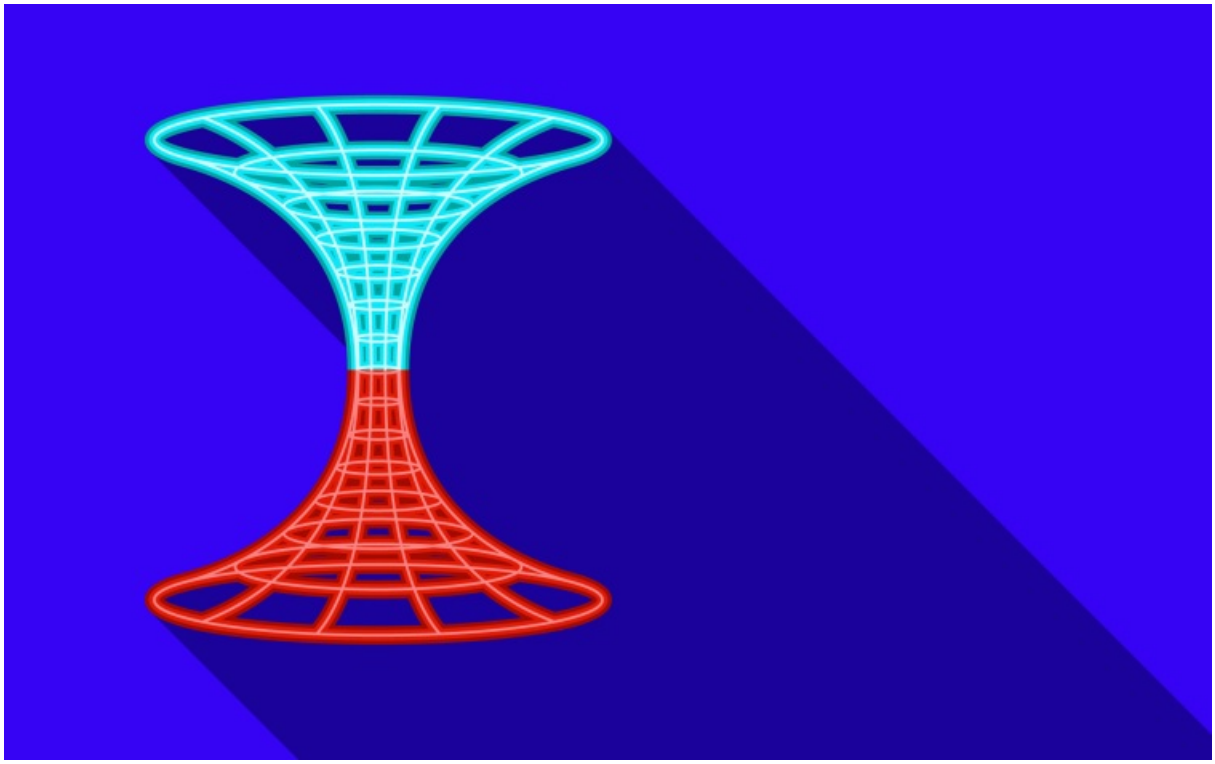
Design Patterns for Fun and Proficiency



With <3 from SymfonyCasts

Chapter 1: Design Patterns & Their Types

Hey friends! Thanks for hanging out and giving me the *privilege* to guide us through some fun, geeky, but also *useful* stuff. We're talking *design patterns*. The idea is simple: The same problems that *we* face in our code every day have been faced a *million* times before. And often, a way or "strategy" to solve that problem has already been perfected. These are called "design patterns".

Why Should we Care?

A design pattern is nothing more than a "strategy" for writing code when you encounter a particular problem. If you can start to *identify* which types of problems are solved by which strategies, you'll walk into situations and immediately know what to do. Learning design patterns gives you:

A) More tools in your developer toolkit when coding and B) A better understanding of core libraries like Symfony, which leverages design patterns a *lot*.

It'll also make you *way* more fun at parties... assuming the only people at the party are programmers... because you'll be able to smartly say things like:

> Yea, I noticed that you refactored to use the decorator pattern - great idea for extending that class without violating the single responsibility principle.

Dang, we're going to be *super* popular.

Design Pattern Types

Ok, so there are *tons* of design patterns. Though... only a small number are likely to be useful to us in the real-world: we just won't ever face the problems that the others solve. These many design patterns fall into three basic groups. You don't need to memorize these... it's just a nice way to think about the three types of *problems* that design patterns solve.

The first type is called "creational patterns", and these are all about helping *instantiate* objects. They include the factory pattern, builder pattern, singleton pattern, and others.

The second type is called "structural patterns". These help you organize things when you have a bunch of objects and you need to identify *relationships* between them. One example of a relationship would be a parent-child relationship, but there are *many* others. Yea, I know: this one can be a little fuzzy. But we *will* see one structural pattern in this tutorial: the "decorator pattern".

The third and final type of patterns is called "behavioral patterns", which help solve problems with how objects *communicate* with each other, as well as assigning responsibilities between objects. That's a fancy way of saying that behavioral patterns help you design classes with specific responsibilities that can then work together... instead of putting all of that code into one *giant* class. We'll talk about *two* behavioral patterns: the "strategy pattern" and the "observer pattern".

Get that Project Set up!

Now that we've defined some of what we'll be looking at, it's time to get technical! We're going to use these patterns in a *real* Symfony project to do *real* stuff. We'll only cover a *few* patterns in this tutorial - some of my favorites - but if you finish and want to see more, let us know!

All right, to be the best design-pattern-er that you can be, you should definitely download the course code from this page and code along with me. After you unzip it, you'll find a `start/` directory that has the same code that you see here. Pop open this `README.md` file for all the setup details. Though, this one's pretty easy: you just need run:

```
composer install
```

Our app is a simple command-line role-playing game where characters battle each other and level up. RPG's are my *favorite* type of game - Shining Force for the win!

To play, run:

```
./bin/console app:game:play
```

Sweet! We have three character types! Let's be a *fighter*. We're battling *another* fighter. Queue epic battle sounds! And... we won! There was 11 rounds of fighting, 94 damage points dealt, 84 damage points received and glory for all!!! We can also battle *again*. And... woohoo! We're on a *roll*!

This *is* a Symfony app, but a very *simple* Symfony app. It has a command class that sets things up and prints the results. You tell it which character you want to be and it starts the battle.

98 lines | src/Command/GameCommand.php

```
... lines 1 - 14
class GameCommand extends Command
{
... lines 17 - 23
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);

        $io->text('Welcome to the game where warriors fight against each other for honor and glory... and !');

        $characters = $this->game->getCharactersList();
        $characterChoice = $io->choice('Select your character', $characters);

        $playerCharacter = $this->game->createCharacter($characterChoice);
        $playerCharacter->setNickname('Player ' . $characterChoice);

        $io->writeln('It\'s time for a fight!');

        $this->play($io, $playerCharacter);

        return Command::SUCCESS;
    }
... lines 42 - 96
}
```

But most of the work is done via the `game` property, which is this `GameApplication` class. This takes these two `Character` objects and it goes through the logic of having them "attack" each other until one of them wins. At the bottom, it also contains the three character types, which are represented by this `Character` class. You can pass in different stats for your character, like `$maxHealth`, the `$baseDamage` that you do, and different `$armor` levels.

70 lines | src/GameApplication.php

```
... lines 1 - 6
class GameApplication
{
    public function play(Character $player, Character $ai): FightResult
    {
        $player->rest();

        $fightResult = new FightResult();
        while (true) {
            $fightResult->addRound();

            $damage = $player->attack();
            if ($damage === 0) {
                $fightResult->addExhaustedTurn();
            }

            $damageDealt = $ai->receiveAttack($damage);
            $fightResult->addDamageDealt($damageDealt);

            if ($this->didPlayerDie($ai)) {
                return $this->finishFightResult($fightResult, $player, $ai);
            }

            $damageReceived = $player->receiveAttack($ai->attack());
            $fightResult->addDamageReceived($damageReceived);

            if ($this->didPlayerDie($player)) {
                return $this->finishFightResult($fightResult, $ai, $player);
            }
        }
    }
... lines 37 - 68
}
```

So `GameApplication` defines the three character types down here... then battles them up above. That's basically it!

70 lines | src/GameApplication.php

```
... lines 1 - 37
    public function createCharacter(string $character): Character
    {
        return match (strtolower($character)) {
            'fighter' => new Character(90, 12, 0.25),
            'archer' => new Character(80, 10, 0.15),
            'mage' => new Character(70, 8, 0.10),
            default => throw new \RuntimeException('Undefined Character'),
        };
    }
... lines 47 - 70
```

Next: let's dive into our first pattern - the "strategy pattern" - where we allow some characters to cast magical spells. To make that possible, we're going to need to make the `Character` class *a lot* more flexible.

Chapter 2: Strategy Pattern

The first pattern we'll talk about is the "strategy pattern". This is a *behavioral* pattern that helps organize code into separate classes that can then interact with each other.

Definition

Let's start with the *technical* definition:

> The strategy pattern defines a family of algorithms, encapsulates each one and makes them *interchangeable*. It lets the algorithm vary independently from clients that use it.

If that made sense to you, congrats! *You* get to teach the rest of the tutorial!

Let's try that again. Here's *my* definition:

> The strategy pattern is a way to allow *part* of a class to be rewritten from the *outside*.

Imaginary Example

Let's talk about an *imaginary* example before we start coding. Suppose we have a `PaymentService` that does a bunch of stuff... including charging people via credit card. But now, we discover that we need to use this *exact* same class to allow people to pay via PayPal... or via pirate treasure - that sounds more fun.

Anyways, how can we do that? The *strategy pattern*! We would allow a new `PaymentStrategyInterface` object to be passed *into* `PaymentService` and then we would call *that*.

Next, we would create two classes that *implement* the new interface: `CreditCardPaymentStrategy` and `PiratesBootyPaymentStrategy`. That's it! *We* now have control of *which* class we pass in. Yep! We just made part of the code *inside* `PaymentService` controllable from the *outside*.

The Real Example

With that in mind, let's actually *code* this pattern.

Right now, we have three characters that are created inside of `GameApplication`. But the `fighter` is *dominating*. To balance the game, I want to add special attack abilities for each character. For example, the `mage` will be able to *cast spells*.

70 lines | src/GameApplication.php

```
... lines 1 - 6
class GameApplication
{
... lines 9 - 37
    public function createCharacter(string $character): Character
    {
        return match (strtolower($character)) {
            'fighter' => new Character(90, 12, 0.25),
            'archer' => new Character(80, 10, 0.15),
            'mage' => new Character(70, 8, 0.10),
            default => throw new \RuntimeException('Undefined Character'),
        };
    }
... lines 47 - 68
}
```

Currently, the attack functionality is pretty *boring*: we take the character's `baseDamage` then use this cool

`Dice::roll()` function to roll a six-sided die for some randomness.

72 lines | src/Character/Character.php

```
... lines 1 - 6
class Character
{
... lines 9 - 25
    public function attack(): int
    {
        $this->currentStamina -= (25 + Dice::roll(20));
        if ($this->currentStamina <= 0) {
            // can't attack this turn
            $this->currentStamina = self::MAX_STAMINA;

            return 0;
        }

        return $this->baseDamage + Dice::roll(6);
    }
... lines 38 - 70
}
```

But when a `mage` casts a spell, the damage it causes will be *much* more variable: sometimes a spell works really well but... other times it makes like tiny fireworks that do almost zero damage.

Basically, for the mage, we need *completely* different code for calculating damage.

Pass in an Option?

So how can we do this? How can we allow *one* character - the mage - to have different damage logic? The first idea that comes to *my* mind is to pass a flag into the character's constructor, like `$canCastSpells`. Then in the `attack()` method, add an `if` statement so that we have both types of attacks.

Cool... but what if an `archer` needs yet a *different* type of attack? We'd then have to pass *another* flag and we'd end up with *three* variations inside of `attack()`. Yikes.

Sub-Class?

Ok then, another solution might be that we sub-class `Character`. We create a `MageCharacter` that extends `Character`, then override the `attack()` method entirely. But, darn it! We don't want to override *all* of `attack()`, we just want to replace *part* of it. We *could* get fancy by moving the part we want to reuse into a protected function so that we can call it from our sub-class... but this is getting a little ugly. Ideally we can solve problems *without* inheritance whenever possible.

Creating the "strategy" Interface

So let's back up. What we *really* want to do is allow this code to be different on a character-by-character basis. And that is *exactly* what the strategy pattern allows.

Let's do this! The logic that we need the flexibility to change is this part here, where we determine how much damage an attack did.

Ok, step 1 to the pattern is to create an interface that *describes* this work. I'm going to add a new `AttackType/` directory to organize things. Inside, create a new PHP class, change the template to "Interface", and call it `AttackType`.

Cool! Inside, add one `public function` called, how about, `performAttack()`. This will accept the character's `$baseDamage` - because that might be useful - then return the final damage that should be applied.

9 lines | src/AttackType/AttackType.php

```
... lines 1 - 4
interface AttackType
{
    public function performAttack(int $baseDamage): int;
}
```

Awesome!

Adding Implementation of the Interface

Step 2 is to create at least one implementation of this interface. Let's pretend our `mage` has a cool fire attack. Inside the same directory, create a class called `FireBoltType` ... and make it implement `AttackType`. Then, go to "Code -> Generate" - or "command" + "N" on a Mac - and select "Implement Methods" as a shortcut to add the method we need.

14 lines | src/AttackType/FireBoltType.php

```
... lines 1 - 6
class FireBoltType implements AttackType
{
    public function performAttack(int $baseDamage): int
    {
... line 11
    }
}
```

For the magic attack, return `Dice::roll(10)` 3 times. So the damage done is the result of rolling 3 10-sided dice.

