Object design

Anna Adamchuk

Let’s apply some design patterns

- What pattern to apply?
- How to implement it?
- Oh, no! Book’s version is a bit difficult, I’ll implement simplified version!

How about other principles?

- SOLID
- KISS
- CQRS
- Architectural
- Testing

Object types

- Service objects that perform task or return piece of information
- Objects that hold some data, and optionally expose some behaviour for manipulating that data

Services

- Have no identity
- Tackle cross-concern operations

Create better services

Create better services

- Inject all dependencies and configuration values as constructor arguments

Create better services

- Inject all dependencies and configuration values as constructor arguments
- All configuration values should be validated

Create better services

- Inject all dependencies and configuration values as constructor arguments
- All configuration values should be validated
- Throw an exception if constructor argument is invalid

Create better services

- Inject all dependencies and configuration values as constructor arguments
- All configuration values should be validated
- Throw an exception if constructor argument is invalid
- Service should be immutable; behaviour can’t be changed by calling any of its methods.

Inject dependencies and configuration values as constructor arguments

```php
interface Logger
{
    public function log(string $message): void;
}

final class FileLogger implements Logger
{
    private $formatter;

    private $logFilePath;

    public function __construct(Formatter $formatter, string $logFilePath)
    {
        // `Formatter` is dependency of `FileLogger`
        $this->formatter  = $formatter;

        /**
         * $logFilePath is a configuration value which tells the FileLogger
         * to which file the message should be written.
         */
        $this->logFilePath = $logFilePath;
    }

    public function log(string $message): void
    {
        $formattedMessage = $this->formatter->format($message);

        file_put_contents(
            $this->logFilePath,
            $formattedMessage,
            FILE_APPEND
        );
    }
}
```

All constructor arguments should be required

Problem 1: Optional dependency

```php
class OrderImporter
{
    private $logger;

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

    public function import($orderId): void
    {
        if ($this->logger instanceof  Logger) {
            //...
        }
        //...
    }
}
```

Problem 1: Optional dependency

```php
class OrderImporter
{
    private $logger;

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

    public function import($orderId): void
    {
        if ($this->logger instanceof  Logger) {
            //...
        }
        //...
    }
}
```

Recipe: Use Null Object to avoid optional dependency

```php
class NullLogger implements LoggerInterface
{
    public function log(string $message)
    {
        //Do nothing
    }
}

class OrderImporter
{
    private $logger;

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

    public function import(int $orderId): void
    {
        $this->logger(sprintf('Import order: %s', $orderId));

        //...
    }
}
```

Problem 2: Optional configuration value

```php
class MetadataFactory
{
    protected $configuration;

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

    public function getDescription():string
    {
        return $this->configuration['description'] ?? '';
    }
}
```

Problem 2: Optional configuration value

```php
class MetadataFactory
{
    protected $configuration;

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

    public function getDescription():string
    {
        return $this->configuration['description'] ?? '';
    }
}
```

Recipe: Provide configuration class with a sensible default state

```php
class MetadataFactory
{
    protected $configuration;

    public function __construct(Configuration $configuration)
    {
        $this->configuration = $configuration;
    }

    public function getDescription():string
    {
        return $this->configuration->getDescription();
    }
}

$metadataFactory = new MetadataFactory(Configuration::createDefault());
```

Make all dependencies explicit

Problem: retrieving dependencies using static accessors

```php
class DashboardController
{
    public function __invoke(): Response
    {
        $recentPosts  = [];

        if (Cache::has('recent_posts')) {
            $recentPosts = Cache::get('recent_posts');
        }

        // ...
    }
}
```

Recipe: Turn static dependencies into object dependencies

```php
class DashboardController
{
    private $cache;

    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }

    public function __invoke(): Response
    {
        $recentPosts  = [];

        if ($this->cache->has('recent_posts')) {
            $recentPosts = $this->cache->get('recent_posts');
        }

        // ...
    }
}
```

Don’t allow the behaviour of the service to change after it has been instantiated

```php
class Importer
{
    public $ignoreErrors = true;

    public function ignoreErrors(bool $ignoreErrors): void
    {
        $this->ignoreErrors = $ignoreErrors;
    }
    // ...
}

$importer = new Importer();

//When we use the `Importer` now, it will ignore errors...

$importer->ignoreErrors(false);

//When we use it now, it won't ignore errors anymore...
```

Don’t allow the behaviour of the service to change after it has been instantiated

```php
class Importer
{
    public $ignoreErrors = true;

    public function ignoreErrors(bool $ignoreErrors): void
    {
        $this->ignoreErrors = $ignoreErrors;
    }
    // ...
}

$importer = new Importer();

//When we use the `Importer` now, it will ignore errors...

$importer->ignoreErrors(false);

//When we use it now, it won't ignore errors anymore...
```

Make sure that it can’t happen.

