OBJECT-ORIENTED PROGRAMMING - OBJECT DESIGN

ARCHITECTURE LOGICIELLE

@gboissinot


Object Design
Style Guide
Matthias Noback
Foreword by Ross Tuck
MANNING

CLASSES AND OBJECTS

Listing 1.1 A minimum viable class

```
class Foo
{
    // There's nothing here
}

object1 = new Foo();
object2 = new Foo();

object1 == object2 // false
```

Two instances of the same class should not be considered the same.

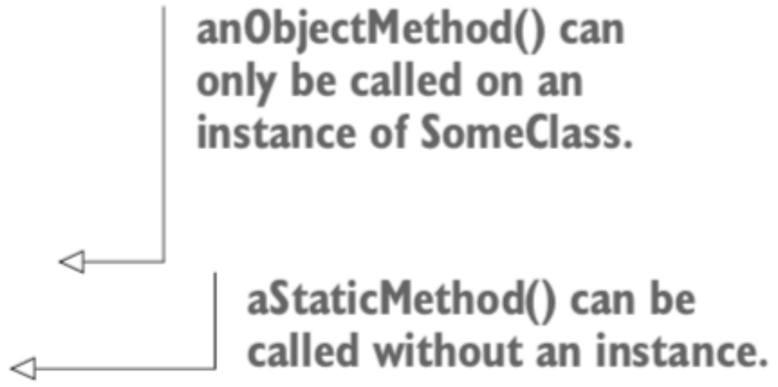
Once you have an instance, you can call methods on it.

Listing 1.2 Calling a method on an instance

```
class Foo
{
    public function someMethod(): void
    {
        // Do something
    }
}

object1 = new Foo();
object1.someMethod();
```

METHODS

```
class Foo
{
    public function anObjectMethod(): void
    {
        // ...
    }

    public static function aStaticMethod(): void
    {
        // ...
    }
}

object1 = new Foo();
object1.anObjectMethod();

Foo.aStaticMethod();
```



CONSTRUCTOR

```
class Foo
{
    public function __construct()
    {
        // Prepare the object
    }
}

object1 = new Foo();
```

__construct() will be implicitly called before a Foo instance gets assigned to objectl.

STATIC FACTORY METHOD

```
class Foo
{
    public static function create(): Foo
    {
        return new Foo();
    }
}

object1 = Foo.create();
object2 = Foo.create();
```

OBJECT STATE & PROPERTIES SCOPING

```
class Foo
{
    private int someNumber;
    private string someString;

    public function __construct()
    {
        this.someNumber = 10;
        this.someString = 'Hello, world!';
    }
}

object1 = new Foo();
```

After instantiation, someNumber and someString will contain 10 and 'Hello, world!' respectively.

SCOPING

```
class Foo
{
    private int someNumber;

    // ...

    public function getSomeNumber(): int
    {
        return this.someNumber;
    }

    public function getSomeNumberFrom(Foo other): int
    {
        return other.someNumber;
    }
}

object1 = new Foo();
object2 = new Foo();

object2.getSomeNumberFrom(object1);
```

Foo, of course, has access to its own someNumber property.

Foo also has access to other's private property someNumber.

This will return the value of objectl's someNumber property.

BEHAVIOR

```
class Foo
{
    public function someMethod(): int
    {
        return /* ... */;
    }

    public function someOtherMethod(): void

    {
        // ...
    }
}

object1 = new Foo();
value = object1.someMethod();

object1.someOtherMethod();
```

someMethod() returns an integer, which we can capture in a variable.

someOtherMethod() doesn't return anything specific, so a client can't capture its return value.

INHERITANCE

1

```
interface Foo
{
    public function foo(): void;
}

class Bar implements Foo
{
}
```

The Foo interface declares a foo() method but doesn't provide an implementation.

Bar is an incorrect implementation of Foo, because it doesn't have an implementation for the foo() method.

2

```
abstract class Foo
{
    abstract public function foo(): void;

    public function bar(): void
    {
        // ...
    }
}

class Baz extends Foo
{
    public function foo(): void
    {
    }
}
```

The foo() method is abstract and has to be defined by a subclass.

Foo provides an actual implementation for the bar() method.

Baz is a correct implementation of Foo, because it provides an implementation for the previously abstract foo() method.

INHERITANCE

3

```
class Foo
{
    public function bar(): void
    {
        // do something
    }
}

class Bar extends Foo
{
    public function bar(): void
    {
        // do something else
    }
}
```

Foo is a regular class, without any abstract methods.

Bar extends Foo, which is now its parent class. It can change the behavior of its bar() method.