14 lines | src/AttackType/FireBoltType.php

```
... lines 1 - 8
    public function performAttack(int $baseDamage): int
    {
        return Dice::roll(10) + Dice::roll(10) + Dice::roll(10);
    }
... lines 13 - 14
```

And... our first attack type is done! While we're here, let's create two *others*. I'll add a `BowType` ... and paste in some code. You can copy this from the code block on this page. This attack has a chance of doing some *critical* damage. Finally, add a `TwoHandedSwordType` ... and I'll paste in that code as well. This one is pretty straightforward: it's the `$baseDamage` plus some random rolls.

16 lines | src/AttackType/BowType.php

```
... lines 1 - 4
use App\Dice;

class BowType implements AttackType
{
    public function performAttack(int $baseDamage): int
    {
        $criticalChance = Dice::roll(100);

        return $criticalChance > 70 ? $baseDamage * 3 : $baseDamage;
    }
}
```

16 lines | src/AttackType/TwoHandedSwordType.php

```
... lines 1 - 4
use App\Dice;

class TwoHandedSwordType implements AttackType
{
    public function performAttack(int $baseDamage): int
    {
        $twoHandledSwordDamage = Dice::roll(12) + Dice::roll(12);

        return $baseDamage + $twoHandledSwordDamage;
    }
}
```

Passing in and Using the Strategy

We're ready for the 3rd and final step for this pattern: allow an `AttackType` interface to be passed into `Character` so that we can use it below. So, quite literally, we're going to add a new argument: `private` - so it's also a property - type-hinted with the `AttackType` interface (so we can allow any `AttackType` to be passed in) and call it `$attackType`.

74 lines | src/Character/Character.php

```
... lines 1 - 4
use App\AttackType\AttackType;
... lines 6 - 7
class Character
{
... lines 10 - 15
    public function __construct(
... lines 17 - 19
        private AttackType $attackType
    ) {
... line 22
    }
... lines 24 - 72
}
```

Below, remove this comment... because now, instead of doing the logic *manually*, we'll say `return $this->attackType->performAttack($this->baseDamage)`.

71 lines | src/Character/Character.php

```
... lines 1 - 24
    public function attack(): int
    {
        $this->currentStamina -= (25 + Dice::roll(20));
        if ($this->currentStamina <= 0) {
            // can't attack this turn
            $this->currentStamina = self::MAX_STAMINA;

            return 0;
        }

        return $this->attackType->performAttack($this->baseDamage);
    }
... lines 37 - 71
```

And we're done! Our `Character` class is now leveraging the *strategy* pattern. It allows someone *outside* of this class to pass in an `AttackType` object, effectively letting them control just *part* of its code.

Taking Advantage of our Flexibility

To take advantage of the new flexibility, open up `GameApplication`, and inside of `createCharacter()`, pass an `AttackType` to each of these, like `new TwoHandedSwordType()` for the `fighter`, `new BowType()` for the `archer`, and

`new FireBoltType()` for the `mage`.

73 lines | src/GameApplication.php

```
... lines 1 - 4
use App\AttackType\BowType;
use App\AttackType\FireBoltType;
use App\AttackType\TwoHandedSwordType;
... lines 8 - 9
class GameApplication
{
... lines 12 - 40
    public function createCharacter(string $character): Character
    {
        return match (strtolower($character)) {
            'fighter' => new Character(90, 12, 0.25, new TwoHandedSwordType()),
            'archer' => new Character(80, 10, 0.15, new BowType()),
            'mage' => new Character(70, 8, 0.10, new FireBoltType()),
... line 47
        };
    }
... lines 50 - 71
}
```

Sweet! To make sure we didn't break anything, head over and try the game.

```
php bin/console app:game:play
```

And... woohoo! It's *still* working!

Adding a Mixed Attack Character

What's great about the "strategy pattern" is that, instead of trying to pass options to `Character` like `$canCastSpells = true` to configure the attack, we have *full* control.

To prove it, let's add a new character - a *mage archer*: a legendary character that has a bow *and* casts spells. Double threat!

To support this idea of having *two* attacks, create a new `AttackType` called `MultiAttackType`. Make it implement the `AttackType` interface and go to "Implement Methods" to add the method.

21 lines | src/AttackType/MultiAttackType.php

```
... lines 1 - 2
namespace App\AttackType;

class MultiAttackType implements AttackType
{
... lines 7 - 13
    public function performAttack(int $baseDamage): int
    {
... lines 16 - 18
    }
}
```

In *this* case, I'm going to create a constructor where we can pass in an `array` of `$attackTypes`. To help out my editor, I'll add some PHPDoc above to note that this is an array specifically of `AttackType` objects.

21 lines | src/AttackType/MultiAttackType.php

```
... lines 1 - 6
    /**
     * @param AttackType[] $attackTypes
     */
    public function __construct(private array $attackTypes)
    {
    }
... lines 13 - 21
```

This class will work by randomly choosing between one of its available `$attackTypes`. So, down here, I'll say `$type = $this->attackTypes[]` - whoops! I meant to call this `attackTypes` with a "s" - then `array_rand($this->attackTypes)`. Return `$type->performAttack($baseDamage)`.

21 lines | src/AttackType/MultiAttackType.php

```
... lines 1 - 13
    public function performAttack(int $baseDamage): int
    {
        $type = $this->attackTypes[array_rand($this->attackTypes)];

        return $type->performAttack($baseDamage);
    }
... lines 20 - 21
```

Done! This is a *very* custom attack, but with the "strategy pattern", it's no problem. Over in `GameApplication`, add the new `mage_archer` character... and I'll copy the code above. Let's have this be... `75, 9, 0.15`. Then, for the `AttackType`, say `new MultiAttackType([])` passing `new BowType()` and `new FireBoltType()`.

75 lines | src/GameApplication.php

```
... lines 1 - 6
use App\AttackType\MultiAttackType;
... lines 8 - 10
class GameApplication
{
... lines 13 - 41
    public function createCharacter(string $character): Character
    {
        return match (strtolower($character)) {
... lines 45 - 47
            'mage_archer' => new Character(75, 9, .15, new MultiAttackType([new BowType(), new FireBoltType()])),
        };
    }
... lines 51 - 73
}
```

Sweet! Below, we also need to update `getCharacterList()` so that it shows up in our character selection list.

75 lines | src/GameApplication.php

```
... lines 1 - 51
    public function getCharactersList(): array
    {
        return [
            'fighter',
            'mage',
            'archer',
            'mage_archer'
        ];
    }
... lines 61 - 75
```

Okay, let's check out the *legendary* new character:

```
php bin/console app:game:play
```

Select `mage_archer` and... oh! A *stunning* victory against a normal `archer`. How cool is that?

Next, let's use the "strategy pattern" one more time to make our `Character` class even *more* flexible. Then, we'll talk about where you can see the "strategy pattern" in the wild and what *specific* benefits it gives us.

Chapter 3: Strategy Part 2: Benefits & In the Wild

We just used the Strategy Pattern to allow things *outside* of the `Character` class to control *how* attacks happen by creating a custom `AttackType` ... then passing it in when you create the `Character`.

Naming Conventions?

If you've read up on this pattern, you might be wondering why we didn't name the interface `AttackStrategy` after the pattern. The answer is... because we don't *have* to. In all seriousness, the clarity and *purpose* of this class are more valuable than hinting the name of a pattern. If we called this "attack strategy"... it might sound like it's responsible for actually *planning* a strategy of attack. That's *not* what we intended. Hence our name: `AttackType`

9 lines | src/AttackType/AttackType.php

```
... lines 1 - 4
interface AttackType
{
    public function performAttack(int $baseDamage): int;
}
```

Another Strategy Pattern Example

Let's do one more quick strategy pattern example to further balance our characters. I want to be able to control the armor of each character beyond just the number that's being passed in right now. This is used down in `receiveAttack()` to figure out how much an attack can be *reduced* by. This was fine before, but *now* I want to start creating very different *types* of armor that each have different properties beyond just a number. We'll need to upgrade our code to allow this.

71 lines | src/Character/Character.php

```
... lines 1 - 7
class Character
{
... lines 10 - 37
    public function receiveAttack(int $damage): int
    {
        $armorReduction = (int) ($damage * $this->armor);
        $damageTaken = $damage - $armorReduction;
        $this->currentHealth -= $damageTaken;

        return $damageTaken;
    }
... lines 46 - 69
}
```

Once again, we *could* solve this by creating *sub-classes*, like `CharacterWithShield`. But now you can hopefully see why that's not a great plan. If we had also used inheritance for customizing how the attacks happen, we might end up with classes like `TwoHandedSwordWithShieldCharacter` or `SpellCastingAndBowUsingWearingLeatherArmorCharacter`. Yikes!

So rather than navigate that nightmare of never-ending sub-classes, we'll use the *Strategy Pattern*. Let's revisit the three steps from earlier. Step one is to *identify* the code that needs to change and create an interface for it.

In our case, we need to determine how much an attack should be reduced by. Cool: create a new `ArmorType/` directory and inside that, a new PHP class... which will actually be an interface... and call it, how about, `ArmorType`.

To hold the armor-reducing code, say `public function getArmorReduction()` where we pass in the `$damage` that

we're about to do, and will return how much damage *reduction* the armor should apply.

9 lines | src/ArmorType/ArmorType.php

```
... lines 1 - 4
interface ArmorType
{
    public function getArmorReduction(int $damage): int;
}
```

Step two is to create at least one implementation of this. Create a new PHP class called `ShieldType` and make it implement `ArmorType`. Below, I'll generate the `getArmorReduction()` method. The shield is cool because it's going to have a 20% chance to block an incoming attack *entirely*. Create a `$chanceToBlock` variable set to `Dice::roll(100)`. Then, if the `$chanceToBlock` is `> 80`, we're going to reduce *all* of the damage. So return `$damage`. *Else* our shield is going to be meaningless and reduce the damage by zero. Ouch!

19 lines | src/ArmorType/ShieldType.php

```
... lines 1 - 4
use App\Dice;

class ShieldType implements ArmorType
{
    /**
     * Has 20% to fully block the attack
     */
    public function getArmorReduction(int $damage): int
    {
        $chanceToBlock = Dice::roll(100);

        return $chanceToBlock > 80 ? $damage : 0;
    }
}
```

While we're here, let's create two other types of armor. The first is a `LeatherArmorType`. I'll paste in the logic: it absorbs 20% of the damage.

15 lines | src/ArmorType/LeatherArmorType.php

```
... lines 1 - 4
class LeatherArmorType implements ArmorType
{
    /**
     * Absorbs 25% of the damage
     */
    public function getArmorReduction(int $damage): int
    {
        return floor($damage * 0.25);
    }
}
```

And *then* create the cool `IceBlockType`: a little something for our magic folk. I'll paste that logic in as well. This will absorb two eight-sided dice rolls added together.

17 lines | src/ArmorType/IceBlockType.php

```
... lines 1 - 6
class IceBlockType implements ArmorType
{
    /**
     * Absorbs 2d8
     */
    public function getArmorReduction(int $damage): int
    {
        return Dice::roll(8) + Dice::roll(8);
    }
}
```

Ok step three: allow an object of the `ArmorType` interface to be passed into `Character` ... then use its logic. In this case, we won't need the `$armor` number at all. Instead, add a `private ArmorType $armorType` argument.

72 lines | src/Character/Character.php

```
... lines 1 - 4
use App\ArmorType\ArmorType;
... lines 6 - 8
class Character
{
... lines 11 - 16
    public function __construct(
... lines 18 - 20
        private ArmorType $armorType
    ) {
... line 23
    }
... lines 25 - 70
}
```

Down below, in `receiveAttack()`, say `$armorReduction = $this->armorType->getArmorReduction()` and pass in `$damage`. And just to make sure things don't drift negative, add a `max()` after `$damageTaken` passing `$damage - $armorReduction` and `0`.

73 lines | src/Character/Character.php

```
... lines 1 - 38
    public function receiveAttack(int $damage): int
    {
        $armorReduction = $this->armorType->getArmorReduction($damage);

        $damageTaken = max($damage - $armorReduction, 0);
... lines 44 - 46
    }
... lines 48 - 73
```

Done! `Character` now leverages the Strategy Pattern... again! Let's go take advantage of that over in `GameApplication`.

Start by removing the armor number on each of these. Then I'll quickly pass in an `ArmorType`: `new ShieldType()`, `new LeatherArmorType()`, and `new IceBlockType()`. For our `mage-archer`, which is our weird character, we'll *keep it* weird by giving them a shield - `new ShieldType()`. That's a lot to carry! Oh, and I also need to make sure I take off the armor for that as well. Perfect!

78 lines | src/GameApplication.php

```
... lines 1 - 4
use App\ArmorType\IceBlockType;
use App\ArmorType\LeatherArmorType;
use App\ArmorType\ShieldType;
... lines 8 - 13
class GameApplication
{
... lines 16 - 44
    public function createCharacter(string $character): Character
    {
        return match (strtolower($character)) {
            'fighter' => new Character(90, 12, new TwoHandedSwordType(), new ShieldType()),
            'archer' => new Character(80, 10, new BowType(), new LeatherArmorType()),
            'mage' => new Character(70, 8, new FireBoltType(), new IceBlockType()),
            'mage_archer' => new Character(75, 9, new MultiAttackType([new BowType(), new FireBoltType()]), new ShieldTyp
        };
    }
... lines 54 - 76
}
```

Let's go try this team. Head over and run:

```
./bin/console app:game:play
```

And... it looks like it's working! Let's play as a `mage-archer` and... sweet! Well, I *lost*. That's *not* sweet, but I tried my best! And you can see that the "damage dealt" and the "damage received" still seem to be working. Awesome!

