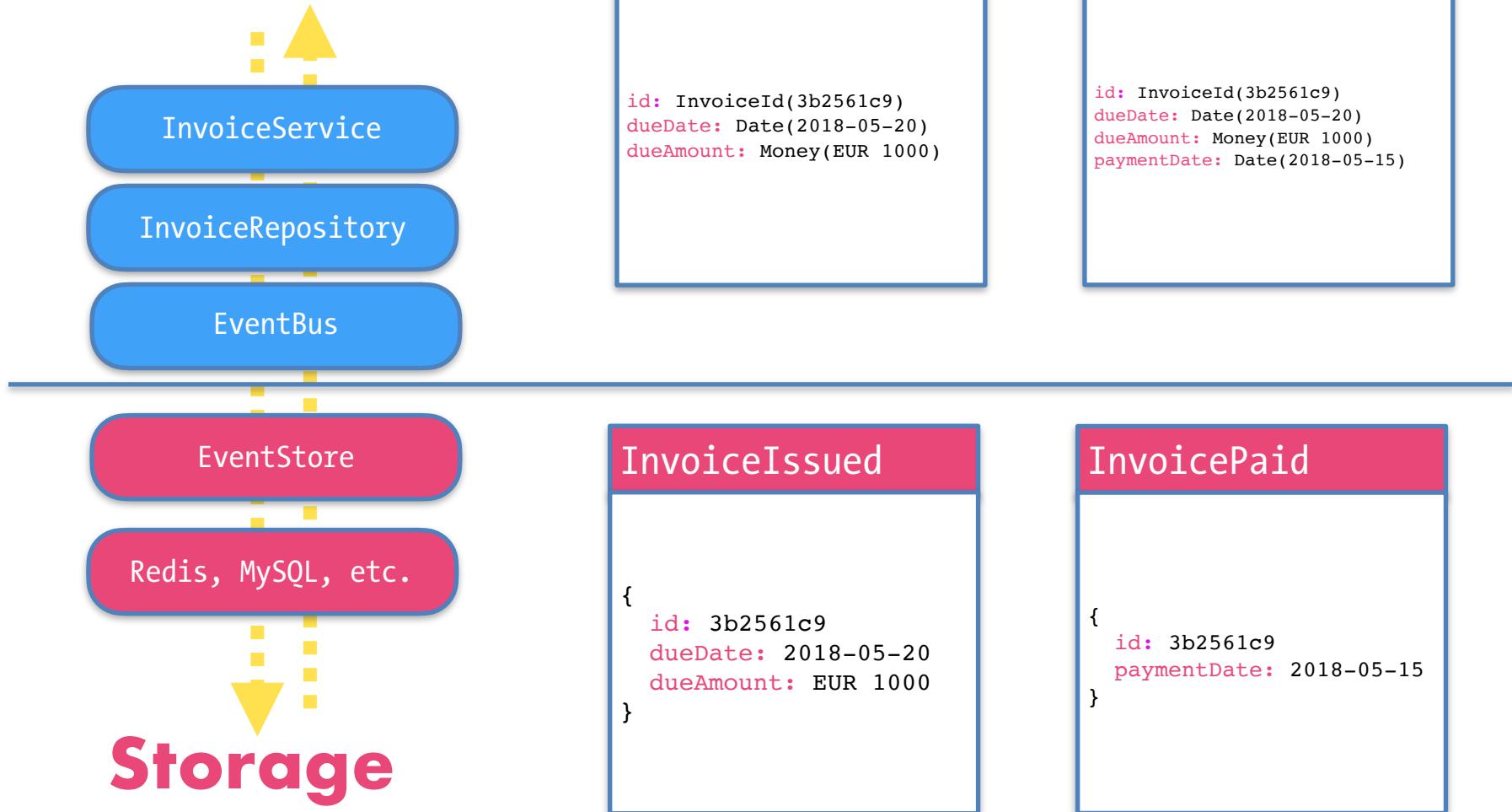


Event Sourcing

Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.



Domain Model



The Domain Entity

```
class Invoice
{
    private $recordedEvents = [ ];
    private $id;

    private function __construct(InvoiceId $id)
    {
        $this->id = $id;
    }

    private function recordThat(DomainEvent $event): void
    {
        $this->recordedEvents[ ] = $event;
    }

    public function getRecordedEvents(): array
    {
        return $this->recordedEvents
    }

    public function getId(): InvoiceId
    {
        return $this->id;
    }
}
```

```
class Invoice
{
    // ...
    private $dueAmount;
    private $dueDate;
    private $paymentDate;

    public static function issue(DueDate $dueDate, Money $dueAmount): self
    {
        $invoice = new static(InvoiceId::generate());
        $invoice->recordThat(new InvoiceIssued($invoice->id, $dueDate, $dueAmount));

        return $invoice;
    }

    public function recordPayment(Payment $payment): void
    {
        Assertion::null($this->paymentDate);
        Assertion::equal($this->dueAmount, $payment->getAmount());

        $this->recordThat(new InvoicePaid($this->id, $payment->getDate()));
    }
}
```

```
class Invoice
{
    // ...
    public static function fromEventStream(Invoice $id, EventStream $stream): self
    {
        $invoice = new static($id);
        foreach ($stream as $event) {
            $invoice->apply($event);
        }

        return $invoice;
    }

    public function apply(DomainEvent $event): void
    {
        switch (true) {
            case $event instanceof InvoiceIssued:
                $this->id = $event->getInvoiceId();
                $this->dueAmount = $event->getDueAmount();
                $this->dueDate = $event->getDueDate();
                break;
            case $event instanceof InvoicePaid:
                $this->paymentDate = $event->getPaymentDate();
                break;
        }
    }
}
```

The Entity Repository

```
class InvoiceRepository
{
    private $bus;
    private $store;

    public function __construct(EventBus $bus, EventStore $store)
    {
        $this->bus = bus;
        $this->store = $store;
    }

    public function save(Invoice $invoice): void
    {
        if (count($events = $invoice->getRecordedEvents())) {
            $this->bus->publishAll($events);
        }
    }

    public function get(InvoiceId $invoiceId): Invoice
    {
        return Invoice::fromEventStream(
            $invoiceId,
            $this->store->getStream($invoiceId)
        );
    }
}
```

The Application Service

```
class InvoiceService
{
    private $repository;

    public function __construct(InvoiceRepository $repository)
    {
        $this->repository = $repository;
    }

    public function issueInvoice(string $dueDate, string $amount, string $currency): InvoiceId
    {
        $invoice = Invoice::issue(
            new DueDate($dueDate),
            new Money($amount, new Currency($currency))
        );

        $this->repository->save($invoice);

        return $invoice->getId();
    }

    public function recordPayment(InvoiceId $invoiceId, Payment $payment): void
    {
        $invoice = $this->repository->get($invoiceId);
        $invoice->recordPayment($payment);
        $this->repository->save($invoice);
    }
}
```

State



State

The State pattern alters the behaviour of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.

