Write SOLID Code & Impress your Friends

With <3 from SymfonyCasts

Chapter 1: SOLID: The Good, The Bad & The Real World

Hey friends! Welcome to our *long* awaited tutorial on the principles of SOLID: single responsibility principle, open closed principle, Liskov substitution principle, interface segregation principle and, my personal favorite: the donut in face principle. Probably... actually known as the dependency inversion principle.

I want to thank my coauthor Diego for helping me *finally* put this tutorial together. And I'm *super* sorry if you've been waiting for this!

SOLID Principles: I don't Love Them

So... why *did* it take us so long to get this tutorial done? The short answer is: I.... kind of don't like the SOLID principles. Okay, let me rephrase that. The SOLID principles are tough to understand. And, in my most humble opinion, they're not always good advice! It depends on the situation. For example, you should write code for your application *differently* than you would write code that's meant to be open sourced and *shared*.

If you want to know a bit more about why SOLID might *not* always be correct, you can read a recent blog post written by Dan North called CUPID – THE BACK STORY. Dan North is known for being the person who first made behavior-driven development famous. You may have heard of him if you're a Behat user.

Anyways, this tutorial is *not* going to be yet another tutorial where we read the definition of each SOLID principle in a monotone voice... and slowly get lost, bored and finally fall asleep. Nope. We're going to dive into each principle, learn what they *really* mean - using normal human words - code some real examples and discuss why and when following these principles makes sense and does *not* make sense. But even when the SOLID principles should *not* be followed, they have a lot to teach us. So strap in for a wild ride.

Project Setup

Since we're going to be doing some *real* coding, let's get the project set up and rocking. Do me a solid by downloading the course code from this page and unzipping it. After you do, you'll find a `start/` directory with the same code you see here. This fancy `README.md` file has all the details about how to get the project up and running. The last step will be to find a terminal, move into the project and start a local web server. I'll use the Symfony binary for this:

```
symfony serve -d
```

Once this finishes, copy that URL, spin back over to your browser, paste and...say hello to "Sasquatch Sightings"! Our latest effort to find the infamous Bigfoot. What this code actually does is... not too important. It talks to a database, lists some big foot sightings and has some calculations. It will be our playground for diving into the SOLID principles.

So next, let's start with the first: the single responsibility principle!

Chapter 2: Single-Responsibility Principle: What is it?

SOLID starts with the Single-Responsibility Principle or SRP.SRP says:

> A module should have only one reason to change.

Um, huh? This sounds... a *little* too "fluffy" to be *actually* useful.

Let's... try again with a... somewhat simpler definition:

> A function or class should be responsible for only one task...or should have only one "responsibility".

Better. But... what *is* a "responsibility" exactly? And why is this rule helpful?

SRP: The Human Definition

On an *even* simpler level, what SRP is *really* trying to say is:

> Gather together the things that change for the same reason and separate things that change for different reasons.

We'll talk more about this definition later, but keep it in mind.

And what problem is SRP trying to help us solve?In theory, if we organize our code into units that all change for the same reason, then when we get a new feature or change request,we will only need to modify one class...instead of making 10 changes to 10 different files... and trying not to break things along the way.

Sending a Confirmation Email

Enough defining stuff! Let's jump into an example. On your browser, click "Sign Up". As you can see, our app has a registration form! Open `src/Controller/RegistrationController.php` to see the code behind this.Most of the logic for saving the user is in this `UserManager::register()` method. Hold Cmd or Ctrl to jump into this: it lives at `src/Manager/UserManager.php` .

30 lines | src/Manager/UserManager.php

```
... lines 1 - 8
class UserManager
{
... lines 11 - 19
    public function register(User $user, string $plainPassword): void
    {
        $user->setPassword(
            $this->passwordEncoder->encodePassword($user, $plainPassword)
        );

        $this->entityManager->persist($user);
        $this->entityManager->flush();
    }
}
```

This method hashes the user's password... and then saves the user to the database.Awesome!

But *now*... we've received a change request! The product manager of Sasquatch Sightings - a suspiciously hairy person - would like us to send a confirmation email after registration to verify the user's email address.

To understand SRP, let's implement this the *wrong* way first. Well "wrong" according to SRP.

Side note: we're going to build a simple email confirmation system by hand.If you have this need in a *real* project, check out

symfonycasts/verify-email-bundle.

Coding up the Confirmation Email System

Anyways, the easiest way I can see to add this feature isto add the logic right inside `UserManager::register()` ... because we will only have to touch one file and it will guarantee that anything that calls this method will *definitely* trigger the confirmation email.

At the bottom of this class, I'm going to start by pastingin a private function called `createToken()` . You can copy this from the code block on this page. This generates a random string that we will include in the confirmation link.

35 lines | src/Manager/UserManager.php

```
... lines 1 - 8
class UserManager
{
... lines 11 - 29
    private function createToken(): string
    {
        return rtrim(strtr(base64_encode(random_bytes(32)), '+/', '-_'), '=');
    }
}
```

Up in register, generate a new token `$token = $this->createToken()` ... and then set it on the user: `$user->setConfirmationToken($token)` .

38 lines | src/Manager/UserManager.php

```
... lines 1 - 19
    public function register(User $user, string $plainPassword): void
    {
        $token = $this->createToken();
        $user->setConfirmationToken($token);
... lines 24 - 30
    }
... lines 32 - 38
```

Before I started recording - if you look at the `User.php` file - I already created a `$confirmationToken` property that saves to the database. So thanks to the new code, when a user registers,they *will* now have a random confirmation token saved onto their row in the database.

225 lines | src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface
{
... lines 18 - 60
    /**
     * @ORM\Column(type="string", unique=true, nullable=true)
     */
    private $confirmationToken;
... lines 65 - 223
}
```

Back in `RegistrationController` ... if you scroll down a bit, I've*also* already built a confirmation actionto confirm their email. A user just needs to go to this pre-made route - where the `{token}` in the URL matches the `confirmationToken` that we've set onto their `User` record - and... bam! They'll be verified!

64 lines | src/Controller/RegistrationController.php

```
... lines 1 - 13
class RegistrationController extends AbstractController
{
... lines 16 - 43
    /**
     * @Route("/confirm/{token}", name="check_confirmation_link")
     */
    public function confirmAction(string $token, UserRepository $userRepository, EntityManagerInterface $entityManager)
    {
        $user = $userRepository->findOneBy(['confirmationToken' => $token]);

        if (!$user) {
            throw $this->createNotFoundException(sprintf('The user with confirmation token "%s" does not exist', $token));
        }

        $user->setConfirmationToken(null);

        $entityManager->flush();

        $this->addFlash('success', 'Your email is confirmed! Let\'s go confirm some Bigfoot!');

        return $this->redirectToRoute('app_homepage');
    }
}
```

So back in `UserManager`, we have two jobs left. First, we need to generate an absolute URL to the `confirmAction` that contains their token. And second, we need to send an email to the user with that URL inside.

Let's generate the URL first. Up in the constructor, autowire `RouterInterface $router`. I'll hit Alt + Enter and go to "Initialize properties" to create that property and set it.

46 lines | src/Manager/UserManager.php

```
... lines 1 - 7
use Symfony\Component\Routing\RouterInterface;
... lines 9 - 10
class UserManager
{
... lines 13 - 14
    private RouterInterface $router;
... line 16
    public function __construct(UserPasswordEncoderInterface $passwordEncoder, EntityManagerInterface $entityManager, RouterInterface
    {
... lines 19 - 20
        $this->router = $router;
    }
... lines 23 - 44
}
```

Now, below, say `$confirmationLink = $this->router->generate()` and... the name of our route... is `check_confirmation_link`. Use that. For the second argument, pass `token` set to `$user->getConfirmationToken()`. And because this URL will go into an email, it needs to be absolute. Pass a third argument to trigger that: `UrlGeneratorInterface::ABSOLUTE_URL`.

46 lines | src/Manager/UserManager.php

```
... lines 1 - 23
    public function register(User $user, string $plainPassword): void
    {
... lines 26 - 28
        $confirmationLink = $this->router->generate('check_confirmation_link', [
            'token' => $user->getConfirmationToken()
        ], UrlGeneratorInterface::ABSOLUTE_URL);
... lines 32 - 38
    }
... lines 40 - 46
```

Now, let's send the email! On top, add one more argument - `MailerInterface $mailer` and use the same Alt + Enter, "Initialize properties", trick to create that property and set it.

49 lines | src/Manager/UserManager.php

```
... lines 1 - 6
use Symfony\Component\Mailer\MailerInterface;
... lines 8 - 11
class UserManager
{
... lines 14 - 16
    private MailerInterface $mailer;
... line 18
    public function __construct(UserPasswordEncoderInterface $passwordEncoder, EntityManagerInterface $entityManager, RouterInterface
    {
... lines 21 - 23
        $this->mailer = $mailer;
    }
... lines 26 - 47
}
```

Beautiful! Below, I'll paste in some email generation code. I'll also re-type the `I` on `TemplatedEmail` and hit tab so that PhpStorm adds the `use` statement on top for me.

59 lines | src/Manager/UserManager.php

```
... lines 1 - 6
use Symfony\Bridge\Twig\Mime\TemplatedEmail;
... lines 8 - 12
class UserManager
{
... lines 15 - 27
    public function register(User $user, string $plainPassword): void
    {
... lines 30 - 36
        $confirmationEmail = (new TemplatedEmail())
            ->from('staff@example.com')
            ->to($user->getEmail())
            ->subject('Confirm your account')
            ->htmlTemplate('emails/registration_confirmation.html.twig')
            ->context([
                'confirmationLink' => $confirmationLink
            ]);
... lines 45 - 51
    }
... lines 53 - 57
}
```

This creates an email to this user, from this address...and the template it references already exists. You can see it in: `templates/emails/registration_confirmation.html.twig` .

71 lines | templates/emails/registration_confirmation.html.twig

```
{% apply inline_css %}
<!doctype html>
<html lang="en">
... lines 4 - 42
<body>
<div class="body">
... lines 45 - 50
        <div class="content">
            <h1 class="text-center">Nice to meet you %name%!</h1>
            <p class="block">
                Please <a href="{{ confirmationLink }}">Confirm your account</a>.
            </p>
            <p class="block">
                Or go directly to this URL: {{ confirmationLink }}
            </p>
        </div>
... lines 60 - 66
</div>
</body>
</html>
{% endapply %}
```

We're passing a `confirmationLink` variable... and that is rendered inside the email.

Finally, all the way at the bottom of `register()` ... so after we know that the user has saved successfully, deliver the mail with: `$this->mailer->send($confirmationEmail)` .

61 lines | src/Manager/UserManager.php

```
... lines 1 - 27
    public function register(User $user, string $plainPassword): void
    {
... lines 30 - 52
        $this->mailer->send($confirmationEmail);
    }
... lines 55 - 61
```

Alright! We did it! And we can even try this! Back at the registration page, register as a new user...any password, hit enter and... awesome! It looks like it worked!

Now, the project is not configured to *actually* deliver the email. But we can see what that imaginary email *would* have looked like by going down to the web debug toolbar, clicking any of these links to go to the profiler...hitting "last 10"... then clicking to get into the profiler for the POST request that we just made to the registration form.

On the left, click into the "Email" section. There's our email! You can even look at its HTML. I'm going to steal the confirmation link... pop it into a new tab and... our email is confirmed! Mission accomplished!

And, all of our code is centralized into one method. But... we *did* just violate SRP: our `UserManager` class now has too many responsibilities! But what do I mean by the word "responsibility"? And what *are* the responsibilities that this class has? And what's the problem with violating SRP anyways? And does the influence of gravity extend out forever?

Let's answers most of these questions next.

Chapter 3: SRP: Responsibilities

We've just been informed that - gasp - from time to time,our confirmation email doesn't reach our user's inbox!Ah! And so: we need to implement a resend feature.

SRP: You Shouldn't Need to Change Unrelated Code

This should be easy, right? After all, we've encapsulated all of our logicfor sending a confirmation email into one method. But... hmm. To get this to work, we're probably going to need to extract*part* of the `register()` method into a separate public function so that we can *just* resend the email... without also creating a new token and re-hashing the password.

61 lines | src/Manager/UserManager.php

```
... lines 1 - 12
class UserManager
{
... lines 15 - 27
    public function register(User $user, string $plainPassword): void
    {
        $token = $this->createToken();
        $user->setConfirmationToken($token);

        $confirmationLink = $this->router->generate('check_confirmation_link', [
            'token' => $user->getConfirmationToken()
        ], UrlGeneratorInterface::ABSOLUTE_URL);

        $confirmationEmail = (new TemplatedEmail())
            ->from('staff@example.com')
            ->to($user->getEmail())
            ->subject('Confirm your account')
            ->htmlTemplate('emails/registration_confirmation.html.twig')
            ->context([
                'confirmationLink' => $confirmationLink
            ]);