Pattern Benefits

So *that's* the Strategy Pattern! When do you *need* it? When you find that you need to swap out just *part* of the code inside of a class. And what are the *benefits*? A bunch! Unlike inheritance, we can now create characters with endless combinations of attack and armor behaviors. We could also swap out an `AttackType` or `ArmorType` at runtime. This means that we could, for example, read some configuration or environment variable and dynamically use it to change one of the attack types of our characters on the fly. That's not possible with inheritance.

Pattern and SOLID Principle

If you watched our SOLID tutorial, the Strategy Pattern is a clear win for SRP - the single responsibility principle - and OCP - the open closed principle. The Strategy Pattern allows us to break big classes like `Character` into smaller, more focused ones, but still have them interact with each other. That pleases SRP.

And OCP is happy because we now have a way to modify or extend the behavior of the `Character` class without actually *changing* the code inside. We can pass in new armor and attack types instead.

Strategy Pattern in the Real World

Finally, where might we see this pattern in the real world? One example, if you hit "shift" + "shift" and type in `Session.php`, is Symfony's `Session` class. The `Session` is a simple key value store, but different apps will need to store that data in different locations, like the filesystem or a database.

Instead of trying to accomplish that with a bunch of code inside of the `Session` class itself, `Session` accepts a `SessionStorageInterface`. We can pass whatever session storage strategy we want. Heck, we could even use environment variables to swap to a different storage at runtime!

Where else is the Strategy Pattern used? Well, it's subtle, but it's actually used in a lot of places. Anytime you have a class that accepts an interface as a constructor argument, especially if that interface comes from the

same library, that's quite possibly the Strategy Pattern. It means that the library author decided that, instead of putting a bunch of code in the middle of the class, it should be abstracted into another class. *And*, by type-hinting an *interface*, they're allowing someone *else* to pass in whatever implementation - or *strategy* they want.

Here's another example. Over on GitHub, I'm on the Symfony repository. Hit "t" and search for `JsonLoginAuthenticator`. This is the code behind the `json_login` security authenticator. One common need with the `JsonLoginAuthenticator` is to use it like normal... but then take control of what happens on success: for example, to control the JSON that's returned after authentication.

To allow for that `JsonLoginAuthenticator` allows you to pass in an `AuthenticationSuccessHandlerInterface`. So instead of *this* class trying to figure out what to do on success, it allows *us* to pass in a custom implementation that gives us complete control.

Think you've got all that? Great! Let's talk about the Builder Pattern *next*.

Chapter 4: Builder Pattern

Time for "design pattern" number *two*: the "builder pattern". This is one of those creational patterns that help you instantiate and configure objects. And, it's a bit easier to understand than the "strategy pattern".

Official Definition

The *official* definition of the "builder pattern" is this:

> A creational design pattern that lets you build and configure complex objects step-by-step.

That... actually made sense. Part *two* of the official definition says:

> the pattern allows you to produce different types and representations of an object using the same construction code.

In other words, you create a builder class that helps *build* other objects... and those object might be of *different* classes or the *same* class with different data.

Simple Example

A simple example might be a pizza parlor that needs to create a bunch of pizzas, each with a different crust, toppings, etc. To make life easier, the owner of the pizza parlor, who's a Symfony developer by night, decides to create a `PizzaBuilder` class with easy methods like `addIngredient()`, `setDough()`, and `addCheese()`. Then, they create a `buildPizza()` method, which takes *all* of that info and does the heavy lifting of *creating* that `Pizza` object and returning it. That `buildPizza()` method can be as complicated as needed. Anyone *using* this class doesn't know or care about any of that. The method could also create different *classes* for different situations if that's what our brave pizza-parlor-owner-slash-Symfony-dev needs for their app.

Creating the Builder Class

Ok, let's create a builder in *our* project. Head over to `GameApplication` and go down to `createCharacter()`. The *problem* is that we're building *four* different `Character` objects and passing *quite* a bit of data to configure each one. And, what if we need to create these `Character` objects in *other* places in our code? They're not *super* easy to build right now. We *could* make some sub-class of `Character` that can set this data up automatically, like by calling the parent constructor. But, like we talked about with the strategy pattern, that could get really ugly when we start having odd combinations of things like a `mage-archer` with an `IceBlockType` shield class.

And what if creating a `Character` object was even *more* difficult? Like, if it required making database queries or other operations? Our goal is to make the instantiation of `Character` objects easier and more clear. And we can accomplish that by creating a builder class.

Add a `src/Builder/` directory for organization and, inside of that, a new PHP class called `CharacterBuilder`. I'm creating this class but I am *not* creating a corresponding interface. Builder classes *often* implement an interface like `CharacterBuilderInterface`, but they don't *need* to. Later, we'll talk about why you *might* decide to add an interface in *some* situations.

8 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 4
class CharacterBuilder
{
}
```

Methods and Method Chaining

Okay, inside, we get to create whatever methods we want to allow the outside world to craft characters. For

example, `public function setMaxHealth()`, which will accept an `int $maxHealth` argument. I'm going to leave this method empty for the moment... but it well eventually return *itself*: it will return `CharacterBuilder`. This is really common in the builder pattern because it allows method chaining, also known as a "fluent interface". But, it's not a *requirement* of the pattern.

All right, let's quickly fill in a few more methods, like `setBaseDamage()` ... and the last two are the armor and attack types. So say `setAttackType()`. Remember, attack types are *objects*. But instead of allowing an `AttackType` interface argument, I'm going to accept a `string` argument called `$attackType`. Why? I don't *have* to this, but I'm trying to make it as *easy* as possible to create characters. So instead of making *someone else* instantiate the attack types, I'm going to allow them to pass a simple string - like the word `bow` - and, in a few minutes, *we* will handle the complexity of instantiating the object for them.

Okay, copy that, and do the same for `setArmorType()`.

34 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 6
class CharacterBuilder
{
    public function setMaxHealth(int $maxHealth): self
    {
... line 11
    }

    public function setBaseDamage(int $baseDamage): self
    {
... line 16
    }

    public function setAttackType(string $attackType): self
    {
... line 21
    }

    public function setArmorType(string $armorType): self
    {
... line 26
    }

... lines 29 - 32
}
```

And... that's it! Those are the only four things that you can control in a character.

The Creational Method

The *final* method that our builder needs is the one that will actually *build* the `Character`. You can call this anything you want, how about `buildCharacter()`. And it is, of course, going to return a `Character` object.

34 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 4
use App\Character\Character;
... line 6
class CharacterBuilder
{
... lines 9 - 28
    public function buildCharacter(): Character
    {
... line 31
    }
}
```

To store the character stats, we're going to create *four* properties, which I'll paste in: `private int $maxHealth`, `private int $baseDamage`, and then `private string $attackType` and `private string $armorType`. Then, in each method,

assign that property and `return $this`. We'll do that for `$baseDamage` ... `$attackType` ... and `$armorType`.

80 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 14
class CharacterBuilder
{
    private int $maxHealth;
    private int $baseDamage;
    private string $attackType;
    private string $armorType;

    public function setMaxHealth(int $maxHealth): self
    {
        $this->maxHealth = $maxHealth;

        return $this;
    }

    public function setBaseDamage(int $baseDamage): self
    {
        $this->baseDamage = $baseDamage;

        return $this;
    }

    public function setAttackType(string $attackType): self
    {
        $this->attackType = $attackType;

        return $this;
    }

    public function setArmorType(string $armorType): self
    {
        $this->armorType = $armorType;

        return $this;
    }
... lines 49 - 78
}
```

Beautiful! The `buildCharacter()` method is fairly straightforward: we do whatever ugly work needed to create the `Character`. So I'll say `return new Character()` passing `$this->maxHealth` and `$this->baseDamage`. The last two arguments require objects... so they're a bit more complex. But that's ok! I don't mind if my builder gets a little complicated.

Doing some Heavy Lifting

I'll go to the bottom of this class and paste in two new `private` methods. These handle creating the `AttackType` and `ArmorType` objects. Except... I need a bunch of `use` statements for this, which I forgot. Whoops. So I'm going to re-type the end of these classes and hit "tab" to add those `use` statements. There we go!

Okay, we can now use the two new `private` methods to map the strings to *objects*. *This* is the heavy lifting - and the real value - of `CharacterBuilder`. Say `$this->createAttackType()` and `$this->createArmorType()`.

80 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 5
use App\ArmorType\IceBlockType;
use App\ArmorType\LeatherArmorType;
use App\ArmorType\ShieldType;
... line 9
use App\AttackType\BowType;
use App\AttackType\FireBoltType;
use App\AttackType\TwoHandedSwordType;
... lines 13 - 14
class CharacterBuilder
{
... lines 17 - 49
    public function buildCharacter(): Character
    {
        return new Character(
            $this->maxHealth,
            $this->baseDamage,
            $this->createAttackType(),
            $this->createArmorType(),
        );
    }

    private function createAttackType(): AttackType
    {
        return match ($this->attackType) {
            'bow' => new BowType(),
            'fire_bolt' => new FireBoltType(),
            'sword' => new TwoHandedSwordType(),
            default => throw new \RuntimeException('Invalid attack type given')
        };
    }

    private function createArmorType(): ArmorType
    {
        return match ($this->armorType) {
            'ice_block' => new IceBlockType(),
            'shield' => new ShieldType(),
            'leather_armor' => new LeatherArmorType(),
            default => throw new \RuntimeException('Invalid armor type given')
        };
    }
}
```

And... our builder is done! Next: let's use this in `GameApplication`. Then, we'll make our builder even *more* flexible (but not more difficult to use) by accounting for characters that use *multiple* attack types.

Chapter 5: Builder Improvements

The first version of our builder class is done! Though, in `GameApplication`, the `mage_archer` has *two* different attack types. Our `CharacterBuilder` doesn't support that right now... but we'll add it in a minute.

Clearing State After Building?

Oh, one more thing about the builder class! In the "build" method, after you create the object, you *may* choose to "reset" the builder object. For example, we could set the `Character` to a variable, then, before we return it, reset `$maxHealth` and all the other properties back to their original state. Why would we do this? Because it would allow for a *single* builder to be used over and over again to create *many* objects - or, characters in this case.

80 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 14
class CharacterBuilder
{
... lines 17 - 49
    public function buildCharacter(): Character
    {
        return new Character(
            $this->maxHealth,
            $this->baseDamage,
            $this->createAttackType(),
            $this->createArmorType(),
        );
    }
... lines 59 - 78
}
```

However, I'm *not* going to do that... which just means that a single `CharacterBuilder` will be meant to be used just *one* time to build *one* character. You can choose either option in your app: there isn't a right or wrong way for the builder pattern.

Using the Builder

All right, let's go use this! Inside of `GameApplication`, first, just to make life easier, I'm going to create a `private function` at the bottom called `createCharacterBuilder()` which will return `CharacterBuilder`. Inside, `return new CharacterBuilder()`.

84 lines | src/GameApplication.php

```
... lines 1 - 4
use App\Builder\CharacterBuilder;
... lines 6 - 14
class GameApplication
{
... lines 17 - 78
    private function createCharacterBuilder(): CharacterBuilder
    {
        return new CharacterBuilder();
    }
}
```

That's going to be nice because... up here in `createCharacter()`, we can use that. I'm going to clear out the old stuff... and now, use the fluid way to make characters: `$this->createCharacterBuilder()`, `->setMaxHealth(90)`, `->setBaseDamage(12)`, `->setAttackType('sword')` and `->setArmorType('shield')`. Oh, and, though I didn't do it, it would be nice to add constants on the builder for these strings, like `sword` and `shield`.

Finally, call `->buildCharacter()` to... *build* that character!

102 lines | src/GameApplication.php

```
... lines 1 - 7
class GameApplication
{
... lines 10 - 38
    public function createCharacter(string $character): Character
    {
        return match (strtolower($character)) {
            'fighter' => $this->createCharacterBuilder()
                ->setMaxHealth(90)
                ->setBaseDamage(12)
                ->setAttackType('sword')
                ->setArmorType('shield')
                ->buildCharacter(),
... lines 48 - 70
        };
    }
... lines 73 - 100
}
```

That's really nice! And it would be even *nicer* if creating a character were even *more* complex, like involving database calls.

To save some time, I'm going to paste in the other three characters, which look similar. Down here for our `mage_archer`, I'm currently using the `fire_bolt` attack type. We *do* need to re-add a way to have both `fire_bolt` *and* `bow`, *but* this should work for now.

102 lines | src/GameApplication.php

```
... lines 1 - 38
    public function createCharacter(string $character): Character
    {
        return match (strtolower($character)) {
... lines 42 - 48
            'archer' => $this->createCharacterBuilder()
                ->setMaxHealth(80)
                ->setBaseDamage(10)
                ->setAttackType('bow')
                ->setArmorType('leather_armor')
                ->buildCharacter(),

            'mage' => $this->createCharacterBuilder()
                ->setMaxHealth(70)
                ->setBaseDamage(8)
                ->setAttackType('fire_bolt')
                ->setArmorType('ice_block')
                ->buildCharacter(),

            'mage_archer' => $this->createCharacterBuilder()
                ->setMaxHealth(75)
                ->setBaseDamage(9)
                ->setAttackType('fire_bolt') // TODO re-add bow!
                ->setArmorType('shield')
                ->buildCharacter(),

            default => throw new \RuntimeException('Undefined Character')
        };
    }
... lines 73 - 102
```

Let's try it out! At your terminal, run:

```
php bin/console app:game:play
```

Hey! It didn't explode! That's always a happy sign. And if I fight as an `archer` ... I win! Our app still works!

