the NeRd Talks

Dependency Injection

NeRdTalker: Marco Ciprietti

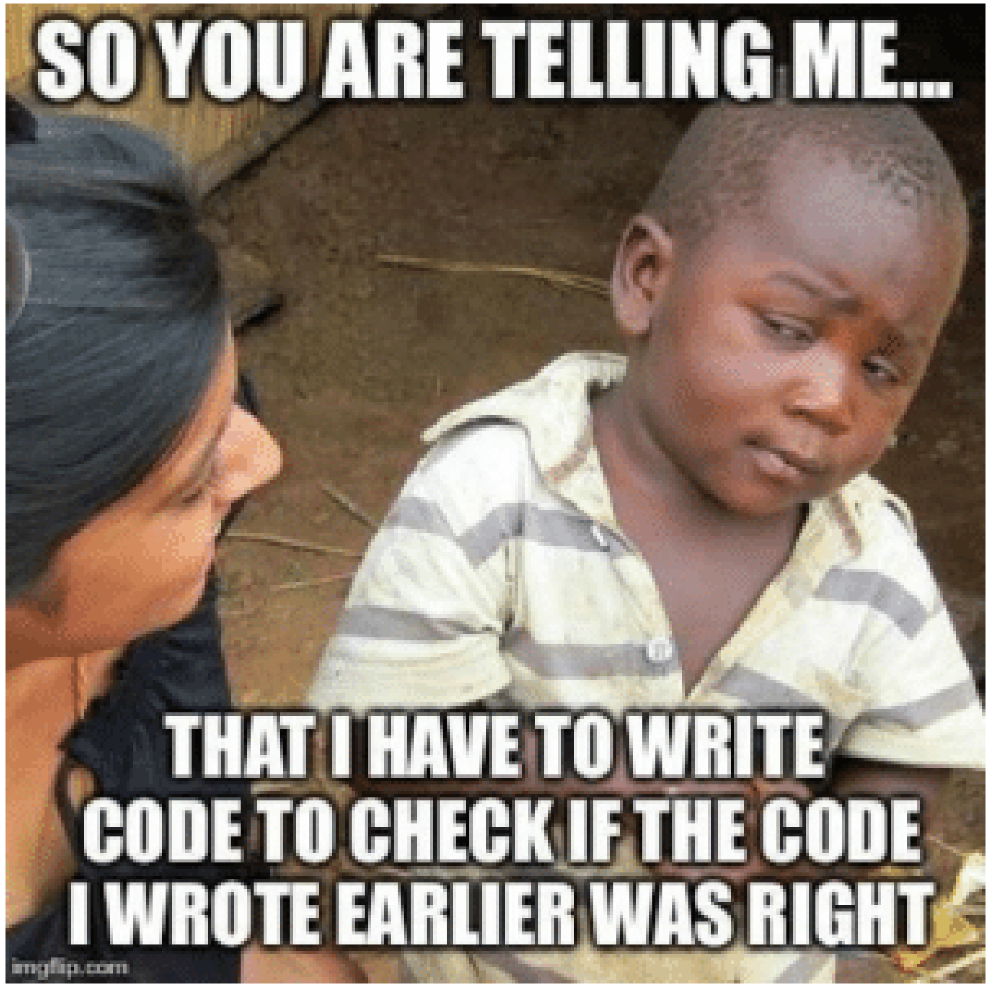
Dependency Injection

Ice Cream Service

```php
class IceCreamController {
    public function iceCreamAction($request) {
        $iceCreamService = new IceCreamService();
        $iceCream = $iceCreamService->makeIceCream($request->get('flavors'));

        return new Response($iceCream);
    }
}
```

```php
class IceCreamService {
    public function getIceCream($flavors) {
        $iceCream = new IceCream();

        foreach ($flavors as $flavor) {
            $iceCreamFlavor = $this->database->findByFlavor($flavor);

            if (!$iceCreamFlavor)
                throw new FlavorNotFound();

            $iceCream->addFlavor($iceCreamFlavor);
        }

        return $iceCream;
    }
}
```

