Composition and inheritance

Object-oriented programming levelled up

Pol Dellaiera - DIGIT B4 - DIGIT PHP SUPPORT

November 2021

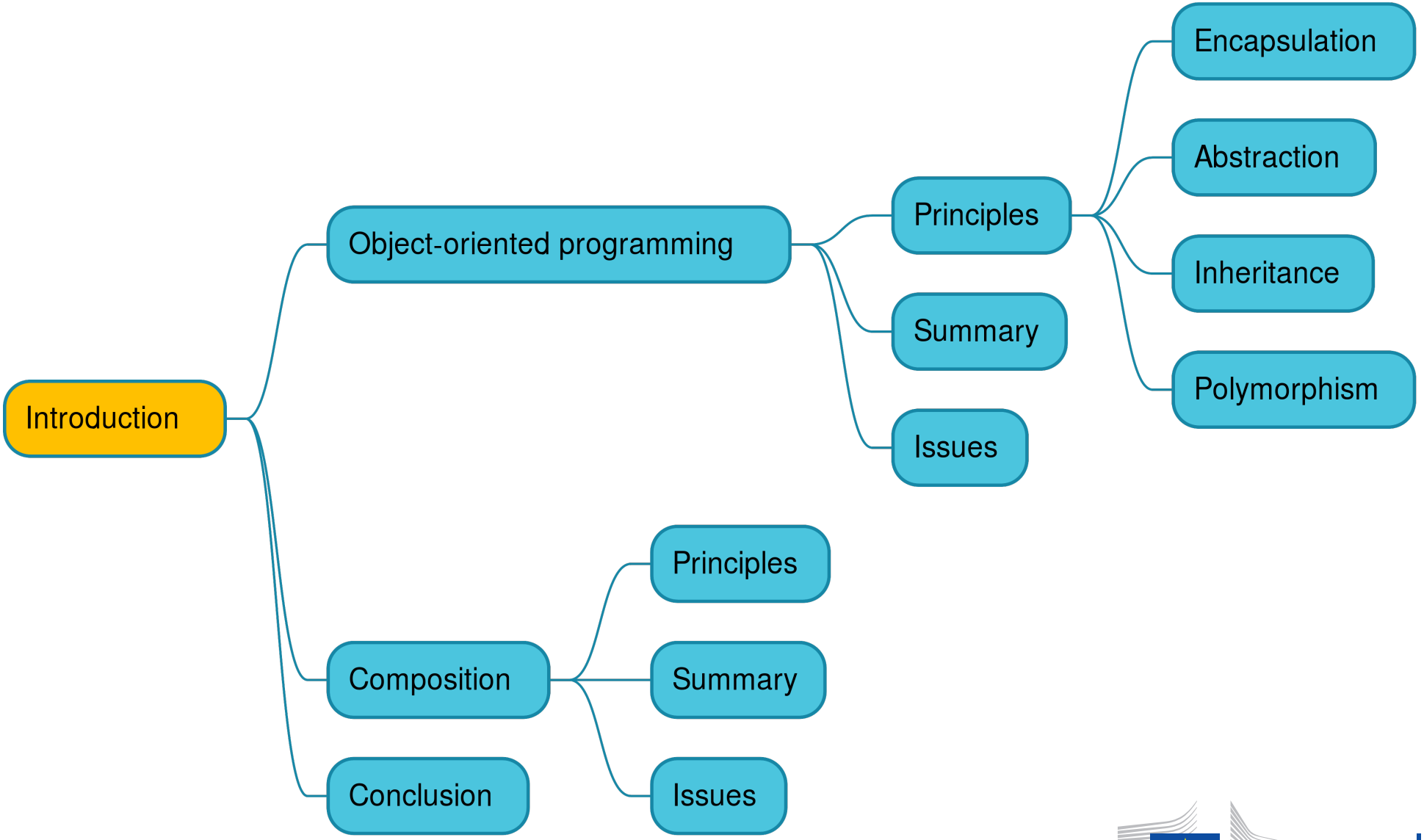
Commit #437e9bb

INTRODUCTION

IF YOU CAN'T EXPLAIN IT SIMPLY, YOU DON'T UNDERSTAND IT WELL ENOUGH.

Composition and inheritance

Map



Introduction

Inheritance and composition are two programming paradigms developers use to establish relationships between classes and objects.

Introduction

Whereas **inheritance** derives one class from another, **composition** defines a class as the sum of its parts.

Introduction

Classes and objects created through **inheritance** are tightly coupled because changing the parent or superclass in an inheritance relationship risks breaking your code.

Introduction

Classes and objects created through **inheritance** are tightly coupled because changing the parent or superclass in an inheritance relationship risks breaking your code.

Classes and objects created through **composition** are loosely coupled, meaning that you can more easily change the component parts without breaking your code.

Introduction

Because loosely coupled code offers more flexibility, many developers have learned that **composition** is a better technique than **inheritance**.

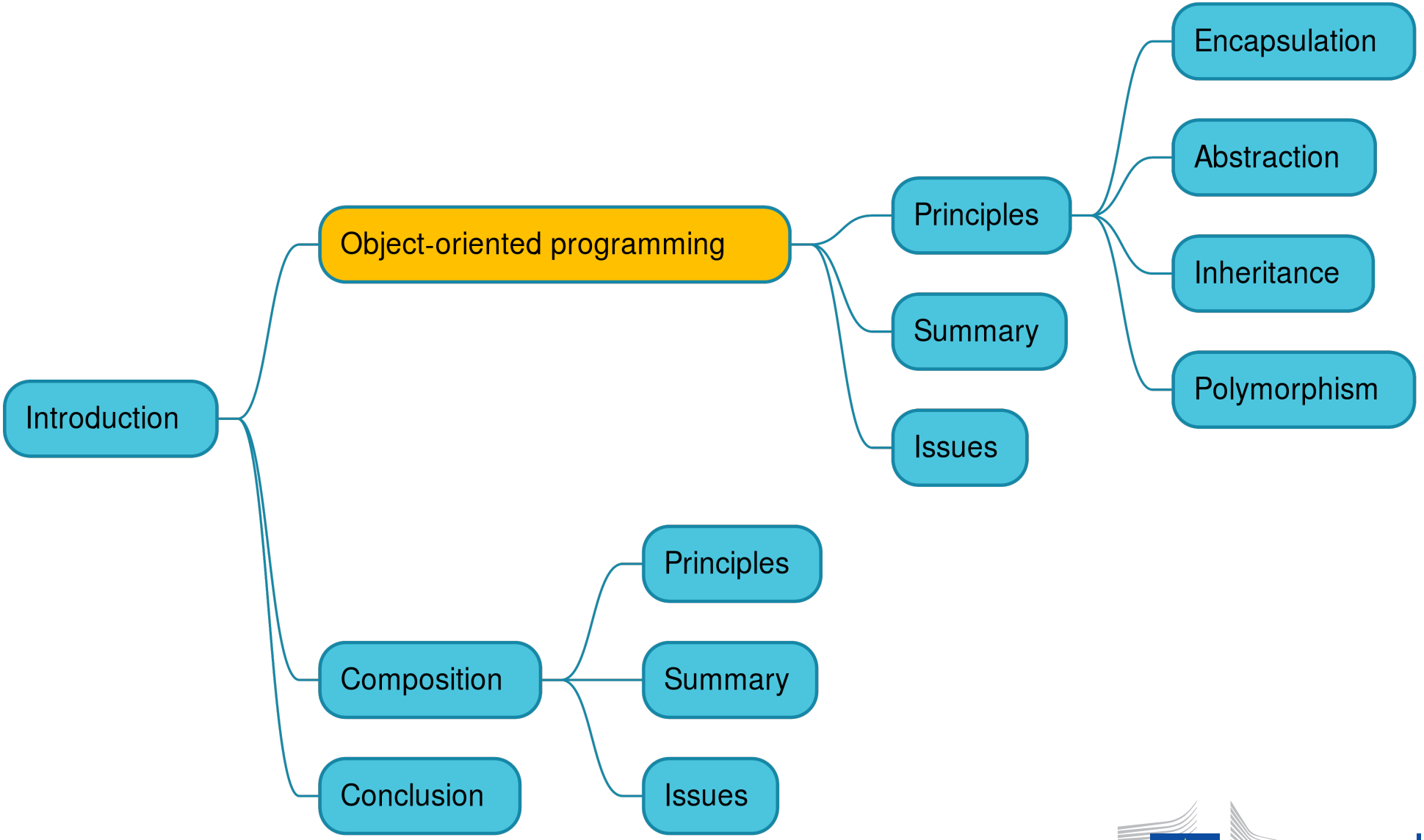
At which cost?

OOP

A QUICK REMINDER

Composition and inheritance

Map



__O__bject-__o__riented __p__rogramming (__OOP__) is more than just classes and objects.

***O**bject-**o**riented **p**rogramming (**OOP**) is more than just classes and objects.*

It's a whole programming paradigm based around objects that contain data fields and methods.

__O__bject-__o__riented __p__rogramming (__OOP__) is more than just classes and objects.

It's a whole programming paradigm based around objects that contain data fields and methods.

It is __essential__ to understand this.

__O__bject-__o__riented __p__rogramming (__OOP__) is more than just classes and objects.

It's a whole programming paradigm based around objects that contain data fields and methods.

It is __essential__ to understand this.

Using classes to organise a bunch of unrelated methods together is __not__ OOP per se.

Junade Ali, Mastering PHP Design Patterns
ISBN-13: 978-1785887130

What

What makes

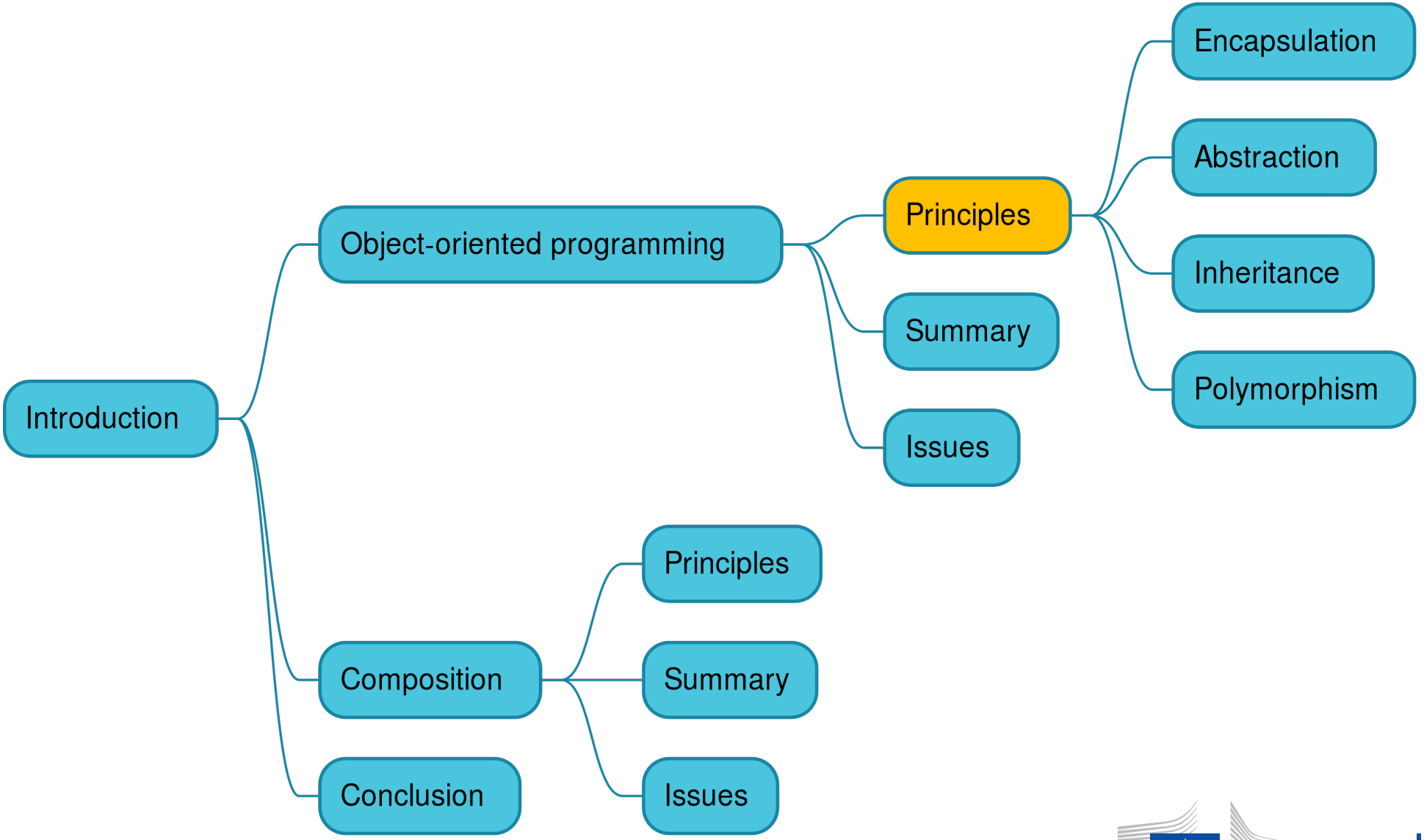
What makes code

What makes code **OOP** ?