Allow for Multiple Attack Types

So what about allowing our mage_archer's two attack types? Well, that's the *beauty* of the builder pattern. Part of our job, when we create the builder class, is to make life as *easy* as possible for whoever uses this class. That's why I chose to use `string $armorType` and `$attackType` instead of *objects*.

We can solve handling *two* different `AttackTypes` however we want. Personally, I think it would be cool to be able to pass multiple arguments. So let's make that happen!

Over in `CharacterBuilder`, change the argument to `...$attackTypes` with an "s", using the fancy `...` to accept any number of arguments. Then, since this will now hold an *array*, change the property to `private array $attackTypes` ... and down here, `$this->attackTypes = $attackTypes`.

88 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 15
class CharacterBuilder
{
... lines 18 - 19
    private array $attackTypes;
... lines 21 - 36
    public function setAttackType(string ...$attackTypes): self
    {
        $this->attackTypes = $attackTypes;

        return $this;
    }
... lines 43 - 86
}
```

Easy. Next we need to make a few changes down in `buildCharacter()`, like changing the `$attackTypes` strings into objects. To do that, I'm going to say `$attackTypes =` and... get a little fancy. You don't *have* to do this, but I'm going to use `array_map()` and the new short `fn` syntax - `(string $attackType) => $this->createAttackType($attackType)`. For the *second* argument of `array_map()` - the array that we *actually* want to map - use `$this->attackTypes`.

88 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 50
    public function buildCharacter(): Character
    {
        $attackTypes = array_map(fn(string $attackType) => $this->createAttackType($attackType), $this->attackTypes);
... lines 54 - 65
    }
... lines 67 - 88
```

Now, in the private method, instead of reading the *property*, read an `$attackType` argument.

88 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 67
    private function createAttackType(string $attackType): AttackType
    {
        return match ($attackType) {
... lines 71 - 74
        };
    }
... lines 77 - 88
```

Ok, we *could* have done this with a `foreach` loop... and if you like `foreach` loops better, *do it*. Honestly, I think I've been writing too much JavaScript lately. Anyways, this basically says:

> I want to loop over all of the "attack type" strings and, for each one, call this function where we change that `$attackType` string into an `AttackType` object. Then set all of those `AttackType` objects onto a *new* `$attackTypes` variable.

In other words, this is now an *array* of `AttackType` objects.

To finish this, say `if (count($attackTypes) === 1)`, *then* `$attackType = $attackTypes[0]` to grab the first and only attack type. Otherwise, say `$attackType = new MultiAttackType()` passing `$attackTypes`. Finally, at the bottom, use the `$attackType` variable.

88 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 50
    public function buildCharacter(): Character
    {
... line 53
        if (count($attackTypes) === 1) {
            $attackType = $attackTypes[0];
        } else {
            $attackType = new MultiAttackType($attackTypes);
        }
... line 59
        return new Character(
... lines 61 - 62
            $attackType,
... line 64
        );
    }
... lines 67 - 88
```

Phew! You can see it's a bit ugly, but that's okay! We're hiding the creation complexity *inside* this class. And we could easily unit test it.

Let's try things out. Run our command...

```
./bin/console app:game:play
```

... let's be a `mage_archer` and... awesome! No error! So... I'm going to assume that's all working.

Ok, in `GameApplication`, we're instantiating the `CharacterBuilder` *manually*. But what if the `CharacterBuilder` needs access to some services to do its job, like the EntityManager so it can make database queries?

Next, let's make this example more useful by seeing how we handle the creation of this `CharacterBuilder` object in a *real* Symfony app by leveraging the service container. We'll also talk about the *benefits* of the builder pattern.

Chapter 6: Builder in Symfony & with a Factory

What if, in order to instantiate the `Character` objects, `CharacterBuilder` needed to, for example, make a *query* to the database? Well, when we need to make a query, we normally give our class a constructor and then autowire the entity manager service. But `CharacterBuilder` *isn't* a service. You could *technically* use it like a service, but a service is a class where you typically only need a *single* instance of it in your app. In `GameApplication` however, we're creating one `CharacterBuilder` *per* character. If we *did* try to autowire `CharacterBuilder` into `GameApplication`, that *would* work. Symfony *would* autowire the EntityManager into `CharacterBuilder` and then it *would* autowire that `CharacterBuilder` object here. The *problem* is that we would then only have *one* `CharacterBuilder` ... when we actually need *four* to create our four `Character` objects.

This is why builder objects are commonly partnered with a builder *factory*. Let me undo all of the changes I just made to `GameApplication` ... and `CharacterBuilder`.

Creating a Factory

Over in the `Builder/` directory, create a new class called `CharacterBuilderFactory`:

12 lines | src/Builder/CharacterBuilderFactory.php

```
... lines 1 - 2
namespace App\Builder;

class CharacterBuilderFactory
{
... lines 7 - 10
}
```

By the way, there *is* a pattern called the *factory pattern*, which we won't *specifically* cover in this tutorial. But a "factory" is just a class whose job is to create *another* class. It, like the builder pattern, is a *creational pattern*. Inside of the factory class, create a new method called, how about... `createBuilder()`, which will return a `CharacterBuilder`. And inside of *that*, just `return new CharacterBuilder()`:

12 lines | src/Builder/CharacterBuilderFactory.php

```
... lines 1 - 4
class CharacterBuilderFactory
{
    public function createBuilder(): CharacterBuilder
    {
        return new CharacterBuilder();
    }
}
```

This `CharacterBuilderFactory` *is* a service. Even if we need *five* `CharacterBuilder` objects in our app, we only need *one* `CharacterBuilderFactory`. We'll just call this method on it *five* times.

That means, over in `GameApplication`, we can create a `public function __construct()` and autowire `CharacterBuilderFactory $characterBuilderFactory`. I'll also add `private` in front to make it a property:

107 lines | src/GameApplication.php

```
... lines 1 - 5
use App\Builder\CharacterBuilderFactory;
... lines 7 - 8
class GameApplication
{
    public function __construct(private CharacterBuilderFactory $characterBuilderFactory)
    {
    }
... lines 14 - 105
}
```

Then, down inside `createCharacterBuilder()`, instead of creating this by hand, rely on the factory: `return $this->characterBuilderFactory->createBuilder()`:

107 lines | src/GameApplication.php

```
... lines 1 - 8
class GameApplication
{
... lines 11 - 101
    private function createCharacterBuilder(): CharacterBuilder
    {
        return $this->characterBuilderFactory->createBuilder();
    }
}
```

The nice thing about this factory (and this is really the *purpose* of the factory pattern in general) is that we have *centralized* the instantiation of this object.

Getting Services into the Builder

How does that help our situation? Remember, the problem I imagined was this: What if our character builder needed a service like the `EntityManager`?

With our new setup, we can make that happen. I don't actually have Doctrine installed in this project, so instead of the `EntityManager`, let's require `LoggerInterface $logger`... and I'll again add `private` in front to turn that into a property:

98 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 14
use Psr\Log\LoggerInterface;

class CharacterBuilder
{
... lines 19 - 23
    public function __construct(private LoggerInterface $logger)
    {
    }
... lines 27 - 96
}
```

Then, down in `buildCharacter()`, just to test that this is working, use it: `$this->logger->info('Creating a character')`. I'll also pass a second argument with some extra info like `'maxHealth' => $this->maxHealth` and `'baseDamage' => $this->baseDamage`:

98 lines | src/Builder/CharacterBuilder.php

```
... lines 1 - 16
class CharacterBuilder
{
... lines 19 - 55
    public function buildCharacter(): Character
    {
        $this->logger->info('Creating a character!', [
            'maxHealth' => $this->maxHealth,
            'baseDamage' => $this->baseDamage,
        ]);
... lines 62 - 75
    }
... lines 77 - 96
}
```

`CharacterBuilder` now requires a `$logger` ... but `CharacterBuilder` is *not* a service that we'll fetch directly from the container. We'll get it via `CharacterBuilderFactory`, which *is* a service. So autowiring `LoggerInterface` will work here:

18 lines | src/Builder/CharacterBuilderFactory.php

```
... lines 1 - 4
use Psr\Log\LoggerInterface;

class CharacterBuilderFactory
{
    public function __construct(private LoggerInterface $logger)
    {
    }
... lines 12 - 16
}
```

Then, pass that *manually* into the builder as `$this->logger`:

18 lines | src/Builder/CharacterBuilderFactory.php

```
... lines 1 - 6
class CharacterBuilderFactory
{
... lines 9 - 12
    public function createBuilder(): CharacterBuilder
    {
        return new CharacterBuilder($this->logger);
    }
}
```

We're seeing some of the benefits of the factory pattern here. Since we've already centralized the instantiation of `CharacterBuilder`, anywhere that *needs* a `CharacterBuilder`, like `GameApplication`, doesn't need to change at all... even though we just added a constructor argument! `GameApplication` was already offloading the instantiation work to `CharacterBuilderFactory`.

To see if this is working, run:

```
./bin/console app:game:play -vv
```

The `-vv` will let us see log messages while we play. And... got it! Look! Our `[info] Creating a character` message popped up. We can't see the other stats on this screen, but they *are* in the log file. *Awesome*.

What does The Builder Pattern Solve?

So *that's* the builder pattern! What problems can it solve? Simple! You have an object that's difficult to instantiate, so you add a builder class to make life easier. It also helps with the Single Responsibility Principle. It's one of the strategies that helps abstract creation logic of a class *away* from the class that will *use* that object. Previously, in `GameApplication`, we had the complexity of both *creating* the `Character` objects *and* using them. We still have code here to use the builder, but most of the complexity now lives in the builder class.

Does my Builder Need an Interface?

Frequently, when you study this pattern, it will tell you that the builder (`CharacterBuilder`, for example) should implement a new interface, like `CharacterBuilderInterface`, which would have methods on it like `setMaxHealth()`, `setBaseDamage()`, etc. This is *optional*. When would you *need* it? Well, like all interfaces, it's useful if you need the flexibility to swap *how* your characters are created for some other implementation.

For example, imagine we created a *second* builder that implemented `CharacterBuilderInterface` called `DoubleMaxHealthCharacterBuilder`. This creates `Character` objects, but in a *slightly* different way... like maybe it *doubles* the `$maxHealth`. If both of those builders implemented `CharacterBuilderInterface`, then inside of our `CharacterBuilderFactory`, which would now *now* return `CharacterBuilderInterface`, we could read some configuration to figure out which `CharacterBuilder` class we want to use.

So creating that interface really has less to do with the builder pattern itself... and more to do with making your code more flexible. Let me undo that fake code inside of `CharacterBuilderFactory`. And... inside of `CharacterBuilder`, I'll remove that make-believe interface.

Where Do We See the Builder Pattern?

And where do we see the builder pattern in the wild? This one is pretty easy to spot because method chaining is such a common feature of builders. The first example that comes to mind is Doctrine's `QueryBuilder`:

```
class CharacterRepository extends ServiceEntityRepository
{
    public function findHealthyCharacters(int $healthMin): array
    {
        return $this->createQueryBuilder('character')
            ->orderBy('character.name', 'DESC')
            ->andWhere('character.maxHealth > :healthMin')
            ->setParameter('healthMin', $healthMin)
            ->getQuery()
            ->getResult();
    }
}
```

It allows us to configure a query with a bunch of nice methods before finally calling `getQuery()` to actually create the `Query` object. It also leverages the factory pattern: to create the builder, you call `createQueryBuilder()`. That method, which lives on the base `EntityRepository` is the "factory" responsible for instantiating the `QueryBuilder`.

Another example is Symfony's `FormBuilder`:

```
public function buildForm(FormBuilderInterface $builder, $options)
{
    $animals = ['', '', '', ''];
    $builder
        ->add('name', TextType::class)
        ->add('animal', ChoiceType::class, [
            'placeholder' => 'Choose an animal',
            'choices' => array_combine($animals, $animals),
        ]);
}
```

In that example, *we* don't call the `buildForm()` method, but *Symfony* eventually *does* call this once we're done configuring it.

Ok team, let's talk about the *observer pattern* next.

Chapter 7: The Observer Pattern

Time for pattern number three - the *observer pattern*. Here's the technical definition:

The Definition

> The observer pattern defines a one-to-many dependency between objects so that when one object changes state, *all* of its dependents are notified and updated automatically.

Okay, *not bad*, but let's try my version:

> The observer pattern allows a bunch of objects to be notified by a *central* object when something happens.

This is the classic situation where you write some code that needs to be called whenever something *else* happens. And there are actually *two* strategies to solve this: the *observer pattern* and the *pub-sub* pattern. We'll talk about both. But first up - the *observer* pattern.

Anatomy of Observer

There are two different types of classes that go into creating this pattern. The first is called the "subject". That's the central object that will do some work and then notify *other* objects before or after that work. Those other objects are the second type, and they're called "observers".

This is pretty simple. Each observer tells the subject that it wants to be notified. Later, the subject loops over all of the observers and "notifies" them... which means it calls a method on them.

The Real-Life Challenge

Back in our app, we're going to make our game more interesting by introducing *levels* to the characters. Each time you win a fight, your character will earn some XP or "experience points". After you've earned enough points, the character will "level up", meaning it's base stats, like `$maxhealth` and `$baseDamage`, will increase.

To write this new functionality, we *could* put the code right here inside of `GameApplication` after the fight finishes. So... maybe down here in `finishFightResult()`, we would do the XP calculation and see if the character can level up:

107 lines | src/GameApplication.php