– GoF

Implementation Goals

- **Finite State Machines / Workflows**
- Isolate an object state into several objects
- Prevent the code from having a lot of conditional statements to check each state combination at runtime.

The Door Example

- **Open** state
- **Closed** state
- **Locked** state
- Transition from one state to another must leave the object in a coherent state.
- Invalid transition operation must be prevented / forbidden.

The State Transition Matrix

From / to	Open	Closed	Locked
Open	Invalid	<code>close()</code>	Invalid
Closed	<code>open()</code>	Invalid	<code>lock()</code>
Locked	Invalid	<code>unlock()</code>	Invalid

```
$door = new Door('open');  
echo "Door is open\n";
```

```
$door->close();  
echo "Door is closed\n";
```

```
$door->lock();  
echo "Door is locked\n";
```

```
$door->unlock();  
echo "Door is closed\n";
```

```
$door->open();  
echo "Door is open\n";
```

```
class Door
{
    private $state = 'open';

    public function close(): void
    {
        if ('open' !== $this->state) {
            throw InvalidDoorStateOperation::doorCannotBeClosed($this->state);
        }

        $this->state = 'closed';
    }

    public function open(): void
    {
        if ('closed' !== $this->state) {
            throw InvalidDoorStateOperation::doorCannotBeOpen($this->state);
        }

        $this->state = 'open';
    }

    // ...
}
```



Extract States in Separate Classes

```
interface DoorState
{
    public function open(): DoorState;
    public function close(): DoorState;
    public function lock(): DoorState;
    public function unlock(): DoorState;
}
```

```
abstract class AbstractDoorState implements DoorState
{
    public function close(): DoorState
    {
        throw InvalidDoorStateOperation::doorCannotBeClosed($this->getCurrentState());
    }

    public function open(): DoorState
    {
        throw InvalidDoorStateOperation::doorCannotBeOpen($this->getCurrentState());
    }

    public function lock(): DoorState
    {
        throw InvalidDoorStateOperation::doorCannotBeLocked($this->getCurrentState());
    }

    public function unlock(): DoorState
    {
        throw InvalidDoorStateOperation::doorCannotBeUnlocked($this->getCurrentState());
    }

    private function getCurrentState(): string
    {
        $class = get_class($this);

        return strtolower(substr($class, 0, strlen($class) - 9));
    }
}
```

```
class OpenDoorState extends AbstractDoorState
{
    public function close(): DoorState
    {
        return new ClosedDoorState();
    }
}
```

```
class ClosedDoorState extends AbstractDoorState
{
    public function open(): DoorState
    {
        return new OpenDoorState();
    }

    public function lock(): DoorState
    {
        return new LockedDoorState();
    }
}
```

```
class LockedDoorState extends AbstractDoorState
{
    public function unlock(): DoorState
    {
        return new ClosedDoorState();
    }
}
```

```
class Door
{
    private $state;

    public function __construct(DoorState $initialState)
    {
        $this->state = $initialState;
    }

    public function close(): void
    {
        $this->state = $this->state->close();
    }

    public function open(): void
    {
        $this->state = $this->state->open();
    }
}
```

```
class Door
{
    // ...
    public function lock(): void
    {
        $this->state = $this->state->lock();
    }

    public function unlock(): void
    {
        $this->state = $this->state->unlock();
    }
}
```

```
class Door
{
    // ...
    public function isOpen(): bool
    {
        return $this->state instanceof OpenDoorState;
    }

    public function isClosed(): bool
    {
        return $this->state instanceof ClosedDoorState;
    }

    public function isLocked(): bool
    {
        return $this->state instanceof LockedDoorState;
    }
}
```

```
$door = new Door(new OpenDoorState());  
echo "Door is open\n";
```

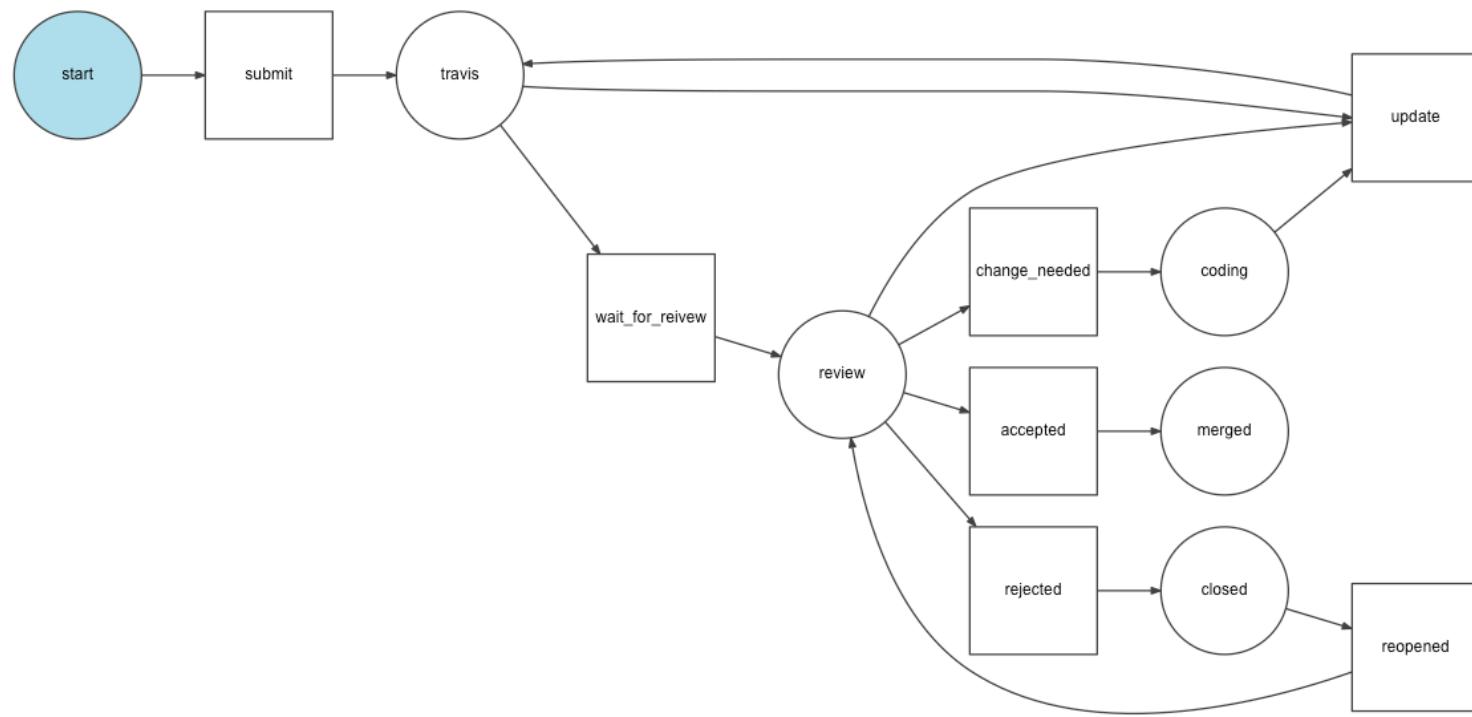
```
$door->close();  
echo "Door is closed\n";
```

```
$door->lock();  
echo "Door is locked\n";
```

```
$door->unlock();  
echo "Door is closed\n";
```

```
$door->open();  
echo "Door is open\n";
```