Don’t allow the behaviour of the service to change after it has been instantiated

```php
class Importer
{
    public $ignoreErrors = true;

    public function ignoreErrors(bool $ignoreErrors): void
    {
        $this->ignoreErrors = $ignoreErrors;
    }
    // ...
}

$importer = new Importer();

//When we use the `Importer` now, it will ignore errors...

$importer->ignoreErrors(false);

//When we use it now, it won't ignore errors anymore...
```

Make sure that it can’t happen.

Do nothing inside a constructor, only assign properties

```php
class FileLogger implements Logger
{
    private $logFilePath;

    public function __construct(string $logFilePath)
    {
        $logFileDirectory = dirname($logFilePath);

        if (!is_dir($logFileDirectory)) {
            // create the directory if it doesn't exist yet
            mkdir($logFileDirectory, 0777, true);
        }

        touch($logFilePath);

        $this->logFilePath = $logFilePath;
    }

    // ...
}
```

Do nothing inside a constructor, only assign properties

```php
class FileLogger implements Logger
{
    private $logFilePath;

    public function __construct(string $logFilePath)
    {
        $logFileDirectory = dirname($logFilePath);
        if (!is_dir($logFileDirectory)) {
            // create the directory if it doesn't exist yet
            mkdir($logFileDirectory, 0777, true);
        }

        touch($logFilePath);

        $this->logFilePath = $logFilePath;
    }

    // ...
}
```

Recipe: Push work outside of the constructor

```php
class FileLogger implements Logger
{
    private $logFilePath;

    public function __construct(string $logFilePath)
    {
        $this->logFilePath = $logFilePath;
    }

    public function log(string $message): void
    {
        $this->ensureLogFileExists();

        // ...
    }

    private function ensureLogFileExists(): void
    {
        if (is_file($this->logFilePath)) {
            return;
        }

        $logFileDirectory = dirname($this->logFilePath);

        if (!is_dir($logFileDirectory)) {
            // create the directory if it doesn't exist yet
            mkdir($logFileDirectory, 0777, true);
        }

        touch($this->logFilePath);
    }
}
```

Throw an exception when argument is invalid

```php
class Alerting
{
    private $minimunLevel;

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

$alerting = new Alerting(-999999);
```

Throw an exception when argument is invalid

```php
class Alerting
{
    private $minimunLevel;

    public function __construct($minimunLevel)
    {
        if ($minimunLevel < 0) {
            throw new InvalidArgumentException(
                'Minimum alerting level should be greater then 0'
            );
        }
        $this->minimunLevel = $minimunLevel;
    }
}

$alerting = new Alerting(-999999);
```

Value Objects

- It’s immutable
- It wraps primitive type data
- It adds meaning by using domain specific terms (e.g. this isn’t just an int, it’s a Year)
- It imposes limitations by means of validation (e.g. this isn’t just any string, it’s a string with ‘@’ and int)

Better Value Objects

Better Value Objects

- Require the minimum amount of data needed to behave consistently

Better Value Objects

- Require the minimum amount of data needed to behave consistently
- Require meaningful data

Better Value Objects

- Require the minimum amount of data needed to behave consistently
- Require meaningful data
- Validate constructor arguments

Better Value Objects

- Require the minimum amount of data needed to behave consistently
- Require meaningful data
- Validate constructor arguments
- Extract new object to represent composite values

Require the minimum amount of data needed to behave consistently

```php
final class Position
{
    private $x;
    private $y;

    public function __construct()
    {
    }

    public function setX(int $x): void
    {
        $this->x = $x;
    }

    public function setY(int $y): void
    {
        $this->y = $y;
    }

    public function distanceTo(Position $other): float
    {
        return sqrt(
            ($other->x - $this->x) ** 2 + ($other->y - $this->y) ** 2
        );
    }
}

$position = new Position();
$position->setX(45);
$position->setY(60);
```

Require the minimum amount of data needed to behave consistently

```php
final class Position
{
    private int $x;
    private int $y;

    public function __construct(int $x, int $y)
    {
        $this->x = $x;
        $this->y = $y;
    }

    public function distanceTo(Position $other): float
    {
        return sqrt(
            ($other->x - $this->x) ** 2 +
            ($other->y - $this->y) ** 2);
    }
}

/*
 * `x` and `y` have to be provided, or you won't be able to get an
 * instance of `Position`
 */
$position = new Position(45, 60);
```