The principles!

Composition and inheritance

Map



Object-oriented programming

OOP Principles

- Encapsulation

Object-oriented programming

OOP Principles

- Encapsulation
- Abstraction

Object-oriented programming

OOP Principles

- Encapsulation
- Abstraction
- Inheritance

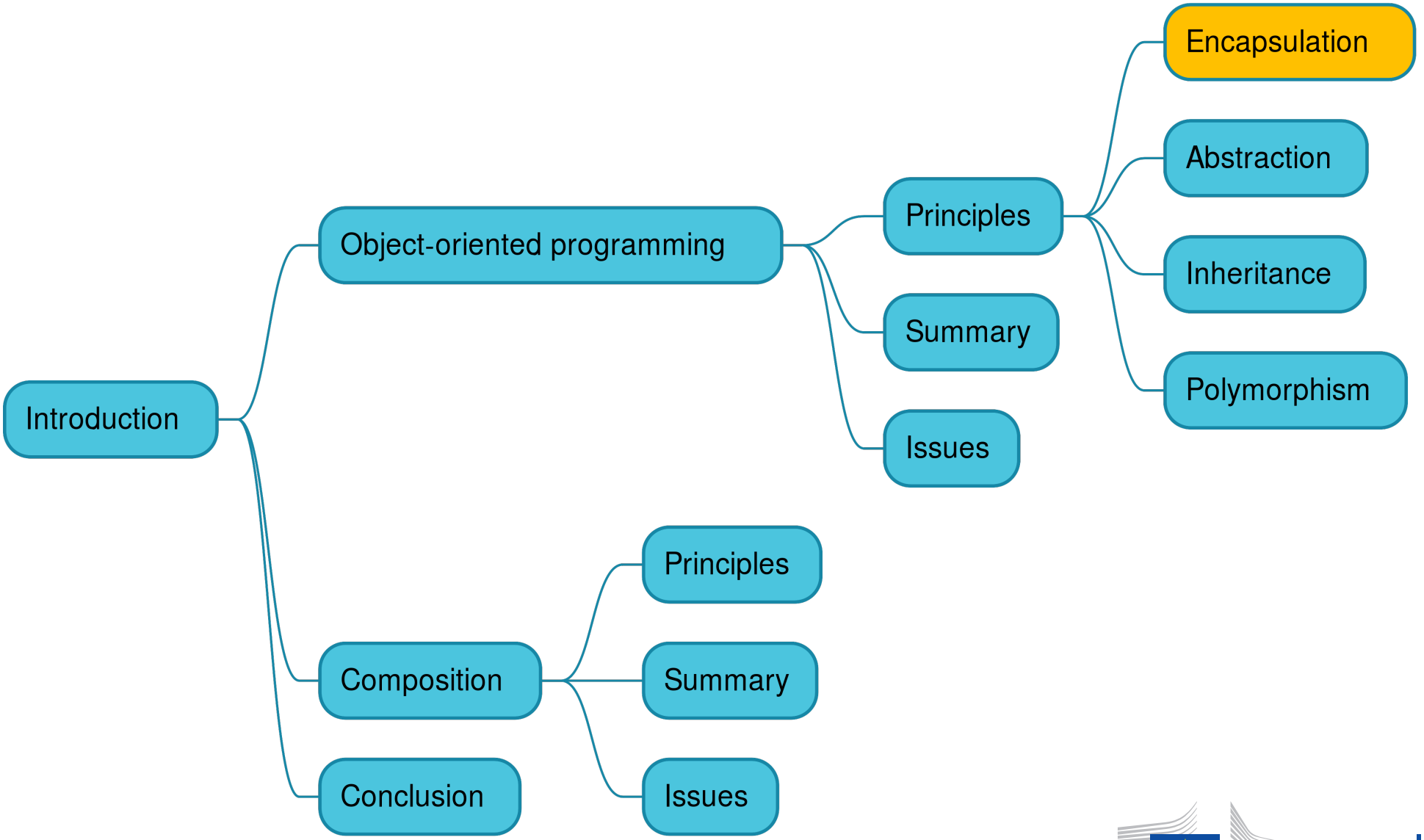
Object-oriented programming

OOP Principles

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Composition and inheritance

Map



OOP Principles

Encapsulation

Encapsulation is an object-oriented programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.

OOP Principles

Encapsulation

- Protect the internal state of an object access to the class properties

OOP Principles

Encapsulation

- Protect the internal state of an object access to the class properties
  Via the use of `getters` and `setters` methods.

OOP Principles

Encapsulation

- Protect the internal state of an object access to the class properties
  Via the use of `getters` and `setters` methods.
- Reduce software development complexity

OOP Principles

Encapsulation

- Protect the internal state of an object access to the class properties
  Via the use of `getters` and `setters` methods.
- Reduce software development complexity
  By hiding the implementation details and only exposing the relevant operations.

OOP Principles

Encapsulation

- Protect the internal state of an object access to the class properties
  Via the use of `getters` and `setters` methods.
- Reduce software development complexity
  By hiding the implementation details and only exposing the relevant operations.
- The internal implementation can be interchanged

OOP Principles

Encapsulation

- Protect the internal state of an object access to the class properties
  Via the use of `getters` and `setters` methods.
- Reduce software development complexity
  By hiding the implementation details and only exposing the relevant operations.
- The internal implementation can be interchanged
  Without worrying about breaking the code using the class.

OOP Principles

Encapsulation / Getters & setters

```php
<?php

class User
{
    public function __construct(
        private string $name,
        private string $email,
    ) {}
}

```

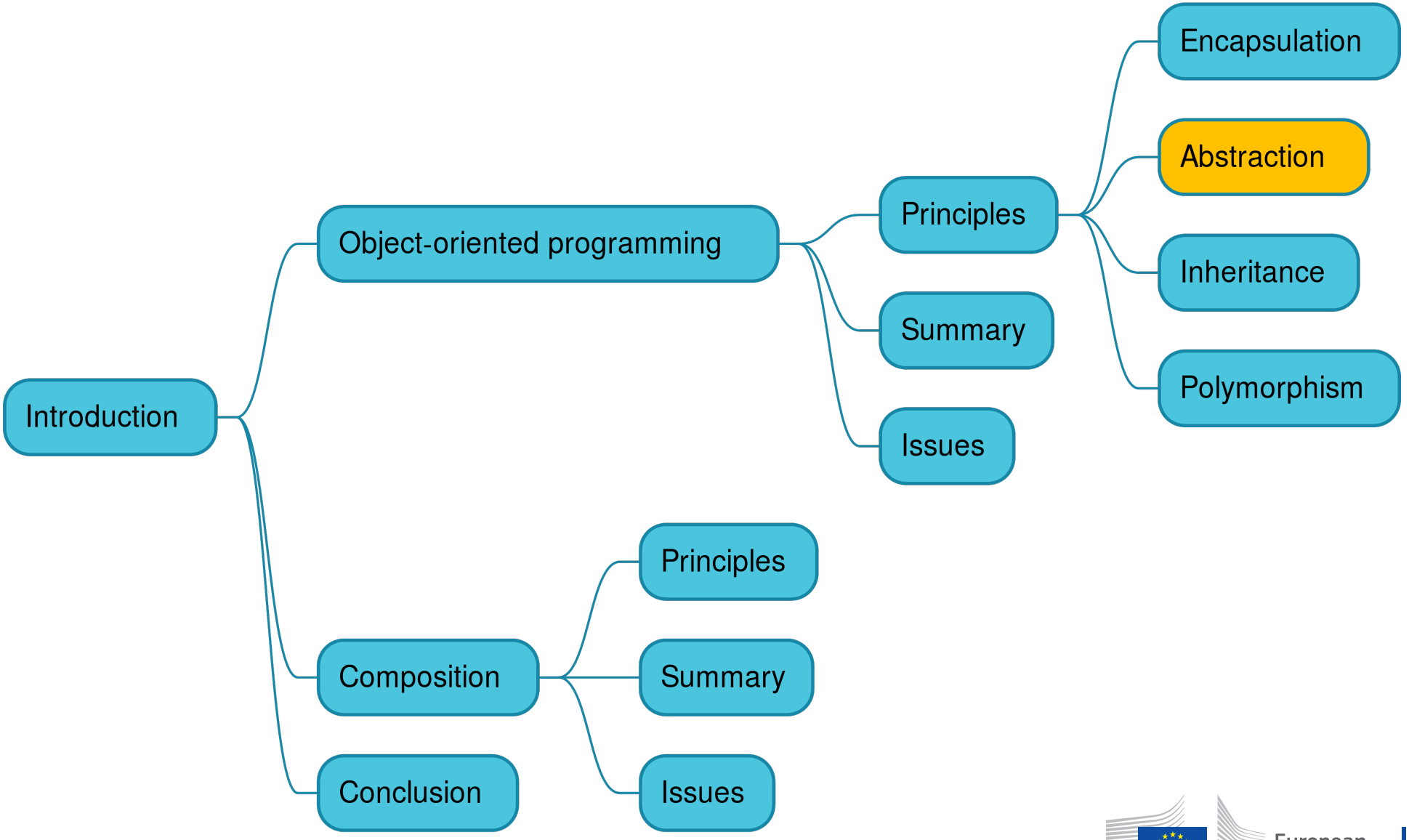
OOP Principles

Encapsulation / Getters & setters

```php
<?php

class User
{
    public function __construct(
        private string $name,
        private string $email,
    ) {}

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

    public function setEmail(string $email): void
    {
        $this->email = $email;
    }
}

```

Composition and inheritance

Map



OOP Principles

Abstraction

- It’s easier to design a program when separating the interface from its implementation

OOP Principles

Abstraction

- It's easier to design a program when separating the interface from its implementation
  And focus on the interface only.

OOP Principles

Abstraction

- It's easier to design a program when separating the interface from its implementation
  And focus on the interface only.
- This is like treating a system as a *black box*

OOP Principles

Abstraction

- It's easier to design a program when separating the interface from its implementation
  And focus on the interface only.
- This is like treating a system as a *black box*
  Where it's not important to understand the inner mechanisms in order to reap the benefits of using it.

OOP Principles

Abstraction

- This process is called *abstraction*

OOP Principles

Abstraction

- This process is called *abstraction*
  Because we are abstracting away the implementation detail and only presenting a clean and *easy-to-use* interface via the class methods.

OOP Principles

Abstraction

- This process is called *abstraction*
  Because we are abstracting away the implementation detail and only presenting a clean and *easy-to-use* interface via the class methods.
- *Abstraction* helps isolate the impact of changes made to the code

OOP Principles

Abstraction

- This process is called *abstraction*
  Because we are abstracting away the implementation detail and only presenting a clean and *easy-to-use* interface via the class methods.
- *Abstraction* helps isolate the impact of changes made to the code
  So that if something goes wrong, the change will only affect the implementation details and not the outside code.