Ice Cream Service – Unit Test


SO YOU ARE TELLING ME...
THAT I HAVE TO WRITE CODE TO CHECK IF THE CODE I WROTE EARLIER WAS RIGHT
imgflip.com

Unit Test

Unit testing is a software development process in which the ***smallest testable parts*** *of an application, called units, are individually and independently scrutinized for proper operation.*

searchsoftwarequality.techtarget.com/definition/unit-testing

Integration Test

Integration testing is the phase in software testing in which individual software modules are ***combined and tested as a group****.*

[…]

It occurs after unit testing and before validation testing.

wikipedia.org/wiki/Integration_testing

Unit Test vs. Integration Test

Unit Testing

- Test the smallest testable part of the application
- Unit tests should have no dependencies on code outside the unit tested.
- Modules are tested independently

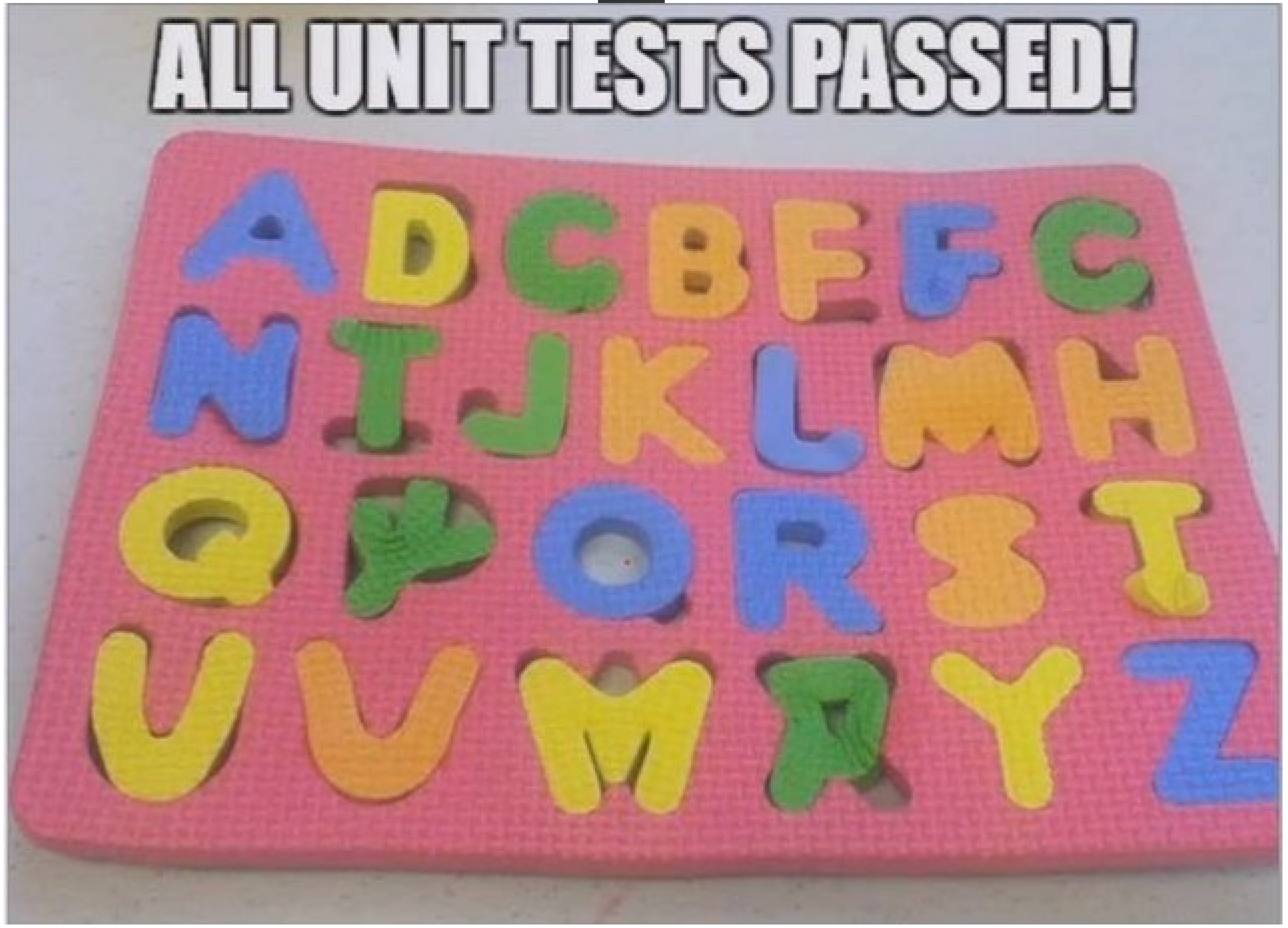
Integration Testing

- Test the real-life operations of the application
- Integration testing is dependent on other outside systems like databases, hardware etc.
- Modules are combined together

Ice Cream Service – Unit Test

```
class IceCreamControllerTest extends TestCase {

    public function testNoIceCream() {
        $iceCreamController = new IceCreamController();

        $this->expectException(FlavorNotFound::class);
        $iceCream = $iceCreamController->iceCreamAction(new Request(['lemon']));
    }



    public function testIceCreamExists() {