        $user->setPassword(
            $this->passwordEncoder->encodePassword($user, $plainPassword)
        );

        $this->entityManager->persist($user);
        $this->entityManager->flush();

        $this->mailer->send($confirmationEmail);
    }
... lines 55 - 59
}
```

Isn't it kind of weird... or at least "not ideal"... that in order to add this "email resend" feature, we're going to be messing with and rearranging code that deals with hashing passwords and persisting user data?In a perfect world, shouldn't I be able to create this "email resend" feature without going *anywhere* near code that's unrelated to this functionality?

This is what SRP is trying to help us with. In that "perfect" SRP world, each time a change is requested in our project, we would only need to touch code that directly relates to that change: we wouldn't need to change -or even work near - unrelated code. The fact that we're going to need to modify a method that*also* deals with saving users and hashing passwords... in order to add a feature that has *nothing* to do with that stuff... is a sign that `UserManager` violates SRP. Our `UserManager` class has too many responsibilities.

What *is* a "Responsibility"?

But what *are* the responsibilities of this class? I can think of 5 at least: generate a confirmation link...which also includes creating the confirmation token, create an email, hash a password, save the user and send an email.

But... hold on a second. And this is a very, very important - and confusing - point about SRP. Defining responsibilities is *not* meant to mean:

> Think of all the different, tiny things that your class does.

Nope! A better way to say this might be:

> Think of all the different reasons that this class might change.

That's much harder... and it *completely* depends on your application and business. To help with this, it's sometimes useful to think of what our class does on a higher level. In my eyes, our register method does two basic things: (1) it prepares & persists the user and (2) it sends an email.

Now let's see if we can think of a person in our "totally-not-fake" business that might ask for a *change* to one of these two things.

For example, for the "high level job" of "preparing and persisting the user", our database administrator might, in the future, want to change how users are stored... or our CTO might want to start using a third party authentication provider instead of storing users in a local database and managing their passwords. This type of change would affect how we hash passwords *and* how we save users. In other words, two of our original, so-called "responsibilities" - hashing the password and persisting the user - will likely change for the same reason. And so, they are really part of the *same*, *one* responsibility: "preparing and persisting the user".

The other "high level" thing the method does it send the confirmation email. That will most likely need to change if a marketing person wants to tweak the subject of an email to be more fun...or pass in some "featured product" variables to the template to try to sell stuff. This means that 3 of the other original so-called "responsibilities" - generating the confirmation URL, creating the email and sending the email - will all most likely change for the same reason. And so, for our project, they would all be considered *one* responsibility: "sending the confirmation email".

Organizing Responsibilities is an Art... at Best

Is this perfect? *Definitely* not! You could *easily* argue that sending the email would change for *another* reason. If someone decides we're going to start sending emails using a *different* email provider service... we're already protected from that change: that would just require some configuration tweaks in a *different* file. But what if we think that it's likely that we might change how our email verification system works in the future? In that case, we would have a legitimate reason to think that the generation of the confirmation token and link would change for a *different* reason than our user persistence or email creation.

Identifying the most likely reasons that a function might need to change and then grouping the functionality into those responsibilities is the hardest part of SRP. Even *our* grouping looks imperfect. But honestly, it's good enough! My advice is to do your best and don't over think it. We're also going to talk about *over* optimization of SRP later... which can lead to a different problem.

It's also helpful to keep our original "human" definition for SRP in mind:

> Gather together the things that change for the same reason and separate those things that change for different reasons.

Next: now that we've identified the two responsibilities that `UserManager` currently has, let's refactor our code to make it more SRP compliant.

Chapter 4: Refactoring for SRP

We've identified that `UserManager::register()` handles two things that might change for different reasons. These are its two responsibilities: one, creating and sending a confirmation email and two, setting up the data for a user and saving it to the database.

We're now going to follow the advice of SRP and "separate those things that change for different reasons".

Clarifying The Responsibility of UserManager

The first thing I want to do is rename `register()` to `create()` ... or you could use `save()` ... or even rename the entire class itself. The point is: I want to make its responsibility more clear: to set all the required data on the user object and save it to the database.

Right click on `register()`, go to Refactor->Rename and call this `create()`.

61 lines | src/Manager/UserManager.php

```
... lines 1 - 12
13  class UserManager
14  {
... lines 15 - 27
28      public function create(User $user, string $plainPassword): void
29      {
... lines 30 - 53
54      }
... lines 55 - 59
60  }
```

When I hit enter, over in `RegistrationController`, PhpStorm renamed the method there too.

64 lines | src/Controller/RegistrationController.php

```
... lines 1 - 13
14  class RegistrationController extends AbstractController
15  {
... lines 16 - 18
19      public function signup(Request $request, UserManager $userManager)
20      {
... lines 21 - 23
24          if ($form->isSubmitted() && $form->isValid()) {
... lines 25 - 31
32              $userManager->create($user, $plainPassword);
... lines 33 - 36
37          }
... lines 38 - 41
42      }
... lines 43 - 62
63  }
```

Creating the ConfirmationEmailSender Class

Next, let's move the email-related logic into a new class in the `Service/` directory... though, it doesn't matter where this lives. Create a new PHP class called, how about, `ConfirmationEmailSender`. This class will need two services: the router so it can generate the link and mailer. Add a public function `__construct()` with those two arguments: `MailerInterface $mailer`, and `RouterInterface $router`. Hit Alt + Enter and go to "Initialize properties" to create both of those properties and set them. We don't need this extra PHPDoc up here.

19 lines | src/Service/ConfirmationEmailSender.php

```
... lines 1 - 4
use Symfony\Component\Mailer\MailerInterface;
use Symfony\Component\Routing\RouterInterface;

class ConfirmationEmailSender
{
    private MailerInterface $mailer;
    private RouterInterface $router;

    public function __construct(MailerInterface $mailer, RouterInterface $router)
    {
        $this->mailer = $mailer;
        $this->router = $router;
    }
}
```

Now we can create a public function called, how about, `send()`, with a `User` object argument that will return `void`.

40 lines | src/Service/ConfirmationEmailSender.php

```
... lines 1 - 10
class ConfirmationEmailSender
{
... lines 13 - 21
    public function send(User $user): void
    {
... lines 24 - 37
    }
}
```

For the inside of this, let's go steal all of the email-related logic from `UserManager`. So... copy the `$confirmationLink` and `$confirmationEmail` parts... delete those... and paste. Yes PhpStorm: I *definitely* want you to import the `use` statements for me.

The last line we need to steal is the `$mailer->send()` line. Paste that into the new class.

40 lines | src/Service/ConfirmationEmailSender.php

```
... lines 1 - 5
use Symfony\Bridge\Twig\Mime\TemplatedEmail;
... line 7
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
... lines 9 - 10
class ConfirmationEmailSender
{
... lines 13 - 21
    public function send(User $user): void
    {
        $confirmationLink = $this->router->generate('check_confirmation_link', [
            'token' => $user->getConfirmationToken()
        ], UrlGeneratorInterface::ABSOLUTE_URL);

        $confirmationEmail = (new TemplatedEmail())
            ->from('staff@example.com')
            ->to($user->getEmail())
            ->subject('Confirm your account')
            ->htmlTemplate('emails/registration_confirmation.html.twig')
            ->context([
                'confirmationLink' => $confirmationLink
            ]);

        $this->mailer->send($confirmationEmail);
    }
}
```

Very nice! Let's celebrate by cleaning things up in `UserManager`: we can remove the last two arguments of the constructor - `$router` and `$mailer` - their properties... and even some `use` statements on top.

38 lines | src/Manager/UserManager.php

```
... lines 1 - 8
class UserManager
{
... lines 11 - 13
    public function __construct(UserPasswordEncoderInterface $passwordEncoder, EntityManagerInterface $entityManager)
    {
... lines 16 - 17
    }
... line 19
    public function create(User $user, string $plainPassword): void
    {
        $token = $this->createToken();
        $user->setConfirmationToken($token);

        $user->setPassword(
            $this->passwordEncoder->encodePassword($user, $plainPassword)
        );

        $this->entityManager->persist($user);
        $this->entityManager->flush();
    }
... lines 32 - 36
}
```

Who Should Generate the Confirmation Token?

Done! Now... let's see... who should be responsible for creating and setting the confirmation token on the User? I'm... not exactly sure. But let's *invert* that question: who should *not* be responsible for creating the token?

That's a bit easier: it *probably* doesn't make sense for the service whose only responsibility is creating an email... to *also* be responsible for generating this cryptographically-secure token and saving it to the database. Yes, this service *does* deal with the confirmation link... but it feels like that logic would change for very different reasons than the email itself.

So if we discard `ConfirmationEmailSender` from our options, then there's only one logical place left `UserManager::create()`. And... it makes sense: this method sets up new `User` objects with *all* the data they need and then saves them. You *could* also choose to isolate the confirmation token creation logic into a *third* class... there's no right or wrong answer, which is what makes this stuff so darn tricky! But over optimizing, by splitting things into *too* many pieces, is also something that we do *not* want to do. We'll talk more about that in the next chapter.

Anyways, now that we've split all of our code into two places, over in `RegistrationController`, we need to call both methods. Autowire a new argument into the method: `ConfirmationEmailSender $confirmationEmailSender`. Then, below, right after we call `$userManager->create()`, say `$confirmationEmailSender->send()` and pass the `$user` object.

66 lines | src/Controller/RegistrationController.php

```
... lines 1 - 13

class RegistrationController extends AbstractController
{
... lines 17 - 19
    public function signup(Request $request, UserManager $userManager, ConfirmationEmailSender $confirmationEmailSender)
    {
... lines 22 - 24
        if ($form->isSubmitted() && $form->isValid()) {
... lines 26 - 32
            $userManager->create($user, $plainPassword);
            $confirmationEmailSender->send($user);
... lines 35 - 38
        }
... lines 40 - 43
    }
... lines 45 - 64
}
```

Done! Our original feature - sending a confirmation email -is now implemented in a more SRP-friendly way.

Creating a "Takes Care of Everything" Service?

By the way, if you *don't* like that you need to call two methods whenever you're registering a new user...I kind of agree! And it's no problem: you could extract these two calls into a *new* class... maybe called `UserRegistrationHandler`.

It's *one* responsibility would be to "orchestrate" all the tasks related to registering a user. This is just *one* responsibility - not many - because it's not actually *doing* any of the real work. So, for example, if we needed to make a change to the confirmation email... or change how users are persisted to the database...neither of those would require us to need to modify this new class. The new class would only change if we added some new "step" to user registration like sending an API call to our newsletter service.

Enjoying SRP: Adding the Resend Feature

Anyways, now that we've refactored to be SRP-compliant, we get to enjoy our hard work by *finally* adding the new feature that our team asked for: the ability to resend a confirmation email.

If you downloaded the course code from this page, you should have a `tutorial/` directory with a `ResendConfirmationController` file inside. Copy this, go up to the `Controller/` directory... and paste. This comes with the boilerplate needed for an endpoint that a user could POST to in order to resend their confirmation email.

24 lines | src/Controller/ResendConfirmationController.php

```
... lines 1 - 8
class ResendConfirmationController extends AbstractController
{
    /**
     * @Route("/resend-confirmation", methods={"POST"})
     */
    public function resend()
    {
        $this->denyAccessUnlessGranted('ROLE_USER');
        $user = $this->getUser();

        // TODO: send confirmation email

        return new Response(null, 204);
    }
}
```

But... the actual *sending* of that confirmation email is still a "TODO". Remove that comment, autowire the `ConfirmationEmailSender` service... and then say `$confirmationEmailSender->send($user)`.

25 lines | src/Controller/ResendConfirmationController.php