Foo is a regular class, without any abstract methods.

INHERITANCE

4

```
class Foo
{
    public function foo(): void
    {
        // do something
    }

    protected function bar(): void
    {
    }

    private function baz(): void
    {
    }
}

class Bar extends Foo
{
    public function someMethod(): void
    {
        $this->foo();

        $this->bar();

        //$this->baz();
    }
}
```

foo() is available because it's a public method.

bar() is available because it's a protected method.

baz() is not available because it's a private method.

FINAL CLASS

```
final class Bar
{
    // ...
}

class Baz extends Bar // won't work
{
    // ...
}
```

Bar is marked as final, so Baz can't extend it.

Donnez des exemples

pour changer

le comportement sans l'usage de l'héritage

POLYMORPHISM

```
class Foo
{
    // ...
}

final class Bar
{
    public function bar(Foo foo): void
    {
        foo.someMethod();
    }
}
```

COMPOSITION

```
final class Bar
{
    private Foo foo;

    public function __construct(Foo foo)
    {
        this.foo = foo;
    }
}
```

Fournir un exemple concret illustrant

la composition dans le cadre

d'une application e-commerce

COMPOSITION - E-COMMERCE EXEMPLE

```
final class Order
{
    private array lines;

    public function __construct(array lines)
    {
        this.lines = lines;
     }
}
```

Each element in lines is a Line object.

DEPENDENCIES

Comment ajouter une dépendance vers un objet Logger ?

DEPENDENCIES

```
class Foo
{
    public function someMethod(): void

    {
        logger = new Logger();
        logger.debug('...');
    }
}
```

Foo instantiates a Logger when needed.

1

```
class Foo
{
    public function someMethod(): void
    {
        logger = ServiceLocator.getLogger();
        logger.debug('...');
    }
}
```

Foo fetches a Logger instance from a known location.

2

DEPENDENCIES - DEPENDENCY INJECTION (DI)

```
class Foo
{
    private Logger logger;

    public function __construct(Logger logger)
    {
        this.logger = logger;
    }

    public function someMethod(): void
    {
        this.logger.debug('...');
    }
}
```

Foo has an instance of Logger provided to it as a constructor argument.

3

DEPENDENCY INJECTION (DI) - EXEMPLE

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

final class FileLogger implements Logger
{
    private Formatter formatter;

    public function __construct(Formatter formatter)
    {
        this.formatter = formatter;
    }

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

        // ...
    }
}

logger = new FileLogger(new DefaultFormatter());
logger.log('A message');
```

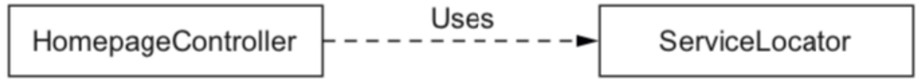
Formatter is a dependency of FileLogger.

SERVICE LOCATOR

```
final class ServiceLocator
{
    private array services;

    public function __construct()
    {
        this.services = [
            'logger' => new FileLogger(/* ... */)
        ];
    }

    public function get(string identifier): object
    {
        if (!isset(this.services[identifier])) {
            throw new LogicException(

                'Unknown service: ' . identifier
            );
        }

        return this.services[identifier];
    }
}
```

You can have any number of services here.

SERVICE LOCATOR



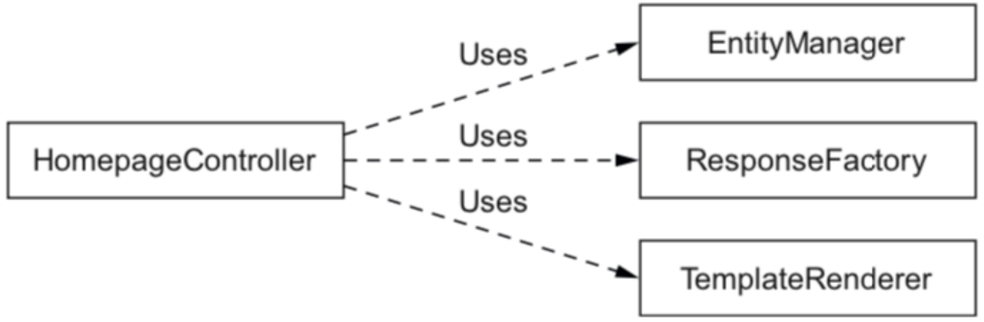