OOP Principles

Abstraction / Using interfaces

```php
<?php

$user = new User('patrick', 'user@example.com');

function printPrettyUsername(User $user): string
{
    return ucfirst($user->getName());
}

printUsername($user); // Patrick

```

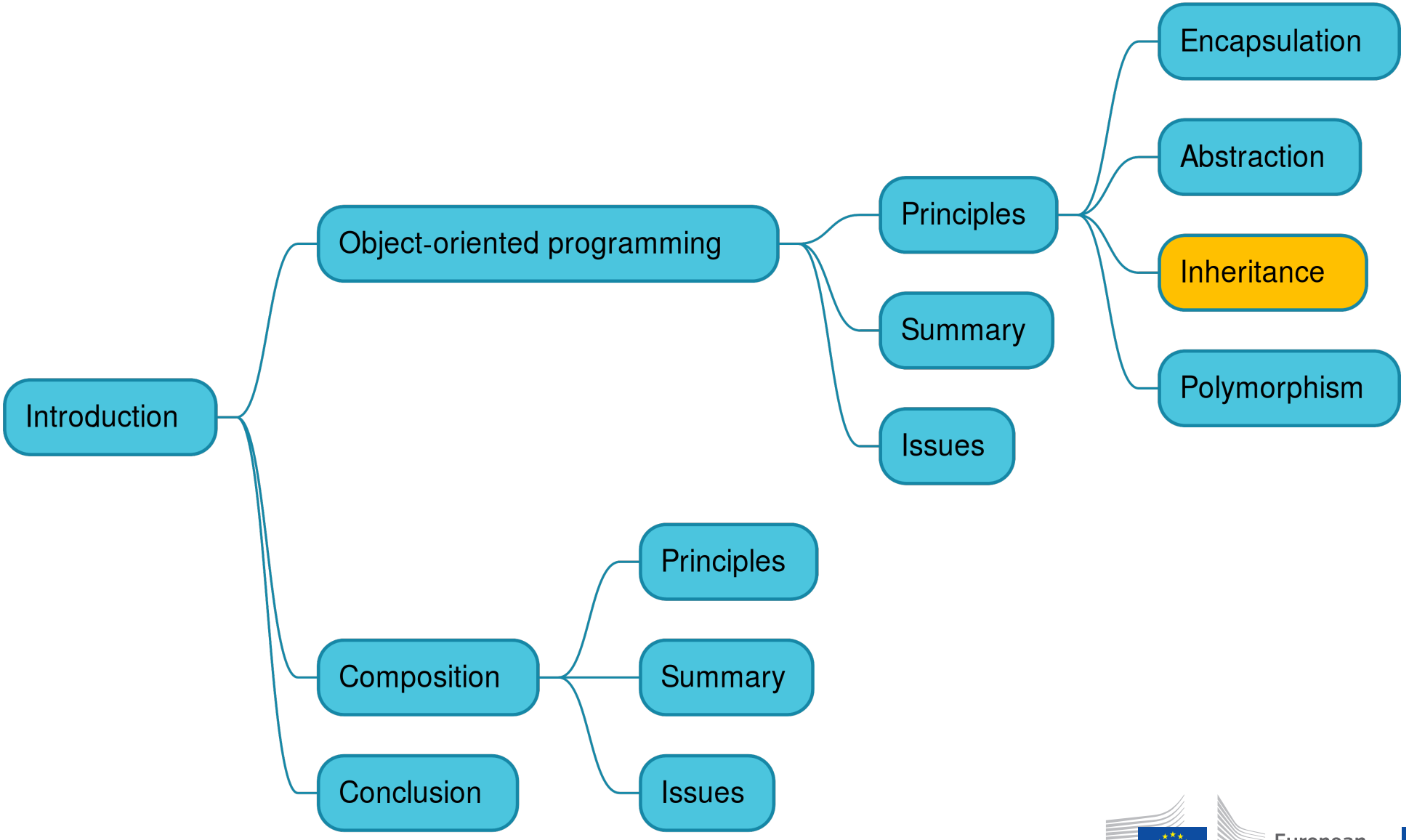
OOP Principles

Abstraction / Using interfaces

```php
<?php

interface UserInterface {
    public function getName(): string;
    public function setEmail(string $email): void;
}

class User implements UserInterface
{
    /* Same content as before */
}

$user = new User('patrick', 'user@example.com');

function printPrettyUsername(UserInterface $user): string
{
    return ucfirst($user->getName());
}

printUsername($user); // Patrick

```

Composition and inheritance

Map



OOP Principles

Inheritance

- It is fundamental to OOP

OOP Principles

Inheritance

- It is fundamental to OOP
  A programming language may have objects and messages, but without inheritance it's not object-oriented.

OOP Principles

Inheritance

- It is fundamental to OOP
  A programming language may have objects and messages, but without inheritance it's not object-oriented.
- Inheritance serves two purposes: semantics and mechanics.

OOP Principles

Inheritance

- It is fundamental to OOP
  A programming language may have objects and messages, but without inheritance it's not object-oriented.
- Inheritance serves two purposes: semantics and mechanics.
- Inheritance will affect the way many classes and objects relate to each other.

OOP Principles

Inheritance

- It is fundamental to OOP
  A programming language may have objects and messages, but without inheritance it's not object-oriented.
- Inheritance serves two purposes: semantics and mechanics.
- Inheritance will affect the way many classes and objects relate to each other.
- Inheritance is achieved by *extending* another class

OOP Principles

Inheritance

- When *extending* a class, the subclass inherits methods, properties and constants from the parent class.

OOP Principles

Inheritance

- When *extending* a class, the subclass inherits methods, properties and constants from the parent class.
- It permits the implementation of additional functionality in similar objects without the need to reimplement all of the shared functionality. (*DRY*)

OOP Principles

Inheritance

- When *extending* a class, the subclass inherits methods, properties and constants from the parent class.
- It permits the implementation of additional functionality in similar objects without the need to reimplement all of the shared functionality. (*DRY*)
- The visibility of methods, properties and constants can be relaxed.

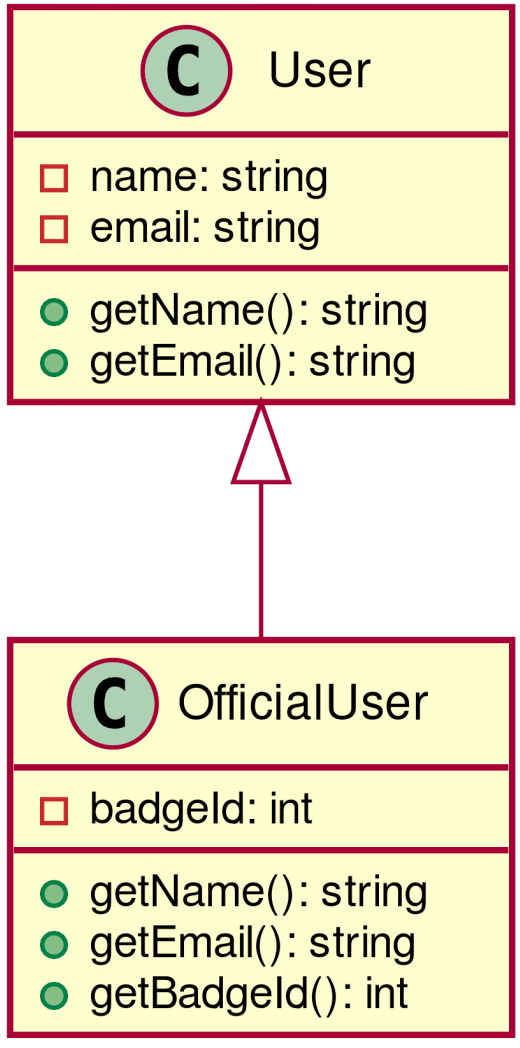
OOP Principles

Inheritance / Example

```php
<?php

class OfficialUser extends User
{
    public function getName(): string
    {
        return sprintf('Mr %s', parent::getName());
    }

    public function getBadgeId(): int
    {
        return array_reduce(
            str_split($this->getName()),
            static fn (int $s, string $letter): int => $s + ord($letter),
            0
        );
    }
}

$user = new OfficialUser('patrick', 'user@example.com');

echo $user->getName(); // Mr patrick
echo $user->getBadgeId(); // 973

```

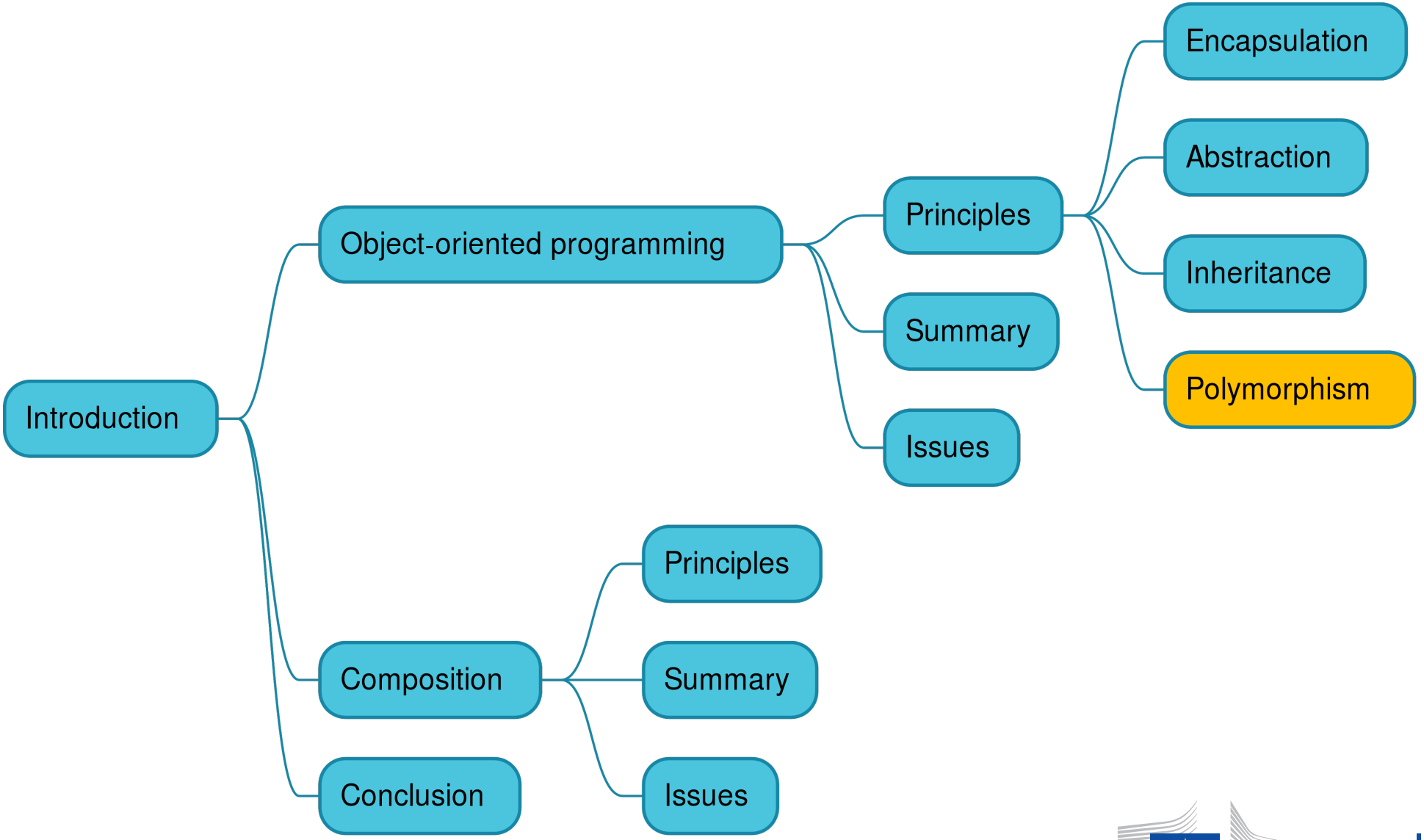
OOP Principles

Inheritance / UML representation



Composition and inheritance

Map



OOP Principles

Polymorphism

- Polymorphism is a Greek word that literally means "many forms".

OOP Principles

Polymorphism

- Polymorphism is a Greek word that literally means "many forms".
- Polymorphism is closely related to inheritance.

OOP Principles

Polymorphism

- Polymorphism is a Greek word that literally means "many forms".
- Polymorphism is closely related to inheritance.
- Polymorphism allows objects of different classes to respond differently based on the same message.

OOP Principles

Polymorphism

- Polymorphism is a Greek word that literally means "many forms".
- Polymorphism is closely related to inheritance.
- Polymorphism allows objects of different classes to respond differently based on the same message.
- To implement polymorphism in PHP, you can use either `abstract` classes or `interfaces`.

OOP Principles

Polymorphism / With an abstract class

```php
<?php

abstract class Person {
    abstract public function greet(): string;
}

class Dutch extends Person
{
    public function greet(): string { return 'Hallo!'; }
}

class English extends Person
{
    public function greet(): string { return 'Hello!'; }
}

$dutch = new Dutch;
$english = new English;

var_dump($dutch instanceof Person);   // true
var_dump($english instanceof Person); // true

```

OOP Principles

Polymorphism / With an abstract class

```php
<?php

abstract class Person {
    abstract public function greet(): string;
}

class Dutch extends Person
{
    public function greet(): string { return 'Hallo!'; }
}

class English extends Person
{
    public function greet(): string { return 'Hello!'; }
}

$greetings = array_map(
    static fn (Person $person): string => $person->greet(),
    [new Dutch, new English]
); // ['Hallo!', 'Hello!']