```
... lines 1 - 8
class GameApplication
{
... lines 11 - 88
    private function finishFightResult(FightResult $fightResult, Character $winner, Character $loser): FightResult
    {
        $fightResult->setWinner($winner);
        $fightResult->setLoser($loser);

        return $fightResult;
    }
... lines 96 - 105
}
```

But, to better organize our code, I want to put this new logic somewhere *else* and use the observer pattern to connect things. `GameApplication` will be the *subject*, which means *it* will be responsible for notifying any *observers* when a fight finishes.

Another reason, beyond code organization, that someone might choose the observer pattern is if

`GameApplication` lived in a third-party vendor library and that vendor library wanted to give *us* - the *user* of the library - some way to run code *after* a battle finishes... since we wouldn't have the luxury to just hack the code in `GameApplication`.

Creating the Observer Interface

Ok, step one to this pattern is to create an interface that all the observers will implement. For organization's sake, I'll create an `Observer/` directory. Inside, add a new PHP class, make sure "Interface" is selected, and call it, how about, `GameObserverInterface` ... since these classes will be "observing" something related to each game. `FightObserverInterface` would also have been a good name:

11 lines | src/Observer/GameObserverInterface.php

```
... lines 1 - 2
namespace App\Observer;
... lines 4 - 6
interface GameObserverInterface
{
... line 9
}
```

Inside we just need one `public` method. We can call it anything: how about `onFightFinished()`:

11 lines | src/Observer/GameObserverInterface.php

```
... lines 1 - 4
use App\FightResult;

interface GameObserverInterface
{
    public function onFightFinished(FightResult $fightResult): void;
}
```

Why do we need this interface? Because, in a minute, we're going to write code that loops over *all* of the observers inside of `GameApplication` and calls a method on them. So... we need a way to *guarantee* that each observer *has* a method, like `onFightFinished()`. And we can actually pass `onFightFinished()` whatever arguments we want. Let's pass it a `FightResult` argument because, if I want to run some code after a fight finishes, it'll probably be useful to know the *result* of that fight. I'll also add a `void` return type:

11 lines | src/Observer/GameObserverInterface.php

```
... lines 1 - 4
use App\FightResult;

interface GameObserverInterface
{
    public function onFightFinished(FightResult $fightResult): void;
}
```

Adding the Subscribe Code

Okay, step two: We need a way for every observer to *subscribe* to be notified on `GameApplication`. To do that, create a `public function` called, how about, `subscribe()`. You can name this anything. This is going to accept any `GameObserverInterface`, I'll call it `$observer` and it will return `void`. I'll fill in the logic in a moment:

118 lines | src/GameApplication.php

```
... lines 1 - 7
use App\Observer\GameObserverInterface;

class GameApplication
{
... lines 12 - 89
    public function subscribe(GameObserverInterface $observer): void
    {
        // TODO: Implement subscribe() method.
    }
... lines 94 - 116
}
```

The *second* part, which is *optional*, is to add a way to *unsubscribe* from the changes. Copy everything we just did... paste... and change this to `unsubscribe()`:

118 lines | src/GameApplication.php

```
... lines 1 - 9
class GameApplication
{
... lines 12 - 94
    public function unsubscribe(GameObserverInterface $observer): void
    {
        // TODO: Implement unsubscribe() method.
    }
... lines 99 - 116
}
```

Perfect!

At the top of the class, create a new array property that's going to hold all of the observers. Say `private array $observers = []` and then, to help my editor, I'll add some documentation: `@var GameObserverInterface[]`:

127 lines | src/GameApplication.php

```
... lines 1 - 9
class GameApplication
{
    /** @var GameObserverInterface[] */
    private array $observers = [];
... lines 14 - 125
}
```

Back down in `subscribe()`, populate this. I'll add a check for uniqueness by saying `if (!in_array($observer, $this->observers, true))`, then `$this->observers[] = $observer`:

127 lines | src/GameApplication.php

```
... lines 1 - 9
class GameApplication
{
... lines 12 - 92
    public function subscribe(GameObserverInterface $observer): void
    {
        if (!in_array($observer, $this->observers, true)) {
            $this->observers[] = $observer;
        }
    }
... lines 99 - 125
}
```

Do something similar down in `unsubscribe()`. Say `$key = array_search($observer, $this->observers)` and then

`if ($key !== false)` - meaning we *did* find that observer - `unset($this->observers[$key])` :

127 lines | src/GameApplication.php

```
... lines 1 - 9
class GameApplication
{
... lines 12 - 99
    public function unsubscribe(GameObserverInterface $observer): void
    {
        $key = array_search($observer, $this->observers, true);

        if ($key !== false) {
            unset($this->observers[$key]);
        }
    }
... lines 108 - 125
}
```

Notifying the Observers

Finally, we're ready to *notify* these observers. Right after the fight ends, `finishFightResult()` is called. So, right here, I'll say `$this->notify($fightResult)` :

136 lines | src/GameApplication.php

```
... lines 1 - 9
class GameApplication
{
... lines 12 - 108
    private function finishFightResult(FightResult $fightResult, Character $winner, Character $loser): FightResult
    {
... lines 111 - 113
        $this->notify($fightResult);

        return $fightResult;
    }
... lines 118 - 134
}
```

We don't *need* to do this... but I'm going to isolate the logic of notifying the observers to a new `private function` down here called `notify()` . It will accept the `FightResult $fightResult` argument and return `void` . Then `foreach` over `$this->observers as $observer` . And because we know that those are all `GameObserverInterface` instances, we can call `$observer->onFightFinished()` and pass `$fightResult` :

136 lines | src/GameApplication.php

```
... lines 1 - 9
class GameApplication
{
... lines 12 - 128
    private function notify(FightResult $fightResult): void
    {
        foreach ($this->observers as $observer) {
            $observer->onFightFinished($fightResult);
        }
    }
}
```

And... the *subject* - `GameApplication` - is *done*! By the way, sometimes the code that notifies the observers - so `notify()` in our case - lives in a `public` method and is meant to be called by someone *outside* of this class. That's just a variation on the pattern. Like with many of the small details of these patterns, you can do whatever you feel is best. I'm showing you the way *I* like to do things.

Next: let's implement an *observer* class, write the level-up logic, then hook it into our system.

Chapter 8: The Observer Class

Now that we've finished our *subject* class - `GameApplication` - where we can call `subscribe()` if we want to be notified after a fight finishes - let's turn to creating an *observer* that will calculate how much XP the winner should earn and whether or not the character should level up.

But first, we need to add a few things to the `Character` class to help. On top, add `private int $level` that will default to `1` and a `private int $xp` that will default to `0`:

87 lines | src/Character/Character.php

```
... lines 1 - 8
class Character
{
... lines 11 - 15
    private int $level = 1;
    private int $xp = 0;
... lines 18 - 85
}
```

Down here a bit, add `public function getLevel(): int` which will `return $this->level` ... and another convenience method called `addXp()` that will accept the new `$xpEarned` and return the *new* XP number. Inside say `$this->xp += $xpEarned` ... and `return $this->xp`:

87 lines | src/Character/Character.php

```
... lines 1 - 8
class Character
{
... lines 11 - 65
    public function getLevel(): int
    {
        return $this->level;
    }

    public function addXp(int $xpEarned): int
    {
        $this->xp += $xpEarned;

        return $this->xp;
    }
... lines 77 - 85
}
```

Finally, right after, I'm going to paste in one more method called `levelUp()`. We'll call *this* when a character levels up: it increases the `$level`, `$maxHealth`, and `$baseDamage`:

99 lines | src/Character/Character.php

```
... lines 1 - 8
class Character
{
... lines 11 - 65
    public function levelUp(): void
    {
        // +%15 bonus to stats
        $bonus = 1.15;

        $this->level++;
        $this->maxHealth = floor($this->maxHealth * $bonus);
        $this->baseDamage = floor($this->baseDamage * $bonus);

        // todo: level up attack and armor type
    }
... lines 77 - 97
}
```

We *could* also level-up the attack and armor types if we wanted.

Creating the Observer Class

Ok, *now* let's create that observer. Inside the `src/Observer/` directory, add a new PHP class. Let's call it `XpEarnedObserver`. And all of our observers need to `implement` the `GameObserverInterface`. Go to "Code generate", or `Command` + `N` on a Mac to implement the `onFightFinished()` method:

14 lines | src/Observer/XpEarnedObserver.php

```
... lines 1 - 2
namespace App\Observer;

use App\FightResult;

class XpEarnedObserver implements GameObserverInterface
{
    public function onFightFinished(FightResult $fightResult): void
    {
        // TODO: Implement onFightFinished() method.
    }
}
```

For the *guts* of `onFightFinished()`, I'm going to delegate the *real* work to a service called `XpCalculator`.

If you downloaded the course code, you should have a `tutorial/` directory with `XpCalculator.php` inside. Copy that, in `src/`, create a new `Service/` directory and paste that inside. You can check this out if you want to, but it's nothing fancy:

59 lines | src/Service/XpCalculator.php

```
... lines 1 - 2
namespace App\Service;

use App\Character\Character;

class XpCalculator
{
    public function addXp(Character $winner, int $enemyLevel): void
    {
        $xpEarned = $this->calculateXpEarned($winner->getLevel(), $enemyLevel);

        $totalXp = $winner->addXp($xpEarned);

        $xpForNextLvl = $this->getXpForNextLvl($winner->getLevel());
        if ($totalXp >= $xpForNextLvl) {
            $winner->levelUp();
        }
    }

    private function calculateXpEarned(int $winnerLevel, int $loserLevel): int
    {
        $baseXp = 30;
        $rawXp = $baseXp * $loserLevel;

        $levelDiff = $winnerLevel - $loserLevel;
        return match (true) {
            $levelDiff === 0 => $rawXp,

            // You get less XP when the opponent is lower level than you
            $levelDiff > 0 => $rawXp - floor($loserLevel * 0.20),

            // You get extra XP when the opponent is higher level than you
            $levelDiff < 0 => $rawXp + floor($loserLevel * 0.20),
        };
    }

    private function getXpForNextLvl(int $currentLvl): int
    {
        $baseXp = 100;
        $xpNeededForCurrentLvl = $this->fibonacciProgressionFormula($baseXp, $currentLvl);
        $xpNeededForNextLvl = $this->fibonacciProgressionFormula($baseXp, $currentLvl + 1);

        // Since the character holds the total amount of XP earned we need to include
        // the XP needed for the current level.
        return $xpNeededForCurrentLvl + $xpNeededForNextLvl;
    }

    private function fibonacciProgressionFormula(int $baseXp, int $currentLvl): int
    {
        $currentLvl--;
        if ($currentLvl === 0) {
            return 0;
        }

        return $baseXp * ($currentLvl-1) + ($baseXp * ($currentLvl));
    }
}
```

It takes the `Character` that won, the enemy's level, and it figures out how much XP it should award to the winner. Then, if they're eligible to level up, it levels-up that character.

Over in `XpEarnedObserver`, we can use that. Create a constructor so that we can autowire in a `private readonly`

(`readonly` just to be super trendy) `XpCalculator $xpCalculator` :

23 lines | src/Observer/XpEarnedObserver.php

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

class XpEarnedObserver implements GameObserverInterface
{
    public function __construct(
        private readonly XpCalculator $xpCalculator
    ) {
    }
... lines 14 - 21
}
```

Below, let's set the `$winner` to a variable - `$fightResult->getWinner()` - and `$loser` to `$fightResult->getLoser()` . Finally, say `$this->xpCalculator->addXp()` and pass `$winner` and `$loser->getLevel()` :

23 lines | src/Observer/XpEarnedObserver.php

```
... lines 1 - 7
class XpEarnedObserver implements GameObserverInterface
{
... lines 10 - 14
    public function onFightFinished(FightResult $fightResult): void
    {
        $winner = $fightResult->getWinner();
        $loser = $fightResult->getLoser();

        $this->xpCalculator->addXp($winner, $loser->getLevel());
    }
}
```

Connecting the Subject & Observer

Beautiful! The subject and observer are now *done*. The final step is to instantiate the observer and make it *subscribe* to the subject: `GameApplication` . We're going to do this manually inside of `GameCommand` .

Open up `src/Command/GameCommand.php` , and find `execute()` , which is where we're currently initializing all of the code inside our app. In a few minutes, we'll see a more *Symfony* way of connecting all of this. For right now, say `$xpObserver = new XpEarnedObserver()` ... and pass that a `new XpCalculator()` service so it's happy. Then, we can say `$this->game` (which is the `GameApplication`) `->subscribe($xpObserver)` :

105 lines | src/Command/GameCommand.php

```
... lines 1 - 7
use App\Observer\XpEarnedObserver;
use App\Service\XpCalculator;
... lines 10 - 16
class GameCommand extends Command
{
... lines 19 - 25
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $xpObserver = new XpEarnedObserver(
            new XpCalculator()
        );
        $this->game->subscribe($xpObserver);
... lines 32 - 47
    }
... lines 49 - 103
}
```

So we're *subscribing* the observer before we actually run our app down here.

This means... we're ready! But, just to make it a bit more obvious if this is working, head back to `Character` and add *one more* function here called `getXp()`, which will return `int` via `return $this->xp`:

104 lines | src/Character/Character.php

```
... lines 1 - 8
class Character
{
... lines 11 - 89
    public function getXp(): int
    {
        return $this->xp;
    }
... lines 94 - 102
}
```

This will allow us, inside of `GameCommand` ... if you scroll down a bit to `printResults()` ... here we go... to add a few things like `$io->writeln('XP: ' . $player->getXp())` ... and the same thing for `Final Level`, with `$player->getLevel()`:

107 lines | src/Command/GameCommand.php

```
... lines 1 - 16
class GameCommand extends Command
{
... lines 19 - 78
    private function printResult(FightResult $fightResult, Character $player, SymfonyStyle $io)
    {
... lines 81 - 99
        $io->writeln('Damage received: ' . $fightResult->getDamageReceived());
        $io->writeln('XP: ' . $player->getXp());
        $io->writeln('Final Level: ' . $player->getLevel());
... lines 103 - 104
    }
}
```

Ok team - testing time! Spin over, run

```
./bin/console app:game:play
```

and let's play as the `fighter`, because that's still one of the toughest characters. And... awesome! Because we won, we received 30 XP. We're *still* Level 1, so let's fight a few more times. Aw... we lost, so no XP. Now we have 60 XP... 90 XP... woo! We leveled up! It says `Final Level: 2`. It's working!