```
final class HomepageController
{
    private ServiceLocator locator;

    public function __construct(ServiceLocator locator)
    {
        this.locator = locator;
    }

    public function execute(Request request): Response
    {
        user = this.locator.get(EntityManager.className)
            .getRepository(User.className)
            .getById(request.get('userId'));

        return this.locator.get(ResponseFactory.className)
            .create()
            .withContent(
                this.locator.get(TemplateRenderer.className)
                    .render(
                        'homepage.html.twig',
                        [
                            'user' => user
                        ]
                    ),
                'text/html'
            );
    }
}
```

Instead of injecting the dependencies we need, we inject the whole ServiceLocator, from which we can later retrieve any specific dependency.

SERVICE LOCATOR

Refactoring du Controller
sans le Service Locator

SERVICE LOCATOR -> DI




```
final class HomepageController
{
    private EntityManager entityManager;
    private ResponseFactory responseFactory;
    private TemplateRenderer templateRenderer;

    public function __construct(
        EntityManager entityManager,
        ResponseFactory responseFactory,
        TemplateRenderer templateRenderer
    ) {
        this.entityManager = entityManager;
        this.responseFactory = responseFactory;
        this.templateRenderer = templateRenderer;
    }

    public function execute(Request request): Response
    {
        user = this.entityManager.getRepository(User.className)
            .getById(request.get('userId'));

        return this.responseFactory
            .create()
            .withContent(
                this.templateRenderer.render(
                    'homepage.html.twig',
                    [
                        'user' => user
                    ]
                ),
                'text/html'
            );
    }
}
```

DI

Fournir un exemple de code
pour le service d'enregistrement
des utilisateurs, en fournissant
des alternatives à l'implémentation
normale (celle en production)
via des implémentations (stubs, fakes, etc)

DEPENDENCIES

Refactoring


```
user = this.entityManager
    .getRepository(User.className)
    .getById(request.get('userId'));
user.changePassword(newPassword);
this.entityManager.flush();
```

REPOSITORY PATTERN

```
user = this.userRepository.getById(request.get('userId'));
user.changePassword(newPassword);
this.userRepository.save(user);
```

CONFIGURATION VALUES

```
final class FileLogger implements Logger
{
    // ...

    private string logFilePath;

    public function __construct(
        Formatter formatter,
        string logFilePath
    ) {
        // ...

        this.logFilePath = logFilePath;
    }

    public function log(string message): void
    {
        // ...

        file_put_contents(
            this.logFilePath,
            formattedMessage,
            FILE_APPEND
        );
    }
}
```

logFilePath is a configuration value that tells the FileLogger to which file the messages should be written.

CONFIGURATION VALUES

```
final class Credentials
{
    private string username;
    private string password;

    public function __construct(string username, string password)
    {
        this.username = username;
        this.password = password;
    }

    public function username(): string
    {
        return this.username;
    }

    public function password(): string
    {
        return this.password;
    }
}

final class ApiClient
{
    private Credentials credentials;