State Machine with Symfony



```
framework:
workflows:
  pull_request:
    type: 'state_machine'
    supports:
      - App\Entity\PullRequest
  initial_place: start
  places: [start, coding, travis, review, merged, closed]
  transitions:
    submit:
      from: start
      to: travis
    update:
      from: [coding, travis, review]
      to: travis
    wait_for_review:
      from: travis
      to: review
    request_change:
      from: review
      to: coding
    accept:
      from: review
      to: merged
    reject:
      from: review
      to: closed
    reopen:
      from: closed
      to: review
```

Strategy



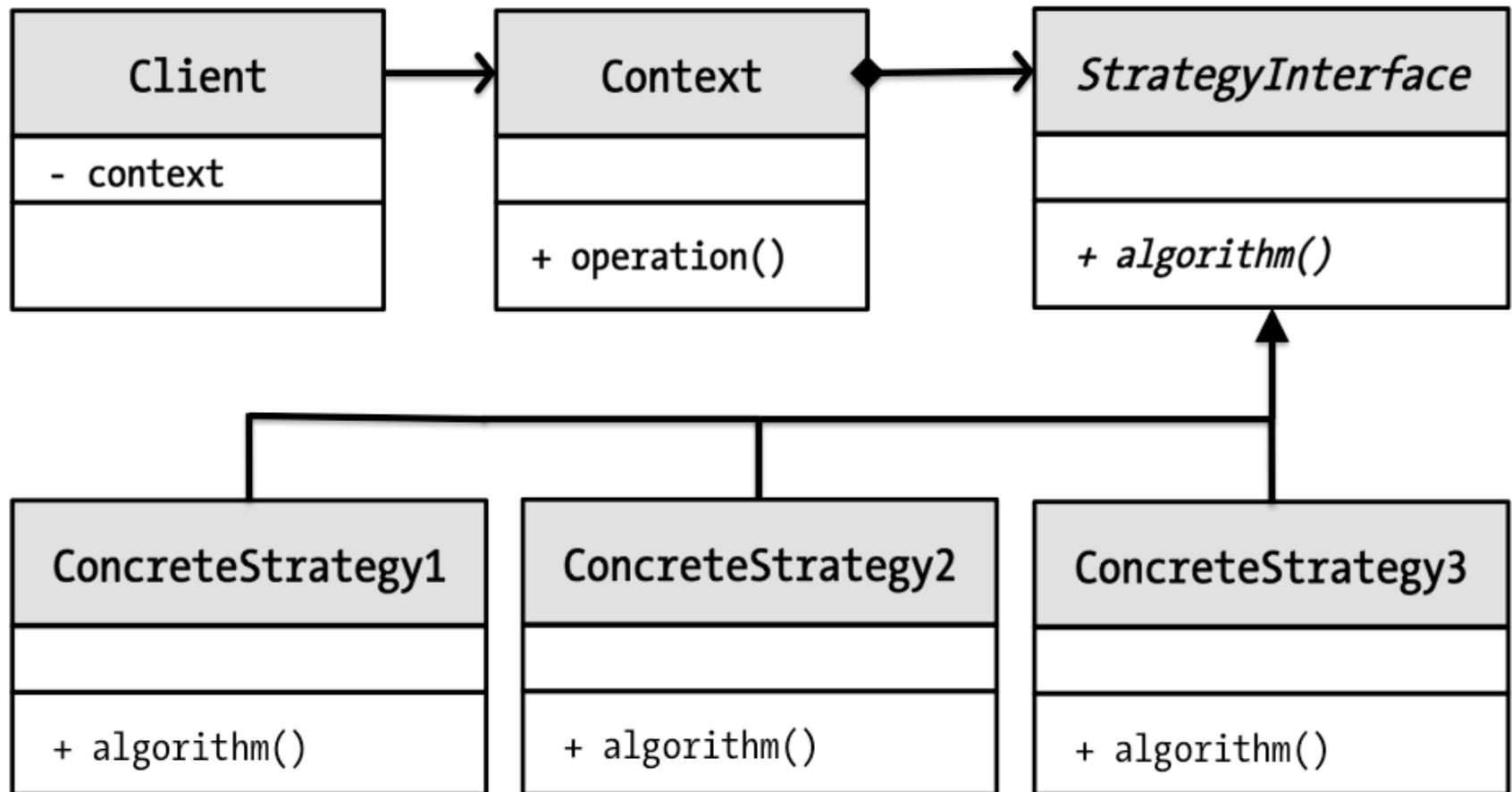
Strategy