```
... lines 1 - 4
use App\Service\ConfirmationEmailSender;
... lines 6 - 9
class ResendConfirmationController extends AbstractController
{
... lines 12 - 14
    public function resend(ConfirmationEmailSender $confirmationEmailSender)
    {
... lines 17 - 19
        $confirmationEmailSender->send($user);
... lines 21 - 22
    }
}
```

It's that easy! I won't bother testing this... but I will repeat the words that every developer loves to say: "it should work".

The important thing is that, thanks to our new organization, if, for example,a marketing person *did* want to tweak the subject on our welcome email, we can make that change without messing around near code that saves things to the database or hashes passwords.

But... I have *more* that I want to say about SRP...like the risks of over-optimizing, which violates a concept called cohesion.I also think that, thanks to inspiration from Dan North, there's an easier way to think about SRP.I'll explain all of that next.

Chapter 5: SRP: Takeaways

We decided that the confirmation email functionality and user creation functionality are likely to change for different reasons. And so, we split these two responsibilities into two separate classes.

Over-Separation & Cohesion

Now, I have some questions. Should we separate the password-hashing logic from the user-persistence responsibility? Meaning, should we move it into its own class? And should we treat the confirmation token generation as *its* own responsibility and move *it* somewhere separate?

If you look quickly at SRP, it kinda sounds like the rule is:

> Put every tiny piece of functionality into its own class and method.

But, thankfully, SRP is *not* saying that... that would make our code a disaster! There's another concept called "cohesion". It says:

> Keep things together that are related.

At first, it seems like cohesion and SRP are opposites. I mean, SRP says "separate things" and cohesion says "no, keep things together!". But on closer inspection, SRP and cohesion are two ways of saying the same thing: keep only *related* things together. This is the push-and-pull of SRP: separate things that will change for different reasons... but do *not* separate any further.

Looking at `UserManager`, we're already somewhat protected from changes to the password-hashing functionality, because we rely on a service that's behind an interface: `UserPasswordEncoderInterface`. How that service works could *completely* change and we wouldn't need to update any code in this class. So the risk of that changing in some way that *would* cause us to need to change *this* class is probably very low.

38 lines | src/Manager/UserManager.php

```
... lines 1 - 8
9  class UserManager
10 {
... lines 11 - 13
14     public function __construct(UserPasswordEncoderInterface $passwordEncoder, EntityManagerInterface $entityManager)
15     {
16         $this->passwordEncoder = $passwordEncoder;
... line 17
18     }
... line 19
20     public function create(User $user, string $plainPassword): void
21     {
... lines 22 - 24
25         $user->setPassword(
26             $this->passwordEncoder->encodePassword($user, $plainPassword)
27         );
... lines 28 - 30
31     }
... lines 32 - 36
37 }
```

What about the token generation logic? Well, do we think it's very likely that we might change how our tokens are generated? This... to me feels like a weak candidate to separate. It's already simple: one line of code down here... and two lines of code up here. And it's unlikely change, especially for a reason that's *different* than the other code in this class.

38 lines | src/Manager/UserManager.php

```
... lines 1 - 8
class UserManager
{
... lines 11 - 19
    public function create(User $user, string $plainPassword): void
    {
        $token = $this->createToken();
        $user->setConfirmationToken($token);
... lines 24 - 30
    }
... line 32
    private function createToken(): string
    {
        return rtrim(strtr(base64_encode(random_bytes(32)), '+/', '-_'), '=');
    }
}
```

Overall, my advice is this: don't *over*-anticipate potential future changes.

Write Code that Fits in your Head

At the beginning of this tutorial, I mentioned a blog post by Dan North, the father of behavior-driven development. He has something delightfully refreshing to say about the single responsibility principle. Instead of thinking about possible changes... and organizing things into responsibilities - which *is* tricky - he suggests something more straightforward: write simple code.... using the measuring stick of: "does this code fit in my head?".

I *love* this. If a method or class has too many things in it, then the total logic of that method won't "fit in your head"... and it will be difficult to think about and work with. So, you should separate it into smaller pieces that *do* fit into your head.

On the other hand, if you split the code for registering a user into 10 different classes, that's *also* going to become complex to think about. The overall goal is to create units of code that fit in your head... so that you can have an overall application that *also* "fits in our head".

If you follow this general advice, I think you'll find that you probably create classes and methods that follow SRP pretty nicely... without the stress of trying to perfect it.

Okay, it's time to dive into the next solid principle: the open-closed principle.

Chapter 6: Open–Closed Principle

The second SOLID principle is the Open-Closed Principal. Or OCP. Ready for the super understandable technical definition? Here we go.

Technical and (Less) Technical Definition

> A module should be open for extension, but closed for modification.

As usual - and hopefully you're a bit quicker than I am - this definition makes no sense to me.... at least at first. Let's try our own definition. OCP says:

> You should be able to change what a class does without actually changing its code.

If that sounds crazy... or downright impossible, it's actually not! And we'll learn one common pattern that makes this possible.

But full disclosure, OCP is *not* my favorite SOLID principle. And later, we'll talk about when it should be used and when... maybe it shouldn't. But more on that once we've got a good understanding of what OCP really is.

Updating our Believability Scoring Algorithm

Now, the whole point of Sasquatch Sightings is for people to be able to submit their *own* sightings. To help sort through all of these, we've developed a proprietary algorithm to give each sighting a "believability score". Ooh. How is that implemented?

Open `src/Service/SightingScorer.php`. After you submit a sighting, we call `score()` ... and all the logic lives right in this class. We look at the latitude and longitude, title, and description for certain keywords. We call each of these "scoring factors".

Now, we've received a change request. We need to add a new scoring factor where we look at the *photos* included with the post. The easiest way to implement this would be to go down here, create a new private method called `evaluatePhotos()` ... and then call that from up here in the `score()` method.

But doing that would violate OCP because we would be changing our existing code in order to add the new feature. OCP tells us that a class's behavior should be able to be modified *without* changing its code. How is that even possible?

The truth is that our class *already* violated OCP before we got this change request. To be able to add the new feature without changing our existing code, we needed to write our class differently from its very beginning. Since it's a little late for that, let's walk through the OCP mindset and refactor this class so that it *does* follow the rules.

"Closing" a Class to a Change

First, we need to identify which kind of change we want to "close" this class against. In other words, what kind of change do we want to allow a future developer to be able to make without modifying this class. Based on the change request, we need to be able to add more scoring factors without modifying the `score()` method itself. Since there's no way to do that right now, we're going to change this method in order to "close" it to this change. How? By separating each scoring factor into its own class and injecting them into the `SightingScorer` service.

Step one is to create an interface that describes what each scoring factor should do. In `src/`, for organization, create a new directory called `Scoring/`. And inside of that, choose "new PHP class"... then change this to be an interface... called `ScoringFactorInterface`.

Each factor *should* need only one method. Let's call it `score()`. It will accept the `BigFootSighting` object that it's going to score.... and will return an integer, which will be the amount to *add* to the total score.

11 lines | src/Scoring/ScoringFactorInterface.php

```
... lines 1 - 4
use App\Entity\BigFootSighting;
... line 6
interface ScoringFactorInterface
{
    public function score(BigFootSighting $sighting): int;
}
```

Perfect! You could also add some documentation above this to describe the method of interface better: probably a good idea.

Step two is to create a new class for each scoring factor and make it implement the new interface. For example, copy, `evaluateCoordinates()`, delete it and then go into the `Scoring` directory and create a new class called `CoordinatesFactor`. We'll make it implement `ScoringFactorInterface` ... I'll paste the method - hit okay to add the `use` statements - rename this to `score()` and make it `public`. It already, correctly, returns an integer, so this is done!

25 lines | src/Scoring/CoordinatesFactor.php

```
... lines 1 - 4
use App\Entity\BigFootSighting;

class CoordinatesFactor implements ScoringFactorInterface
{
    public function score(BigFootSighting $sighting): int
    {
        $score = 0;
        $lat = (float)$sighting->getLatitude();
        $lng = (float)$sighting->getLongitude();

        // California edge to edge coordinates
        if ($lat >= 32.5121 && $lat <= 42.0126
            && $lng >= -114.1315 && $lng <= -124.6509
        ) {
            $score += 30;
        }

        return $score;
    }
}
```

Let's repeat this for `evaluateTitle()`. Create a class called `TitleFactor`, implement the `ScoringFactorInterface`, paste, make it `public` and rename it to `score()`.

25 lines | src/Scoring/TitleFactor.php

```
... lines 1 - 4
use App\Entity\BigFootSighting;

class TitleFactor implements ScoringFactorInterface
{
    public function score(BigFootSighting $sighting): int
    {
        $score = 0;
        $title = strtolower($sighting->getTitle());

        if (stripos($title, 'hairy') !== false) {
            $score += 10;
        }

        if (stripos($title, 'chased me') !== false) {
            $score += 20;
        }

        return $score;
    }
}
```

And one more: copy, `evaluateDescription()`, delete that, create our last factor classfor now, which will be `DescriptionFactor`, implement `ScoringFactorInterface` paste in the logic, clean things up...and rename to `score()`.

29 lines | src/Scoring/DescriptionFactor.php

```
... lines 1 - 4
use App\Entity\BigFootSighting;

class DescriptionFactor implements ScoringFactorInterface
{
    public function score(BigFootSighting $sighting): int
    {
        $score = 0;
        $title = strtolower($sighting->getDescription());

        if (stripos($title, 'hairy') !== false) {
            $score += 10;
        }

        if (stripos($title, 'chased me') !== false) {
            $score += 20;
        }

        if (stripos($title, 'using an iPhone') !== false) {
            $score -= 50;
        }

        return $score;
    }
}
```

That looks happy! Now we can work our magic in `SightingScorer`. Add a `__construct()` method that will accept an `array` of scoring factors. I'll hit Alt + Enter and go to "Initialize properties" to create that property and set itAbove the property, I like to add extra PHPDoc so my editor knows this isn't just an array of *anything*, it's an array of `ScoringFactorInterface[]` objects.

31 lines | src/Service/SightingScorer.php

```
... lines 1 - 8
class SightingScorer
{
    /**
     * @var ScoringFactorInterface[]
     */
    private array $scoringFactors;

    public function __construct(array $scoringFactors)
    {
        $this->scoringFactors = $scoringFactors;
    }
... lines 20 - 29
}
```

Down in `score()`, instead of calling each method individually, we can now loop over `$this->scoringFactors` and say `$score += $scoringFactor->score($sighting)`.

31 lines | src/Service/SightingScorer.php

```
... lines 1 - 20
    public function score(BigFootSighting $sighting): BigFootSightingScore
    {
        $score = 0;
        foreach ($this->scoringFactors as $scoringFactor) {
            $score += $scoringFactor->score($sighting);
        }

        return new BigFootSightingScore($score);
    }
... lines 30 - 31
```

That's it! Our SightingScorer is now *closed* to one type of change that we may need to make in the future: adding scoring factors. In other words, we can now add *new* scoring factors, *without* modifying this method.

Wiring the $scoringFactors Argument

Yaaay! But... on a technical level, this won't work yet. At your browser, click to submit a new sighting. Instant error! Of course. This isn't really related to OCP, but Symfony doesn't know what to *pass* for the new `$scoringFactors` argument.

Next, let's look at two ways to fix this: the simple way... and the fancier way, which involves a tagged iterator. After, we'll look at some takeaways for the open-closed principle.

Chapter 7: OCP: Autoconfiguration & tagged_iterator

When we went to the "submit" page, we got this gigantic error. It's the middle that's most relevant:

> Cannot autowire service `SightingScorer`, argument `$scoringFactors` of method `__construct` is type-hinted array. You should configure its value explicitly.

That makes sense! We haven't told Symfony what to pass to the new argument of `SightingScorer`.

Manually Wiring the Argument

What *do* we want to pass there? An array of all of our "scoring factor" services. The simplest way to do that is to configure it manually in `config/services.yaml`. Down at the bottom, we want to configure the `App\Service\SightingScorer` ... service and we want to control its `arguments:`, specifically this `$scoringFactors` argument. Copy that, paste, and this will be an array: I'll use the multi-line syntax. Each entry in the array with be one of the scoring factor services. So `@App\Scoring\TitleFactor`, copy that, paste... fix the indentation... then pass `DescriptionFactor` and `CoordinatesFactor`.

39 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 32
    App\Service\SightingScorer:
        arguments:
            $scoringFactors:
                - '@App\Scoring\TitleFactor'
                - '@App\Scoring\DescriptionFactor'
                - '@App\Scoring\CoordinatesFactor'
```

This will now pass an array with these three service objects inside.

Try it again. Refresh and... the error is gone... and now it kicked us to the log-in page. Copy the email above, enter the password, hit "sign in" and... beautiful! The page loads. Let's give it a try. Fill in the details of your most recent interaction with Bigfoot. Oh, but before I submit this, I'm going to add some keywords to the description that I know our scoring factor is looking for.

Submit and... it works! Ah man, a believability score of only 10!? I really thought that was a Bigfoot.

Enabling Autoconfiguration

Before we talk more about OCP, on a technical, Symfony level, there is one other way to inject these services. It's called a "tagged iterator"... and it's a pretty cool idea. It's also commonly used in the core of Symfony itself.

Open up `src/Kernel.php`. I know, we almost never open this file. Inside, go to Code -> Generate, or Command + N on a Mac, and select Override methods. Override one called `build()` ... let me find it. There it is.

This is a hook where we can do extra processing on the container while it's being built. The parent method is empty... but I'll leave the parent call. Add `$container->registerForAutoconfiguration()`, pass this `ScoringFactorInterface::class`, then `->addTag('scoring.factor')`.