        IceCreamService::insertFlavor('lemon');

        $iceCreamController = new IceCreamController();
        $iceCream = $iceCreamController->iceCreamAction(new Request(['lemon']));

        $this->assertNotNull($iceCream);
        $this->assertEquals(['lemon'], $iceCream->getFlavors());
    }
}
```

Problem 1



Unit Test vs. Integration Test

nit Testing

- Test the te t of the applicatio
- Unit tests sh o dependen ide the unit tes
- Mod tested indep y

Integration Testing

- Test the real-life operations of the application
- Integration testing is dependent on other outside systems like databases, hardware etc.
- Modules are combined together

Problem 2

Extensibility

Inversion of Control

Inversion of Control (IoC) is a programming principle which inverts the flow of control as compared to traditional control flow.
In IoC, custom-written portions of a computer program receive the flow of control from a generic framework.

wikipedia.org/wiki/Inversion_of_control

Inversion of Control

Traditional

In *traditional programming*, the custom code calls into reusable libraries to take care of generic tasks.

Inversion of Control

With *Inversion of Control*, it is the framework that calls into the custom, or task-specific, code.

Inversion of Control

Separation of the *what-to-do* part of the code from the *when-to-do* part.

- Clients provide the *when-to-do* (*IceCreamController)*
- Services provide the *what-to-do* (*IceCreamService*)

Advantages:

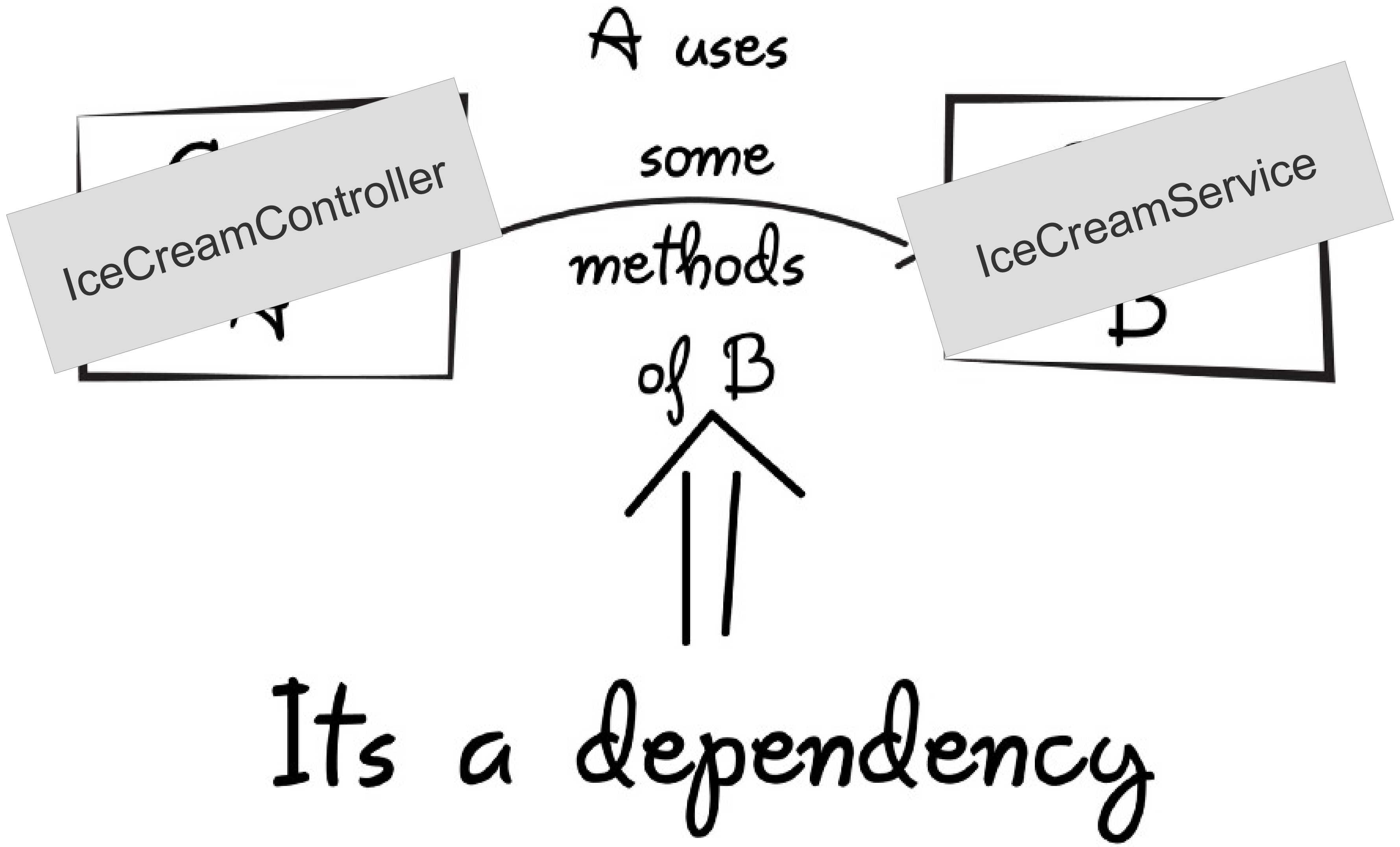
- Decouple the execution of a task from implementation
- Focus a module on the task it is designed for
- To prevent side effects when replacing a module
- Increase modularity and extensibility
- Free modules from assumptions about how other systems do what they do and instead rely on contracts