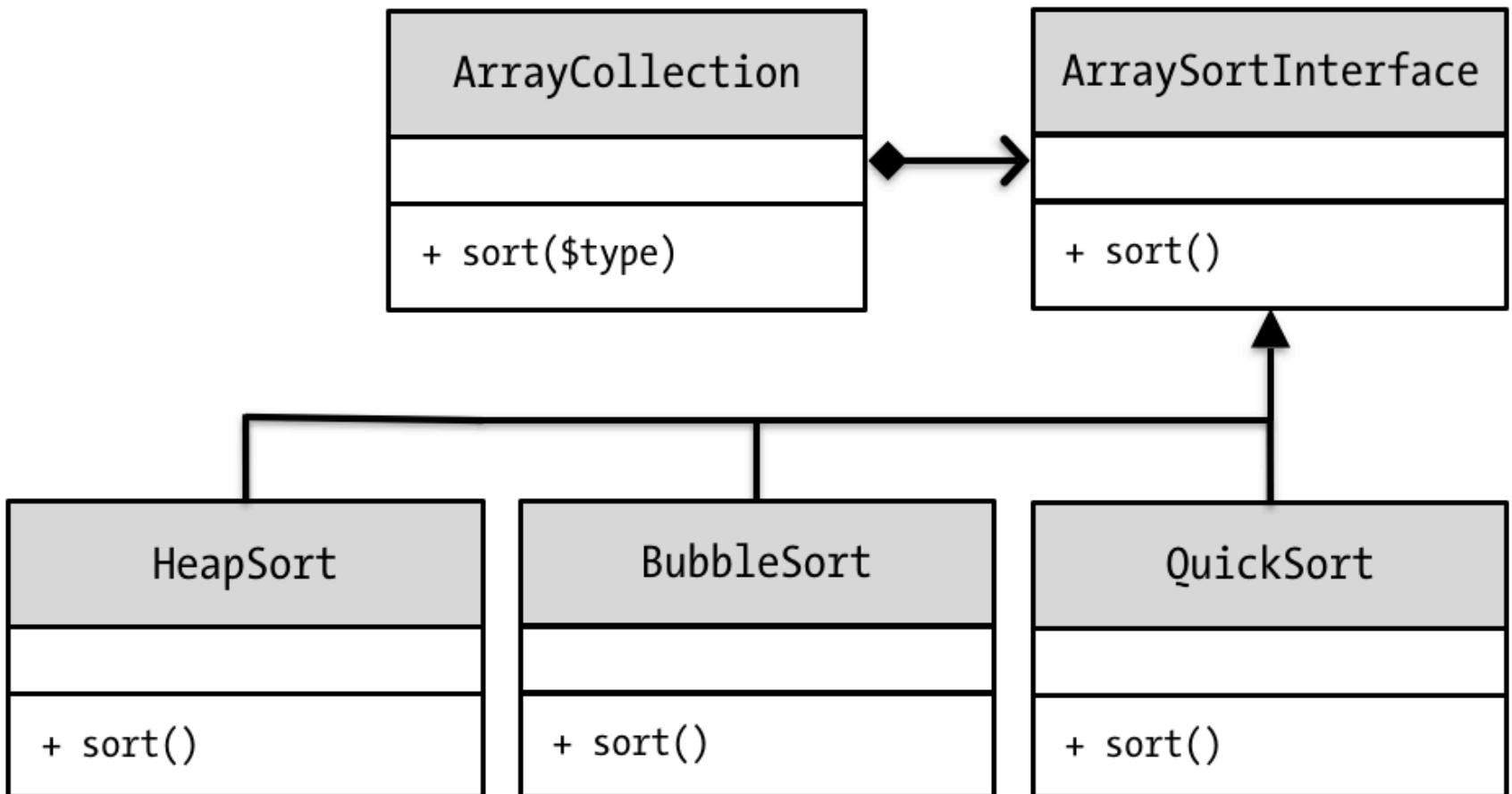
The Strategy pattern creates an interchangeable family of algorithms from which the required process is chosen at run-time.

– GoF

Main goals of Strategy

- Encapsulating algorithms of the same nature in separate objects
- Exposing a unified interface for these concrete algorithm
- Choosing the right strategy to rely on at run-time
- Preventing code from having large conditional blocks statements (if, elseif, else, switch, case)



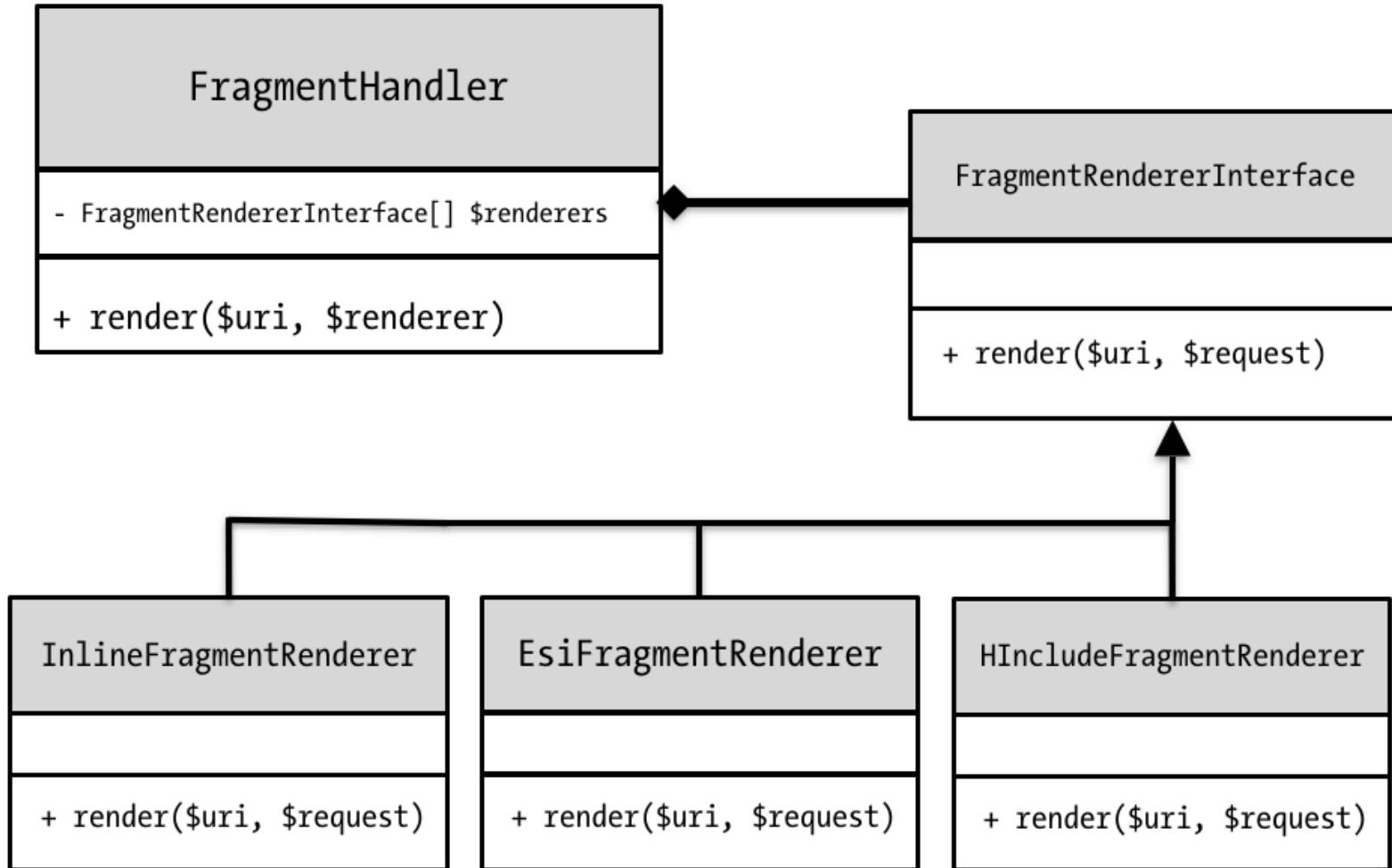


HttpKernel

The HttpKernel component comes with a fragment rendering system allowing the application to choose the strategy to use to render a dynamic fragment.