49 lines | src/Kernel.php

```
... lines 1 - 4
use App\Scoring\ScoringFactorInterface;
... lines 6 - 11
class Kernel extends BaseKernel
{
... lines 14 - 40
    protected function build(ContainerBuilder $container)
    {
        parent::build($container);

        $container->registerForAutoconfiguration(ScoringFactorInterface::class)
            ->addTag('scoring.factor');
    }
}
```

Thanks to this, any autoconfigurable service, which is all of our services, that implements `ScoringFactorInterface`, will automatically be tagged with `scoring.factor`. That `scoring.factor` is a name that I *totally* just made up.

This line, on its own, won't make any real change. But now, back in `services.yaml` we can simplify: set the `$scoringFactors` argument to a special YAML syntax: `!tagged_iterator scoring.factor`.

36 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 32
    App\Service\SightingScorer:
        arguments:
            $scoringFactors: !tagged_iterator scoring.factor # Inject all services tagged with "scoring.factor"
```

This says: please inject all services that are tagged with `scoring.factor`. So autoconfiguration adds the tag to our scoring factor services... and this handles passing them in. Pretty cool, right?

The only gotcha is that we need to change the type-hint in `SightingScorer` to be an `iterable`. This won't pass us an array... but it *will* pass us something that we can `foreach` over. As a bonus, it's a "lazy" iterable: the scoring factor services won't be instantiated until and unless we run the `foreach`. Oh, and change the property type to `iterable` also.

31 lines | src/Service/SightingScorer.php

```
... lines 1 - 8
class SightingScorer
{
... lines 11 - 13
    private iterable $scoringFactors;
... line 15
    public function __construct(iterable $scoringFactors)
... lines 17 - 18
    }
... lines 20 - 29
}
```

Next: now that we understand the type of change that OCP wants us to make to our code, let's talk about why we should care - or not care - about OCP and when we should and should not follow it.

Chapter 8: OCP: Takeaways

The big thing that OCP wants us to take away from this conversation is this: try to imagine the future changes you are most likely to need to make, and architect, your code so that you will be able to make those changes without modifying existing classes.

OCP Design Patterns

We showed one common pattern to do this: by injecting an array or - iterable - of services instead of hardcoding all the logic right inside the class. There are also other patterns that you can use to accomplish OCP, including the "strategy pattern" - which is similar to what we did, but where you allow just *one* service to be passed in to handle some work - and the template method pattern. All of these are different flavors of the same thing: allowing functionality to be passed *into* a class, instead of living *inside* the class.

OCP is Never Fully Achievable

But the truth is, I don't love OCP. And I've got three reasons. First, even Uncle Bob - the father of the SOLID principles - knows that OCP is a "lie". OCP promises that, if you follow it correctly, you will *never* need to mess around with your old code. But a system can't be 100% OCP-compliant. Our `SightingScorer` class is "closed" against the change of "adding new scoring factors". But what would happen if we suddenly needed a scoring factor to be able to *multiply* the existing score by a number... instead of just adding *to* it.

31 lines | src/Service/SightingScorer.php

```
... lines 1 - 8
class SightingScorer
{
... lines 11 - 20
    public function score(BigFootSighting $sighting): BigFootSightingScore
    {
        $score = 0;
        foreach ($this->scoringFactors as $scoringFactor) {
            $score += $scoringFactor->score($sighting);
        }

        return new BigFootSightingScore($score);
    }
}
```

This unexpected change would require us to, yup, modify the code in `SightingScorer`. If we had anticipated this change, we could have added an abstraction to `SightingScorer` to protect us from this new kind of change. But no one can perfectly predict the future: we can do our best... but often, we'll be wrong.

Unnecessary Abstractions add Complexity

Of course, just because a principle isn't perfect doesn't meant we should never use it. But that leads me to the second reason that I don't love OCP: It creates unnecessary abstractions... which make our code harder to understand.

`SightingScorer` is now closed against new scoring factors, which means we can add new scoring factors to our system without modifying the class. But at what cost? I can no longer open up this class and quickly understand how the believability score is calculated. Now I need to dig around to figure out which factors are injected... then go look at each individual factor class.

If you have a large team, being able to separate things into smaller pieces like this becomes more desirable. But, for example here at SymfonyCasts - with our brave team of about four - we would probably *not* make this change. It adds misdirection to our code, with a limited benefit.

Changing Code is... Ok!

And that leads me to my third and final reason for not loving OCP. And this one comes from Dan North's blog post.

He argues that the open-closed principle comes from an era when changes were expensive because of the need to compile a code, the fact that we hadn't really mastered the science of refactoring code yet, and because version control was done with CVS, which according to him, added to a mentality of wanting to make changes by adding *new* code, instead of modifying *existing* code.

In other words... OCP is a dinosaur! Dan's advice, which I agree with, is quite different than OCP. He says:

> If you need code to do something else, change the code to make it do something else.

Quoting Dan, he says:

> Code is not an asset to be carefully shrink-wrapped, and preserved, but a cost, a debt. All code is cost. So if I can take a big pile of existing code and replace it with smaller, more specific costs, than I'm winning at code.

I love that.

So how do I *personally* navigate OCP in the real world? It's pretty simple. If I'm building an open source library where the people who use my code will *literally* not be able to modify it, then I *do* follow a pattern like we used in SightingScorer whenever I identify a change that a user might need to make. This gives my users the ability to *make* that change... without modifying the code in the class... which would be impossible for them.

But if I'm coding in a private application, I'm *much* more likely to keep all the code right inside the class. But this is *not* an absolute rule. Separating the code makes it easier to unit test and can help us follow the advice from SRP: writing code that "fits in your head". Larger teams will also probably want to split things more readily than smaller teams. As with all the SOLID principles, do your best to write simple code and... don't overthink it.

Next, let's turn to SOLID principle number three: the Liskov Substitution Principle.

Chapter 9: Liskov Substitution Principle

Solid principle number three is, I think, a pretty cool one. It's the Liskov Substitution Principle, developed by Barbara Liskov: a researcher at MIT and winner of the Turing award, which is, I've learned, sort of the Nobel prize for computer science. No biggie.

Liskov Defined

Liskov's principle states:

> Subtypes must be substitutable for their base types.

That's... actually not a terrible definition. A "subtype" basically means a class: any class that extends a base class *or* that implements an interface.

So let me rephrase the definition. I'm going to stick to just talking about classes and parent classes, but this applies equally to a class that implements an interface. Here it is:

> You should be able to substitute a class for a sub-class without breaking your app or needing to change any code.

Dan North refers to this as simply:

> The principle of least surprise, applied to classes that have a parent class or implement an interface.

In other words, a class should behave in a way that most users expect it should behave like its parent class or interface *intended*.

Okay, that sounds great! But... what does that mean *specifically*?

The 4 Aspects that (Mostly) Define Liskov

It means four specific things. Pretend that we have a class that extends a base class or implements an interface. It also has a protected property and a method, both of which live in that parent class. Or in the case of the method, it lives on the interface.

Given this setup, Liskov says 4 things.

One: you cannot change the *type* of a protected property.

Two: you can't *narrow* the type hint of an argument. Like, if the parent class uses the `object` type-hint, you can't make this *narrower* in your subclass by requiring something more *specific*, like a `DateTime` object.

Three, which is both similar and *opposite* to the previous rule, you can't *widen* the *return* type. If the parent class says a method returns a `DateTime` object, you can't change this in the subclass to suddenly return something *wider*, like *any* object.

And finally, four, you should follow your parent class's - or interface's - rules around whether or not you should throw an exception under certain conditions.

There may be some edge-case things that I've missed with these 4 rules, but this is the basic idea. By violating any of these rules, you are making your class behave *differently* than its parent class or interface *intended*. That's bad because if part of your code expects an instance of that interface and you pass in your class, even though it implements the interface, the class's violations may cause weird stuff to happen. We'll see *specific* examples of this over the next few chapters.

Now here's what I really *love* about this principle. Those first three rules? Yeah, they're *impossible* to violate in PHP. If you change the property type on a protected property, narrow the type-hint on an argument or widen a return type on a method, PHP will give you a syntax error. Yup, Liskov's principle makes *so* much sense, that its rules are codified right into the language.

So, we now know the rules of Liskov.But to get a deeper feeling for *why* these rules exist and - almost more importantly -what things we *are* allowed to do in a "subtype", let's jump into two real-world examples next.

Chapter 10: Liskov: Unexpected Exceptions

Let's jump into our first example where we learn how we can violate the Liskov principle! And... maybe more importantly, why... that's not such a great idea.

Creating a new Scoring Factor

In the `src/Scoring/` directory, create a new scoring factor class called `PhotoFactor` ... and make it implement the `ScoringFactorInterface`. We'll finally fulfill the change request we received earlier: to add a scoring factor that reads the images for each sighting.

23 lines | src/Scoring/PhotoFactor.php

```
... lines 1 - 6
class PhotoFactor implements ScoringFactorInterface
{
... lines 9 - 21
}
```

Thanks to our work with the open-closed principle, we can now add this scoring factor without touching `SightingScorer`. And to be extra cool, thanks to this `tagged_iterator` thing in `services.yaml`, the new `PhotoFactor` service will be instantly passed into `SightingScorer`. Yay!

In `PhotoFactor`, go to Code -> Generate - or Command + N on a Mac - and select "Implement Methods" to generate the `score()` method. Inside, I'll paste some code.

23 lines | src/Scoring/PhotoFactor.php

```
... lines 1 - 8
    public function score(BigFootSighting $sighting): int
    {
        if (count($sighting->getImages()) === 0) {
            throw new \InvalidArgumentException('Invalid BigFootSighting, it should have at least one photo');
        }

        $score = 0;
        foreach ($sighting->getImages() as $image) {
            $score += rand(1, 100); // todo analyze image
        }

        return $score;
    }
... lines 22 - 23
```

This is pretty simple: we loop over the images...and pretend that we're analyzing them in some super advanced way. Shh, don't tell our users. Oh, and if there are no images for this sighting, we throw an exception.

Cool! Let's try it. Go back to our homepage, click to add a new post and fill in some details! I'll leave images empty for simplicity. And... ah! A 500 error! That's our new exception! We broke our app! And it broke because we violated Liskov's principle! She tried to warn us!

Our new scoring factor class - or subtype - to use the more technical word just did something unexpected: it threw an exception!

The Ugly Work-Around

One way to fix this, which might seem silly...but there's a reason we're doing this...is to add some conditional code inside of `SightingScorer`. If `PhotoFactor` doesn't like sightings with zero images, let's just skip that factor when that happens!

Inside the `foreach`, if `ScoringFactor` is an `instanceof PhotoFactor` and count of `$sighting->getImages()` equals zero, then `continue`.

37 lines | src/Service/SightingScorer.php

```
... lines 1 - 5
use App\Scoring\PhotoFactor;
... lines 7 - 8

class SightingScorer
... lines 11 - 21
    public function score(BigFootSighting $sighting): BigFootSightingScore
    {
... line 24
        foreach ($this->scoringFactors as $scoringFactor) {
            // LSP violation and also OCP violation
            if ($scoringFactor instanceof PhotoFactor && count($sighting->getImages()) === 0) {
                continue;
            }
... lines 30 - 31
        }
... lines 33 - 34
    }
}
```

In addition to this *not* being the best way to fix this - more on that in a minute this also violates the open-closed principle. But... it *does* fix things: if we resubmit the form...our app works again!

Exceptions are a "Soft" Part of an Interface

But... let's back up. Open `ScoringFactorInterface`. Unlike argument types and return types, there's no way in PHP to*codify* whether or not a method should throw an exception or which types of exceptions should be usedBut this *can*, at least, be described in the documentation above the method... which we totally skipped!

Let's fill that in. We don't need the `@return` or `@param` because they're redundant...unless we want to add some more information about their meaning. I'll add a quick description... and then let's be very clear about the exception behavior we expect:

> This method should not throw an exception for any normal reason.

16 lines | src/Scoring/ScoringFactorInterface.php

```
... lines 1 - 6
interface ScoringFactorInterface
{
    /**
     * Return the score that should be added to the overall score.
     *
     * This method should not throw an exception for any normal reason.
     */
    public function score(BigFootSighting $sighting): int;
}
```

In the real-world, if a method *is* allowed to throw an exceptionwhen some expected situation happens, you would typically see an `@throws` that describes that. And if you *don't* see that, you can assume that you are *not* allowed to throw an exception for any normal situation.

Our Class Behaves Unexpectedly

Anyways, now that we've clarified this,it's easy to see that our `PhotoFactor` breaks Liskov's principle: `PhotoFactor` behaves in a way that the class that uses it - `SightingScorer`, sometimes called the "client class" - was not expecting.That "bad behavior" caused us to need to hack in this code to get it to work.

Another way to think about it, which explains why this is called Liskov's*substitution* principle, is that, if any of our code relies on a `ScoringFactorInterface` object - like `DescriptionFactor` - we could *not* "replace" or "substitute" that object for our `PhotoFactor` without breaking things.

If this substitution aspect doesn't make complete sense yet, don't worry. Our next example will illustrate it even better.

instanceof Checks Indicate Liskov Violation

So: we violated Liskov's principle by throwing an exception. And then, I lazily worked around the problem by adding some `instanceof` code to `SightingScorer` ... to *literally* work "around" the problem.

When you have an `instanceof` conditional like this, it's often a signal that you're violating Liskov because it means that you have a specific implementation of a class or interface that is behaving *differently* than the rest... which you then need to code for.

So let's remove this: take out the if statement and let's even go clean out the extra `use` statement on top.

31 lines | src/Service/SightingScorer.php