```

OOP Principles

Polymorphism / With an interface

```php
<?php

interface Greatable
{
    public function greet(): string;
}

class Dutch implements Greatable
{
    public function greet(): string { return 'Hallo!'; }
}

class English implements Greatable
{
    public function greet(): string { return 'Hello!'; }
}

$dutch = new Dutch;
$english = new English;

var_dump($dutch instanceof Greatable);   // true
var_dump($english instanceof Greatable); // true

```

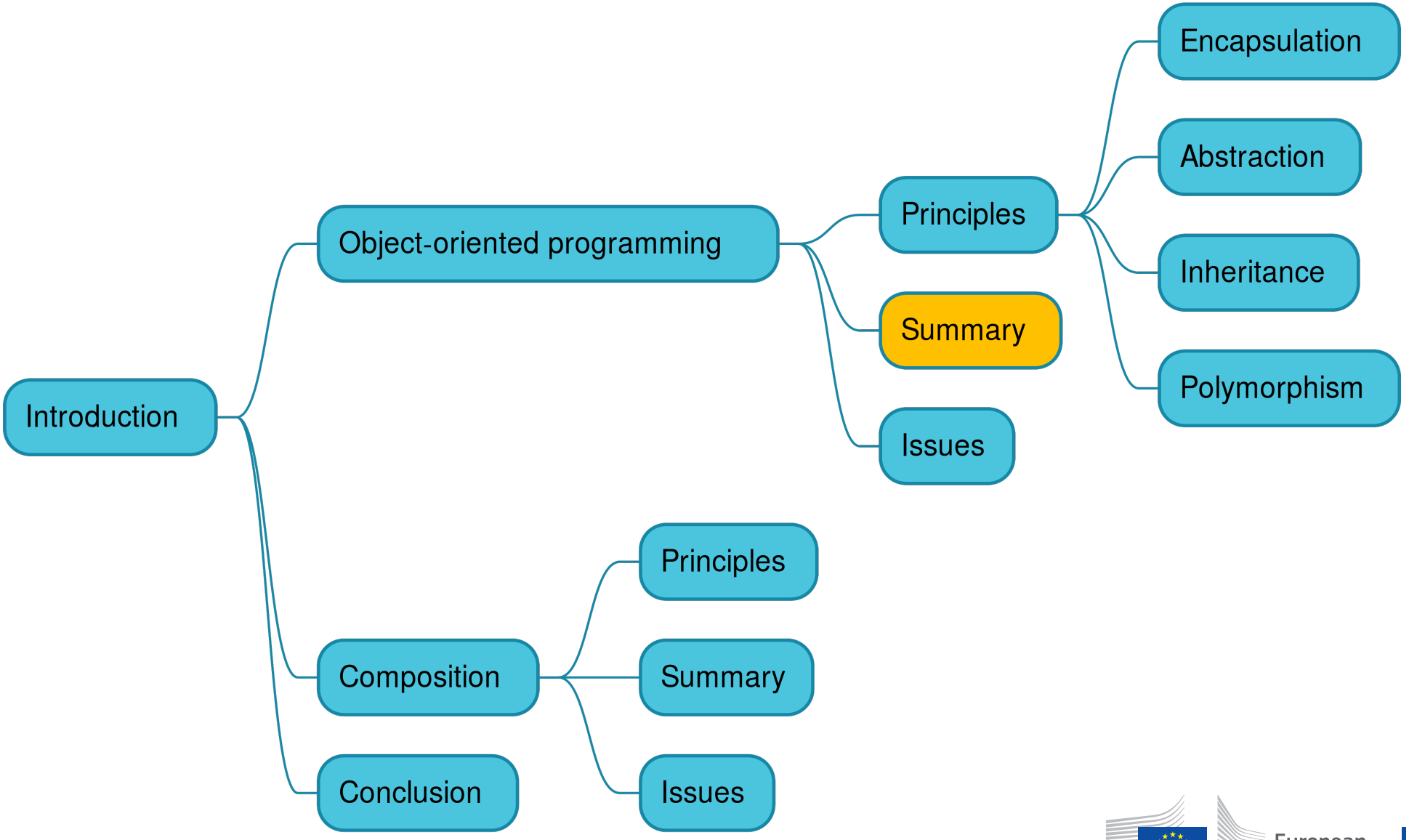
OOP Principles

Polymorphism / With an interface

```php
<?php

interface Greatable
{
    public function greet(): string;
}

class Dutch implements Greatable
{
    public function greet(): string { return 'Hallo!'; }
}

class English implements Greatable
{
    public function greet(): string { return 'Hello!'; }
}

$greetings = array_map(
    static fn (Greatable $user): string => $user->greet(),
    [new Dutch, new English]
); // ['Hallo!', 'Hello!']

```

Composition and inheritance

Map



OOP Principles

At a glance

- Encapsulation

OOP Principles

At a glance

- Encapsulation: `public`, `protected`, `private`

OOP Principles

At a glance

- Encapsulation: `public`, `protected`, `private`
- Abstraction

OOP Principles

At a glance

- Encapsulation: `public`, `protected`, `private`
- Abstraction: `interface`

OOP Principles

At a glance

- Encapsulation: `public`, `protected`, `private`
- Abstraction: `interface`
- Inheritance

OOP Principles

At a glance

- Encapsulation: `public`, `protected`, `private`
- Abstraction: `interface`
- Inheritance: `extends`

OOP Principles

At a glance

- Encapsulation: `public`, `protected`, `private`
- Abstraction: `interface`
- Inheritance: `extends`
- Polymorphism

OOP Principles

At a glance

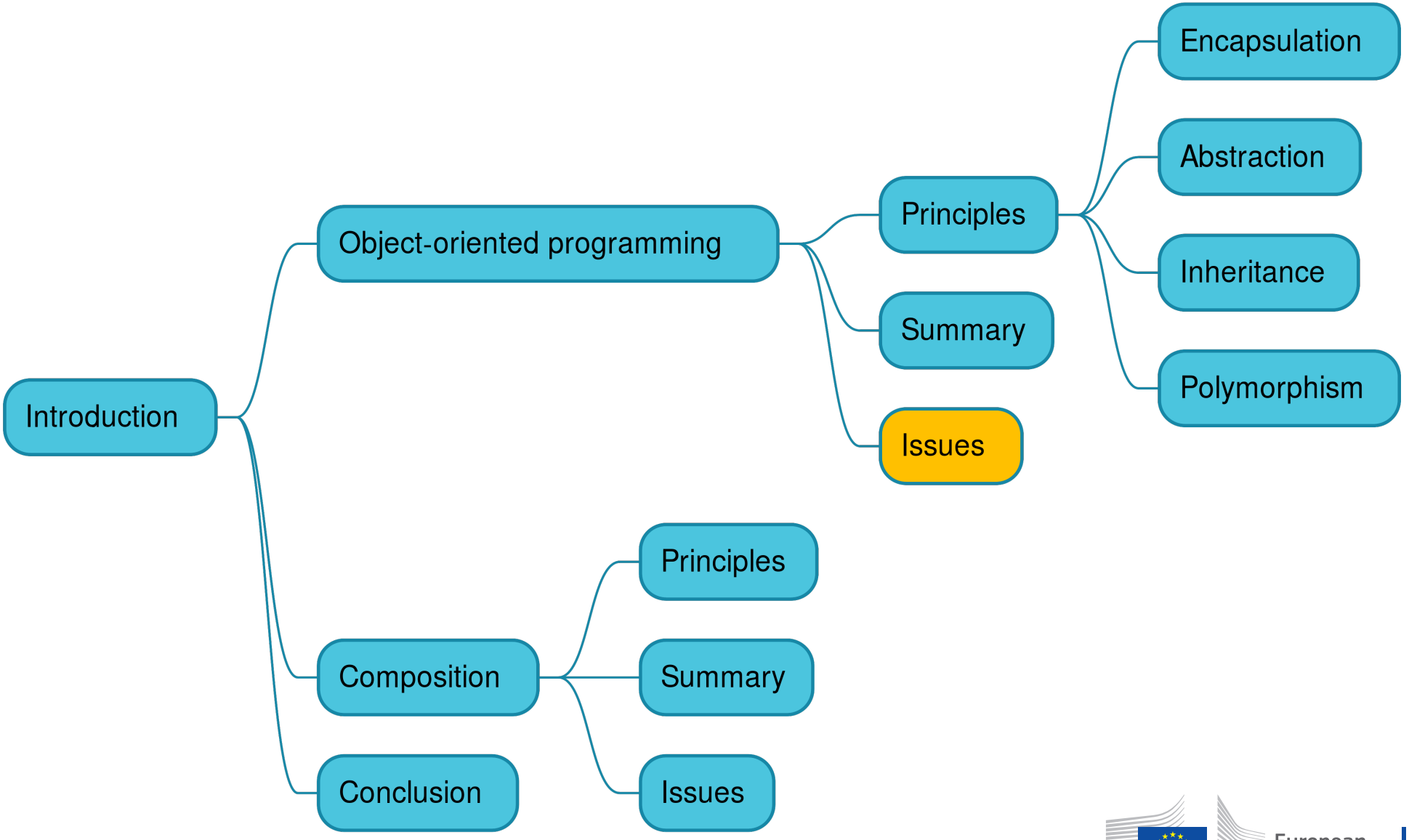
- Encapsulation: `public`, `protected`, `private`
- Abstraction: `interface`
- Inheritance: `extends`
- Polymorphism: `abstract`

One day, I’m going to live in *theory*

One day, I’m going to live in *theory* because in *theory* everything goes perfectly.

Composition and inheritance

Map



- The *Yo-Yo* problem

- The *Yo-Yo* problem
- It breaks encapsulation

OOP

Issues

- The *Yo-Yo* problem
- It breaks encapsulation
- The inheritance of unnecessary methods

OOP

Issues

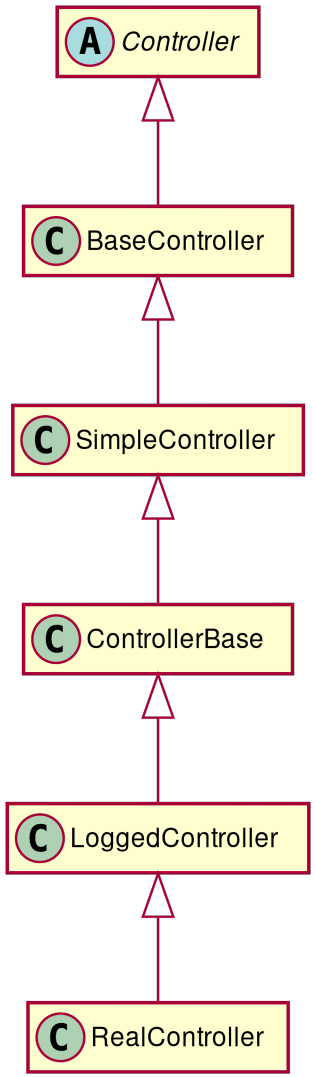
- The *Yo-Yo* problem
- It breaks encapsulation
- The inheritance of unnecessary methods
- Lack of flexibility

- The *Yo-Yo* problem
- It breaks encapsulation
- The inheritance of unnecessary methods
- Lack of flexibility
- Creates an `Is-a` relation

Anti-pattern that occurs when a programmer has to read and understand a program whose inheritance graph is so long and complicated that the programmer has to keep flipping between many different class definitions in order to follow the control flow of the program.

OOP

Issues / The *Yo-Yo* problem




```php
<?php

abstract class Controller {}

class BaseController extends Controller {}

class SimpleController extends BaseController {}

class ControllerBase extends SimpleController {}

class LoggedController extends ControllerBase {}

class RealController extends LoggedController {}
```

- Breaks encapsulation

- Breaks encapsulation
  Inheritance creates dependency between child and parent.

- Breaks encapsulation

  Inheritance creates dependency between child and parent.

  When a class inherit another class, we include all methods and attributes from parent class and expose to the child class, therefore we break the encapsulation.

- Breaks encapsulation

  Inheritance creates dependency between child and parent.

  When a class inherit another class, we include all methods and attributes from parent class and expose to the child class, therefore we break the encapsulation.

  The child object can access all the methods in parent object and overwrite them.

- Breaks encapsulation

  Inheritance creates dependency between child and parent.

  When a class inherit another class, we include all methods and attributes from parent class and expose to the child class, therefore we break the encapsulation.

  The child object can access all the methods in parent object and overwrite them.

  That creates a tightly coupled relation between child and parent class, also against the idea of OOP, which is to hide the complexity in the object and interact by interface.