    public function __construct(Credentials credentials)
    {
        this.credentials = credentials;
    }
}
```

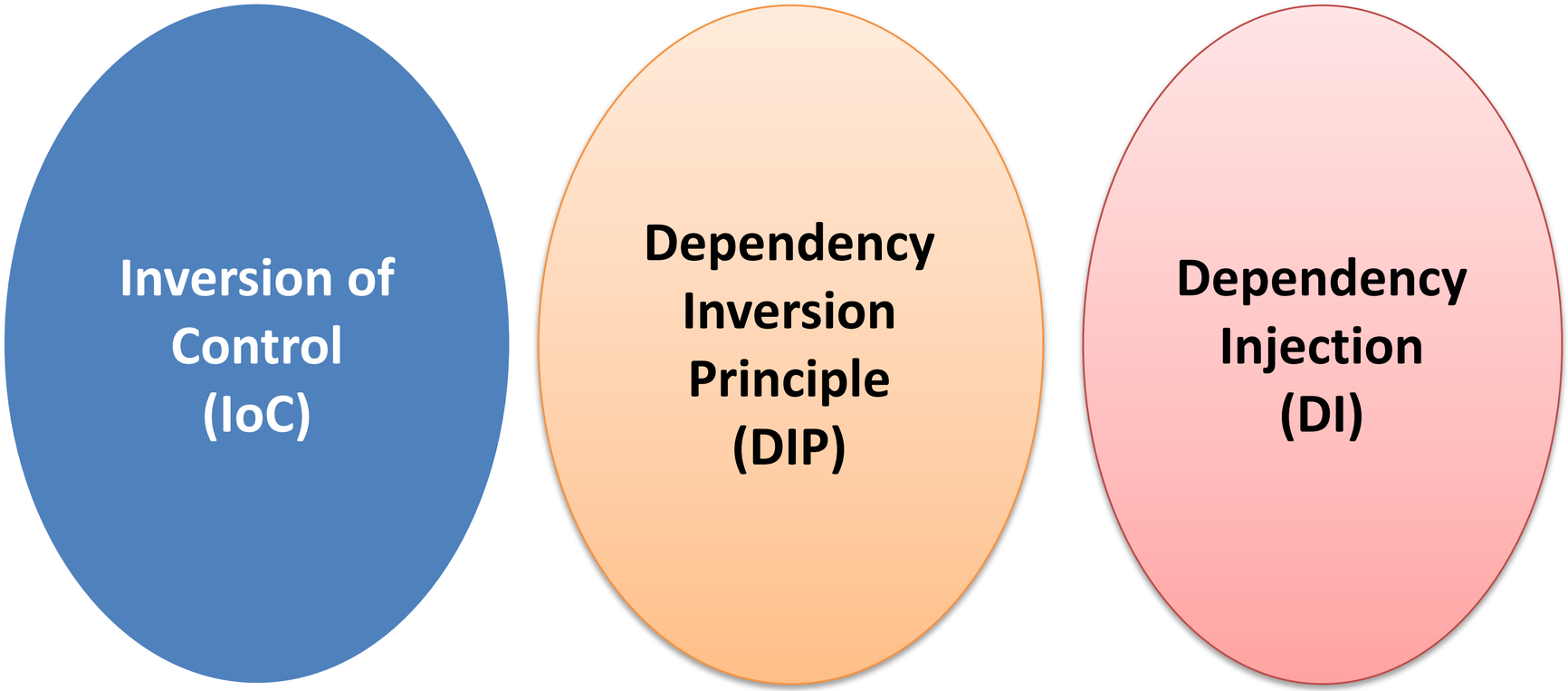
CONFIGURATION VALUES - EXERCISE REFACTORING

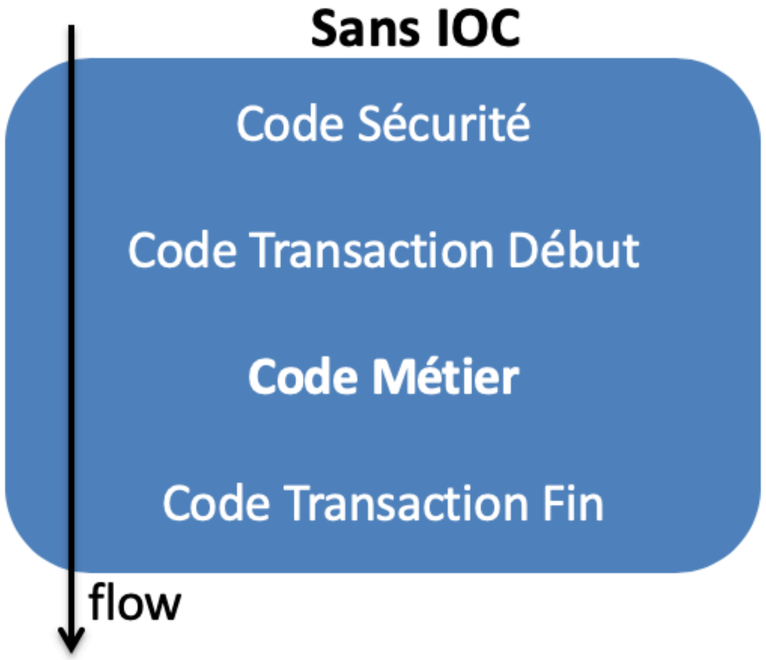