```
... lines 1 - 8
class SightingScorer
{
... lines 11 - 20
    public function score(BigFootSighting $sighting): BigFootSightingScore
    {
        $score = 0;
        foreach ($this->scoringFactors as $scoringFactor) {
            $score += $scoringFactor->score($sighting);
        }

        return new BigFootSightingScore($score);
    }
}
```

Now that we've clarified that the `score()` method should *not* throw an exception in normal situations, the real fix is... kinda obvious: stop throwing the exception! Replace the exception with `return 0` .

23 lines | src/Scoring/PhotoFactor.php

```
... lines 1 - 6
class PhotoFactor implements ScoringFactorInterface
{
    public function score(BigFootSighting $sighting): int
    {
        if (count($sighting->getImages()) === 0) {
            return 0;
        }
... lines 14 - 20
    }
}
```

That's it. The class now acts like we expect: no surprises.

By the way, all of this does does not mean that it is *illegal* for our `score()` method to *ever* throw an exception. If the method, for example, needed to query a database... and the database connection was down... then yeah! You should totally throw an exception! That is an *unexpected* situation. But for all the, expected, normal cases, we should follow the rules of our parent class or interface.

Next let's look at one more example of Liskov's principle where we create a subclass of an existing class... then secretly substitute it into our system without breaking anything. Liskov would be so proud!

Chapter 11: Liskov: Substituting a Class

Our highly-advanced, proprietary, believability score system is having some performance problems. To help debug it, we want to measure how long calculating a score takes. The simplest way to implement this would be almost entirely inside `SightingScorer`. We could set a start time on top, then use that down here to calculate a duration. And then we could pass that `$duration` into the `BigFootSightingScore` class. Hold Command or Ctrl and click to open it: it's in the `src/Model/` directory. Inside here, we could create a new property called `$duration` ... with a getter so that we could use that value.

Lets: Substitute a Class!

But... let me undo that. Let's make things more interesting! To keep our application as *skinny* as possible on production, I only want to run this new timing code when we're in Symfony's `dev` environment. And yes, we *could* inject some `$shouldCalculateDuration` value into `SightingScorer` *based* on the environment and use *it* to determine if we should do that work.

But, in the spirit of Liskov, instead of *changing* `SightingScorer`, I want to create a *subclass* that does the timing and *substitute* that class into our system as the `SightingScorer` service.

It's gonna be kinda fun! And it's a pattern you'll find inside Symfony itself, like with the `TraceableEventDispatcher`: a class that is substituted in for the *real* event dispatcher only while developing. It adds debugging info. Well, *technically*, that class uses *decoration* instead of being a subclass. That's a different, and usually better design pattern when you want to *replace* an existing class. But, to really understand Liskov, we'll use a subclass.

Creating the Subclass

Let's start by creating that new subclass. Over in the `Service/` directory... so that it's right next to our normal `SightingScorer`, add a new class called `DebuggableSightingScorer`. Make it extend the normal `SightingScorer`.

9 lines | src/Service/DebuggableSightingScorer.php

```
... lines 1 - 4
class DebuggableSightingScorer extends SightingScorer
{

}
```

Since our subtype is currently making *no* changes to the parent class, Liskov would definitely be happy with it. What I mean is: we should *definitely* be able to *substitute* this class into our app in place of the original, with no problems.

Substituting the Real Class

But where *is* the normal `SightingScorer` service actually used? Open `src/Controller/BigFootSightingController.php`. This `upload()` action is the one that is executed when, from the homepage, we click to submit a sighting. Yep, down here, you can see that this is the `upload()` method.

58 lines | src/Controller/BigFootSightingController.php

```
... lines 1 - 13
class BigFootSightingController extends AbstractController
{
... lines 16 - 19
    public function upload(Request $request, SightingScorer $sightingScorer, EntityManagerInterface $entityManager)
    {
        $form = $this->createForm(BigFootSightingType::class);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
... lines 26 - 40
        }

        return $this->render('main/sighting_new.html.twig', [
            'form' => $form->createView()
        ]);
    }
... lines 47 - 56
}
```

One of the arguments that's being autowired to this method is the `SightingScorer` ... which is used down here on submit to calculate the score.

Now I want to change this service to use our new class: I want to substitute it. How? Open `config/services.yaml`. I mentioned earlier that we were going to swap in our `DebuggableSightingScorer` *only* in the `dev` environment. But to keep things simple, I'm *actually* going to do it in *all* environments. If you *did* want to have this only affect your `dev` environment, you could make the same changes we're about to make in a `services_dev.yaml` file.

Anyways, to suddenly start using our new class everywhere that the `SightingScorer` is used, add `class:` and then `App\Service\DebuggableSightingScorer`.

37 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 32
    App\Service\SightingScorer:
        class: App\Service\DebuggableSightingScorer
... lines 35 - 37
```

I know, this looks a little funny. This first line is still the service id. But now instead of using that as the class, Symfony will use `DebuggableSightingScorer`. The end result is that whenever someone autowires `SightingScorer` - like we do in our controller - Symfony will instantiate an instance of our `DebuggableSightingScorer` ... and pass the normal `$scoringFactors` argument. Yep, we just substituted our subclass into the system!

To prove it, find your terminal and run:

```
php bin/console debug:container Sighting
```

I want to look at the `SightingScorer` service, so I'll hit 5. And... perfect! The service id is `App\Service\SightingScorer`, but the class is `App\Service\DebuggableSightingScorer`.

Another way to show this would be to go into our `BigFootSightingController` and temporarily `dd($sightingScorer)`.

Back at your browser, refresh and... there it is! `DebuggableSightingScorer`

Let's go take that out... then refresh again. The page works and... even though I won't test it, if we submitted, our `DebuggableSightingScorer` *would* correctly calculate the believability score.

In other words, no surprise: if you create a subclass and change *nothing* in it, you *can* substitute that class for its parent class. It follow's Liskov's principle.

Method Changes that are NOT Allowed

Let's start adding our timing mechanism. In the class, go to Code -> Generate - or Command + N on a Mac - select "Override methods" and override the `score()` method. If you override a method and keep the same argument type hints and return type, this class is *still* substitutable: I can refresh and PHP is still happy.

15 lines | src/Service/DebuggableSightingScorer.php

```
... lines 1 - 4
use App\Entity\BigFootSighting;
use App\Model\BigFootSightingScore;

class DebuggableSightingScorer extends SightingScorer
{
    public function score(BigFootSighting $sighting): BigFootSightingScore
    {
        return parent::score($sighting);
    }
}
```

But if we *did* change the argument type-hints or return type to something totally *different*, then even PHP will tell us to knock it off. For example, let's completely change the return type to `int`.

15 lines | src/Service/DebuggableSightingScorer.php

```
... lines 1 - 9
    public function score(BigFootSighting $sighting): int
    {
        return parent::score($sighting);
    }
... lines 14 - 15
```

PhpStorm is mad! And if we refresh, PHP is mad too!

> `DebuggableSightingScorer::score()` must be compatible with the parent `score()`, which returns `BigFootSightingScore`.

Our signature is incompatible and, nicely, PHP does *not* let us violate Liskov's principle in this way. Go and undo that change.

So does this mean that we can *never* change the return type or argument type-hints in a subclass? Actually... no! Remember the rules from earlier: you *can* change a return type if you make it more *narrow*, meaning more *specific*. And you can *also* change an argument type-hint... as long as you make it accept a *wider*, or *less* specific type.

Let's see this in action by finishing our timing feature next.

Chapter 12: Liskov: What Changes *Are* Allowed?

Calculating how long it takes for the parent `score()` method to execute will be easy. But then... what do we *do* with that number? This method returns a `BigFootSightingScore` instance.... so we can't suddenly change this to return an `int` for the duration. How can this method return both the `BigFootSightingScore` *and* info about how long it took for the score to calculate?

Creating a Subclass for the Return Value

The answer is: create another subclass! A subclass of `BigFootSightingScore` that holds the extra info. `BigFootSightingScore` lives in the `src/Model/` directory: there it is. Right next to it, add a new class called, how about, `DebuggableBigFootSightingScore`. Make it extend the normal `BigFootSightingScore`.

9 lines | src/Model/DebuggableBigFootSightingScore.php

```
... lines 1 - 4
class DebuggableBigFootSightingScore extends BigFootSightingScore
{

}
```

Now we have two subclasses to play with! This time, override the constructor: do that by going to Code -> Generate - or Command + N on a Mac. Override `__construct()`.

This calls the parent constructor with the score, which is great! Add a new argument: `float $calculationTime`. I'll hit Alt + Enter and go to "Initialize properties"... select just `$calculationTime` ... to create that property and set it. To make the `$calculationTime` accessible, at the bottom, go back to Code -> Generate and make a "getter" method for this!

21 lines | src/Model/DebuggableBigFootSightingScore.php

```
... lines 1 - 4
class DebuggableBigFootSightingScore extends BigFootSightingScore
{
    private float $calculationTime;

    public function __construct(int $score, float $calculationTime)
    {
        parent::__construct($score);

        $this->calculationTime = $calculationTime;
    }

    public function getCalculationTime(): float
    {
        return $this->calculationTime;
    }
}
```

Wait: Does __construct need to Follow Liskov's Rules?

By the way, adding a required argument to a method that you are overriding - like we're doing in `__construct` - is normally *another* way to violate Liskov's principle. Let's think about it using a different example: `SightingScorer`. When we use this, we can normally call `score()` and pass it a single argument. If we suddenly substituted in a *different* class whose `score()` method required *two* arguments... well, that would make our code explode. That new class would *not* be substitutable for the old one.

However, the constructor does *not* need to follow Liskov's principle... which took me a minute to wrap my head around. Why not? Because if you are instantiating a `DebuggableBigFootSightingScore` - with `new DebuggableBigFootSightingScore` - then you know *exactly* which class you are instantiating. And so, you can figure out *exactly* which arguments you need to pass.

This is different than being *passed* a `BigFootSightingScore` object... where the *true* class might be a *subclass*. In that situation, you need any of the methods that you *call* on that object to behave like the *original* class's methods. Since the constructor is

never called on an object, that's not an issue.

Anyways, back in `DebuggableSightingScorer`, let's return our new `DebuggableBigFootSightingScore` class with a dummy duration. Say `$bfScore = parent::score()` ... and then return a `new DebuggableBigFootSightingScore` passing the `int` score - `$bfScore->getScore()` - and `100` for a fake duration. Let's also advertise that we return this new class: `DebuggableBigFootSightingScore`

21 lines | src/Service/DebuggableSightingScorer.php

```
... lines 1 - 6
use App\Model\DebuggableBigFootSightingScore;
... line 8
class DebuggableSightingScorer extends SightingScorer
{
    public function score(BigFootSighting $sighting): DebuggableBigFootSightingScore
    {
        $bfsScore = parent::score($sighting);

        return new DebuggableBigFootSightingScore(
            $bfsScore->getScore(),
            100
        );
    }
}
```

But wait: we just changed the return-type to something *different* than our parent class! Is that allowed?

Narrower Return Types are Allowed

Find your browser, refresh and... PHP totally *does* allow this! That's because this *does* follow Liskov's principle: we are making the return type more *narrow*... or more specific.

But why is making a return type more *narrow* allowed? Look at `BigFootSightingController`: the class that uses the `SightingScorer`. This code requires a `SightingScorer` instance. And so, when we call the `score()` method later, we know that it will return a `BigFootSightingScore` object. We know that because, if we jump into the `SightingScorer` class, yep! The `score()` method returns a `BigFootSightingScore`.

And so, we know the `$bfsScore` variable is an instance of `BigFootSightingScore` ... and we know that *that* class has a `getScore()` method on it. I'll, once again, jump into the class. This is the original `BigFootSightingScore` and here is its `getScore()` method. We use that in our controller to get the integer score and... everything is happy!