OOP

Issues / Breaks encapsulation

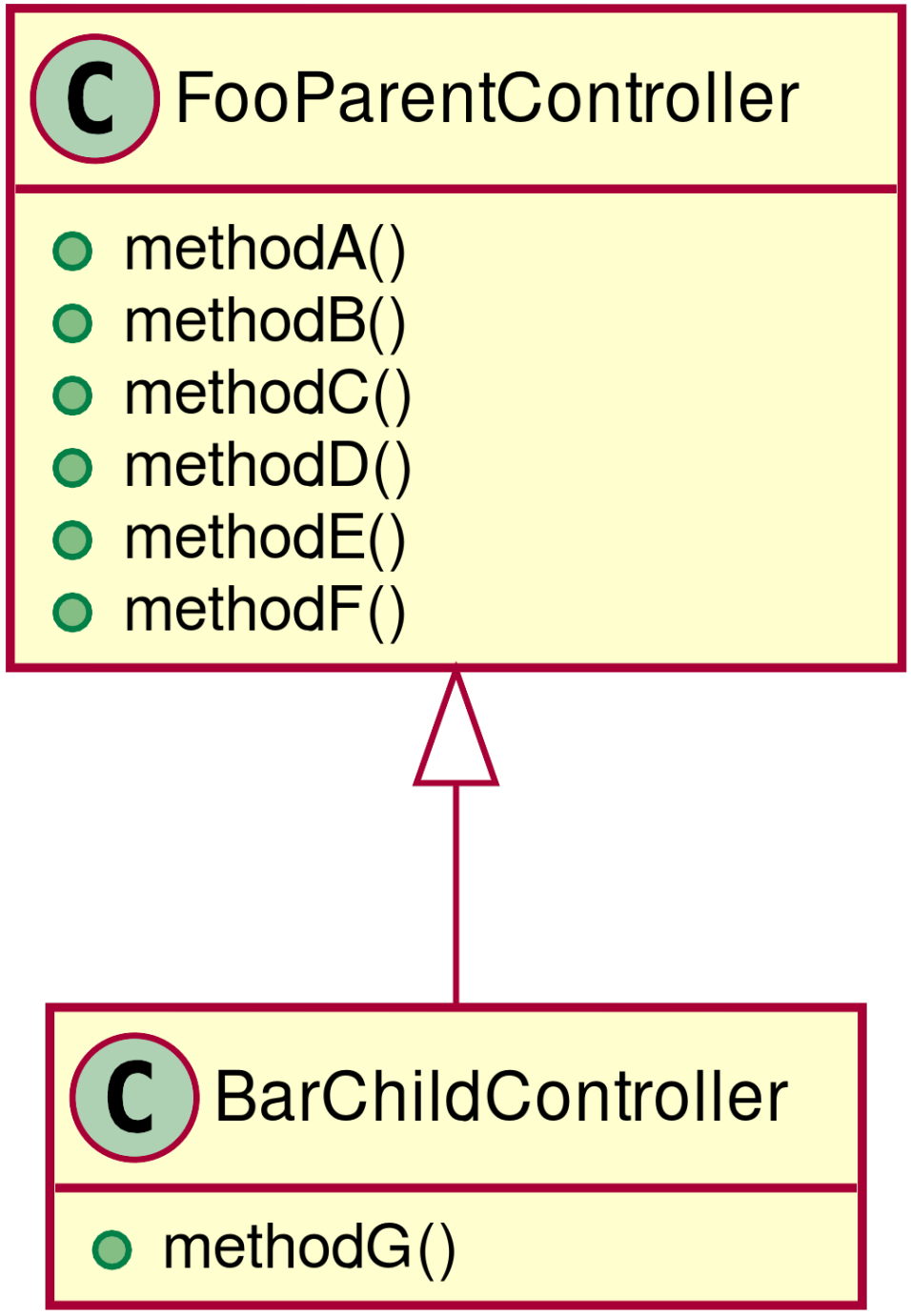
- Inheritance of unnecessary methods

OOP

Issues / Breaks encapsulation

- Inheritance of unnecessary methods

  Inheritance makes a child class inherit all the methods and properties from parent class, even if it is not used or not needed, that creates more complexity than the child class needs to.



OOP
Issues

- Flexibility

- Flexibility
  A class can only inherit from one parent class.

- Flexibility
  A class can only inherit from one parent class.
- `Is-a` relation

- Flexibility
  A class can only inherit from one parent class.
- `Is-a` relation
  By extending a class, we enforce an `is-a` relationship between parent and child, which sometimes does not reflect the object's real relationships.

OOP

Issues / is-a relation

```php
/**
 * Last-In-First-Out (LIFO) stack.
 */
class Stack extends ArrayObject {
    /**
     * Push a value in the stack.
     */
    public function push(mixed $value): void
    {
        $this->append($value);
    }

    /**
     * Get and remove current stack value.
     */
    public function pop(): mixed
    {
        $arrayCopy = $this->getArrayCopy();
        $result = array_pop($arrayCopy);
        $this->exchangeArray($arrayCopy);

        return $result;
    }
}

```

OOP

Issues / `is-a` relation

- Semantically, the statement "a `Stack is a ArrayObject`" is not true.

- Semantically, the statement "a `Stack is a ArrayObject`" is not true.

  `Stack` is not a proper subtype of `ArrayObject`. A stack is supposed to enforce *Last-In-First-Out (LIFO)*, a constraint easily satisfied by the `Stack::push()` and `Stack::pop()` methods, but not enforced by the `ArrayObject`'s methods.

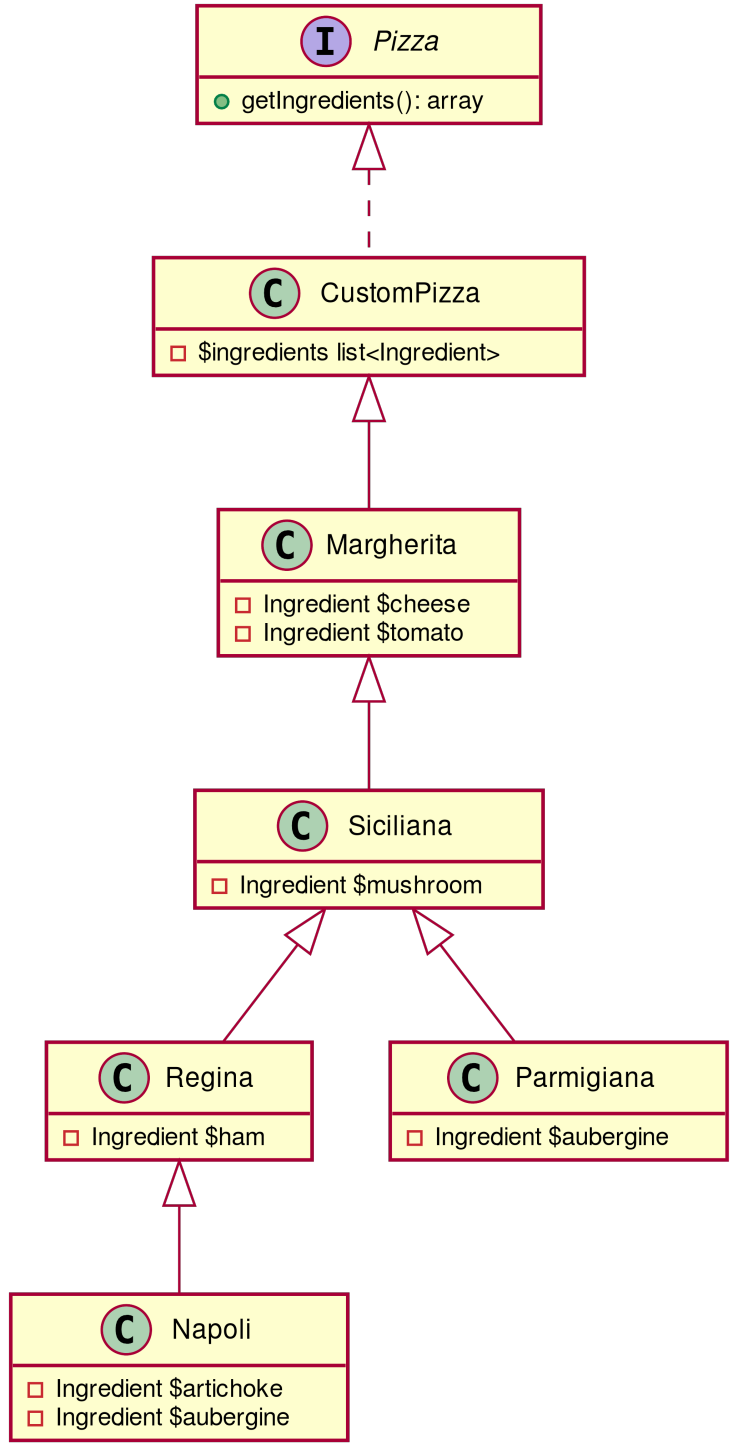
- Semantically, the statement "a `Stack is a ArrayObject`" is not true.
  `Stack` is not a proper subtype of `ArrayObject`. A stack is supposed to enforce *Last-In-First-Out (LIFO)*, a constraint easily satisfied by the `Stack::push()` and `Stack::pop()` methods, but not enforced by the `ArrayObject`'s methods.
- Mechanically, inheriting from `ArrayObject` violates encapsulation.

- Semantically, the statement "a `Stack is a ArrayObject`" is not true.
  `Stack` is not a proper subtype of `ArrayObject`. A stack is supposed to enforce *Last-In-First-Out (LIFO)*, a constraint easily satisfied by the `Stack::push()` and `Stack::pop()` methods, but not enforced by the `ArrayObject`'s methods.
- Mechanically, inheriting from `ArrayObject` violates encapsulation.
  Using `ArrayObject` is an implementation choice that should be hidden from consumers.

Issues / `is-a` relation

- Semantically, the statement "a `Stack is a ArrayObject`" is not true.
  `Stack` is not a proper subtype of `ArrayObject`. A stack is supposed to enforce *Last-In-First-Out (LIFO)*, a constraint easily satisfied by the `Stack::push()` and `Stack::pop()` methods, but not enforced by the `ArrayObject`'s methods.
- Mechanically, inheriting from `ArrayObject` violates encapsulation.
  Using `ArrayObject` is an implementation choice that should be hidden from consumers.
- Finally, implementing a stack by inheriting from `ArrayObject` is a cross-domain relationship.

- Semantically, the statement "a `Stack is a ArrayObject`" is not true.
  `Stack` is not a proper subtype of `ArrayObject`. A stack is supposed to enforce *Last-In-First-Out (LIFO)*, a constraint easily satisfied by the `Stack::push()` and `Stack::pop()` methods, but not enforced by the `ArrayObject`'s methods.
- Mechanically, inheriting from `ArrayObject` violates encapsulation.
  Using `ArrayObject` is an implementation choice that should be hidden from consumers.
- Finally, implementing a stack by inheriting from `ArrayObject` is a cross-domain relationship.
  `ArrayObject` is a randomly-accessible collection while a stack is a queuing concept.



```php
<?php

interface Ingredient {}

interface Pizza {
    public function getIngredients(): array;
}

```

```php
<?php

final class CustomPizza implements Pizza
{
    public function __construct(
        Ingredient ...$ingredients
    ) {}

    public function getIngredients(): array
    {
        return $this->ingredients;
    }
}

```

```php
<?php

class Margherita extends CustomPizza
{
    public function __construct(
        Ingredient $tomato,
        Ingredient $cheese
    ) {
        parent::__construct($tomato, $cheese);
    }

    public function getIngredients(): array
    {
        return parent::getIngredients();
    }
}

```

```php
<?php

class Siciliana extends Margherita
{
    public function __construct(
        Ingredient $tomato,
        Ingredient $cheese,
        private Ingredient $mushroom
    ) {
        parent::__construct($tomato, $cheese);
    }

    public function getIngredients(): array
    {
        return [...parent::getIngredients(), $this->mushroom];
    }
}

```

```php
<?php

class Regina extends Siciliana
{
    public function __construct(
        Ingredient $tomato,
        Ingredient $cheese,
        Ingredient $mushroom,
        private Ingredient $ham
    ) {
        parent::__construct($tomato, $cheese, $mushroom);
    }

    public function getIngredients(): array
    {
        return [...parent::getIngredients(), $this->ham];
    }
}

```

```php
<?php

class Parmigiana extends Siciliana
{
    public function __construct(
        Ingredient $tomato,
        Ingredient $cheese,
        Ingredient $mushroom,
        private Ingredient $aubergine
    ) {
        parent::__construct($tomato, $cheese, $mushroom);
    }

    public function getIngredients(): array
    {
        return [...parent::getIngredients(), $this->aubergine];
    }
}

```

```php
<?php

class Napoli extends Regina
{
    public function __construct(
        Ingredient $tomato,
        Ingredient $cheese,
        Ingredient $mushroom,
        Ingredient $ham,
        private Ingredient $aubergine,
        private Ingredient $artichoke
    ) {
        parent::__construct($tomato, $cheese, $mushroom, $ham);
    }

    public function getIngredients(): array
    {
        return [...parent::getIngredients(), $this->aubergine, $this->artichoke];
    }
}