Require the data that is meaningful

```php
class Coordinates
{
    private $latitude;
    private $longitude;

    public function __construct(float $latitude, float $longitude)
    {
        $this->latitude = $latitude;
        $this->longitude = $longitude;
    }

    // ...
}

$meaningfulCoordinates = new Coordinates(45.0, -60.0);
/*
 * There's nothing that stops us from creating a `Coordinates`
 * object that doesn't make any sense at all:
 */
$offThePlanet = new Coordinates(1000.0, -20000);
```

Require the data that is meaningful

```php
class Coordinates
{
    private $latitude;
    private $longitude;

    public function __construct(float $latitude, float $longitude)
    {
        $this->latitude = $latitude;
        $this->longitude = $longitude;
    }

    // ...
}

$meaningfulCoordinates = new Coordinates(45.0, -60.0);
/*
 * There's nothing that stops us from creating a `Coordinates`
 * object that doesn't make any sense at all:
 */
$offThePlanet = new Coordinates(1000.0, -20000);
```

Require the data that is meaningful

```php
final class Coordinates
{
    // ...

    public function __construct(float $latitude, float $longitude)
    {
        if ($latitude > 90 || $latitude < -90) {
            throw new InvalidArgumentException(
                'Latitude should be between -90 and 90'
            );
        }
        $this->latitude = $latitude;
        if ($longitude > 180 || $longitude < -180) {
            throw new InvalidArgumentException(
                'Longitude should be between -180 and 180'
            );
        }
        $this->longitude = $longitude;
    }
}
```

Extract new objects to represent composite values

```php
final class Amount
{
    // ...
}

final class Currency
{
    // ...
}


// Amount and Currency always go together, like here:
final class Product
{
    public function setPrice(Amount $amount,Currency $currency): void
    {
        // ...
    }
}

// And here:
final class Converter
{
    public function convert(
      Amount $localAmount,
      Currency $localCurrency,
      Currency $targetCurrency
    ): Amount {
        // ...
    }
}
```

Extract new objects to represent composite values

```php
final class Amount
{
    // ...
}

final class Currency
{
    // ...
}


// Amount and Currency always go together, like here:
final class Product
{
    public function setPrice(Amount $amount,Currency $currency): void
    {
        // ...
    }
}

// And here:
final class Converter
{
    public function convert(
      Amount $localAmount,
      Currency $localCurrency,
      Currency $targetCurrency
    ): Amount {
        // ...
    }
}
```

Extract new objects to represent composite values

```
final class Money
{
    public function __construct(Amount $amount, Currency $currency)
    {
        // ...
    }
}
```

A template for implementing methods

```
[scope] function methodName(type name,  ...): void|[returnType]
{
    [pre-conditions checks]

    [failure scenarios]

    [happy path]

    [post-conditions checks]

    [return void|specific-return-type]
}
```

Pre-condition checks

```php
if ([/* some pre-condition wasn't met */) {
    throw new InvalidArgumentException(...);
}
```

```php
Assertion::inArray($value, ['value1', 'value2', 'value3']);
```

```php
Assertion::allIsInstanceOf($value, EventListenet::class);
```

Failure scenario

```php
public function getRowById(int $id): array
{
    /*
     * This could throw an InvalidArgumentException:
     */
    Assertion::greaterThan($id, 0, 'ID should be greater than 0');

    /*
     * This could cause either an InvalidArgumentException or a
     * RuntimeException to be thrown from the code that calls the
     * database:
     */
    $record = $this->db->find($id);

    /*
     * This is _our_ failure scenario: we couldn't find the record,
     * so we throw a `RuntimeException`.
     */
    if ($record === null) {
        throw new RuntimeException(
          sprintf(
            'Could not find record with ID "%d"',
            $id
        );
    }

    return $record;
}
```

Failure scenario

```php
public function getRowById(int $id): array
{
    /*
     * This could throw an InvalidArgumentException:
     */
    Assertion::greaterThan($id, 0, 'ID should be greater than 0');

    /*
     * This could cause either an InvalidArgumentException or a
     * RuntimeException to be thrown from the code that calls the
     * database:
     */
    $record = $this->db->find($id);

    /*
     * This is _our_ failure scenario: we couldn't find the record,
     * so we throw a `RuntimeException`.
     */
    if ($record === null) {
        throw new RuntimeException(
          sprintf(
            'Could not find record with ID "%d"',
            $id
        );
    }

    return $record;
}
```