But *now* we know that we have *substituted* the `SightingScorer` for a `DebuggableSightingScorer` ... and we know that *its* `score()` method returns a `DebuggableBigFootSightingScore`. But that's okay! Why? Because `DebuggableBigFootSightingScore` extends `BigFootSightingScore`. So we are *still* returning a `BigFootSightingScore` instance, which, of course, *still* has a `getScore()` method. The fact that we return a subclass... that potentially has extra methods on it, does *not* break its substitutability.

But if we had changed its return type to something *less* specific, like *any* object, then there would be no guarantee that what we return from this method has a `getScore()` method. And so, that *would* break Liskov's principle. PHP would be *so* mad at us, that it would generate a syntax error. Let's undo that.

We won't talk about it in detail, but the same philosophy can be applied to argument types, but in the opposite direction. It's okay to change an argument type as long as you support at *least* the original type. It's not okay to be *more* restrictive with the type you allow, but it *is* okay to be *less* specific: I *am* allowed to say that the `score()` method supports *any* object. Well, in *this* example, that would be problematic because we're passing the argument to the parent class...which still *does* require a `BigFootSighting` ... but in general, allowing for a *less* specific, or *wider* argument type *is* allowed by Liskov. And you can see this if we refresh: no syntax error from PHP.

Let's change that back.

Next: it's time to celebrate our new system by *using* the new duration value, tweaking a few things in Symfony's config and listing the takeaways from Liskov's principle.

Chapter 13: Liskov Takeaways & Service Alias

To celebrate our new system, let's see it in action.In `BigFootSightingController`, after the `addFlash()`, let's also add some duration information. But since we don't know for sure if we're using the "debuggable" version of the service,add if `$bfsScore` is an instance of `DebuggableBigFootSightingScore`, then `$this->addFlash('success', sprintf(...))` with:

> Btw, the scoring took %f milliseconds

Passing `$bfsScore->getCalculationTime()` times 1000 to convert from microseconds to milliseconds.

66 lines | src/Controller/BigFootSightingController.php

```
... lines 1 - 6
use App\Model\DebuggableBigFootSightingScore;
... lines 8 - 14
class BigFootSightingController extends AbstractController
... lines 16 - 20
    public function upload(Request $request, SightingScorer $sightingScorer, EntityManagerInterface $entityManager)
    {
... lines 23 - 25
        if ($form->isSubmitted() && $form->isValid()) {
... lines 27 - 38
            if ($bfsScore instanceof DebuggableBigFootSightingScore) {
                $this->addFlash('success', sprintf(
                    'Btw, the scoring took %f milliseconds',
                    $bfsScore->getCalculationTime() * 1000
                ));
            }
... lines 45 - 48
        }
... lines 50 - 53
    }
... lines 55 - 64
}
```

Cool! But... wait: didn't I say that `instanceof` is a signal that we may be breaking Liskov's principle?Yep! But I'm not too worried about it here, for a few reasons. First, this is my controller... whose job is to tie all the ugly pieces of my app together.And second, I'm using the `instanceof` to detect if I can *add* functionality... not to work-around a misbehaving subclass.

However, another solution, depending on if you really *do* need to substitute this class only in one environment, is to explicitly say that you require the debuggable version of the service. So instead of saying, "I allow any `SightingScorer`", we could say, "I specifically need a `DebuggableSightingScorer`".

If we did that, we wouldn't need the `instanceof` because we would know that *that* service returns a `DebuggableBigFootSightingScore`, which has the `getCalculationTime()` method on it.

65 lines | src/Controller/BigFootSightingController.php

```
... lines 1 - 21
22     public function upload(Request $request, DebuggableSightingScorer $sightingScorer, EntityManagerInterface $entityManager)
23     {
... lines 24 - 26
27         if ($form->isSubmitted() && $form->isValid()) {
... lines 28 - 39
40             $this->addFlash('success', sprintf(
41                 'Btw, the scoring took %f milliseconds',
42                 $bfsScore->getCalculationTime() * 1000
43             ));
... lines 44 - 47
48         }
... lines 49 - 52
53     }
... lines 54 - 65
```

But... we're missing one tiny config detail in Symfony. Try to refresh the page. It breaks!

> Cannot autowire service `DebuggableSightingScorer`: argument $scoringFactors is type-hinted `iterable`. You should configure its value explicitly.

Wait... we hit this error when we worked on the open-closed principle. And, in `config/services.yaml`, we fixed it by specifically wiring the `$scoringFactors` argument. Why isn't that working anymore?

Thanks to auto-registration - the feature that automatically registers all classes in `src/` as a service - there is a *separate* service in our container called `DebuggableSightingScorer`. You can see it if you run:

```
php bin/console debug:container Sighting
```

Yup! There's a `DebuggableSightingScorer` service and a *separate* service for `SightingScorer`. This is... *not* what we want. Really, I want Symfony to pass us the *same* service, regardless of whether we type-hint `DebuggableSightingScorer` or `SightingScorer`.

We can do that by adding an alias. Inside `services.yaml`, say `App\Service\DebuggableSightingScorer`, colon, an `@` symbol and then `App\Service\SightingScorer`.

39 lines | config/services.yaml

```
... lines 1 - 7
8  services:
... lines 9 - 32
33     App\Service\DebuggableSightingScorer: '@App\Service\SightingScorer'
... lines 34 - 39
```

This says: whenever someone tries to autowire or use the `DebuggableSightingScorer` service, you should *actually* pass them the `SightingScorer` service... which, I know, is actually an *instance* of the `DebuggableSightingScorer` class. It *can* be a bit confusing.

Back at your terminal, run `debug:container` again:

```
php bin/console debug:container Sighting
```

It *looks* like there are still 2 services, but if you hit "6" to look at the "Debuggable" one, on top, it says:

> This is an alias for the service `App\Service\SightingScorer`.

And over in the browser, when we refresh...it works again!

Liskov Principle Takeaways

So the big takeaway from Liskov's principle is this: make sure that when you have a "subtype" - a class that extends another or that implements an interface - it follows the rules of that parent type. It doesn't do anything surprising. That's it. And PHP even prevents us from *most* Liskov violations.

The most interesting part of Liskov for *me* is learning about the things that we *are* allowed to do. Like, you *are* allowed to change the return type of a method as long as you make it more *specific*. Or, the opposite for argument types: you can change them... as long as you make them *less* specific.

Okay, next up is solid principle number 4: the interface segregation principle.

Chapter 14: Interface Segregation Principle

Ready for principle number 4? It's the interface segregation principle - or ISP. It says:

> Clients should not be forced to depend on interfaces that they do not use.

That's not a bad definition! But I want to clarify that word "interface". It is *not* necessarily referring to a *literal* interface. It's referring to the abstract concept of an interface, which generally means "the public methods" of a class... even if it doesn't technically implement an interface. The meaning of interface *here* is: the "stuff that you can do with an object" when I give it to you.

The Simpler Definition

So let me try to give this an even simpler definition:

> Build small, focused classes instead of big, giant classes.

This definition reminds me a lot of the single responsibility principle... and that's true! But the interface segregation principle kind of looks at this from the other direction: from the perspective of who *uses* the class, not from the perspective of the class itself. Again, the original definition is:

> *Clients* should not be forced to depend upon interfaces - so basically methods - that they do not use.

For example, suppose you've accidentally built a giant class called `ProductManager` with a *ton* of methods on it. Whoops! Then, somewhere in your code, you need to call just *one* of those methods. This other class is called the "client" because it is *using* our giant `ProductManager` class. And unfortunately, even though it only needs one method from the `ProductManager`, it needs to inject the whole giant object. It's forced to depend on an object whose interface - whose public methods - are many more than it actually needs.

New Feature: Adjusting a Score

Why is this a problem? Let's answer that question a bit later after we play with a real world example. Because... management has asked us to make yet *another* change to our believability score system! If a sighting receives a score of *less* than 50 points... but it has three or more photos, we will give it a boost: 5 extra points per photo. This... was not a change we anticipated! Darn! Our scoring factors *do* have the ability to add to the score... but they *don't* have the ability to *see* the final score and then modify it.

Adding another Method to the Interface

No problem: let's add a second method to the interface that has the ability to do that. Call it, how about, public function `adjustScore()`. In this case, it's going to receive the `int $finalScore` that's just been calculated and the `BigFootSighting` that we're scoring. It will return the new `int` final score. You can add some PHPDoc above this to better explain the purpose of the method if you want.

18 lines | src/Scoring/ScoringFactorInterface.php

```
... lines 1 - 6
interface ScoringFactorInterface
{
... lines 9 - 15
    public function adjustScore(int $finalScore, BigFootSighting $sighting): int;
}
```

In a minute, we're going to call this from inside of `SightingScorer` *after* the initial scoring is done. But first, let's open `PhotoFactor` and add the new bonus logic.

Implementing the new Method

At the bottom, go to Code -> Generate - or Command + N on a Mac select "Implement Methods" and implement `adjustScore()`. Say `$photosCount = $sighting->getImages()` - don't forget to *count* these - then if the `$finalScore` is less than 50 and `$photosCount` is greater than two - the `$finalScore` should get plus equals `$photosCount * 5`. At the bottom, return `$finalScore`.

33 lines | src/Scoring/PhotoFactor.php

```
... lines 1 - 6
class PhotoFactor implements ScoringFactorInterface
{
... lines 9 - 22
    public function adjustScore(int $finalScore, BigFootSighting $sighting): int
    {
        $photosCount = count($sighting->getImages());
        if ($finalScore < 50 && $photosCount > 2) {
            $finalScore += $photosCount * 5;
        }

        return $finalScore;
    }
}
```

New logic done! But now... what do we do with all the other classes that implement `ScoringFactorInterface`? Unfortunately, for PHP to even run, we do need to add the new method to *each* class. But we can just make it return `$finalScore`.

So at the bottom of `CoordinatesFactor`, go back to Code -> Generate -select "Implement Methods", generate `adjustScore()`, and return `$finalScore`.

30 lines | src/Scoring/CoordinatesFactor.php

```
... lines 1 - 6
class CoordinatesFactor implements ScoringFactorInterface
{
... lines 9 - 24
    public function adjustScore(int $finalScore, BigFootSighting $sighting): int
    {
        return $finalScore;
    }
}
```

Copy, this close `CoordinatesFactor`, go to `DescriptionFactor` and add it to the bottom. Do the same thing inside of `TitleFactor`.

30 lines | src/Scoring/TitleFactor.php

```
... lines 1 - 6
class TitleFactor implements ScoringFactorInterface
{
... lines 9 - 24
    public function adjustScore(int $finalScore, BigFootSighting $sighting): int
    {
        return $finalScore;
    }
}
```

Finally, we can update `SightingScorer`. Add a second loop after calculating the score: for each `$this->scoringFactors` as `$scoringFactor`, this time say `$score = $scoringFactor->adjustScore()` ... and pass in `$score` and `$sighting`.

35 lines | src/Service/SightingScorer.php

```
... lines 1 - 8
class SightingScorer
{
... lines 11 - 20
    public function score(BigFootSighting $sighting): BigFootSightingScore
    {
... lines 23 - 27
        foreach ($this->scoringFactors as $scoringFactor) {
            $score = $scoringFactor->adjustScore($score, $sighting);
        }
... lines 31 - 32
    }
}
```

Done! By the way, you might argue that the *order* of scoring factors is now relevant. That's true! But... we're not going to worry about that for simplicity... and because that isn't relevant to this principle. But, there *is* a way to give a tagged service a higher priority in Symfony so that it is passed earlier or later than other scoring factors.

We Violated OCP!

If, at this point, something is itching you, that might be because we just violated the open-closed principle! We had to modify the `score()` method in order to add this new behavior. But that's okay! It highlights the tricky nature of OCP: we didn't anticipate this kind of change! You can't "close" a class against *all* kinds of changes: you can only close it against the changes that you correctly predict.

Looking at our new interface and the classes that implement it, you can probably feel that it's not... *ideal* that all of these classes need to implement this method... even though they don't really *care* about it. Next: we're going to make this even *more* obvious, refactor to a better solution, and finally discuss the key takeaways from the interface segregation principle.

Chapter 15: ISP: Refactoring & Takeaways

We've just finished adding the ability to add a bonus to the score if the score is less than 50*and* there are 3 photos or more on a sighting. And... management is *already* requesting another change: we need to make sure that -no matter what - a score *never* receives more than a 100 points.

No problem! We can create another scoring factor class to check for this.In the `Scoring/` directory, add a class called, how about, `MaxScoreAdjuster`. I'm giving this a slightly different name, even though it's a scoring factor,because it's real job is going to be to adjust the score. Make it implement `ScoringFactorInterface`.

Now go to Code -> Generate - or Command + N on a Mac - and just generate,`adjustScore()` to start. For the logic, return the minimum of `$finalScore` or 100. So if the `$finalScore` is over a hundred, this will return 100.