```

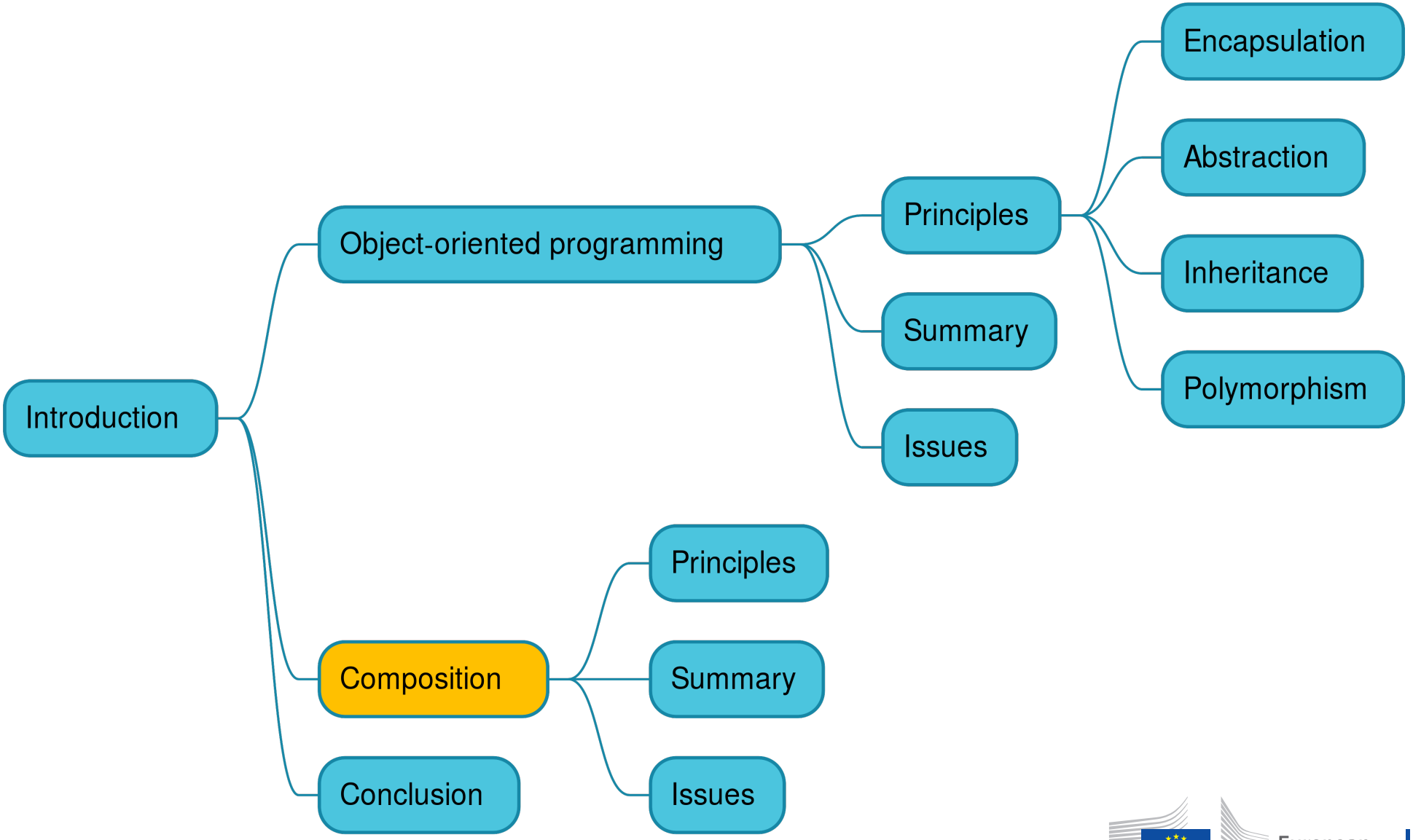
```php
<?php

var_dump($margherita instanceof Pizza);       // true
var_dump($margherita instanceof $siciliana);  // true
var_dump($margherita instanceof $regina);     // true
var_dump($margherita instanceof $parmigiana); // true
var_dump($margherita instanceof $napoli);     // true

```

COMPOSITION

MOVING FROM IS-A TO HAS-A

Composition and inheritance

Map



Object-oriented programming with composition

Principles

- Encapsulation

Object-oriented programming with composition

Principles

- Encapsulation
- Abstraction

Object-oriented programming with composition

Principles

- Encapsulation
- Abstraction
- Inheritance

Object-oriented programming with composition

Principles

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Object-oriented programming with composition

Principles

- Encapsulation
- Abstraction
- ~~Inheritance~~ **Composition**
- Polymorphism

Object-oriented programming with composition

The good things

- We use it every day,

Object-oriented programming with composition

The good things

- We use it every day,
- Natural and very simple to use.

```php
<?php

interface Ingredient {}

interface Pizza {
    public function getIngredients(): array;
}

```

```php
<?php

final class Margherita implements Pizza
{
    public function __construct(
        private Ingredient $tomato,
        private Ingredient $cheese
    ) {}

    public function getIngredients(): array
    {
        return [$this->tomato, $this->cheese];
    }
}

```

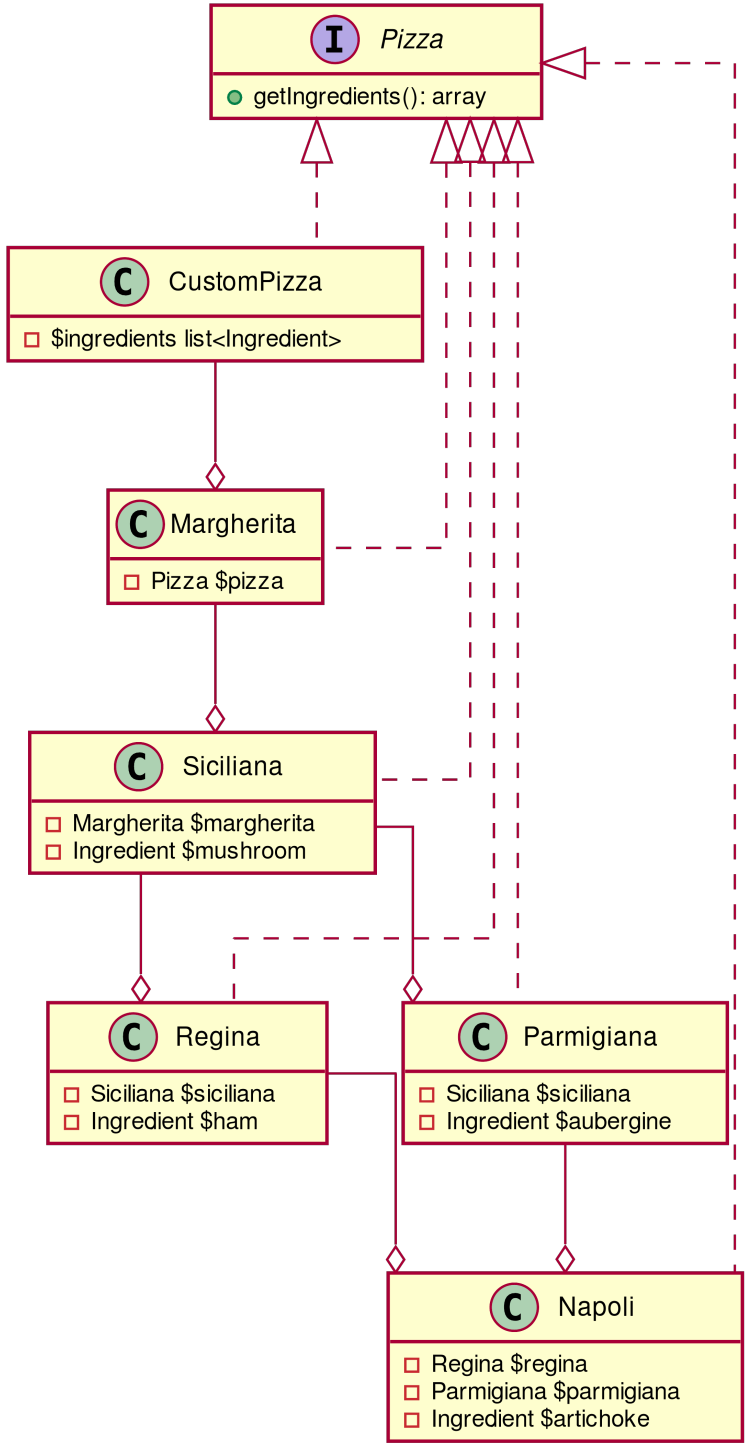
```php
<?php

final class Siciliana implements Pizza
{
    public function __construct(
        private Ingredient $tomato,
        private Ingredient $cheese
        private Ingredient $mushroom
    ) {}

    public function getIngredients(): array
    {
        return [$this->tomato, $this->cheese, $this->mushroom];
    }
}

```

```php
<?php

final class Regina implements Pizza
{
    public function __construct(
        private Ingredient $tomato,
        private Ingredient $cheese,
        private Ingredient $mushroom,
        private Ingredient $ham
    ) {}

    public function getIngredients(): array
    {
        return [$this->tomato, $this->cheese, $this->mushroom, $this->ham];
    }
}

```

```php
<?php

final class Parmigiana implements Pizza
{
    public function __construct(
        private Ingredient $tomato,
        private Ingredient $cheese,
        private Ingredient $mushroom,
        private Ingredient $aubergine
    ) {}

    public function getIngredients(): array
    {
        return [$this->tomato, $this->cheese, $this->mushroom, $this->aubergine];
    }
}

```

```php
<?php

final class Napoli implements Pizza
{
    public function __construct(
        private Ingredient $tomato,
        private Ingredient $cheese,
        private Ingredient $mushroom,
        private Ingredient $ham,
        private Ingredient $artichoke,
        private Ingredient $aubergine
    ) {}

    public function getIngredients(): array
    {
        return [
            $this->tomato,
            $this->cheese,
            $this->mushroom,
            $this->ham,
            $this->artichoke,
            $this->aubergine
        ];
    }
}

```

```php
<?php

$tomato = new Tomato;
$cheese = new Cheese;
$mushroom = new Mushroom;
$ham = new Ham;
$aubergine = new Aubergine;
$artichoke = new Artichoke;

$marguerita = new Margherita($tomato, $cheese);

$siciliana = new Siciliana($tomato, $cheese, $muschroom);

$regina = new Regina($tomato, $cheese, $mushroom, $ham);

$parmigiana = new Parmigiana($tomato, $cheese, $mushroom, $aubergine);

$napoli = new Napoli($tomato, $cheese, $mushroom, $ham, $aubergine, $artichoke);

```



```php
<?php

interface Ingredient {}

interface Pizza {
    public function getIngredients(): array;
}

```

```php
<?php

final class CustomPizza implements Pizza
{
    public function __construct(
        private Ingredient ...$ingredients
    ) {}

    public function getIngredients(): array
    {
        return $this->ingredients;
    }
}

```

```php
<?php

final class Margherita implements Pizza
{
    public function __construct(
        private Pizza $customPizza
    ) {}

    public function getIngredients(): array
    {
        return $this->customPizza->getIngredients();
    }
}

```

```php
<?php

final class Siciliana implements Pizza
{
    public function __construct(
        private Marguerita $margherita,
        private Ingredient $mushroom
    ) {}

    public function getIngredients(): array
    {
        return [...$this->margherita->getIngredients(), $this->mushroom];
    }
}

```

```php
<?php

final class Regina implements Pizza
{
    public function __construct(
        private Siciliana $siciliana,
        private Ingredient $ham
    ) {}

    public function getIngredients(): array
    {
        return [...$this->siciliana->getIngredients(), $this->ham];
    }
}

```

```php
<?php

final class Parmigiana implements Pizza
{
    public function __construct(
        private Siciliana $siciliana,
        private Ingredient $aubergine
    ) {}

    public function getIngredients(): array
    {
        return [...$this->siciliana->getIngredients(), $this->aubergine];
    }
}

```

```php
<?php

final class Napoli implements Pizza
{
    public function __construct(
        private Regina $regina,
        private Parmigiana $parmigiana,
        private Ingredient $artichoke
    ) {}

    public function getIngredients(): array
    {
        return [
            ...$this->regina->getIngredients(),
            ...$this->parmigiana->getIngredients(),
            $this->artichoke
        ];
    }
}

```

```php
<?php

var_dump($margherita instanceof Pizza);       // true
var_dump($margherita instanceof $siciliana);  // false
var_dump($margherita instanceof $regina);     // false
var_dump($margherita instanceof $parmigiana); // false
var_dump($margherita instanceof $napoli);     // false

```

```php
<?php

$tomato = new Tomato;
$cheese = new Cheese;
$mushroom = new Mushroom;
$ham = new Ham;
$aubergine = new Aubergine;
$artichoke = new Artichoke;

$basePizza = new CustomPizza($tomato, $cheese);

$margherita = new Margherita($basePizza);

$siciliana = new Siciliana($margherita, $muschroom);

$regina = new Regina($siciliana, $ham);

$parmigiana = new Parmigiana($siciliana, $aubergine);

$napoli = new Napoli($regina, $parmigiana, $artichoke);

```

Object-oriented programming with composition

Real life example

- Each dependency is created,

Object-oriented programming with composition

Real life example

- Each dependency is created,
- Each dependency is injected when it's needed.

```php
<?php

/**
 * Last-In-First-Out (LIFO) stack.
 */
final class Stack {
    private ArrayObject $stack;

    public function __construct() {
        $this->stack = new ArrayObject;
    }

    public function push(mixed $value): void
    {
        $this->stack->append($value);
    }

    public function pop(): mixed
    {
        $arrayCopy = $this->stack->getArrayCopy();
        $result = array_pop($arrayCopy);
        $this->stack->exchangeArray($arrayCopy);

        return $result;
    }
}

```

Object-oriented programming with composition

Key points

- No `extends` needed,

Object-oriented programming with composition

Key points

- No `extends` needed,
- Internals are hidden from public interface,

Object-oriented programming with composition

Key points

- No `extends` needed,
- Internals are hidden from public interface,
- Internals are *swap-able* without altering the public interface,

Object-oriented programming with composition

Key points

- No `extends` needed,
- Internals are hidden from public interface,
- Internals are *swap-able* without altering the public interface,
- The `final` keyword enforces the `Composition` pattern.