Dependency Injection

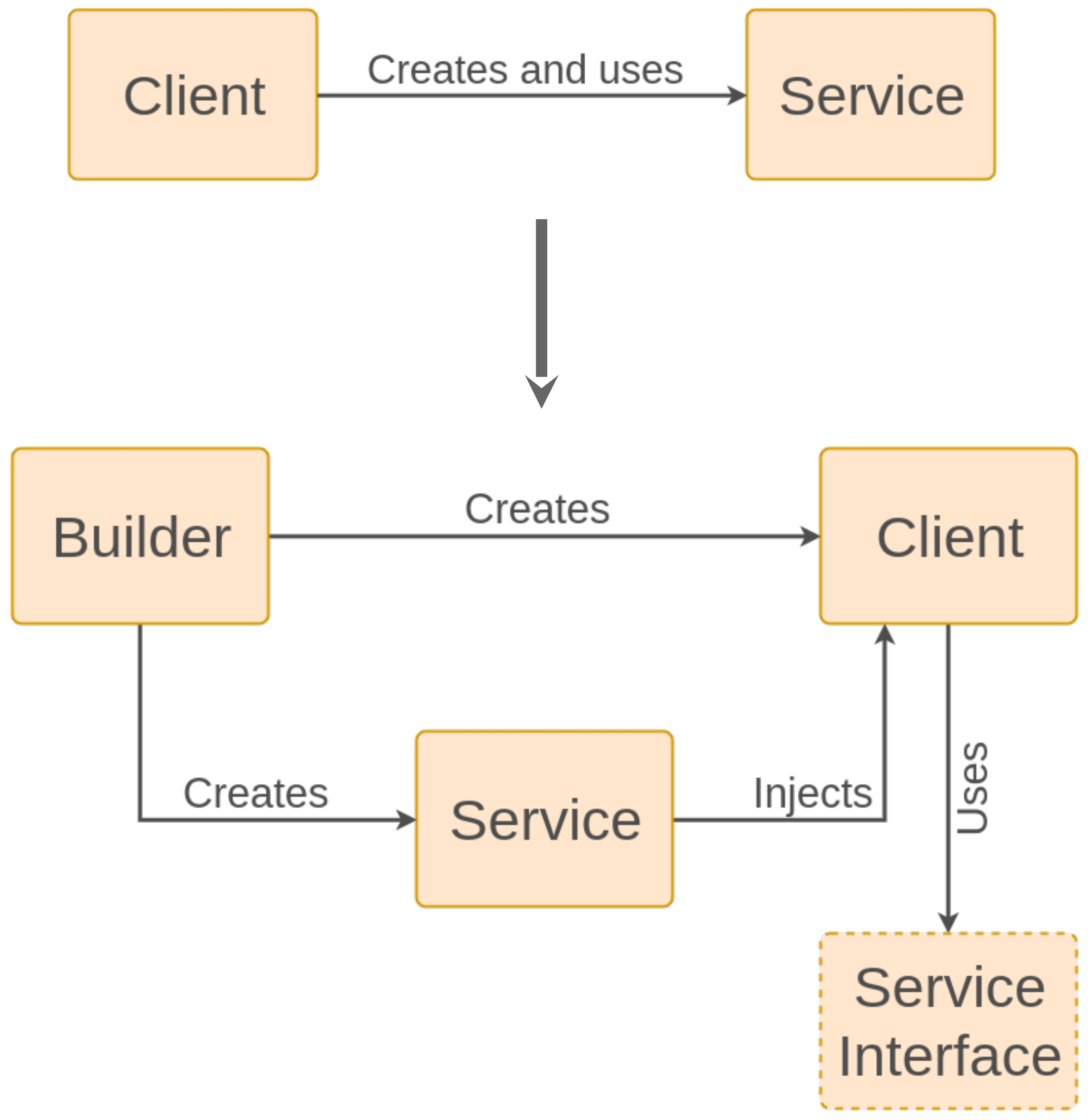
In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object.
A dependency is an object that can be used.

wikipedia.org/wiki/Dependency_injection

Dependency Injection



Dependency Injection



Dependency Injection

Constructor Injection

```php
class IceCreamController {
    /** @var IceCreamService */
    private $iceCreamService;

    public function __construct(IceCreamService $iceCreamService) {
        $this->iceCreamService = $iceCreamService;
    }
}
```

Pros

- If the dependency is a requirement and the class cannot work without it.
- The dependency won't change during the object's lifecycle.

Cons

- Extend the class and override the constructor can be a mess.

Dependency Injection

Setter Injection

```php
class IceCreamController {
    /** @var IceCreamService */
    private $iceCreamService;

    public function setIceCreamService(IceCreamService $iceCreamService) {
        $this->iceCreamService = $iceCreamService;
    }
}
```

Pros

- Optional dependencies. If you do not need the dependency, then just do not call the setter.
- You can call the setter multiple times, usefull to add dependencies to a collection.

Cons

- You can call the setter multiple times, so you cannot be sure the dependency is not replaced during the lifetime.
- You cannot be sure the setter will be called.

Dependency Injection

Property Injection

```
class IceCreamController {

    /** @var IceCreamService */
    public $iceCreamService;
}
```

Pros

- Optional dependencies.