Supported rendering strategies

- **Inline**
- **Esi (Edge Side Include)**
- **Ssi (Server Side Include)**
- **HInclude (HTML Include)**



Defining the Fragment Renderer Interface

```
interface FragmentRendererInterface
{
    /**
     * Renders a URI and returns the Response content.
     *
     * @param string|ControllerReference $uri
     * @param Request                  $request A Request instance
     * @param array                     $options An array of options
     *
     * @return Response A Response instance
     */
    public function render($uri, Request $request, array $options = []);

    /**
     * @return string The strategy name
     */
    public function getName();
}
```

Defining the
concrete renderer
strategies

```
class InlineFragmentRenderer implements FragmentRendererInterface
{
    // ...
    private $kernel;

    public function render($uri, Request $request, array $options = array())
    {
        // ...
        $subRequest = $this->createSubRequest($uri, $request);
        // ...
        $level = ob_get_level();
        try {
            return $this
                ->kernel
                ->handle($subRequest, HttpKernelInterface::SUB_REQUEST, false);
        } catch (\Exception $e) {
            // ...
            return new Response();
        }
    }

    public function getName()
    {
        return 'inline';
    }
}
```

```
class HIncludeFragmentRenderer implements FragmentRendererInterface
{
    // ...

    public function render($uri, Request $request, array $options = [])
    {
        // ...

        return new Response(sprintf(
            '<hx:include src="%s"%s>%s</hx:include>',
            $uri,
            $renderedAttributes,
            $this->templating->render($options['default']));
    });

    public function getName()
    {
        return 'hinclude';
    }
}
```

```
class EsiFragmentRenderer implements FragmentRendererInterface
{
    // ...
    private $surrogate;

    public function render($uri, Request $request, array $options = [])
    {
        // ...

        $salt = isset($options['alt']) ? $options['alt'] : null;
        if ($salt instanceof ControllerReference) {
            $salt = $this->generateSignedFragmentUri($salt, $request);
        }

        return new Response($this->surrogate->renderIncludeTag(
            $uri,
            $salt,
            isset($options['ignore_errors']) ? $options['ignore_errors'] : false,
            isset($options['comment']) ? $options['comment'] : ''
        ));
    }

    public function getName()
    {
        return 'esi';
    }
}
```

Implementing the Context Client Code

```
class FragmentHandler
{
    private $debug;
    private $renderers = [ ];
    private $requestStack;

    public function __construct(
        RequestStack $requestStack,
        array $renderers,
        bool $debug = false
    ) {
        $this->requestStack = $requestStack;
        foreach ($renderers as $renderer) {
            $this->addRenderer($renderer);
        }
        $this->debug = $debug;
    }

    public function addRenderer(FragmentRendererInterface $renderer)
    {
        $this->renderers[$renderer->getName()] = $renderer;
    }
}
```

```
class FragmentHandler
{
    // ...

    public function render($uri, $renderer = 'inline', array $options = [])
    {
        if (!isset($options['ignore_errors'])) {
            $options['ignore_errors'] = !$this->debug;
        }

        if (!isset($this->renderers[$renderer])) {
            throw new \InvalidArgumentException(...);
        }

        if (!$request = $this->requestStack->getCurrentRequest()) {
            throw new \LogicException('...');
        }

        return $this->deliver(
            $this->renderers[$renderer]->render($uri, $request, $options)
        );
    }
}
```

Initializing the Fragment Handler

```
$handler = new FragmentHandler($kernel);
$handler->addRenderer(new InlineFragmentRenderer(...));
$handler->addRenderer(new EsiFragmentRenderer(...));
$handler->addRenderer(new SsiFragmentRenderer(...));
$handler->addRenderer(new HIncludeFragmentRenderer(...));

$handler->render('/yolo', 'inline', ['ignore_errors' => false]);
$handler->render('/yolo', 'hinclude', ['ignore_errors' => false]);
$handler->render('/yolo', 'esi', ['ignore_errors' => false]);
$handler->render('/yolo', 'ssi', ['ignore_errors' => false]);
```

Calling the fragment handler in Twig

```
 {{ render(uri('yolo')) , {ignore_errors: false}) }}  
 {{ render_hinclude(uri('yolo')) , {ignore_errors: false}) }}  
 {{ render_esi(uri('yolo')) , {ignore_errors: false}) }}  
 {{ render_ssi(uri('yolo')) , {ignore_errors: false}) }}
```

Benefits

- **Easy to implement**
- **Make the code's behavior vary at run-time**
- **Great to combine with other patterns like Composite**
- **Each algorithm lives in its own class**
- **Fullfill SRP, OCP & DIP principles of SOLID**