Object-oriented programming with composition

Issues

- No *Yo-Yo* problem

Object-oriented programming with composition

Issues

- No *Yo-Yo* problem
- No hard relation between a class and another

Object-oriented programming with composition

Issues

- No *Yo-Yo* problem
- No hard relation between a class and another
- No inheritance of unnecessary methods

Object-oriented programming with composition

Issues

- No *Yo-Yo* problem
- No hard relation between a class and another
- No inheritance of unnecessary methods
- No flexibility issue as a class is not supposed to extend another

Object-oriented programming with composition

Issues

- No *Yo-Yo* problem
- No hard relation between a class and another
- No inheritance of unnecessary methods
- No flexibility issue as a class is not supposed to extend another
- No `is-a` relation, now replaced with `has-a`.

European Commission

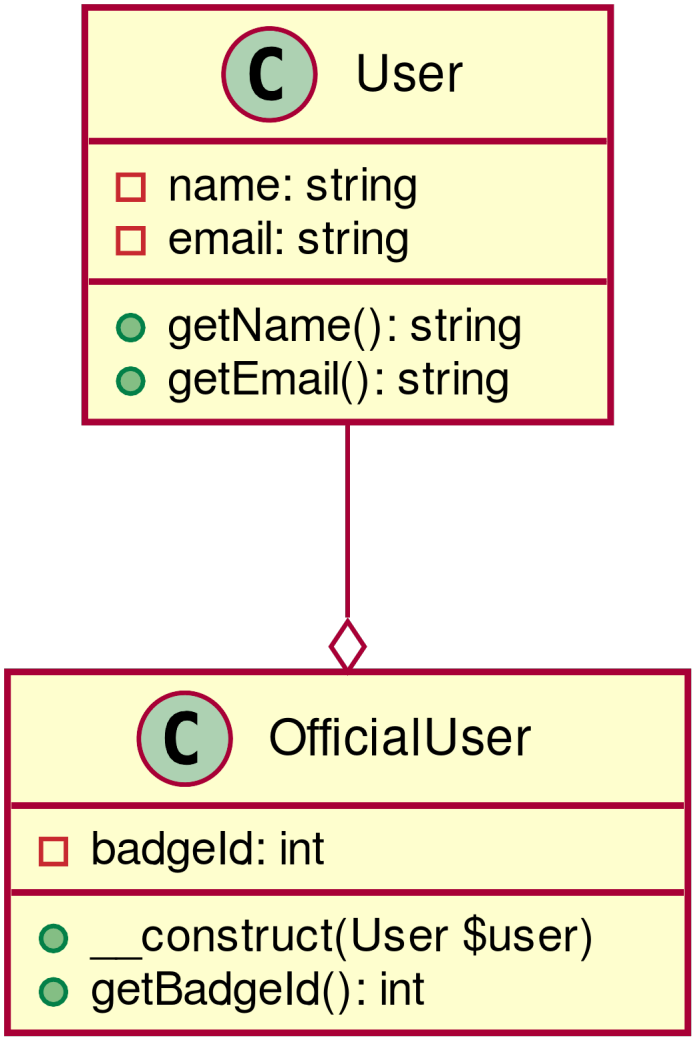
Simplicity is the ultimate sophistication.

Leonardo Da Vinci

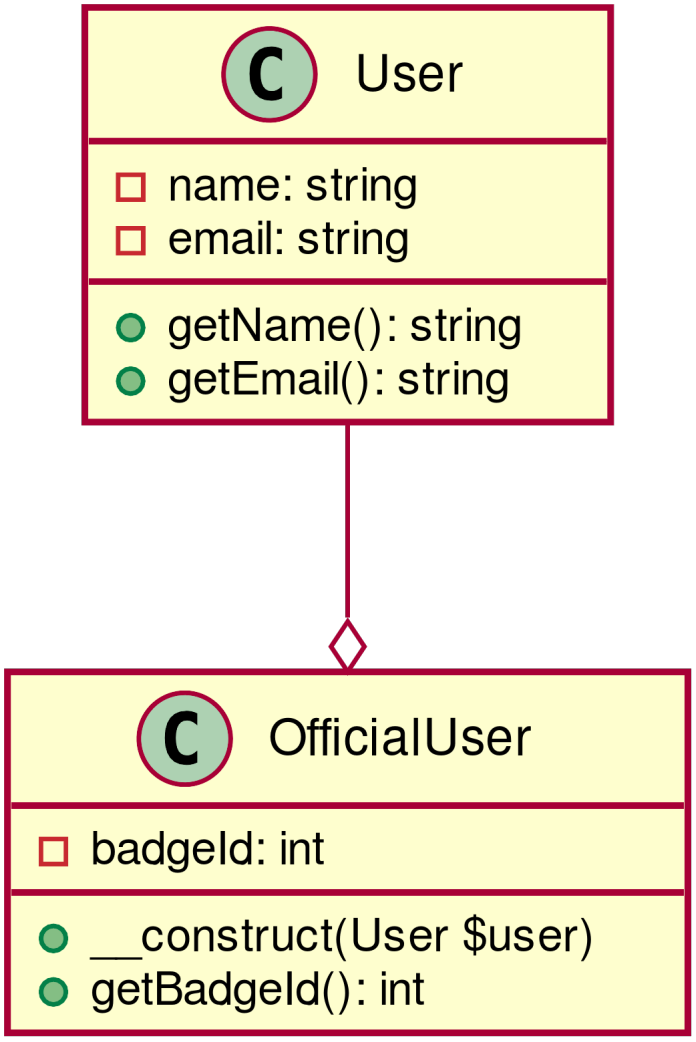
But wait. . .

But wait. . . I'm pretty sure I already saw that somewhere!