Cons

- Dependency is out of control, it can be changed at any point in the object's lifetime.
- You cannot use type hinting so you cannot be sure what dependency is injected (*Symfony*)

Ice Cream Service

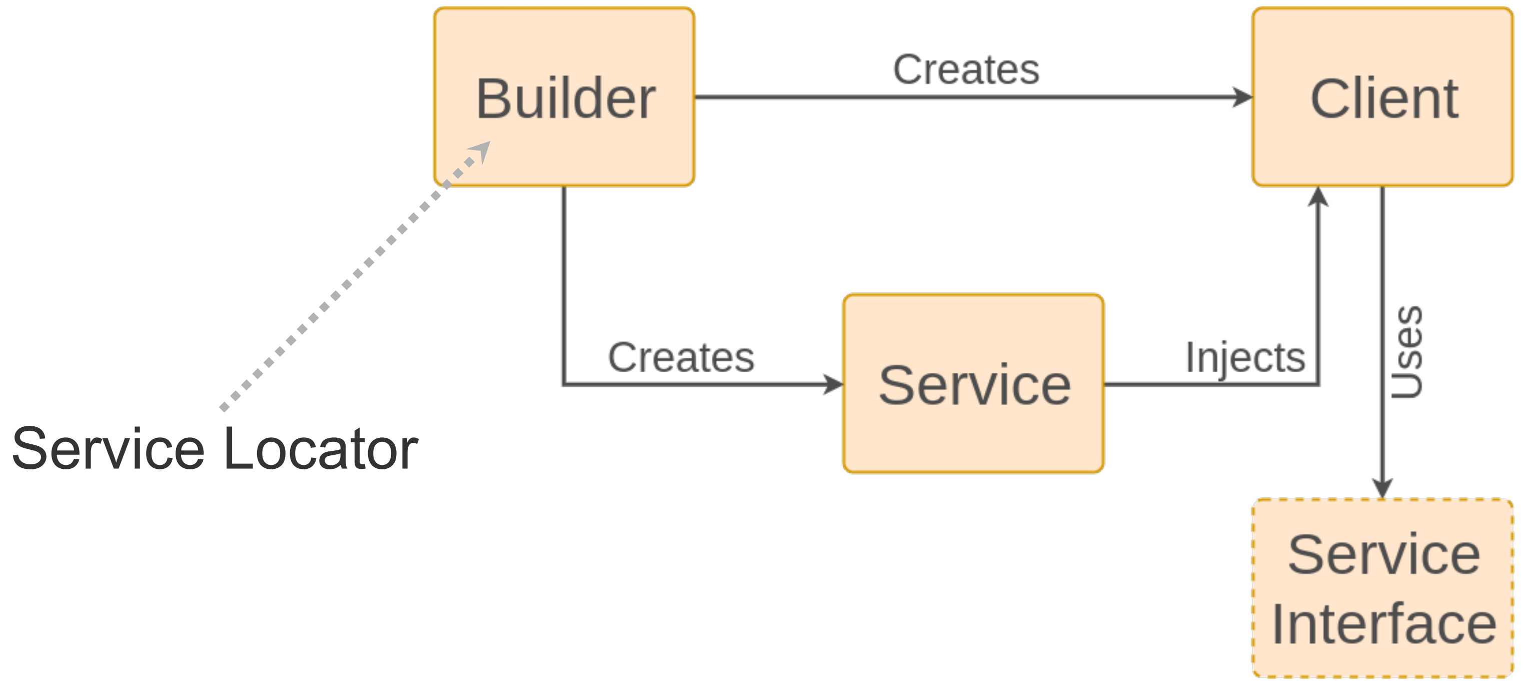
```php
class IceCreamController {

    /** @var IceCreamService */
    private $iceCreamService;

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

    public function makeAction(Request $request) {
        $iceCream = $this->iceCreamService->makeIceCream($request->get('flavors'));

        return new Response($iceCream);
    }
}
```

```php
interface IceCreamService {
    /**
     * @param string[] $flavors
     * @return IceCream
     * @throws FlavorNotFound if one of the requested flavors doesn't exist
     */
    function makeIceCream($flavors);
}
```

Dependency injected!