```
final class MySQLTableGateway
{
    public function __construct(
        string host,
        int port,
        string username,
        string password,
        string database,
        string table
    ) {
        // ...
    }
}
```

CONFIGURATION VALUES – EXERCISE SOLUTION

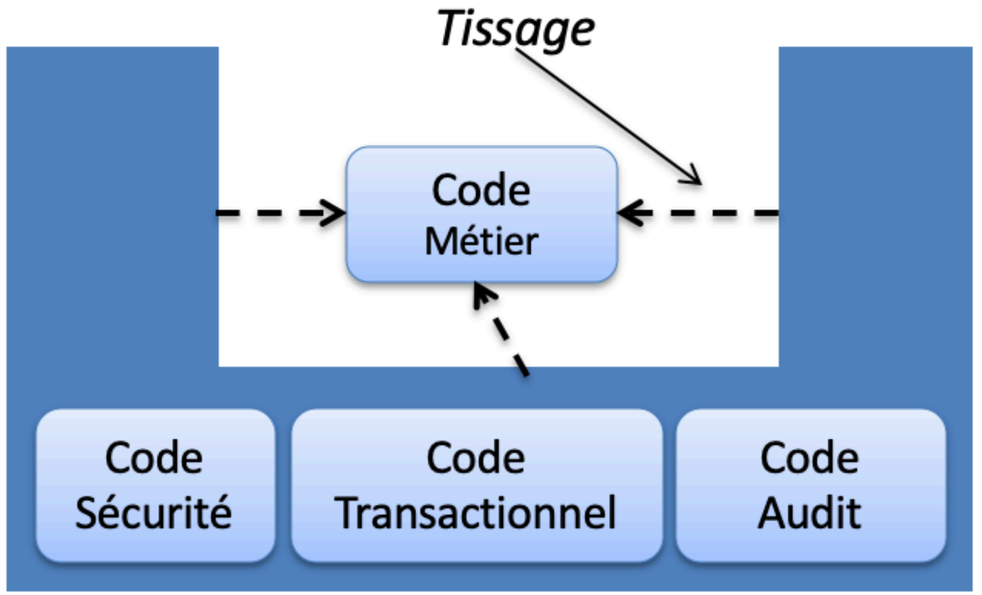
```
final class MySQLTableGateway
{
    public function __construct(
        ConnectionConfiguration connectionConfiguration,
        string table
    ) {
        // ...
    }
}
```

The name of the table isn't part of the information needed to make the connection to the database, so it isn't moved to the new ConnectionConfiguration object.


Inversion of Control (IoC)
Dependency Inversion Principle (DIP)
Dependency Injection (DI)