What's great about this is that `GameApplication` doesn't need to know or care about the XP and the leveling up logic. It just notifies its subscribers and they can do whatever they want.

Next, let's see how we could wire all of this up using Symfony's *container*. We'll also talk about the *benefits* of this pattern and what parts of SOLID it helps with.

Chapter 9: Observer Inside Symfony + Benefits

We've implemented the Observer Pattern! The `GameApplication` is our subject, which notifies all of the observers... and we have *one* at the moment: `XpEarnedObserver`. Inside `GameCommand`, we connected all of this by *manually* instantiating the observer and `XpCalculator`... then calling `$this->game->subscribe()`:

107 lines | src/Command/GameCommand.php

```
... lines 1 - 16
class GameCommand extends Command
{
... lines 19 - 25
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $xpObserver = new XpEarnedObserver(
            new XpCalculator()
        );
        $this->game->subscribe($xpObserver);
... lines 32 - 47
    }
... lines 49 - 105
}
```

But... that isn't very Symfony-like.

Both `XpEarnedObserver` and `XpCalculator` are *services*. So we would *normally* autowire them from the container, not instantiate them manually. We *are* autowiring `GameApplication`... but our overall situation isn't quite right. In a perfect world, by the time Symfony gives us this `GameApplication`, Symfony's container would have *already* hooked up all of its observers so that it's ready to use *immediately*. How can we do that? Let's do it the simple way first.

Manually Specifying the Services

Remove all of the manual code inside of `GameCommand`:

107 lines | src/Command/GameCommand.php

```
... lines 1 - 16
class GameCommand extends Command
{
... lines 19 - 25
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $xpObserver = new XpEarnedObserver(
            new XpCalculator()
        );
        $this->game->subscribe($xpObserver);
... lines 32 - 47
    }
... lines 49 - 105
}
```

We're going to recreate this same setup... but inside `services.yaml`. Open that... and at the bottom, we need to modify the service `App\GameApplication`. But we don't need to configure any arguments. In this case, we need to configure some `calls`. Here, I'm basically telling Symfony:

> Yo! After you instantiate `GameApplication`, call the `subscribe()` method on it and pass, as an argument, the `@App\Observer\XpEarnedObserver` service.

29 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 25
    App\GameApplication:
        calls:
            - subscribe: ['@App\Observer\XpEarnedObserver']
```

So when we autowire `GameApplication`, Symfony will go grab the `XpEarnedObserver` service and *that* service will, of course, get `XpCalculator` autowired into *it*. This is pretty normal autowiring: the only special part is that Symfony will now call the `subscribe()` method on `GameApplication` before it passes that object to `GameCommand`.

In other words, this *should* work. Let's give it a try! Run:

```
./bin/console app:game:play
```

There are no errors so far and... oh. We lost. Bad luck. Let's try again! We won *and* we received 30 XP. It's working!

Setting up Autoconfiguration

The downside to this solution is that every time we add a new observer, we'll need to go to `services.yaml` and wire it *manually*. *Gasp*, how *undignified*...

Could we *automatically* subscribe all services that implement `GameObserverInterface`? Why, yes! And what an *excellent* idea! We can do that in two steps.

First, open `src/Kernel.php`. This isn't a file we work with much, but we're about to do some deeper things with the container and so this is exactly where we want to be. Go to Code Generate or `Command + O` and select "Override Methods". We're going to override one called `build()`:

20 lines | src/Kernel.php

```
... lines 1 - 6
use Symfony\Component\DependencyInjection\ContainerBuilder;
... lines 8 - 9
class Kernel extends BaseKernel
{
... lines 12 - 13
    protected function build(ContainerBuilder $container)
    {
... lines 16 - 17
    }
}
```

Perfect! The parent method is empty, so we don't need to call it at all. Instead, say `$container->registerForAutoconfiguration()`, pass it `GameObserverInterface::class`, and then say `->addTag()`. I'm going to invent a new tag here called `game.observer`:

20 lines | src/Kernel.php

```
... lines 1 - 9
class Kernel extends BaseKernel
{
... lines 12 - 13
    protected function build(ContainerBuilder $container)
    {
        $container->registerForAutoconfiguration(GameObserverInterface::class)
            ->addTag('game.observer');
    }
}
```

This probably isn't something you see very often (or ever) in *your* code, but it's really common in third-party bundles. This says that any service that implements `GameObserverInterface` should automatically be given this `game.observer` tag... assuming that service has `autoconfigure` enabled, which all of our services do.

That tag name could be *any* string... and it doesn't do anything at the moment: it's just a random string that's now attached to our service.

But we *should*, at least, be able to see it. Spin over and run:

```
./bin/console debug:container xpearnedobserver
```

It found our service! And check it out: `Tags - game.observer`.

Ok, now that our service has a *tag*, we're going to write a little more code that automatically calls the `subscribe` method on `GameApplication` for *every* service *with* that tag. This is *also* going to go in `Kernel`, but in a *different* method. In this case, we're going to implement something called a "compiler pass".

Add a new interface called `CompilerPassInterface`. Then, below, go back to "Code Generate", "Implement Methods", and select `process()`:

31 lines | src/Kernel.php

```
... lines 1 - 6
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
... lines 8 - 11
class Kernel extends BaseKernel implements CompilerPassInterface
{
... lines 14 - 21
    public function process(ContainerBuilder $container)
    {
... lines 24 - 28
    }
}
```

Compiler passes are a bit more advanced, but super cool! It's a piece of code that runs at the *very* end of the container and services being built... and you can do *whatever* you want inside.

Check it out! Say `$definition = $container->findDefinition(GameApplication::class)`:

31 lines | src/Kernel.php

```
... lines 1 - 4
use App\Observer\GameObserverInterface;
... lines 6 - 11
class Kernel extends BaseKernel implements CompilerPassInterface
{
... lines 14 - 21
    public function process(ContainerBuilder $container)
    {
        $definition = $container->findDefinition(GameApplication::class);
... lines 25 - 28
    }
}
```

No, this does *not* return the `GameApplication` object. It returns a `Definition` object that knows everything about *how* to instantiate a `GameApplication`, like its class, constructor arguments, and any calls it might have on it.

Next, say `$taggedObservers = $container->findTaggedServiceIds('game.observer')`:

31 lines | src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel implements CompilerPassInterface
{
... lines 14 - 21
    public function process(ContainerBuilder $container)
    {
        $definition = $container->findDefinition(GameApplication::class);
        $taggedObservers = $container->findTaggedServiceIds('game.observer');
... lines 26 - 28
    }
}
```

This will return an array of all the services that have the `game.observer` tag. Then we can loop over them with `foreach ($taggedObservers as $id => $tags)`. The `$id` is the service id... and `$tags` is an array because you can technically put the same tag on a service multiple times... but that's not something we care about:

31 lines | src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel implements CompilerPassInterface
{
... lines 14 - 21
    public function process(ContainerBuilder $container)
    {
        $definition = $container->findDefinition(GameApplication::class);
        $taggedObservers = $container->findTaggedServiceIds('game.observer');
        foreach ($taggedObservers as $id => $tags) {
... line 27
        }
    }
}
```

Now say `$definition->addMethodCall()`, which is the PHP version of `calls` in YAML. Pass this the `subscribe` method and, for the arguments, a `new Reference()` (the one from `DependencyInjection`), with `id`:

31 lines | src/Kernel.php

```
... lines 1 - 8
use Symfony\Component\DependencyInjection\Reference;
... lines 10 - 11
class Kernel extends BaseKernel implements CompilerPassInterface
{
... lines 14 - 21
    public function process(ContainerBuilder $container)
    {
        $definition = $container->findDefinition(GameApplication::class);
        $taggedObservers = $container->findTaggedServiceIds('game.observer');
        foreach ($taggedObservers as $id => $tags) {
            $definition->addMethodCall('subscribe', [new Reference($id)]);
        }
    }
}
```

This is a fancy way of saying that we want the `subscribe()` method to be called on `GameApplication` ... and for it to pass the *service* that holds the `game.observer` tag.

The end result is the same as what we had before in `services.yaml` ... just more dynamic and better for impressing your programmer friends. So, remove all of the YAML code we added:

29 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 25
    App\GameApplication:
        calls:
            - subscribe: ['@App\Observer\XpEarnedObserver']
```

If we try our game again...

```
./bin/console app:game:play
```

No errors! And... yes! It *still* works! If we need to add another observer later, we can just create a class, make it implement `GameObserverInterface` and... done! It will *automatically* be subscribed to `GameApplication`.

Observer Pattern in the Wild

So *that* is the observer pattern. How it looks can differ, with different method names for subscribing. Heck, sometimes the observers are passed in through the constructor! But the idea is *always* the same: a central object loops over and calls a method on a collection of other objects when something happens.

Where do we see this in the wild? It shows up in a lot of places, but here's *one* example. Over on Symfony's GitHub page, I'm going to hit "T" and search for a class called `LocaleSwitcher`. If you need to do something in your application each time the locale switches, you can register your code with the `LocaleSwitcher` and it will call you. In this case, the observers are passed through the constructor. And then you can see down here, after the locale is set, it loops over all of those and calls `setLocale()`. So `LocaleSwitcher` is the subject, and these are the observers.

How do you register an observer? Not surprisingly, it's by creating a class that implements `LocaleAwareInterface`. Thanks to autoconfiguration, Symfony will automatically tag your service with `kernel.locale_aware`. Yup, it uses the same mechanism for hooking all of this up that we just used!

Benefits of the Observer Pattern

The *benefits* of the observer pattern are actually best described by looking at the SOLID principles. This pattern helps the Single Responsibility pattern because you can encapsulate (or isolate) code into smaller classes.

Instead of putting everything into `GameApplication`, like all of our XP logic right here, we were able to isolate things in `XpEarnedObserver` and keep both classes more focused. This pattern also helps with the Open-closed Principle, because we can now extend the behavior of `GameApplication` *without* modifying its code.

The observer pattern also follows the Dependency Inversion Principle or DIP, which is one of the trickier principles if you ask me. Anyways, DIP is happy because the high-level class - `GameApplication` - accepts an interface - `GameObserverInterface` - and that interface was designed for the purpose of how `GameApplication` will use it. From GameApplication's perspective, this interface represents something that wants to "observe" what happens when something occurs within the game. Namely, the fight finishing. And so, `GameObserverInterface` is a good name.

But, if we had named it based on how the *observers* will *use* the interface, that would have made DIP sad. For example, had we called it `XpChangerInterface` and the method `timeToChangeTheXp`, *that* would be a violation of the Dependency Inversion Principle. If that's fuzzy and you want to know more, check out our SOLID tutorial.

Next, let's quickly turn to the brother pattern of observer: *Pub/sub*.

Chapter 10: Publish-Subscriber (PubSub)

The next pattern I want to talk about maybe *isn't* its own pattern? In reality, it's more of a *variation* of the observer pattern. It's called "pub/sub" or "publish-subscribe".

PubSub vs Observer

The key difference between observer and pub/sub is simply *who* handles notifying the observers. With the observer pattern, it's the *subject* - the thing (like `GameApplication`) that does the work. With pub/sub, there's a third object - usually called a "publisher" - whose *only* job is to handle this kind of stuff. Except, instead of calling it a "publisher", I'm going to use a word that's probably more familiar to you: *event dispatcher*.

With pub/sub, the observers (also called "listeners") tell the *dispatcher* which events they want to listen to. Then, the subject (whatever is doing the work) tells the *dispatcher* to dispatch the event. The dispatcher is then responsible for *calling* the listener methods.

You *could* argue that pub/sub better follows the Single Responsibility pattern. Battling characters and also registering and calling the observers are two separate responsibilities that we've jammed into `GameApplication`.

Creating the Event

So here's the new goal: add the ability to run code *before* a battle starts by using pub/sub.

Step one is to create an event class. This will be the object that is passed as an argument to all of the listener methods. Its purpose is pretty much *identical* to the `FightResult` that we're passing to our observers: it holds whatever data might be useful to a listener.

With the pub/sub pattern, it's customary to create an event class *just* for the event system. So inside of `src/`, I'm going to create a new `Event/` directory. Then a new PHP class. You can call it whatever you want, but for this tutorial, let's call it `FightStartingEvent`:

8 lines | src/Event/FightStartingEvent.php

```
... lines 1 - 2
namespace App\Event;

class FightStartingEvent
{
}
```

This class doesn't need to look like or extend anything... and we'll talk more about it in a minute.

Dispatching the Event

Step *two* is to dispatch this event *inside* of `GameApplication`. Instead of writing our own event dispatcher, we're going to use Symfony's. Let me break the constructor onto multiple lines... and then add a new `private EventDispatcherInterface $eventDispatcher`:

143 lines | src/GameApplication.php

```
... lines 1 - 10
use Symfony\Contracts\EventDispatcher\EventDispatcherInterface;

class GameApplication
{
... lines 15 - 17
    public function __construct(
        private CharacterBuilderFactory $characterBuilderFactory,
        private EventDispatcherInterface $eventDispatcher,
    )
    {
    }
... lines 24 - 141
}
```

Down in `play()`, right at the top, say `$this->eventDispatcher->dispatch()` passing `new FightStartingEvent()`:

143 lines | src/GameApplication.php

```
... lines 1 - 7
use App\Event\FightStartingEvent;
... lines 9 - 12
class GameApplication
{
... lines 15 - 24
    public function play(Character $player, Character $ai): FightResult
    {
        $this->eventDispatcher->dispatch(new FightStartingEvent());
... lines 28 - 52
    }
... lines 54 - 141
}
```

That's *it*! That's enough for the dispatcher to notify all of the code that is listening to the `FightStartingEvent`. Of course... at the moment, *nothing* is listening!