14 lines | src/Scoring/MaxScoreAdjuster.php

```
... lines 1 - 4
use App\Entity\BigFootSighting;

class MaxScoreAdjuster implements ScoringFactorInterface
{
    public function adjustScore(int $finalScore, BigFootSighting $sighting): int
    {
        return min($finalScore, 100);
    }
}
```

Now, setting the priority of the scoring factorsso that this is the final one would be*especially* important. But since that doesn't relate to ISP, we won't worry about it.

Of course, in this new class, we*also* need to implement the*other* method: `score()`. We can just return 0 since we don't care about that.

19 lines | src/Scoring/MaxScoreAdjuster.php

```
... lines 1 - 6
class MaxScoreAdjuster implements ScoringFactorInterface
{
... lines 9 - 13
    public function score(BigFootSighting $sighting): int
    {
        return 0;
    }
}
```

Okay, we've got this working! But we've violated ISP! A lot of the classes that implement `ScoringFactorInterface` - like `MaxScoreAdjuster` and `CoordinatesFactor` - have a dummy method... which we added *just* to satisfy the needs of the interface.

The Signs that You're Violating ISP

When you see something like this, it's a signal that your interface is polluted...or has gotten fat. But again, even though we're using an interface in our example, this also applies to classes in general. If you have a class with multiple public methods... and other parts of your code only use one or some of its methods...that's *also* a violation of ISP. In fact, that's the *main* purpose of ISP. You're requiring clients of your class to depend on interfaces -in other words, methods - that they do*not* need.

What's the solution? Categorize the methods based on their purpose and how they're used...and split them into multiple classes.

For example, if you have a class with 3 methodsand 2 of those methods are always called together, then the class should be

split into only two pieces: one class with those 2 methodsand another class with only the third method.

Splitting our Interface

In our example, it's pretty obvious that splitting the interfaceinto two pieces would make the classes that implement them simpler. So in this `Scoring/` directory, create a new class - or really an interface - and call it `ScoreAdjusterInterface`. What we'll do is go into `ScoringFactorInterface`, steal the `adjustScore()` method and move it into the new interface. Hit okay to import that `use` statement.

11 lines | src/Scoring/ScoreAdjusterInterface.php

```
... lines 1 - 4
use App\Entity\BigFootSighting;

interface ScoreAdjusterInterface
{
    public function adjustScore(int $finalScore, BigFootSighting $sighting): int;
}
```

Thanks to this, we can now go into `CoordinatesFactor` and remove the dummy `adjustScore()` ... and then do the same thing in `TitleFactor` ... and also in `DescriptionFactor`, which feels pretty good! In `MaxScoreAdjuster`, change this to implement `ScoreAdjusterInterface` ... and then we no longer need the dummy `score()` method.

14 lines | src/Scoring/MaxScoreAdjuster.php

```
... lines 1 - 6
class MaxScoreAdjuster implements ScoreAdjusterInterface
{
    public function adjustScore(int $finalScore, BigFootSighting $sighting): int
    {
        return min($finalScore, 100);
    }
}
```

Injecting the Collection of Scoring Adjusters

Finally, the `PhotoFactor` class is interesting: it needs to implement both interfaces, which is totally allowed. Add `ScoreAdjusterInterface`.

33 lines | src/Scoring/PhotoFactor.php

```
... lines 1 - 6
class PhotoFactor implements ScoringFactorInterface, ScoreAdjusterInterface
... lines 8 - 33
```

The last thing to do is make our `SightingScorer` support using *both* interfaces by repeating the trick of injecting a collection of services for `ScoreAdjusterInterface`. In other words, we're now going to inject an `iterable` of scoring factors and a *second* `iterable` of scoring adjusters.

Start in: `src/Kernel.php`. Copy the `registerForAutoConfiguration()` ... and we're going to repeat the same thing, but this time for `ScoreAdjusterInterface` and call the tag `scoring.adjuster`.

53 lines | src/Kernel.php

```
... lines 1 - 12
class Kernel extends BaseKernel
{
... lines 15 - 41
    protected function build(ContainerBuilder $container)
    {
... lines 44 - 48
        $container->registerForAutoconfiguration(ScoreAdjusterInterface::class)
            ->addTag('scoring.adjuster');
    }
}
```

Next, over in `services.yaml`, down on our service, copy the `$scoringFactors` argument, paste, rename to `$scoringAdjusters` and use the new tag name: `scoring.adjuster`.

40 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 34
    App\Service\SightingScorer:
        class: App\Service\DebuggableSightingScorer
        arguments:
... line 38
            $scoreAdjusters: !tagged_iterator scoring.adjuster
```

Copy that argument name and head into `SightingScorer`. Add this as a second `iterable` argument. Then hit Alt + Enter and go to Initialize Properties to create that property and set it. I'll steal the PHPDoc from above the old property to help my editor know that this will hold an iterable of `ScoreAdjusterInterface` objects.

42 lines | src/Service/SightingScorer.php

```
... lines 1 - 9
class SightingScorer
{
... lines 12 - 16
    /**
     * @var ScoreAdjusterInterface[]
     */
    private $scoreAdjusters;

    public function __construct(iterable $scoringFactors, iterable $scoreAdjusters)
    {
... line 24
        $this->scoreAdjusters = $scoreAdjusters;
    }
... lines 27 - 40
}
```

Now loop over *these* instead. You can already see that PhpStorm is not happy because there is no `adjustScore()` method on the scoring factors. Change this to `$scoringAdjusters` ... and I'll rename the variable to `$scoringAdjuster` here and here.

42 lines | src/Service/SightingScorer.php

```
... lines 1 - 27
    public function score(BigFootSighting $sighting): BigFootSightingScore
    {
... lines 30 - 34
        foreach ($this->scoreAdjusters as $scoreAdjuster) {
            $score = $scoreAdjuster->adjustScore($score, $sighting);
        }
... lines 38 - 39
    }
... lines 41 - 42
```

Done! We made our interface smaller, which allowed us to remove all of the dummy methods.

Why Should We Care about ISP?

So, other than being forced to create dummy methods just to make an interface happy, why should we care about ISP? I can think of three reasons.

The first is *naming*. Whether you have a class that's too big or an interface like in our example, splitting it into smaller pieces allows you to give each a more descriptive name that fits its purposes. We can see this in `SightingScorer`. We're now working with scoring *adjusters*, which better describes the purpose of those services than just a "scoring factor"... which does multiple things.

The second is that ISP is a good signal that you might be violating the single responsibility principle. If you notice that you often only call one or two methods from a class... but not it's *other* public methods, that is a violation of ISP. This forces you to think about the *responsibilities* of that class, which may result in organizing into smaller classes *based* on those responsibilities.

The third reason we should care about ISP is that it keeps your dependencies *lighter*. We didn't see that in *this* specific example, but we *did* see it earlier when we talked about SRP. In that case... let me actually close all of my classes... we split a `UserManager` class into two pieces: `UserManager` and `ConfirmationEmailSender`. The `send()` method simply sends the confirmation email, and we use it both after registration *and* when requesting a re-send of that email.

If we had kept these two public function inside of `UserManager` - then resending the confirmation would have be a violation of the interface segregation principle. That would have been a situation where we only needed to call *one* of the two public methods on the class.

And, in order to resend the email, Symfony would need to instantiate a class which depends on, for example, the password encoder service. Why is that a problem? Well, it's minor, but this would force Symfony to instantiate the password encoder so that it could instantiate the `UserManager` ... so that we could send a confirmation email... but we would never actually *use* the password encoder. That's a waste of resources!

Anyways, the tl;dr on the interface segregation principle is this: when you have an interface with a method that not all of its classes need... *or* if you have a class where you routinely use only *some* of its public methods... it may be time to split it into smaller pieces. Or, more simply, you can remember to not build giant classes. But, like everything, it's not an absolute rule. If I had, for example, a `GitHubApiClient` that helped me talked to GitHub's API... I might be ok putting 5 methods in this service, even though I routinely only use one or two of them at a time. After all, the name of the class is still pretty clear... and having more methods probably doesn't increase the number of dependencies that I need to inject into that service.

Next: we're on to principle number five! And this one *really* made my head spin at first. It's: the dependency inversion principle!

Chapter 16: Dependency Inversion Principle

We've made it to the fifth and final SOLID principle:the dependency inversion principle, or DIP. This puppy has a *two* part definition. Ready? One:

> High level modules should not depend on low level modules, both should dependon abstractions - for example, interfaces.

And part two says:

> Abstractions should not depend on details. Details - meaning concrete implementations - should depend on abstractions.

Uhh... if that makes sense to you, you are *awesome*! And... I am jealous of you!

Simpler Definition

How would *I* rephrase this? Um, yikes. How about this. One:

> Classes should depend on interfaces instead of concrete classes.

And two:

> Those interfaces should be designed by the class that *uses* them, not by the classes that will *implement* them.

That's probably still fuzzy... but don't sweat it. This requires a real example.

Our Spam Detection System!

Here's our new problem. We've been getting *so* popular - no surprise -that some of our sightings are getting a lot of spam comments... like comments that say that Bigfoot is *not* real. Those are definitely bots!

So we need a way to determine whether or not a comment is spam basedon some business logic that we've created. If you downloaded the course code from this page, then you should have a `tutorial/` directory with a `CommentSpamManager` class inside. Copy that, then go create a new directory in `src/` called `Comment/` ... and paste the class there.

35 lines | src/Comment/CommentSpamManager.php

```
... lines 1 - 2
namespace App\Comment;
... lines 4 - 6
class CommentSpamManager
{
    public function validate(Comment $comment): void
    {
        $content = $comment->getContent();
        $badWordsOnComment = [];

        $regex = implode('|', $this->spamWords());

        preg_match_all("/$regex/i", $content, $badWordsOnComment);

        if (count($badWordsOnComment[0]) >= 2) {
            // We could throw a custom exception if needed
            throw new \RuntimeException('Message detected as spam');
        }
    }
... lines 23 - 33
}
```

This class basically determines if a comment should be flagged as spam by running a regular expression on the content using a list of predefined spam words. If the content contains two or more of those words, then we consider the comment as spam and throw an exception.

If you think about the single responsibility principle, you could argue that this class *already* has two responsibilities: the low-level regular expression logic that looks for the spam words and a higher level business logic that decides that two spam words is the limit.

Splitting the Class

Let's pretend that we *do* think that these are two different responsibilities. And so, we decide to split this class into two pieces. In the `Service/` directory, create a new class called `RegexSpamWordHelper`. Let's see: move the private `spamWords()` method to the new class... and then create a new public function called `getMatchedSpamWords()` where we pass it the `string $content` and return an array of the matched spam words.

23 lines | src/Service/RegexSpamWordHelper.php

```
... lines 1 - 4
class RegexSpamWordHelper
{
    public function getMatchedSpamWords(string $content): array
    {

    }

    private function spamWords(): array
    {
        return [
            'follow me',
            'twitter',
            'facebook',
            'earn money',
            'SymfonyCats',
        ];
    }
}
```

Next, move the regex logic itself into the class. Copy the entire contents of the existing method.... but leave it... then paste. Let's see... we don't need `$comment->getContent()` anymore.... it's just called `$content` ... and the 0 index of `$badWordsOnComment` will contain the matches, so we can return that.

29 lines | src/Service/RegexSpamWordHelper.php

```
... lines 1 - 6
    public function getMatchedSpamWords(string $content): array
    {
        $badWordsOnComment = [];

        $regex = implode('|', $this->spamWords());

        preg_match_all("/$regex/i", $content, $badWordsOnComment);

        return $badWordsOnComment[0];
    }
... lines 17 - 29
```

Beautiful! Now that this class is ready, let's inject it into `CommentSpamManager`. Add public function `__construct()` with `RegexSpamWordHelper $spamWordHelper`. I'll press Alt + Enter and select "Initialize properties" to create that property and set it.

43 lines | src/Comment/CommentSpamManager.php

```
... lines 1 - 5
use App\Service\RegexSpamWordHelper;

class CommentSpamManager
{
    private RegexSpamWordHelper $spamWordHelper;

    public function __construct(RegexSpamWordHelper $spamWordHelper)
    {
        $this->spamWordHelper = $spamWordHelper;
    }
... lines 16 - 41
}
```

Below, now we can say `$badWordsOnComment = $this->spamWordHelper->getMatchedSpamWords()` and pass that `$content` from above. We don't need any of the logic in the middle anymore.Finally, `$badWordsOnComment` will contain the array of matches, so we don't need to use the 0 index anymore: just count that entire variable.

27 lines | src/Comment/CommentSpamManager.php