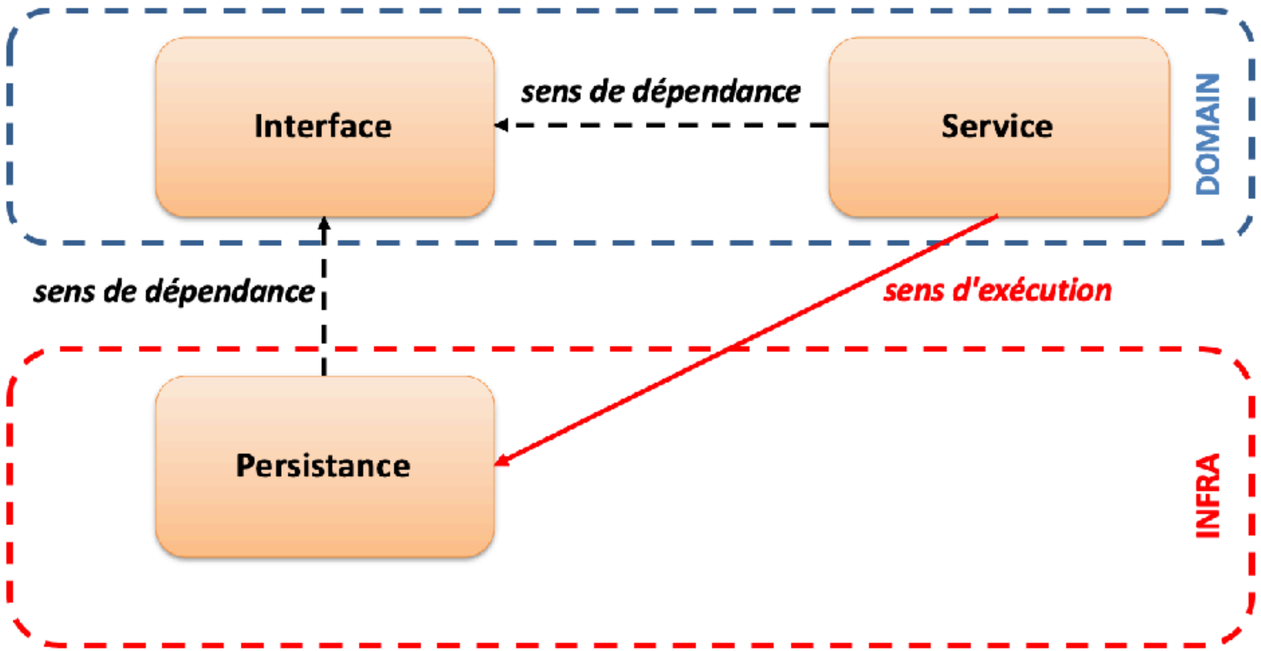


- Appel impératif des éléments d'infrastructure (Sécurité, Transaction, etc)
- Mélange du code métier et des responsabilités techniques



- Déclaration des éléments d'infrastructure : métadonnées XML ou Annotations
- Invocation de ces éléments par le container (ou framework) : souvent implémenté par AOP

(DIP)



- L'ordre de dépendance est l'inverse de l'ordre d'exécution
- Cela fonctionne grâce au polymorphisme (langage OO)

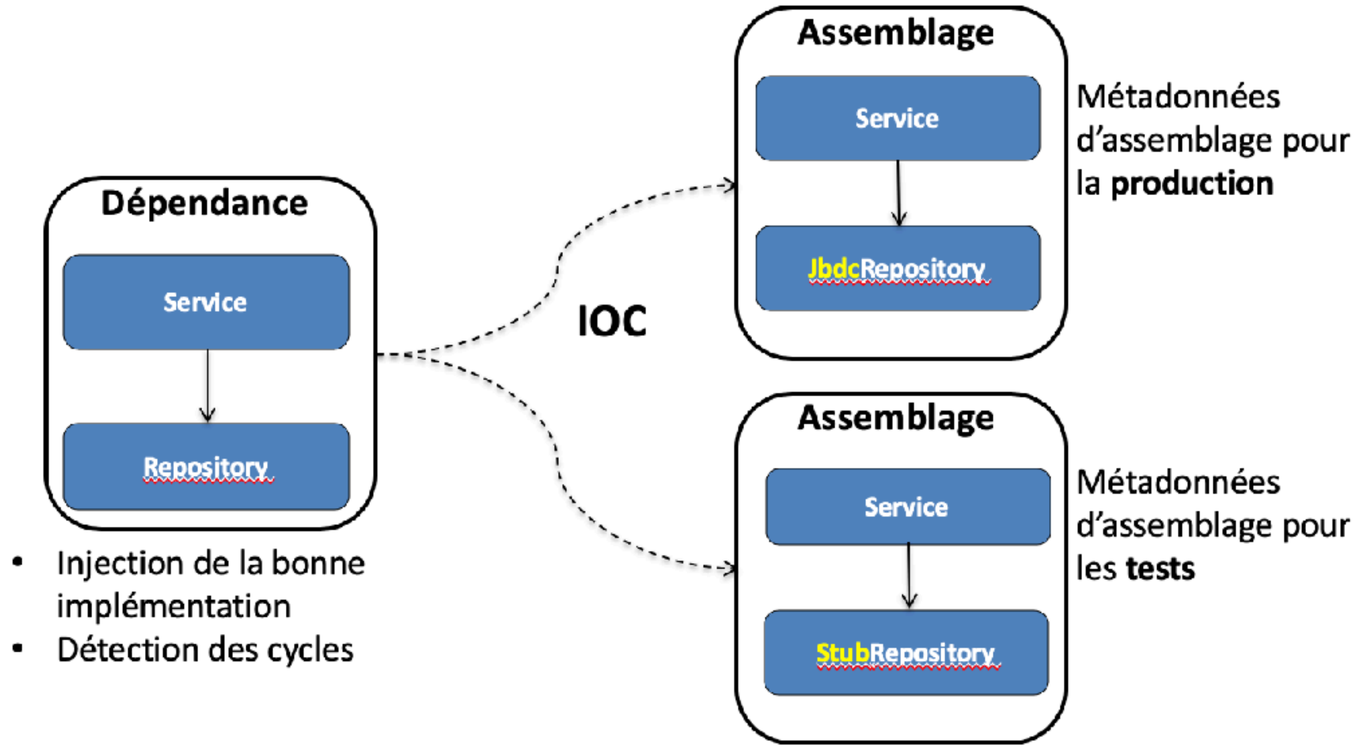
- Permet à un module métier (abstrait, général, stable) d'invoquer un module d'infrastructure (concret, détail, instable) sans en dépendre
- Permet au module métier d'avoir un comportement riche sans introduire de dépendances vers l'infrastructure (BD, réseau, etc)