Registering Listeners... Manually

So *finally*, let's register a listener to this event. Open `GameCommand`: the place where we're initializing our app. We'll see how to do all of this properly with Symfony's container in a minute, but I want to keep it simple to start. In the constructor, add `private readonly EventDispatcherInterface $eventDispatcher`:

106 lines | src/Command/GameCommand.php

```
... lines 1 - 13
use Symfony\Component\EventDispatcher\EventDispatcherInterface;
... lines 15 - 16
class GameCommand extends Command
{
    public function __construct(
... line 20
        private readonly EventDispatcherInterface $eventDispatcher,
    )
    {
... line 24
    }
... lines 26 - 104
}
```

I know, I *am* being a little inconsistent between when I use `readonly` and not. Technically, I *could* use `readonly` on *all* of the constructor arguments... it's just not something I care about all that much. It does *look* cool though.

Choosing the Correct EventDispatcherInterface

Down here, anywhere before our app actually starts, say `$this->eventDispatcher->` . Notice that the only method this has is `dispatch()` . I made a... *tiny* mistake. Let's back up. In `GameApplication` , when I autowired `EventDispatcherInterface` , I chose the one from `Psr\EventDispatcher\EventDispatcherInterface` , which contains the `dispatch()` method we need. So that's *great*.

Inside of `GameCommand` , we autowired that *same* interface. But if you want the ability to attach listeners at *run time*, you need to autowire `EventDispatcherInterface` from `Symfony\Component\EventDispatcher` instead of `Psr` :

```
106 lines | src/Command/GameCommand.php
... lines 1 - 13
use Symfony\Component\EventDispatcher\EventDispatcherInterface;
... lines 15 - 106
```

The one from Symfony *extends* the one from `Psr` :

In reality, regardless of which interface you use, Symfony will *always* pass us the *same* object. That object *does* have a method on it called `addListener()` . So even if I had used the `Psr` interface, this method *would* have existed... it just would have looked funny inside of my editor.

Anyways, the first argument of this is the *name* of the event, which is going to match the class name that we're dispatching. So we can say `FightStartingEvent::class` . And then, to keep it simple, I'm going to be lazy and pass an inline `function()` . I'll also `use ($io)` ... so that inside I can say `$io->note('Fight is starting...')` :

```
106 lines | src/Command/GameCommand.php
... lines 1 - 16
class GameCommand extends Command
{
... lines 19 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
... line 29
        $this->eventDispatcher->addListener(FightStartingEvent::class, function() use ($io) {
            $io->note('Fight is starting...');
        });
... lines 33 - 46
    }
... lines 48 - 104
}
```

And... done! We're dispatching the event inside of `GameApplication` ... and since we've registered the listener here, it should be called!

Let's try it! At your terminal, say:

```
php ./bin/console app:game:play
```

We'll choose our character and... got it - `[NOTE] Fight is starting...` . If we battle again... we get the *same* message. *Awesome*!

Next, let's make this more powerful by passing information to our listener, like *who* is about to battle. Plus, we'll see how the event listener system is used in a *real* Symfony app by leveraging the container to wire everything up.

Chapter 11: Pub Sub Event Class & Subscribers in Symfony

We *are* able to run code right *before* a battle starts by registering what's called a "listener" to `FightStartingEvent`. As you can see, a listener can be any function... though what we see here is a bit less common. Usually a listener will be a method inside a class. And we'll refactor to that in a few minutes.

Passing Data to Listeners

But before we do, it might be useful to have a little bit more info in our listener function, like *who* is about to battle. That's the job of this event class. It can carry *whatever* data we want. For example, create a `public function __construct()` with two properties... which I'm going to make public for simplicity: `$player` and `$ai`:

13 lines | src/Event/FightStartingEvent.php

```
... lines 1 - 4
use App\Character\Character;

class FightStartingEvent
{
    public function __construct(public Character $player, public Character $ai)
    {
    }
}
```

Cool! Over in `GameApplication`, we need to pass those in: `$player` and `$ai`:

143 lines | src/GameApplication.php

```
... lines 1 - 12
class GameApplication
{
... lines 15 - 24
    public function play(Character $player, Character $ai): FightResult
    {
        $this->eventDispatcher->dispatch(new FightStartingEvent($player, $ai));
... lines 28 - 52
    }
... lines 54 - 141
}
```

Back over in our listener, this function will be passed a `FightStartingEvent` object. In fact, it was *always* being passed... it just wasn't useful before. Now we can say `Fight is starting against`, followed by `$event->ai->getNickname()`:

106 lines | src/Command/GameCommand.php

```
... lines 1 - 16
class GameCommand extends Command
{
... lines 19 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
... line 29
        $this->eventDispatcher->addListener(FightStartingEvent::class, function(FightStartingEvent $event) use ($io) {
            $io->note('Fight is starting against ' . $event->ai->getNickname());
        });
... lines 33 - 46
    }
... lines 48 - 104
}
```

Super nice. Give it a try! I'll run the command again and... sweet! We see

> ! [NOTE] Fight is starting against AI: Mage

The only thing I missed is the space after "against" so it looks nicer. I'll fix that really quick:

106 lines | src/Command/GameCommand.php

```
... lines 1 - 16
class GameCommand extends Command
{
... lines 19 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
... line 29
        $this->eventDispatcher->addListener(FightStartingEvent::class, function(FightStartingEvent $event) use ($io) {
            $io->note('Fight is starting against ' . $event->ai->getNickname());
        });
... lines 33 - 46
    }
... lines 48 - 104
}
```

Allowing Listeners to Control Behavior

As I mentioned, you can really put *whatever* data you want inside `FightStartingEvent`. Heck, you could create a `public $shouldBattle = true` property if you wanted. Then, in a listener, you could say `$event->shouldBattle = false` ... maybe because the characters have used *communication* and *honesty* to solve their problems. Brave move!

Anyways, in `GameApplication`, you could then set this event to a new `$event` object, dispatch it, and if they *shouldn't* battle, just `return`. Or you could `return new FightResult()` or throw an exception. Either way, you see the point. Your listeners can, in a sense, communicate *back* to the central object to control its behavior.

I'll undo all of that inside of `GameApplication`, `FightStartingEvent` and also `GameCommand`.

Creating an Event Subscriber

As easy as this inline listener is, it's more common to create a separate class for your listener. You can either create a *listener* class, which is basically a class that has this code here as a public function, *or* you can create a class called a *subscriber*. Both are completely valid ways to use the pub/sub pattern. The only difference is how you *register* a listener versus a subscriber, which is pretty minor, and you'll see that in a minute. Let's refactor to a subscriber because they're easier to set up in Symfony.

In the `Event/` directory, create a new PHP class called... how about... `OutputFightStartingSubscriber`, since this subscriber is going to *output* that a battle is beginning:

26 lines | src/Event/OutputFightStartingSubscriber.php

```
... lines 1 - 2
namespace App\Event;
... lines 4 - 9
class OutputFightStartingSubscriber implements EventSubscriberInterface
{
... lines 12 - 24
}
```

Event listeners don't need to extend any base class or implement any interface, but event *subscribers* do. They need to implement `EventSubscriberInterface`:

26 lines | src/Event/OutputFightStartingSubscriber.php

```
... lines 1 - 7
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class OutputFightStartingSubscriber implements EventSubscriberInterface
{
... lines 12 - 24
}
```

Go to "Code" -> "Generate" or `Command + N` on a Mac and select "Implement methods" to generate `getSubscribedEvents()`:

26 lines | src/Event/OutputFightStartingSubscriber.php

```
... lines 1 - 9
class OutputFightStartingSubscriber implements EventSubscriberInterface
{
... lines 12 - 18
    public static function getSubscribedEvents(): array
    {
... lines 21 - 23
    }
}
```

Nice! With an event subscriber, you'll list which events you subscribe to right inside this class. So we'll say `FightStartingEvent::class => 'onFightStart'`:

26 lines | src/Event/OutputFightStartingSubscriber.php

```
... lines 1 - 9
class OutputFightStartingSubscriber implements EventSubscriberInterface
{
... lines 12 - 18
    public static function getSubscribedEvents(): array
    {
        return [
            FightStartingEvent::class => 'onFightStart',
        ];
    }
}
```

This says:

> When the `FightStartingEvent` happens, I want you to call the `onFightStart()` method right inside this class!

Create that: `public function onFightStart()` ... which will receive a `FightStartingEvent` argument:

26 lines | src/Event/OutputFightStartingSubscriber.php

```
... lines 1 - 9
class OutputFightStartingSubscriber implements EventSubscriberInterface
{
    public function onFightStart(FightStartingEvent $event)
    {
... lines 14 - 16
    }
... lines 18 - 24
}
```

For the guts of this, go over to `GameCommand` and steal the `$io` line:

26 lines | src/Event/OutputFightStartingSubscriber.php

```
... lines 1 - 9
class OutputFightStartingSubscriber implements EventSubscriberInterface
{
    public function onFightStart(FightStartingEvent $event)
    {
... lines 14 - 15
        $io->note('Fight is starting against ' . $event->ai->getNickname());
    }
... lines 18 - 24
}
```

By the way, the `$io` object is kind of hard to pass from console commands into other parts of your code... so I'm going to ignore that complexity here and just create a new one with `$io = new SymfonyStyle(new ArrayInput([]), new ConsoleOutput()`:

26 lines | src/Event/OutputFightStartingSubscriber.php

```
... lines 1 - 4
use Symfony\Component\Console\Input\ArrayInput;
use Symfony\Component\Console\Output\ConsoleOutput;
use Symfony\Component\Console\Style\SymfonyStyle;
... lines 8 - 9
class OutputFightStartingSubscriber implements EventSubscriberInterface
{
    public function onFightStart(FightStartingEvent $event)
    {
        $io = new SymfonyStyle(new ArrayInput([]), new ConsoleOutput());

        $io->note('Fight is starting against ' . $event->ai->getNickname());
    }
... lines 18 - 24
}
```

Now that we have a subscriber, back in `GameCommand`, let's hook that up! Instead of `addListener()`, say `addSubscriber()`, and inside of that, `new OutputFightStartingSubscriber()`:

104 lines | src/Command/GameCommand.php

```
... lines 1 - 5
use App\Event\OutputFightStartingSubscriber;
... lines 7 - 16
class GameCommand extends Command
{
... lines 19 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
... line 29
        $this->eventDispatcher->addSubscriber(new OutputFightStartingSubscriber());
... lines 31 - 44
    }
... lines 46 - 102
}
```

Easy! Testing time! I'll exit, choose my character and... wow! It's working *so* well, it's outputting *twice*. We're amazing!

But... seriously, why is it printing twice? This is, once again, thanks to auto-configuration! Whenever you create a class that implements `EventSubscriberInterface`, Symfony's container is *already* taking that and registering it on the `EventDispatcher`. In other words, Symfony, internally, is already calling this line right here. So, we can delete it!

104 lines | src/Command/GameCommand.php

```
... lines 1 - 29
30          $this->eventDispatcher->addSubscriber(new OutputFightStartingSubscriber());
... lines 31 - 104
```

I guess that answers the question of:

> How do we use the pub/sub pattern in Symfony?

Just create a class, make it implement `EventSubscriberInterface` and... done! Symfony will automatically register it. To *dispatch* an event, create a new event class and dispatch that event anywhere in your code.

If we try this again (I'll exit the battle first)... it only outputs *once*. Great!

And... what are the benefits of pub/sub? They're really the same as the observer, though, in practice, pub/sub is a bit more common... probably because Symfony already has this great event dispatcher. Half of the work is already done *for* us!

Next, let's dive into our final pattern! It's one of my favorites *and*, I think, the most powerful in Symfony: The *decorator* pattern.

Chapter 12: The Decorator Pattern

One more design pattern to go! And honestly, I think we may have saved the best for last. It's the *decorator* pattern. This pattern is a *structural* pattern, so it's all about how you organize and connect related classes. That will make more sense as we uncover it.

Definition

Here's the technical definition:

> The decorator pattern allows you to attach new behaviors to objects by placing these objects inside special *wrapper* objects that contain the behaviors.

Yeah... Let's try this definition instead:

> The decorator pattern is like an intentional man-in-the-middle attack. You replace a class with your *custom* implementation, run some code, then call the true method.

Before we get any deeper and nerdier, let's see it in action.

The Goal

Here's the goal: I want to print something onto the screen whenever a player levels up. The logic for leveling up lives inside of `XpCalculator`:

59 lines | src/Service/XpCalculator.php

```
... lines 1 - 6
class XpCalculator
{
    public function addXp(Character $winner, int $enemyLevel): void
    {
        $xpEarned = $this->calculateXpEarned($winner->getLevel(), $enemyLevel);

        $totalXp = $winner->addXp($xpEarned);

        $xpForNextLvl = $this->getXpForNextLvl($winner->getLevel());
        if ($totalXp >= $xpForNextLvl) {
            $winner->levelUp();
        }
    }
... lines 20 - 57
}
```

But instead of changing the code in *this* class, we're going to apply the decorator pattern, which will allow us to run code before or *after* this logic... without actually *changing* the code inside.

This is a particularly common pattern to leverage if the class you want to modify is a *vendor* service that... you *can't* actually change. And *especially* if that class doesn't give us any *other* way to hook into it, like by implementing the observer or strategy patterns.