Failure scenario

```php
public function getRowById(int $id): array
{
    /*
     * This could throw an InvalidArgumentException:
     */
    Assertion::greaterThan($id, 0, 'ID should be greater than 0');

    /*
     * This could cause either an InvalidArgumentException or a
     * RuntimeException to be thrown from the code that calls the
     * database:
     */
    $record = $this->db->find($id);

    /*
     * This is _our_ failure scenario: we couldn't find the record,
     * so we throw a `RuntimeException`.
     */
    if ($record === null) {
        throw new RuntimeException(
          sprintf(
            'Could not find record with ID "%d"',
            $id
          );
    }

    return $record;
}
```

Happy path

Post-condition checks

```php
public function someVeryComplicatedCalculation(): int
{
    // ...
    $result = ...;

    /*
     * This post-condition check is just a safety check, a.k.a.
     * "This should never happen".
     */
    Assertion::greaterThan(0, $result); return $result;
}
```

Return value

Some rules for exceptions

Use custom exception classes only if needed

1. To catch specific exception type

```
try {
    // possibly throws SomeSpecific exception
} catch (SomeSpecific $exception) {
    // ...
}
```

Use custom exception classes only if needed

2. If there are multiple ways to instantiate exception class

```php
final class CouldNotDeliverOrder extends RuntimeException
{
    public static function itWasAlreadyDelivered(): CouldNotDeliverOrder
    {
        // ...
    }

    public static function insufficientQuantitiesInStock(): CouldNotDeliverOrder
    {
        // ...
    }
}
```

Use custom exception classes only if needed

3. To use named constructor

```php
final class CouldNotFindProduct extends RuntimeException
{
    public static function withId(ProductId $productId): CouldNotFindProduct
    {
        return new self(
            sprintf(
                'Could not find a product with ID "%s"',
                $productId
            )
        );
    }

}

throw CouldNotFindProduct::forId(...);
```

Use custom exception classes only if needed

3. To use named constructor

```php
final class CouldNotFindProduct extends RuntimeException
{
    public static function withId(ProductId $productId): CouldNotFindProduct
    {
        return new self(
            sprintf(
                'Could not find a product with ID "%s"',
                $productId
            )
        );
    }

}
```

```php
throw CouldNotFindProduct::forId(...);
```

Use named constructors to indicate reasons for failure

```php
final class CouldNotFindStreetName extends RuntimeException
{
    public static function withPostalCode(PostalCode $postalCode): CouldNotFindStreetName
    {
      // ...
    }
}
```

```php
final class InvalidTargetPosition extends LogicException
{
    public static function becauseItIsOutsideTheMap(...): InvalidTargetPosition
    {
      // ...
    }
}
```

Add detailed messages

```php
// Before:
final class CouldNotFindProduct extends RuntimeException
{
}

// At the call site:
throw new CouldNotFindProduct(
    sprintf(
        'Could not find a product with ID "%s"',
        $productId
    )
);
```

```php
//After:
final class CouldNotFindProduct extends RuntimeException
{
    public static function withId(ProductId $productId): CouldNotFindProduct
    {
        return new self(
            sprintf(
                'Could not find a product with ID "%s"',
                $productId
            )
        );
    }
}

// At the call site:
throw CouldNotFindProduct::withId($productId);
```

Changing the behaviour of services

Introduce constructor arguments to make behaviour configurable

```php
final class FileLogger
{
    public function log($message): void {
        file_put_contents(
            '/var/log/app.log',
            $message,
            FILE_APPEND
        );
    }
}
```

Introduce constructor arguments to make behaviour configurable

```php
final class FileLogger
{
    public function log($message): void {
        file_put_contents(
            '/var/log/app.log',
            $message,
            FILE_APPEND
        );
    }
}
```

Introduce constructor arguments to make behaviour configurable

```php
final class FileLogger
{
    private $filePath;

    public function __construct(string $filePath)
    {
        $this->filePath = $filePath;
    }

    public function log($message): void
    {
        file_put_contents($this->filePath, $message, FILE_APPEND);
    }
}

$logger = new FileLogger('/var/log/app.log');
```

Introduce constructor arguments to make behaviour replaceable

```php
final class ParameterLoader
{
    public function load($filePath): array
    {
        /*
         * Load parameters from the file and merge add them
         * to the already loaded ones.
         */
        $rawParameters = json_decode(
            file_get_contents($filePath), true
        );

        $parameters = [];

        foreach ($rawParameters as $key => $value) {
            $parameters[] = new Parameter($key, $value);
        }

        return $parameters;
    }
}

$loader = new ServiceConfigurationLoader(__DIR__ . '/parameters.json');
```