DIP

Fournir un exemple de code de code d'un service qui enregistre

les utilisateurs d'une application

(DI)



- La DI est un type particulier d'IoC où la préoccupation transverse est la construction d'un graphe de composants interdépendants
- De multiples objectifs :
  - Abstraire l'implémentation concrète injectée
  - Déléguer au conteneur l'initialisation ordonnée d'un graphe de composants dépendants

OBJECT TYPES

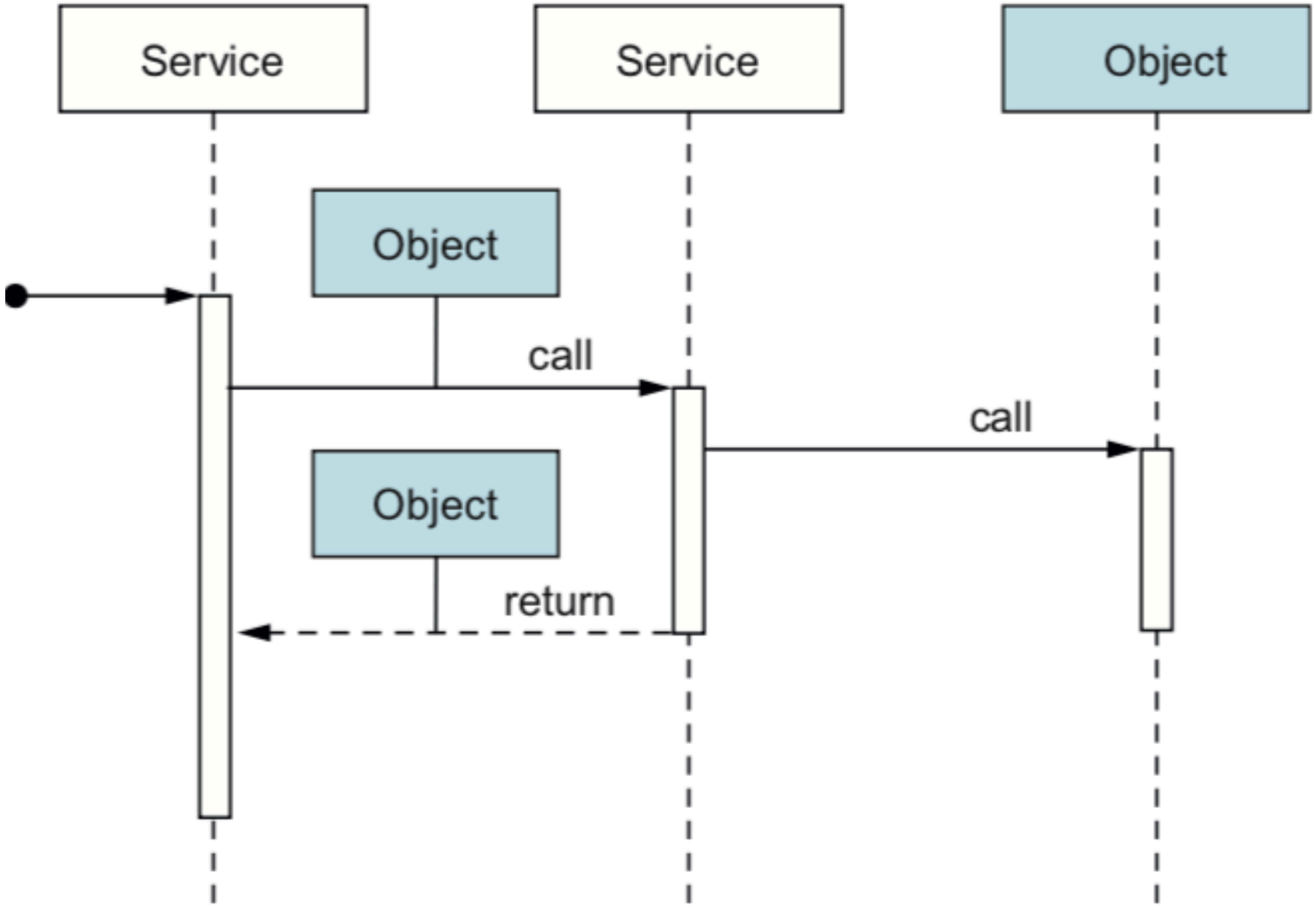
SERVICE OBJECTS
(CONTROLLER, RENDERER, CALCULATOR, ETC)

OBJECTS
(HOLD SOME DATA AND OPTIONALLY EXPOSE SOME BEHAVIOUR)

<==>

ENTITIES, VALUE OBJECTS, DTO

OBJECT TYPES



OBJECT TYPES

Donner des exemples de noms de classes

pour chaque type

DEPENDENCIES - OPTIONAL