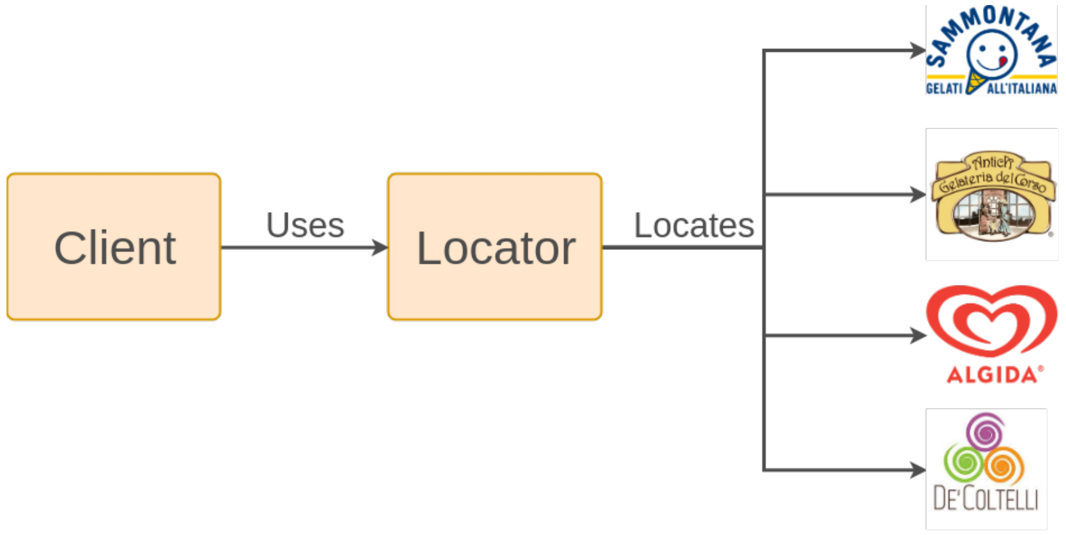
Ice Cream Service

Dependency Injection



Dependency Injection

```
class ServiceLocator {

    private $services = [
        DatabaseIceCreamService::class   => new DatabaseIceCreamService(),
        SammontanaIceCreamService::class => new SammontanaIceCreamService(),
        AlgidaIceCreamService::class     => new AlgidaIceCreamService(),
        [...]
    ];

    public function get($serviceId) {
        return $this->services[$serviceId];
    }
}
```



Dependency Injection

```
class IceCreamController {

    private $iceCreamService;
    private $mailerService;
    private $deliveryService;

    public function __construct(ContainerInterface $container) {
        $this->iceCreamService = $container->get(IceCreamService::class);
        $this->mailerService   = $container->get(MailerService::class);
        $this->deliveryService = $container->get(DeliveryService::class);
        [...]
    }
}
```

Dependency Injection

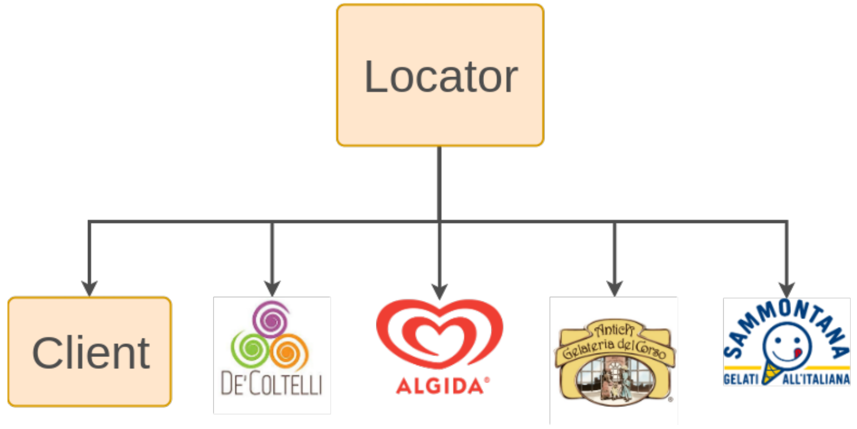
Avoiding your Code Becoming Dependent on the Container

Whilst you can retrieve services from the container directly it is best to minimize this.
[...]
You could have injected the container in and retrieved the ice cream service from it but it would then be tied to this particular container making it difficult to reuse the class elsewhere.