Template Method



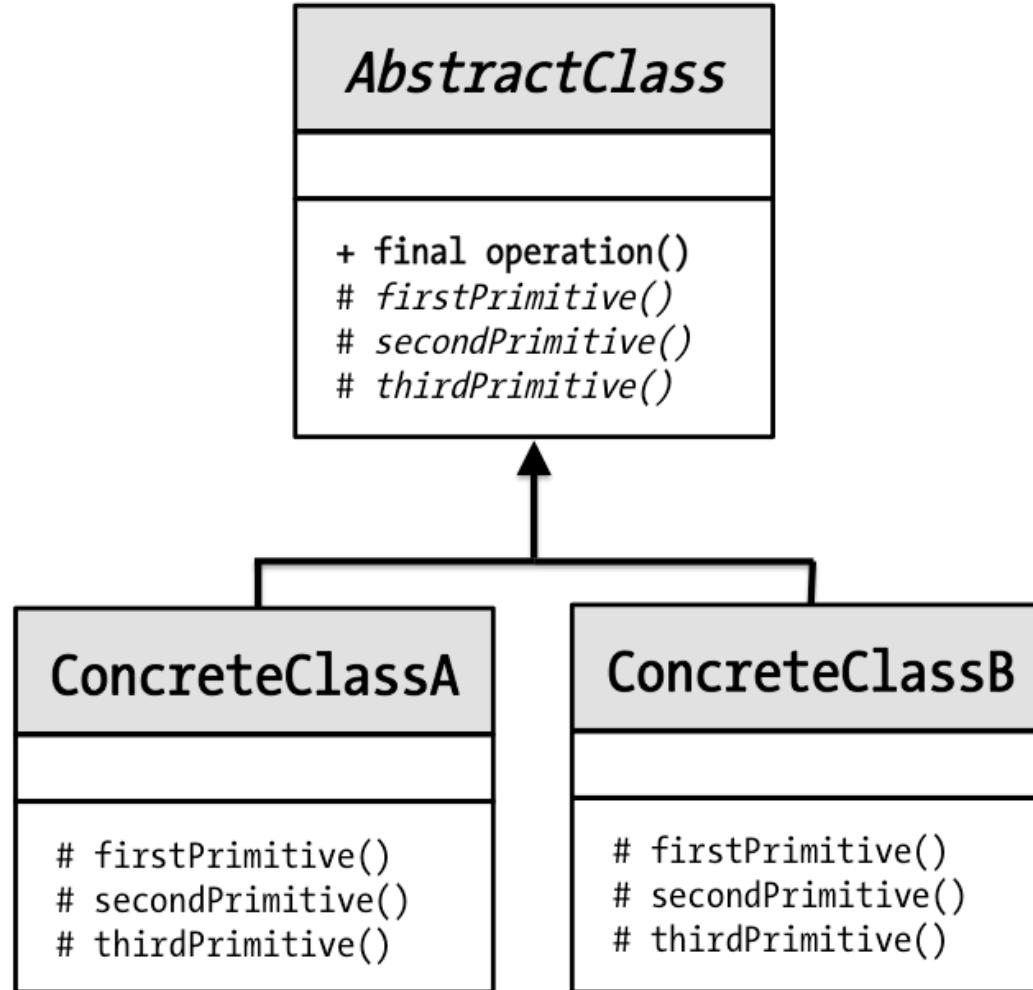
Template Method

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

– GoF

Problems to solve

- Encapsulating an algorithm and preventing it from being overridden by subclasses.
- Allowing subclasses to override some of the steps of this algorithm.
- Leverage the «**Hollywood Principle**»



```
abstract class AbstractClass
{
    final public function operation()
    {
        $this->firstPrimitive();
        $this->secondPrimitive();

        return $this->thirdPrimitive();
    }

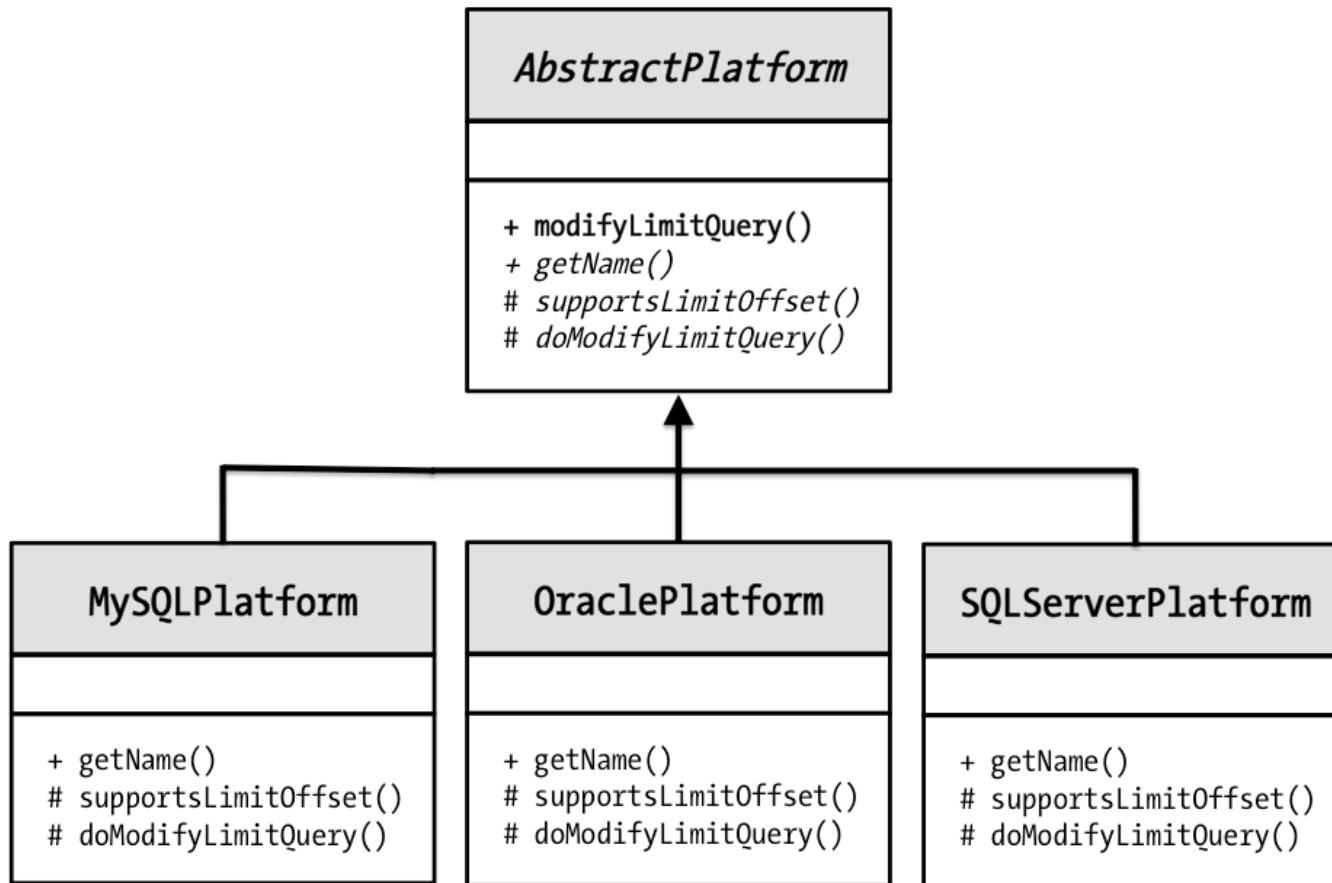
    abstract protected function firstPrimitive();
    abstract protected function secondPrimitive();
    abstract protected function thirdPrimitive();
}
```

A stylized megaphone is positioned on the left side of the image. It has a dark, textured handle and a light-colored, slightly worn circular mouthpiece. Behind the megaphone is a large, jagged red starburst or sunburst pattern.

**Don't call us!
We'll call you!**

Doctrine DBAL

The Doctrine DBAL library provides the algorithm to paginate a SQL query. The implementation of the steps of this algorithm is delegated to the concrete vendor platforms.