```
final class NullLogger implements Logger
{
    public function log(string message): void
    {
        // Do nothing
    }
}
```

ALL IS DEPENDENCIES

Fournir un refactoring


```
// Before:

final class ResponseFactory
{
    public function createApiResponse(array data): Response
    {
        return new Response(
            json_encode(data, JSON_THROW_ON_ERROR | JSON_FORCE_OBJECT),
            [
                'Content-Type' => 'application/json'
            ]
        );
    }
}
```

json_encode() is a hidden dependency.

ALL IS DEPENDENCIES – SOLUTION


```
// After:

final class JsonEncoder
{
    /**
     * @throws RuntimeException
     */
    public function encode(array data): string
    {
        try {
            return json_encode(
                data,
                JSON_THROW_ON_ERROR | JSON_FORCE_OBJECT
            );
        } catch (RuntimeException previous) {
            throw new RuntimeException(
                'Failed to encode data: ' . var_export(data, true),
                0,
                previous
            );
        }
    }
}

final class ResponseFactory
{
    private JsonEncoder jsonEncoder;

    public function __construct(JsonEncoder jsonEncoder)
    {
        this.jsonEncoder = jsonEncoder;
    }

    public function createApiResponse(data): Response
    {
        return new Response(
            this.jsonEncoder.encode(data),
            [
```

From now on, a call to json_encode() will always have the right arguments.

We can throw our own exception now, providing more information that will help us with debugging.

A JsonEncoder instance can now be injected as an actual, explicit dependency.

Proposer une solution pour éviter un appel système

FUNCTIONS AS OBJECT DEPENDENCIES


```
final class MeetupRepository
{
    private Connection connection;


public function __construct(Connection connection)
{
    this.connection = connection;
}

public function findUpcomingMeetups(string area): array
{
    now = new DateTime();

    return this.findMeetupsScheduledAfter(now, area);
}

public function findMeetupsScheduledAfter(
    DateTime time,
    string area
): array {
    // ...
}
```

Instantiating a DateTime object with no arguments will implicitly ask the system what the current time is.

FUNCTIONS AS OBJECT DEPENDENCIES – SOLUTION


```
interface Clock
{
    public function currentTime(): DateTime;
}

final class SystemClock implements Clock
{
    public function currentTime(): DateTime
    {
        return new DateTime();
    }
}

final class MeetupRepository
{
    // ...
    private Clock clock;

    public function __construct(
        Clock clock,
        /* ... */
    ) {
        this.clock = clock;
    }

    public function findUpcomingMeetups(string area): array

    {
        now = this.clock.currentTime();

        // ...
    }
}

meetupRepository = new MeetupRepository(new SystemClock());
meetupRepository.findUpcomingMeetups('NL');
```

A suitable name for this new service, which can tell us the current time, would simply be "Clock."

The standard implementation for this service will use the system's clock to return a DateTime object representing the current time.

Instead of "creating" the current time on the spot, we can now ask the Clock service for it.

CONSTRUCTOR

Fournir un refactoring


```
final class FileLogger implements Logger
{
    private string logFilePath;

    public function __construct(string logFilePath)

    {
        logFileDirectory = dirname(logFilePath);
        if (!is_dir(logFileDirectory)) {
            mkdir(logFileDirectory, 0777, true);
        }

        touch(logFilePath);

        this.logFilePath = logFilePath;
    }

    // ...
}
```

Create the directory if it doesn't exist yet.

CONSTRUCTOR – SOLUTION 1


```
final class FileLogger implements Logger
{
    private string 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)) {
            mkdir(logFileDirectory, 0777, true);
        }

        touch(this.logFilePath);
    }
}
```

Only copy values into properties.

CONSTRUCTOR - SOLUTION 2


```
final class FileLogger implements Logger
{
    private string logFilePath;

    /**
     * @param string logFilePath Absolute path to a log file that
     *                           already exists and is writable.
     */
    public function __construct(string logFilePath)
    {
        this.logFilePath = logFilePath;
    }

    // ...
}

final class LoggerFactory
{
    public function createFileLogger(string logFilePath): FileLogger
    {
        if (!is_file(logFilePath)) {
            logFileDirectory = dirname(logFilePath);
            if (!is_dir(logFileDirectory)) {
                mkdir(logFileDirectory, 0777, true);
            }

            touch(logFilePath);
        }

        if (!is_writable(logFilePath)) {
            throw new InvalidArgumentException(
                'Log file path "{logFilePath}" should be writable'
            );
        }

        return new FileLogger(logFilePath);
    }
}
```

Besides taking care of the directory, LoggerFactory now also makes sure that the log file exists and is writable.

CONSTRUCTOR EXCEPTION

```
final class Alerting
{
    private int minimumLevel;

    public function __construct(int minimumLevel)
    {
        if (minimumLevel <= 0) {
            throw new InvalidArgumentException(
                'Minimum alerting level should be greater than 0'
            );
        }
        this.minimumLevel = minimumLevel;
    }
}

alerting = new Alerting(-99999999);
```

This will throw an InvalidArgumentException.