```
... lines 1 - 16
    public function validate(Comment $comment): void
    {
        $content = $comment->getContent();
        $badWordsOnComment = $this->spamWordHelper->getMatchedSpamWords($content);

        if (count($badWordsOnComment) >= 2) {
            throw new \Exception('Message detected as spam');
        }
    }
... lines 26 - 27
```

Done!

High Level and Low Level Modules

At this point, we've separated the high-level business logic -deciding how many spam words should cause a comment to be marked as spam - from the low level *details*: matching and finding the spam words.The dependency inversion principle doesn't necessarily tell us whether or not we should split the original logic into two classes like we just did.That's probably more the concern of the single responsibility principle.

But DIP *does* teach us to think about our code in terms of "high-level" modules (or classes)like `CommentSpamManager` - that depend on "low level" modules (or classes) like `RegexSpamWordHelper`. And it gives us concrete rules about*how* this relationship should be handled.

Next, let's refactor the relationship between these two classesto be dependency inversion principle compliant. We'll see, in real terms, *exactly* what changes each of the two partsof this principle want us to make.

Chapter 17: Refactoring Towards Dependency Inversion

Our code, specifically the code in these two classes, does *not* follow the dependency inversion principle. Why not? Let's go through the two parts of the definition, one by one.

The first part is:

> High level modules should not depend on low level modules. Both should depend on abstractions, for example, interfaces.

This is a fancy way of saying that classes should depend on interfaces instead of concrete classes. Yep! This part of the rule is that simple. It says that instead of type-hinting - so "depending on" - the concrete `RegexSpamWordHelper`, we should type-hint an interface.

Okay! So we just need to create a new interface, make `RegexSpamWordHelper` *implement* the interface, then change the type-hint to *use* that interface, right? Yes, exactly!

Thinking about the Design of your Interface

But... the *second* part of DIP tells us something about how we should create and *design* that interface. That part says:

> Abstractions should not depend on details. Details - which are concrete implementations - should depend on abstractions.

We simplified this to:

> An interface should be designed by the class that will *use* it, not by the class that will *implement* it.

Let me explain. The most natural way to create the new interface would be to look at the class that will implement it - so `RegexSpamWordHelper` - and create an interface that matches it! So a `RegexSpamWordHelperInterface` with a `getMatchedSpamWords()` method. Done!

But by doing this, we are allowing the interface to, sort of, be "owned" by the *lower level* class, sometimes known as the "details" class. In other words, the way the interface *looks* is being "controlled" by the lower-level `RegexSpamWordHelper` class.

But DIP says that the *higher* level class - `CommentSpamManager` - should be in charge of creating the interface, allowing *it* to design its dependency in *just* the way that *it* wants.

Creating the Interface

Let's put this into practice. If you look at `CommentSpamManager`, all it really needs is to be able to call a method that will return the *number* of spammy words... because that *count* is ultimately all we use: we don't *really* need the matched words themselves.

So in the `Comment/` directory, which I'm using to highlight that this interface is *owned* by `CommentSpamManager`, create a new interface: select PHP class, change to interface and call it, how about, `CommentSpamCounterInterface`.

Inside, add one method: public function `countSpamWords()`, which will accept the `string $content` and return an `int`.

9 lines | src/Comment/CommentSpamCounterInterface.php

```
... lines 1 - 4
interface CommentSpamCounterInterface
{
    public function countSpamWords(string $content): int;
}
```

Beautiful! Notice that just by inverting, *who* we think should be in charge of creating the interface -or who should "own" it - we ended up with a very different result. Instead of forcing the *interface* to look like the low level `RegexSpamWordHelper` class, *that* class is now going to be forced to change *itself* to implement the interface.

Add `implements CommentSpamCounterInterface`, then I'll go to Code -> Generate -or Command + N on a Mac - and select "Implement Methods" to generate `countSpamWords()`. Inside, return the `count()` of `$this->getMatchedSpamWords($content)`.

36 lines | src/Service/RegexSpamWordHelper.php

```
... lines 1 - 6
class RegexSpamWordHelper implements CommentSpamCounterInterface
{
    public function countSpamWords(string $content): int
    {
        return count($this->getMatchedSpamWords($content));
    }
... lines 13 - 34
}
```

Back in `CommentSpamManager`, let's follow the first part of DIP and change this to depend on the new interface. Change the type-hint to `CommentSpamCounterInterface` ... change the type on the property... and let's also rename the property itself to be more clear: call it `$spamWordCounter`. Rename the argument too.

26 lines | src/Comment/CommentSpamManager.php

```
... lines 1 - 6
class CommentSpamManager
{
    private CommentSpamCounterInterface $spamWordCounter;

    public function __construct(CommentSpamCounterInterface $spamWordCounter)
    {
        $this->spamWordCounter = $spamWordCounter;
    }
... lines 15 - 24
}
```

Down in `validate()`, change `$badWordsOnComment` to `$badWordsCount`. Then, instead of calling `getMatchedSpamWords()`, call the new `countSpamWords()`. Below, we don't need the `count()` anymore: just check if `$badWordsCount` is greater than or equal to 2.

26 lines | src/Comment/CommentSpamManager.php

```
... lines 1 - 15
    public function validate(Comment $comment): void
    {
... line 18
        $badWordsCount = $this->spamWordCounter->countSpamWords($content);
... line 20
        if ($badWordsCount >= 2) {
... line 22
        }
    }
... lines 25 - 26
```

Congratulations! Our code now follows the two parts of the dependency inversion principle! One, our high level class - `CommentSpamManager` - depends on an interface. And two, that interface was designed for - and is controlled by - the high-level class, instead of being designed and controlled by the low level, or "details" class: `RegexSpamWordHelper`.

How Symfony Autowires Interfaces

Before we talk about the takeaways from the dependency inversion principle, I want to mention two things.

First, over in `RegexSpamWordHelper`, you *are* allowed to have this public function `getMatchedSpamWords()` method if you're using it somewhere else in your code. Since we're not, I'm going to clean things up and make it `private`.

36 lines | src/Service/RegexSpamWordHelper.php

```
... lines 1 - 6
class RegexSpamWordHelper implements CommentSpamCounterInterface
{
... lines 9 - 13
    private function getMatchedSpamWords(string $content): array
    {
... lines 16 - 22
    }
... lines 24 - 34
}
```

Second... well... this is more of a question: will Symfony know which service to autowire when it sees the `CommentSpamCounterInterface` type-hint? Will it know that it should actually pass us the `RegexSpamWordHelper` service?

Actually... it will! Find your terminal and run:

```
php bin/console debug:autowiring Comment --all
```

I'm passing `--all` just so we can see *all* the results. And... this proves it! As this shows, when Symfony sees a `CommentSpamCounterInterface` type-hint, it will autowire the `RegexSpamWordHelper` service.

This works thanks to a nice feature inside Symfony's container. If Symfony sees an interface in *our* code - like `CommentSpamCounterInterface` - and only one of our classes implements it, then it automatically assumes that this class should be autowired for that interface. If you ever created a *second* class that implemented the interface, Symfony would throw a clear exception telling us that we need to choose which one to autowire.

Next: let's talk about the takeaways of the dependency inversion principle, and also... what that word "inversion" means and doesn't mean.

Chapter 18: DIP: Takeaways

The two rules of the dependency inversion principle give us clear instructionson how two classes - like `CommentSpamManager` and `RegexSpamWordHelper` - should interact.

"Inversion"? What got Inverted?

But before we talk about the pros and cons of DIP...why is this called dependency *inversion*? What is the "inversion"?

This took me a *long* time to wrap my head around. I expected that dependency inversion somehow meantthat the two classes *literally* started depending on each other in some...different way. Like suddenly we would inject the `CommentSpamManager` into `RegexSpamWordHelper` ... instead of the other way around, actually "inverting" the dependency.

But, as you can see... that is *not* the case. On a high level, these two classes depend on each other in the*exact* same way as they always did: the low level, details class - `RegexSpamWordHelper` - is injected into the high-level class - `CommentSpamManager` .

The "inversion" part is... more of an abstract concept. *Before* we refactored our code to create and use the interface, I would have said:

> `CommentSpamManager` depends on `RegexSpamWordHelper` . If we decide to modify `RegexSpamWordHelper` , we will then need to update `CommentSpamManager` to make it work with those changes. `RegexSpamWordHelper` is the boss.

But *after* the refactoring, specifically, after we created an interface based on the needsof `CommentSpamManager` , I would *now* say this:

> `CommentSpamManager` depends on any classthat implements `CommentSpamCounterInterface` . In reality, this is the `RegexSpamWordHelper` class. But if we decided to refactor how `RegexSpamWordHelper` works, *it* would *still* be responsible for implementing `CommentSpamCounterInterface` . In other words, when `RegexSpamWordHelper` changes, our high level `CommentSpamManager` class will *not* need to change.

That is the inversion: it's an inversion of control: a "reversal" of who is in charge.Thanks to the new interface, the high-level class - `CommentSpamManager` - has taken control over what its dependency needs to look like.

Pros and Cons of DIP

So now that we understand the dependency inversion principle, what are its benefits?

Simply put: DIP is all about *decoupling*. `CommentSpamManager` is now *decoupled* from `RegexSpamWordHelper` . We could even *replace* it with a different class that implements this interfacewithout touching *any* code from the high-level class.

This is one of the core strategies to writing "framework agnostic" code.In this situation, developers create interfaces in*their* code and only depend on *those* interfaces, instead of on the interfaces or classesfrom whatever framework they're using.

However, in my code, I rarely follow the dependency inversion principle.Well, let me clarify. If I were working on an open source, reusable library, like Symfony itself, I would *definitely* create interfaces, like we just did. Why? Because I want to allow the users of my code to *replace* this service with some other class, like maybe someone wants to replace our simple `RegexSpamWordHelper` in their app with a class that uses an API to find these spam words.

But if I were writing this in my*own* application, I would skip creating the interface: I would make my code look like it originally did with `CommentSpamManager` relying directly on `RegexSpamWordHelper` with no interface.

Most Dependencies Don't Need Inverting

Why? As Dan North points out in his blog post: not all dependencies*need* to be inverted. If something you depend on will *truly* need to be swapped out for a different class or implementation later, then that dependency is almost more of an "option".If we

had that situation, we probably *would* want to apply DIP. By creating and type-hinting an interface, we're saying:

> Please pass me the "option" that you would like to use for counting spam words.

But, most of the time, to partially quote Dan:

> Dependencies aren't options: they're just the way we are going to count spam words in this situation.

If you followed DIP perfectly, you end up with a code base with a lot of interfaces which are implemented by only one class each. That adds flexibility... which you likely won't need. The "cost" is misdirection: your code is harder to follow.

For example, in `CommentSpamManager`, it now takes a bit more work to figure out which class counts the spam words and how everything is working. And if you ever *do* try to change a dependency to use a different, concrete class, you might discover that, even though you followed DIP, it's not so easy change!

For example, changing from one database system to another is probably going to be an ugly job...even if you created an interface to abstract away the differences beforehand. It might *still* be worth doing... if you *do* think your database will change, but it's not a silver bullet that will make that an easy task.

So my advice is this: unless you're writing code that will be shared across projects, do not create an interface until you have more than one class that would implement it... which we actually saw earlier with our scoring factors. This is a perfectly nice use of interfaces.

But! I fully admit that not everyone agrees with my opinion on this! And if you *do* disagree, awesome! Do what you think is best. There are plenty of smart people out there that *do* create extra interfaces in their code to decouple from whatever frameworks or libraries they're using. I'm just not one of them.

SOLID in Review

Ok friends, that's it! We are done with the SOLID principles! Let's do a quick recap... using our simplified definitions.

One: the single responsibility principle says:

> Write classes so that your code "fits in your head".

Two: the open-closed principle says:

> Design your classes so that you can change their behavior without changing their code.

This is never entirely possible... and in my app code, I rarely follow this.

Three: the Liskov substitution principle says:

> If a class extends a base class or implements an interface, make your class behave like it is supposed to.

PHP protects against most violations of this principle by throwing syntax errors.

Four: the interface segregation principle says:

> If a class has a large interface - so a lot of methods - and you often inject the class and only use *some* of these methods - consider splitting your class into smaller pieces.

And five: the dependency inversion principle says:

> Prefer type-hinting interfaces and allow each interface to be designed for the "high level" class that will use it, instead of for the low-level class that will implement it.

In my app, I *do* type-hint interfaces whenever they exist, usually because services from Symfony or other libraries provide an interface. But I don't create my *own* interfaces until I have multiple classes that need to implement them.

My opinions are, of course, just that: opinions! And I tend to be much more pragmatic than dogmatic... for better or worse. People will definitely disagree... and that's great! SOLID forces us to think critically.

Also the SOLID principles aren't the only "game" in town when it comes to writing clean code. There are design patterns, composition over inheritance, the law of demeter and other principles to guide your path.

If you have any questions or ideas, as always, we would *love* to hear from you down in the comments.

Alright, friends, seeya next time!