I demand ParameterLoader to support loading an XML and Yaml files

Introduce constructor arguments to make behaviour replaceable

```php
final class ParameterLoader
{
    public function load($filePath): array
    {
        /*
         * Load parameters from the file and merge add them
         * to the already loaded ones.
         */
        $rawParameters = json_decode(
            file_get_contents($filePath), true
        );

        $parameters = [];

        foreach ($rawParameters as $key => $value) {
            $parameters[] = new Parameter($key, $value);
        }

        return $parameters;
    }
}

$loader = new ServiceConfigurationLoader(__DIR__ . '/parameters.json');
```

Introduce constructor arguments to make behaviour replaceable

```php
interface FileLoader
{
    /**
     * Load an array of key/value pairs representing parameters
     * stored in a file at the given location.
     */
    public function loadFile(string $filePath): array
}

final class JsonFileLoader implements FileLoader
{
    public function loadFile(string $filePath): array
    {
        Assertion::isFile($filePath);
        $result = json_decode(file_get_contents($filePath), true);

        if (!is_array($result)) {
            throw new RuntimeException(
                sprintf(
                    'Decoding "%s" did not result in an array',
                    $filePath
                )
            );
        }

        return $result;
    }
}
```

Introduce constructor arguments to make behaviour replaceable

```php
interface FileLoader
{
    /**
     * Load an array of key/value pairs representing parameters
     * stored in a file at the given location.
     */
    public function loadFile(string $filePath): array
}

final class JsonFileLoader implements FileLoader
{
    public function loadFile(string $filePath): array
    {
        Assertion::isFile($filePath);
        $result = json_decode(file_get_contents($filePath), true);

        if (!is_array($result)) {
            throw new RuntimeException(
                sprintf(
                    'Decoding "%s" did not result in an array',
                    $filePath
                )
            );
        }

        return $result;
    }
}
```

Introduce constructor arguments to make behaviour replaceable

```php
interface FileLoader
{
    /**
     * Load an array of key/value pairs representing parameters
     * stored in a file at the given location.
     */
    public function loadFile(string $filePath): array
}

final class JsonFileLoader implements FileLoader
{
    public function loadFile(string $filePath): array
    {
        Assertion::isFile($filePath);
        $result = json_decode(file_get_contents($filePath), true);

        if (!is_array($result)) {
            throw new RuntimeException(
                sprintf(
                    'Decoding "%s" did not result in an array',
                    $filePath
                )
            );
        }

        return $result;
    }
}
```

Introduce constructor arguments to make behaviour replaceable

```php
final class ParameterLoader
{
    private FileLoader $fileLoader;

    public function __construct(FileLoader $fileLoader)
    {
        $this->fileLoader = $fileLoader;
    }

    public function load($filePath): array
    {
        // ...
        foreach (...) {
            if (...) {
                $rawParameters = $this->fileLoader->loadFile($filePath);
            }
        }
        // ...
    }
}

$parameterLoader = new ParameterLoader(new JsonFileLoader());
$parameterLoader->load(__DIR__ . '/parameters.json');
```

Introduce constructor arguments to make behaviour replaceable

```php
final class ParameterLoader
{
    private FileLoader $fileLoader;

    public function __construct(FileLoader $fileLoader)
    {
        $this->fileLoader = $fileLoader;
    }

    public function load($filePath): array
    {
        // ...
        foreach (...) {
            if (...) {
                $rawParameters = $this->fileLoader->loadFile($filePath);
            }
        }
        // ...
    }
}

$parameterLoader = new ParameterLoader(new JsonFileLoader());
$parameterLoader->load(__DIR__ . '/parameters.json');
```

Introduce constructor arguments to make behaviour replaceable

```php
final XmlFileLoader implements FileLoader {
    // ...
}

$parameterLoader = new ParameterLoader(new XmlFileLoader());
$parameterLoader->load(__DIR__ . '/parameters.xml');
```

What other ways to change object's behaviour?

Create better objects

Create better objects

- Introduce more types

Create better objects

- Introduce more types
- Be more strict about them

Create better objects

- Introduce more types
- Be more strict about them
- Design objects that only accept valid data

Create better objects

- Introduce more types
- Be more strict about them
- Design objects that only accept valid data
- Design objects that can only be used in valid ways

Create better objects

- Introduce more types
- Be more strict about them
- Design objects that only accept valid data
- Design objects that can only be used in valid ways
- Use composition over inheritance to change object's behaviour

Questions

Thanks