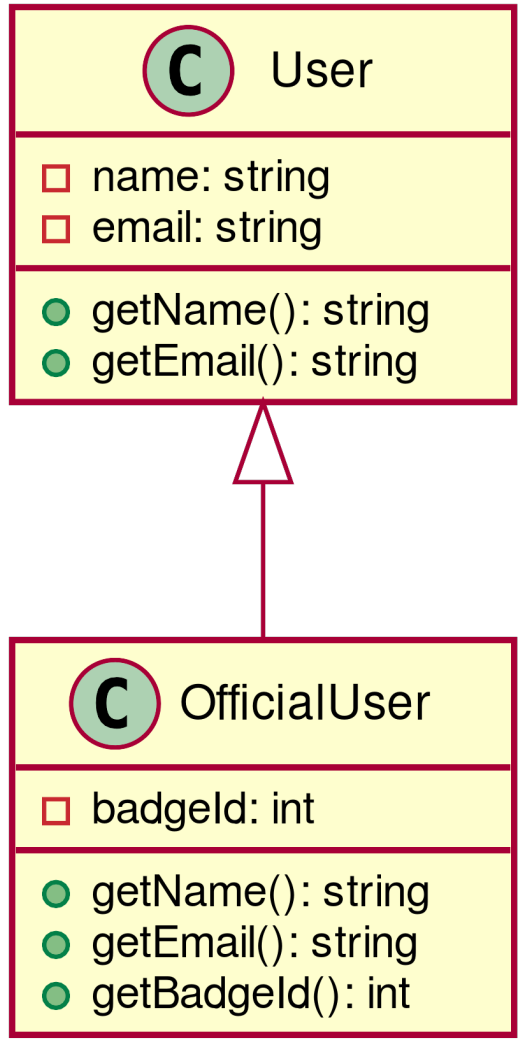
Composition



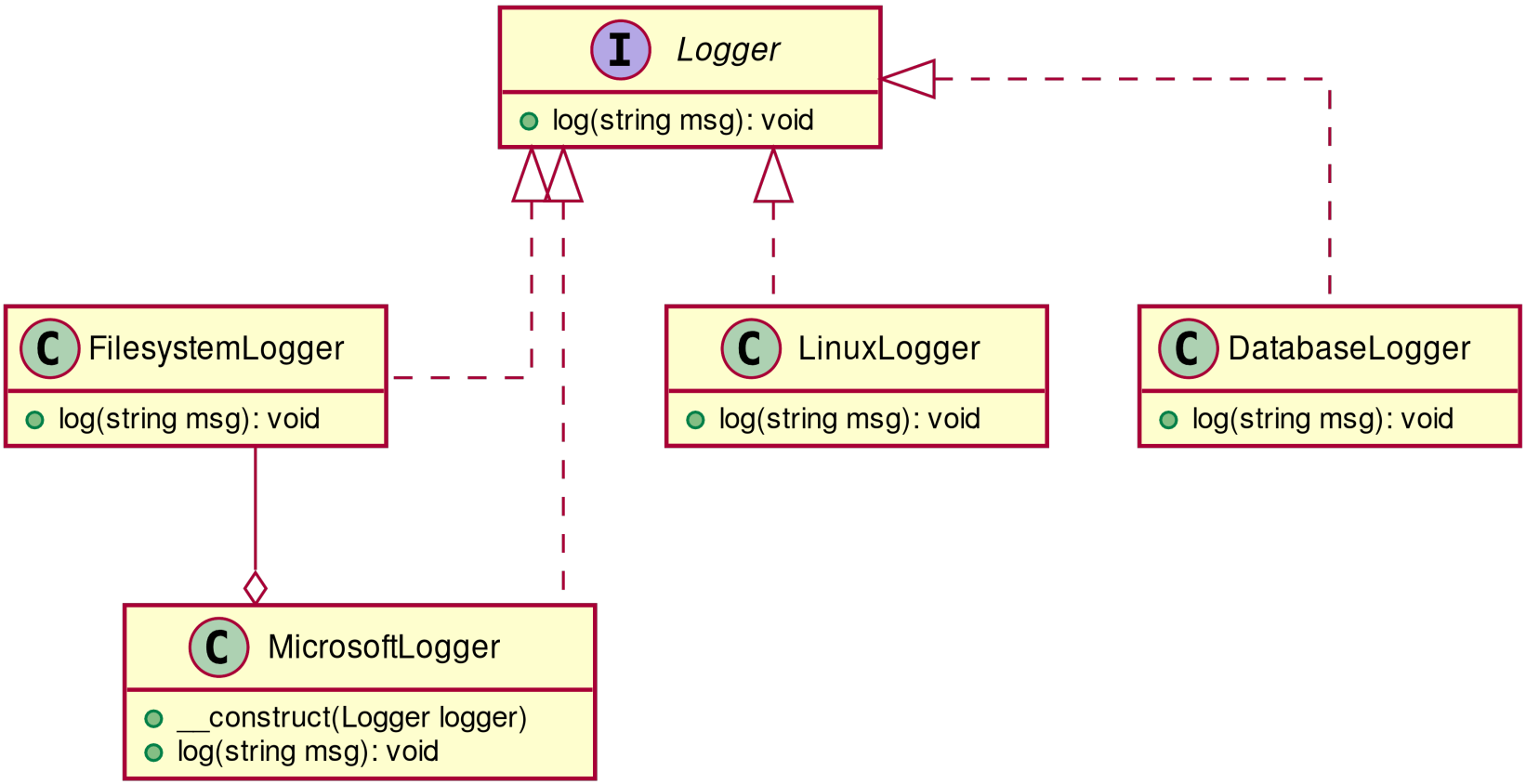
Composition



Inheritance



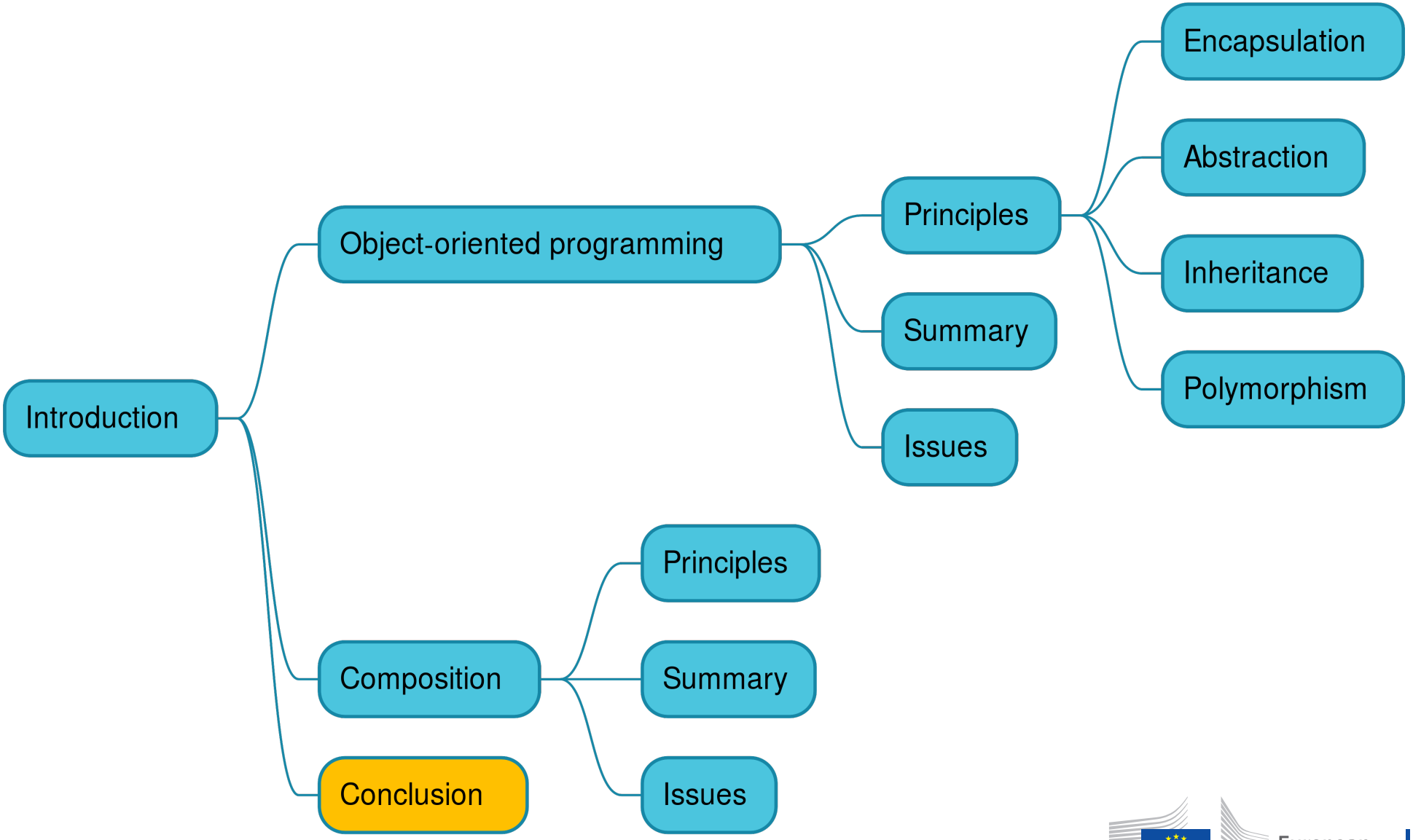
Decorator pattern



CONCLUSION

Composition and inheritance

Map



Composition and inheritance

Conclusion / When to use composition?

- As much as you want,

- As much as you want,
- Use language constructions to enforce this pattern (interfaces, final keyword, $\cdots$).

Composition and inheritance

Conclusion / Where to use it?

- Anywhere,

Composition and inheritance

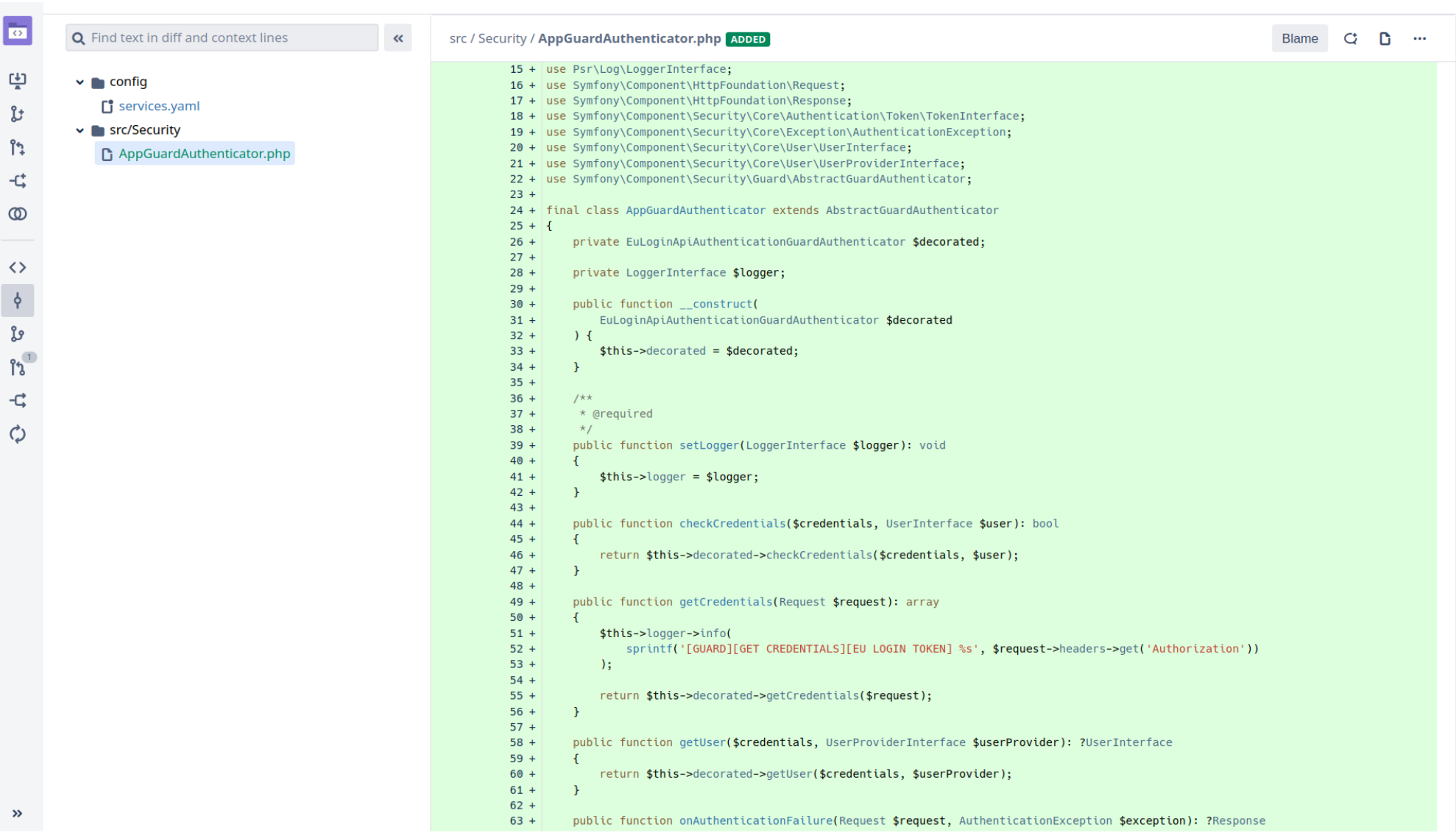
Conclusion / Where to use it?

- Anywhere,
- When customizing part of a framework's component,

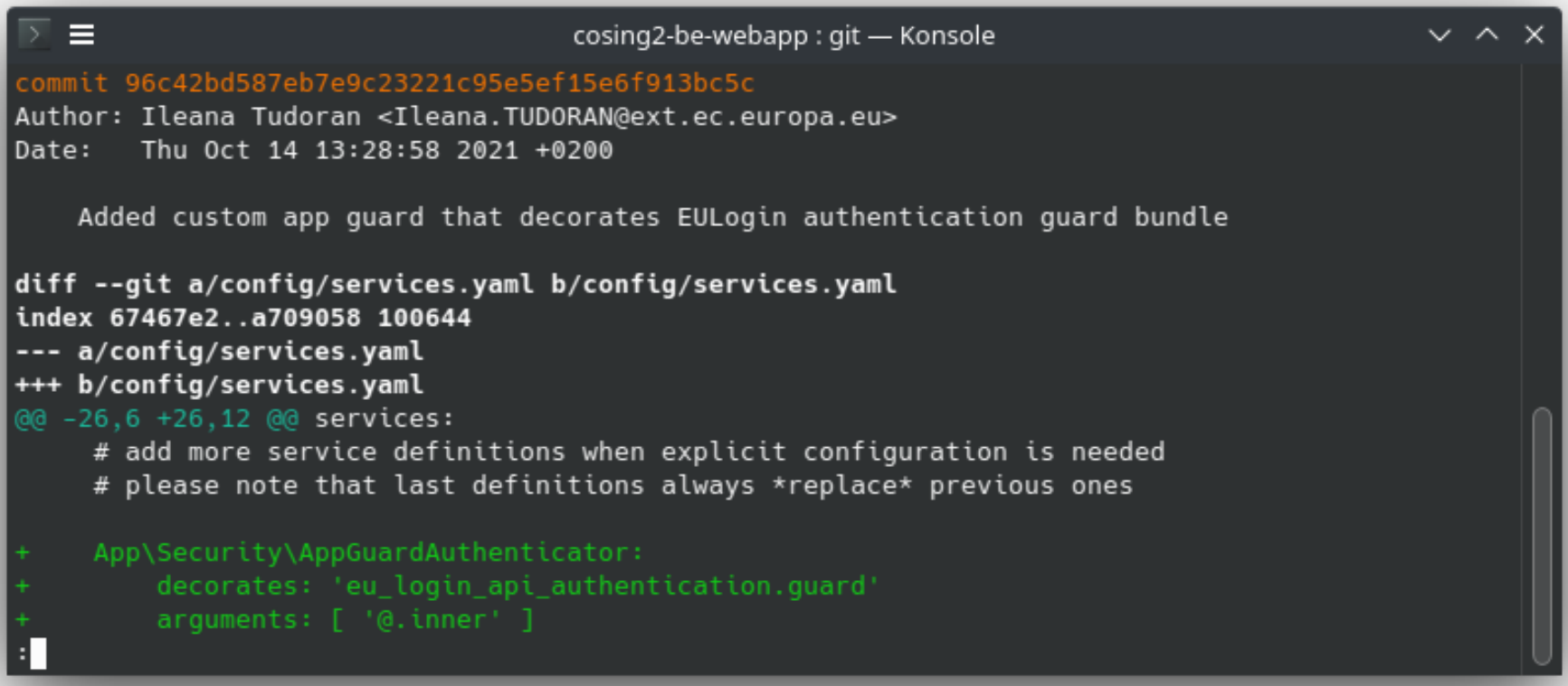
Composition and inheritance

Conclusion / Where to use it?

- Anywhere,
- When customizing part of a framework's component,
- When building custom library.



```php
<?php

final class AppGuardAuthenticator extends AbstractGuardAuthenticator
{
    public function __construct(
        private EuLoginApiAuthenticationGuardAuthenticator $decorated,
        private LoggerInterface $logger
    ) {}

    public function getCredentials(Request $request): array
    {
        $this->logger->info(
            sprintf(
                '[GUARD][GET CREDENTIALS][EU LOGIN TOKEN] %s',
                $request->headers->get('Authorization')
            )
        );

        return $this->decorated->getCredentials($request);
    }

    public function checkCredentials($credentials, UserInterface $user): bool
    {
        return $this->decorated->checkCredentials($credentials, $user);
    }
}

```



European Commission

- Clearer code,

Composition and inheritance

Conclusion / Advantages of composition over inheritance

- Clearer code,
- Ease of testing,

Composition and inheritance

Conclusion / Advantages of composition over inheritance

- Clearer code,
- Ease of testing,
- Works well with `S.O.L.I.D.` principles,

- Clearer code,
- Ease of testing,
- Works well with `S.O.L.I.D.` principles,
- Code **does not** run if an interface has been updated upstream and is not fulfilled.

Composition and inheritance

Conclusion / Drawbacks of composition over inheritance

- Code is more verbose,

Composition and inheritance

Conclusion / Drawbacks of composition over inheritance

- Code is more verbose,
- $\cdots$ (*I'm still looking for them*)

Composition and inheritance

Conclusion / A few tips

- Avoid extending classes,

Composition and inheritance

Conclusion / A few tips

- Avoid extending classes,
- Avoid `abstract` classes,

- Avoid extending classes,
- Avoid `abstract` classes,
- Setting your classes as `final` should be the default construction pattern,

- Avoid extending classes,
- Avoid `abstract` classes,
- Setting your classes as `final` should be the default construction pattern,
- Create an `interface` for each of your services,

- Avoid extending classes,
- Avoid `abstract` classes,
- Setting your classes as `final` should be the default construction pattern,
- Create an `interface` for each of your services,
- Prefer stateless services to stateful,

- Avoid extending classes,
- Avoid `abstract` classes,
- Setting your classes as `final` should be the default construction pattern,
- Create an `interface` for each of your services,
- Prefer stateless services to stateful,
- Read, understand and apply the `S.O.L.I.D.`[1] principles.

[1] Wikipedia - SOLID

Now when someone says...

Now when someone says...

"*The class is final, It is not possible to extend it...*"

Now when someone says...

"*The class is final, It is not possible to extend it...*"

You know what to answer!

Thanks