symfony.com/doc/current/components/dependency_injection.html

Dependency Injection

```
class IceCreamController {

    private $iceCreamService;
    private $mailerService;
    private $deliveryService;

    public function __construct(IceCreamService $iceCreamService,
                                MailerService $mailerService,
                                DeliveryService $deliveryService) {
        $this->iceCreamService = $iceCreamService;
        $this->mailerService = $mailerService;
        $this->deliveryService = $deliveryService;
    }
}
```



Dependency Injection

services.yml

```yaml
services:

  # actual service class
  Service\SammontanaIceCreamService: ~

  # ice cream service interface
  Service\IceCreamService: '@Service\SammontanaIceCreamService'

  # controller with injected service
  Controller\IceCreamController:
    arguments: ['@Service\IceCreamService']
```

Dependency Injection

services.yml

```yaml
services:
  _defaults:
    autowire: true  # enable autowiring for every service
    autoconfigure: true
    public: false

  # actual service class
  Service\SammontanaIceCreamService: ~

  # ice cream service interface
  IceCreamService: '@Service\SammontanaIceCreamService'

  # controller with injected service
  Controller\IceCreamController: ~
```

Ice Cream Service – Unit Test

```
class NoIceCreamServiceStub implements IceCreamService {

    public function makeIceCream($flavors) {
        throw new FlavorNotFound();
    }
}
```

```
class IceCreamServiceStub implements IceCreamService {

    public function makeIceCream($flavors) {
        return new IceCream($flavors);
    }
}
```

Ice Cream Service – Unit Test

```
class IceCreamControllerTest extends TestCase {

    public function testNoIceCream() {
        $iceCreamController = new IceCreamController(new NoIceCreamServiceStub());

        $this->expectException(FlavorNotFound::class);
        $iceCream = $iceCreamController->iceCreamAction(new Request(['lemon']));
    }


    public function testIceCreamExists() {
        $iceCreamController = new IceCreamController(new IceCreamServiceStub());

        $iceCream = $iceCreamController->iceCreamAction(new Request(['lemon']));

        $this->assertNotNull($iceCream);
        $this->assertEquals(['lemon'], $iceCream->getFlavors());
    }
}
```

Unit Test vs. Integration Test

Unit Testing

- Test the smallest testable part of the application
- Unit tests should have no dependencies on code outside the unit tested.
- Modules are tested independently

Integration Testing

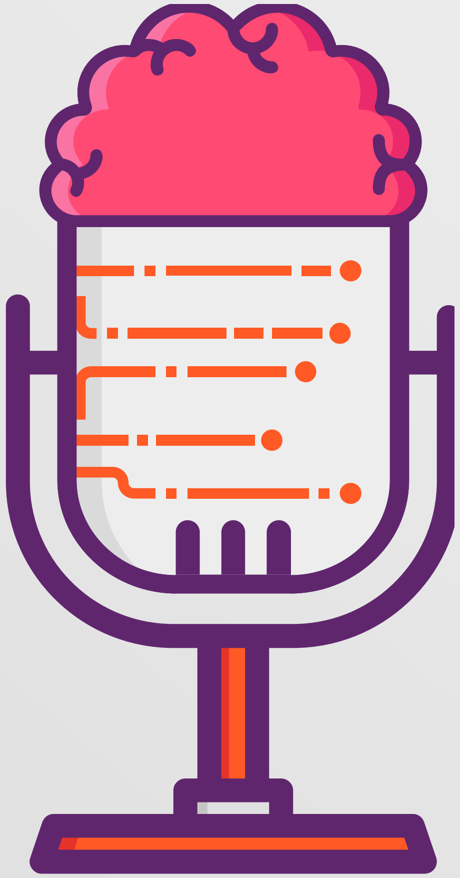
- Test the real-life operations of the application
- Integration testing is dependent on other outside systems like databases, hardware etc.
- Modules are combined together

Summary

- Use Dependency Injection
  - More flexible code
  - Easier to unit test
  - Easier extendable

- Inject services not the Service Locator
  - Clients are not bound to Service Locator
  - Easier to use mock and stubs in tests
  - It not hides dependencies

- Let the Service Container autowire your services
  - Manage services with minimal configuration
  - It is predictable
  - No runtime overhead



That's all Folks!