Defining the
abstract platform

```
abstract class AbstractPlatform implements PlatformInterface
{
    /**
     * Appends the LIMIT clause to the SQL query.
     *
     * @param string $query  The SQL query to modify
     * @param int $limit   The max number of records to fetch
     * @param int $offset  The offset from where to fetch records
     *
     * @return string The modified SQL query
     */
    final public function modifyLimitQuery($query, $limit, $offset = null)
    {
        // ...
    }

    abstract protected function doModifyLimitQuery($query, $limit, $offset);

    protected function supportsLimitOffset()
    {
        return true;
    }
}
```

```
abstract class AbstractPlatform implements PlatformInterface
{
    final public function modifyLimitQuery($query, $limit, $offset = null)
    {
        if ($limit !== null) {
            $limit = (int) $limit;
        }

        if ($offset !== null) {
            $offset = (int) $offset;

            if ($offset < 0) {
                throw new PlatformException(sprintf(
                    'LIMIT offset must be greater or equal than 0, %u given.',
                    $offset
                ));
            }
        }

        if ($offset > 0 && ! $this->supportsLimitOffset()) {
            throw new PlatformException(sprintf(
                'Platform %s does not support offset values in limit queries.',
                $this->getName()
            ));
        }
    }

    return $this->doModifyLimitQuery($query, $limit, $offset);
}
```

Paginating a SQL Query on MySQL

```
class MySQLPlatform extends AbstractPlatform
{
    protected function doModifyLimitQuery($query, $limit, $offset)
    {
        if (null !== $limit) {
            $query .= ' LIMIT ' . $limit;
            if (null !== $offset) {
                $query .= ' OFFSET ' . $offset;
            }
        } elseif (null !== $offset) {
            $query .= ' LIMIT 18446744073709551615 OFFSET ' . $offset;
        }

        return $query;
    }

    public function getName()
    {
        return 'mysql';
    }
}
```

```
$query = 'SELECT id, username FROM user';
$platform = new MySQLPlatform();
$platform->modifyLimitQuery($query, null);
$platform->modifyLimitQuery($query, 10);
$platform->modifyLimitQuery($query, 10, 50);
$platform->modifyLimitQuery($query, null, 50);
```

```
SELECT id, username FROM user
SELECT id, username FROM user LIMIT 10
SELECT id, username FROM user LIMIT 10 OFFSET 50
SELECT id, username FROM user LIMIT 18446744073709551615 OFFSET 50
```

Paginating a SQL Query on Oracle

```
class OraclePlatform extends AbstractPlatform
{
    protected function doModifyLimitQuery($query, $limit, $offset = null)
    {
        if (!preg_match('/^\s*SELECT/i', $query)) {
            return $query;
        }

        if (!preg_match('/\s*FROM\s/i', $query)) {
            $query .= ' FROM dual';
        }

        $limit = (int) $limit; $offset = (int) $offset;
        if ($limit > 0) {
            $max = $offset + $limit;
            if ($offset > 0) {
                $min = $offset + 1;
                $query = sprintf(
                    'SELECT * FROM (SELECT a.*,
ROWNUM AS dbal_rownum' .
                    ' FROM (%s) a WHERE ROWNUM <= %u) WHERE dbal_rownum >= %u',
                    $query,
                    $max,
                    $min
                );
            } else {
                $query = sprintf('SELECT a.* FROM (%s) a WHERE ROWNUM <= %u', $query, $max);
            }
        }

        return $query;
    }
}
```

```
$query = 'SELECT id, username FROM user';
$platform = new OraclePlatform();
$platform->modifyLimitQuery($query, null);
$platform->modifyLimitQuery($query, 10);
$platform->modifyLimitQuery($query, 10, 50);
```

```
SELECT id, username FROM user
```

```
SELECT a.* FROM (SELECT id, username FROM user) a WHERE ROWNUM <= 10
```

```
SELECT * FROM (SELECT a.* , ROWNUM AS dbal_rownum FROM (SELECT id,
username FROM user) a WHERE ROWNUM <= 60) WHERE dbal_rownum >= 51)
```

Security

The Symfony Security component provides an abstract class that defines the algorithm for authenticating a user. Although the algorithm is final, its steps can be however overriden by subclasses.

AbstractAuthenticationListener

```
+ handle(GetResponseEvent $event) [final]  
# attemptAuthentication(Request $request)
```

SimpleFormAuthenticationListener

```
# attemptAuthentication(Request $request)
```

UsernamePasswordFormAuthenticationListener

```
# attemptAuthentication(Request $request)
```

Defining the abstract authentication listener

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

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

Defining the concrete
authentication
listeners

```
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);  
    }  
}
```

```
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);  
    }  
}
```

Benefits

- **Easy to implement**
- **Ensure an algorithm is fully executed**
- **Help eliminate duplicated code**

Downsides

- **May break the Liskov Substitution principle**
- **May become harder to maintain with many steps**
- **The final skeleton can be a limit to extension**

Visitor



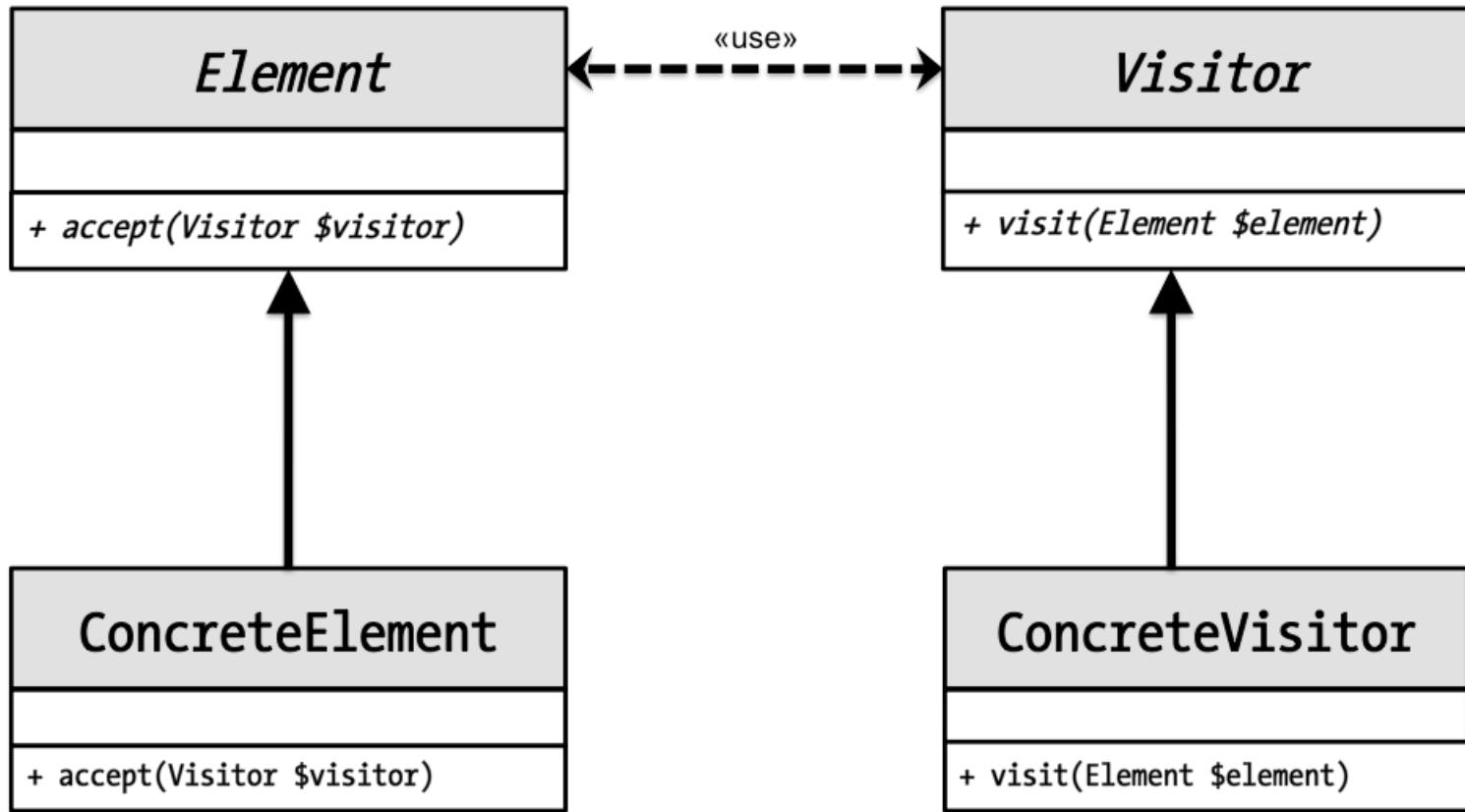
Visitor

The Visitor pattern separates a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

– GoF

Main goals of Visitor

- Separate object's state from its operations
- Ensure the Open/Close Principle
- Leverage Single Responsibility Principle



Doctrine DBAL

The Doctrine DBAL library uses the Visitor pattern to visit a Schema object graph in order to validate it or generate it.

Defining the Visitor interface

```
namespace Doctrine\DBAL\Schema\Visitor;

use Doctrine\DBAL\Schema\Table;
use Doctrine\DBAL\Schema\Schema;
use Doctrine\DBAL\Schema\Column;
use Doctrine\DBAL\Schema\ForeignKeyConstraint;
use Doctrine\DBAL\Schema\Sequence;
use Doctrine\DBAL\Schema\Index;

interface Visitor
{
    function acceptSchema(Schema $schema);
    function acceptTable(Table $table);
    function acceptColumn(Table $table, Column $column);
    function acceptForeignKey(Table $table, ForeignKeyConstraint $fkc);
    function acceptIndex(Table $table, Index $index);
    function acceptSequence(Sequence $sequence);
}
```

Defining the Visitable Data Structures

```
class Schema extends AbstractAsset
{
    // ...
    public function visit(Visitor $visitor)
    {
        $visitor->acceptSchema($this);

        if ($visitor instanceof NamespaceVisitor) {
            foreach ($this->namespaces as $namespace) {
                $visitor->acceptNamespace($namespace);
            }
        }

        foreach ($this->_tables as $table) {
            $table->visit($visitor);
        }

        foreach ($this->_sequences as $sequence) {
            $sequence->visit($visitor);
        }
    }
}
```

```
// ...
class Table extends AbstractAsset
{
    // ...
    public function visit(Visitor $visitor)
    {
        $visitor->acceptTable($this);

        foreach ($this->getColumns() as $column) {
            $visitor->acceptColumn($this, $column);
        }

        foreach ($this->getIndexes() as $index) {
            $visitor->acceptIndex($this, $index);
        }

        foreach ($this->getForeignKeys() as $constraint) {
            $visitor->acceptForeignKey($this, $constraint);
        }
    }
}
```

Defining the Concrete Visitors

```
class DropSchemaSqlCollector extends AbstractVisitor
{
    private $constraints;
    private $sequences;
    private $tables;
    private $tables;

    public function __construct(AbstractPlatform $platform)
    {
        $this->platform = $platform;
        $this->constraints = new \SplObjectStorage();
        $this->sequences = new \SplObjectStorage();
        $this->tables = new \SplObjectStorage();
    }

    public function getQueries()
    {
        $sql = [];
        foreach ($this->constraints as $fkConstraint) {
            $localTable = $this->constraints[$fkConstraint];
            $sql[] = $this->platform->getDropForeignKeysSQL($fkConstraint, $localTable);
        }

        foreach ($this->sequences as $sequence) {
            $sql[] = $this->platform->getDropSequenceSQL($sequence);
        }

        foreach ($this->tables as $table) {
            $sql[] = $this->platform->getDropTableSQL($table);
        }
        return $sql;
    }
}
```

```
class DropSchemaSqlCollector extends AbstractVisitor
{
    // ...
    public function acceptTable(Table $table)
    {
        $this->tables->attach($table);
    }

    public function acceptForeignKey(Table $table, ForeignKeyConstraint $fk)
    {
        if (strlen($fk->getName()) == 0) {
            throw SchemaException::namedForeignKeyRequired($table, $fk);
        }

        $this->constraints->attach($fk, $table);
    }

    public function acceptSequence(Sequence $sequence)
    {
        $this->sequences->attach($sequence);
    }
}
```

```
class SingleDatabaseSynchronizer extends  
AbstractSchemaSynchronizer  
{  
    // ...  
  
    public function getDropAllSchema()  
    {  
        $sm = $this->conn->getSchemaManager();  
        $visitor = new DropSchemaSqlCollector($this->platform);  
  
        /* @var $schema \Doctrine\DBAL\Schema\Schema */  
        $schema = $sm->createSchema();  
        $schema->visit($visitor);  
  
        return $visitor->getQueries();  
    }  
}
```

Benefits

- **Easy to implement**
- **Guarantee SRP and OPC of SOLID**
- **Easy to add new visitors without changing visitee**
- **Visitors can accumulate state**

Downsides

- **Visitors are usually designed stateful**
- **Visitee must expose its state with public methods**
- **Double dispatch / polymorphism not supported in PHP**

Thank you for attending!