Adding the Interface to Support Decoration

For the decorator pattern to work, there's just one rule: the class that we want to decorate (meaning the class we want to extend or modify - `XpCalculator` in our case) needs to implement an interface. You'll see why in a few minutes. If `XpCalculator` were a *vendor* package, we... would just have to hope they did a good job and made it implement an interface.

But since this is *our* code, we can add one. In the `Service/` directory, create a new class... but change it to an interface. Let's call it `XpCalculatorInterface`. Then, I'll go steal the method signature for `addXp()`, paste that here, add a `use` statement and a semicolon:

11 lines | src/Service/XpCalculatorInterface.php

```
... lines 1 - 2
namespace App\Service;

use App\Character\Character;

interface XpCalculatorInterface
{
    public function addXp(Character $winner, int $enemyLevel): void;
}
```

Easy enough!

Over in `XpCalculator`, implement `XpCalculatorInterface`:

59 lines | src/Service/XpCalculator.php

```
... lines 1 - 6
class XpCalculator implements XpCalculatorInterface
{
... lines 9 - 57
}
```

And finally, open up `XpEarnedObserver`. This is the *one* place in our code that *uses* `XpCalculator`. Change this to allow *any* `XpCalculatorInterface`:

23 lines | src/Observer/XpEarnedObserver.php

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

class XpEarnedObserver implements GameObserverInterface
{
    public function __construct(
        private readonly XpCalculatorInterface $xpCalculator
    ) {
    }
... lines 14 - 21
}
```

This shows us *why* a class must implement an interface to support decoration. Because the classes that use our `XpCalculator` can now type-hint an *interface* instead of the concrete class, we're going to be able to swap out the *true* `XpCalculator` for our own class, known as the *decorator*. Let's create that class now!

Creating the Decorator

In the `src/Service/` directory, add a new PHP class and call it, how about, `OutputtingXpCalculator`, since it's an `XpCalculator` that will *output* things to the screen:

13 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 2
namespace App\Service;

class OutputtingXpCalculator implements XpCalculatorInterface
{
... lines 7 - 11
}
```

The most important thing about the decorator class is that it *must* call all of the *real* methods on the *real* service. Yup, we're literally going to pass the *real* `XpCalculator` *into* this one so we can call methods on it.

Create a `public function __construct()` and accept a `private readonly XpCalculatorInterface` called, how about, `$innerCalculator`. Our `OutputtingXpCalculator` *also* needs to implement `XpCalculatorInterface` so that it can be passed into things like our observer:

13 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 4
class OutputtingXpCalculator implements XpCalculatorInterface
{
    public function __construct(
        private readonly XpCalculatorInterface $innerCalculator
    )
    {
... line 11
}
```

Go to "Code"->"Generate" and select "Implement methods" to generate `addXp()`. I'll add the missing `use` statement and:

20 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 4
use App\Character\Character;

class OutputtingXpCalculator implements XpCalculatorInterface
{
... lines 9 - 14
    public function addXp(Character $winner, int $enemyLevel): void
    {
... line 17
    }
}
```

Perfect!

As I mentioned, the most important thing the decorator must *always* do is call that inner service in all of the public interface methods. In other words, say `$this->addXp($winner, $enemyLevel)` ... oh I mean `$this->innerCalculator->addXp()`:

20 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 6
class OutputtingXpCalculator implements XpCalculatorInterface
{
... lines 9 - 14
    public function addXp(Character $winner, int $enemyLevel): void
    {
        $this->innerCalculator->addXp($winner, $enemyLevel);
    }
}
```

A Chain of Decorators

Much better! With decorators, you create a chain of objects. In this case, we have two: the `OutputtingXpCalculator` will call into the true `XpCalculator`. One of the benefits of decorators is that you could have as *many* as you want: we could decorate our decorator to create *three* classes! We'll see this later!

Adding Custom Logic

Anyways, down here, we now have the ability to run code before *or* after we call the inner service. So *before*, say `$beforeLevel = $winner->getLevel()` to store the initial level. Then, below, `$afterLevel = $winner->getLevel()`. Finally, `if ($afterLevel > $beforeLevel)`, we know that we just leveled up!

32 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 7
class OutputtingXpCalculator implements XpCalculatorInterface
{
... lines 10 - 15
    public function addXp(Character $winner, int $enemyLevel): void
    {
        $beforeLevel = $winner->getLevel();

        $this->innerCalculator->addXp($winner, $enemyLevel);

        $afterLevel = $winner->getLevel();
        if ($afterLevel > $beforeLevel) {
... lines 24 - 28
        }
    }
}
```

And *that* calls for a celebration... like printing some stuff! I'll say `$output = new ConsoleOutput()` ... which is just a cheap way to write to the console, and then I'll paste in a few lines to output a nice message:

32 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 7
class OutputtingXpCalculator implements XpCalculatorInterface
{
... lines 10 - 15
    public function addXp(Character $winner, int $enemyLevel): void
    {
        $beforeLevel = $winner->getLevel();

        $this->innerCalculator->addXp($winner, $enemyLevel);

        $afterLevel = $winner->getLevel();
        if ($afterLevel > $beforeLevel) {
            $output = new ConsoleOutput();
            $output->writeln('--------------------------------');
            $output->writeln('<bg=green;fg=white>Congratulations! You\'ve leveled up!</>');
            $output->writeln(sprintf('You are now level "%d"', $winner->getLevel()));
            $output->writeln('--------------------------------');
        }
    }
}
```

Swapping in the Decorated Class into your App

Ok, our decorator class is done! But... how do we hook this up? What we need to do is *replace all* instances of `XpCalculator` in our system with our *new* `OutputtingXpCalculator` .

Let's do this manually first, without Symfony's fancy container stuff. There's only one place in our code that uses `XpCalculator` : `XpEarnedObserver` . Open up `src/Kernel.php` and temporarily comment-out the "subscribe" magic that we added earlier:

31 lines | src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel implements CompilerPassInterface
{
... lines 14 - 21
    public function process(ContainerBuilder $container)
    {
... lines 24 - 25
        foreach ($taggedObservers as $id => $tags) {
//          $definition->addMethodCall('subscribe', [new Reference($id)]);
        }
    }
}
```

I'm doing this because, for the moment, I want to *manually* instantiate `XpEarnedObserver` and *manually* subscribe it in `GameApplication` ... *just* so we can see how decoration works.

Over in `src/Command/GameCommand.php`, let's put back our manual observer pattern setup logic from earlier: `$xpCalculator = new XpCalculator()` and then `$this->game->subscribe(new XpEarnedObserver()` passing `$xpCalculator`:

105 lines | src/Command/GameCommand.php

```
... lines 1 - 7
use App\Observer\XpEarnedObserver;
use App\Service\XpCalculator;
... lines 10 - 16
class GameCommand extends Command
{
... lines 19 - 25
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $xpCalculator = new XpCalculator();
        $this->game->subscribe(new XpEarnedObserver($xpCalculator));
... lines 30 - 45
    }
... lines 47 - 103
}
```

We're not using the decorator yet... but this *should* be enough to keep our app working like before. When we try the command:

```
php ./bin/console app:game:play
```

We win! And we got some XP, which means `XpEarnedObserver` *is* doing its job.

So how do we use the decorator? By sneakily replacing the *real* `XpCalculator` with the fake one. Say `$xpCalculator = new OutputtingXpCalculator()`, and pass it the original `$xpCalculator`:

107 lines | src/Command/GameCommand.php

```
... lines 1 - 8
use App\Service\OutputtingXpCalculator;
... lines 10 - 17
class GameCommand extends Command
{
... lines 20 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $xpCalculator = new XpCalculator();
        $xpCalculator = new OutputtingXpCalculator($xpCalculator);
        $this->game->subscribe(new XpEarnedObserver($xpCalculator));
... lines 32 - 47
    }
... lines 49 - 105
}
```

That's *it*! Suddenly, even though it has no idea, `XpEarnedObserver` is being passed our decorator service! I told you it was sneaky!

So let's start over. Run the game again and battle a few times. The new decorator *should* print a special message the moment that we level up. I'll fight one more time and... got it! We're now level *two*. It works!

If you're wondering why the message printed *before* the battle actually started... that "might" be because these brave battle icons are... really just fancy decoration: technically the battle finishes before those show up.

Okay, we have *successfully* created a decorator class. Awesome! But how could we replace the `XpCalculator` service with the decorator via Symfony's *container*? Let's find out *one* way next. Then we'll do something even *cooler* with decoration after.

Chapter 13: Decoration with Symfony's Container

We just implemented the decorator pattern, where we basically wrapped the original `XpCalculator` in a warm hug with our `OutputtingXpCalculator`. Then... we quietly slipped *that* into the system in place of the *original*... without anyone else - like `XpEarnedObserver` - knowing or caring that we did that:

107 lines | src/Command/GameCommand.php

```
... lines 1 - 17
class GameCommand extends Command
{
... lines 20 - 26
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $xpCalculator = new XpCalculator();
        $xpCalculator = new OutputtingXpCalculator($xpCalculator);
        $this->game->subscribe(new XpEarnedObserver($xpCalculator));
... lines 32 - 47
    }
... lines 49 - 105
}
```

But to set up the decoration, I'm instantiating the objects *manually*, which is not very realistic in a Symfony app. What we really want is for `XpEarnedObserver` to autowire `XpCalculatorInterface` like normal, *without* us needing to do any of this manual instantiation. But we need the container to pass it our `OutputtingXpCalculator` decorator service, *not* the *original* `XpCalculator`. How can we accomplish that? How can we tell the container that whenever someone type-hints `XpCalculatorInterface`, it should pass our decorator service?

To answer that, let's start by undoing our manual code: In both `GameCommand` ... and then `Kernel` ... put back the fancy code that attaches the observer to `GameApplication`:

100 lines | src/Command/GameCommand.php

```
... lines 1 - 14
class GameCommand extends Command
{
... lines 17 - 23
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);

        $io->text('Welcome to the game where warriors fight against each other for honor and glory... and !');
... lines 29 - 40
    }
... lines 42 - 98
}
```

31 lines | src/Kernel.php

```
... lines 1 - 11
class Kernel extends BaseKernel implements CompilerPassInterface
{
... lines 14 - 21
    public function process(ContainerBuilder $container)
    {
... lines 24 - 25
        foreach ($taggedObservers as $id => $tags) {
            $definition->addMethodCall('subscribe', [new Reference($id)]);
        }
    }
}
```

If we try the command now:

```
php ./bin/console app:game:play
```

It *fails*:

> Cannot autowire service `XpEarnedObserver`: argument `$xpCalculator` references interface `XpCalculatorInterface` but no such service exists. You should maybe alias this interface to one of these existing services: `OutputtingXpCalculator` or `XpCalculator`.

Manually Wiring up the Service Decoration: Alias

That's a *great* error... and it makes sense. Inside of our observer, we're type-hinting the *interface* instead of a concrete class. And, unless we do a little more work, Symfony doesn't know *which* `XpCalculatorInterface` service to pass us. How do we tell it? By creating a service *alias*.

In `config/services.yaml`, say `App\Service\XpCalculatorInterface` set to `@App\Service\OutputtingXpCalculator`:

27 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 25
    App\Service\XpCalculatorInterface: '@App\Service\OutputtingXpCalculator'
```

This creates a service whose id is `App\Service\XpCalculatorInterface` ... but it's *really* just a "pointer", or "alias" to the `OutputtingXpCalculator` service. And remember, during autowiring, when Symfony sees an argument type-hinted with `XpCalculatorInterface`, to figure out which service to pass, it simply looks in the container for a service whose id matches that, so `App\Service\XpCalculatorInterface`. And now, it finds one!

So, let's try it again.

```
php ./bin/console app:game:play
```

And... it *still* doesn't work. We're on a roll!

> Circular reference detected for service `OutputtingXpCalculator`, path: `OutputtingXpCalculator` -> `OutputtingXpCalculator`

Oh! Symfony is autowiring `OutputtingXpCalculator` into `XpEarnedObserver` ... but it's *also* autowiring

`OutputtingXpCalculator` into *itself*:

32 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 7
class OutputtingXpCalculator implements XpCalculatorInterface
{
    public function __construct(
        private readonly XpCalculatorInterface $innerCalculator
    )
    {
    }
... lines 15 - 30
}
```

Whoops! We want `OutputtingXpCalculator` to be used *everywhere* in the system that autowires `XpCalculatorInterface` ... *except* for itself.

To accomplish that, back in `services.yaml`, we can manually configure the service. Down here, add `App\Service\OutputtingXpCalculator` with `arguments`, `$innerCalculator` (that's the name of our argument) set to `@App\Service\XpCalculator`:

31 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 27
    App\Service\OutputtingXpCalculator:
        arguments:
            $innerCalculator: '@App\Service\XpCalculator'
```

This will override the argument for *just* this one case. And now...

```
php ./bin/console app:game:play
```

It work? I mean, of course it works! If we play a few rounds and fast forward... yes! There's the "you've leveled up" message! It *did* go through our decorator!

This way of wiring the decorator is *not* our final solution. But before we get there, I have an even *bigger* challenge: let's completely *replace* a core Symfony service with our *own* via decoration. That's next!

Chapter 14: Decoration: Override Core Services & AsDecorator

In Symfony, decoration has a secret super-power: it allows us to customize nearly *any* service inside of Symfony. Woh.

For example, imagine that there's a core Symfony service and we need to extend its behavior with our own. How could we do that? Well, we *could* subclass the core service... and reconfigure things so that Symfony's container uses *our* class instead of the core one. That *might* work... but *this* is where decoration shines.

So, as a challenge, let's extend the behavior of Symfony's core `EventDispatcher` service so that whenever an event is dispatched, we dump a debugging message.

Investigating the Event Dispatcher

The ID of the service that we want to decorate is `event_dispatcher`

```
php ./bin/console debug:contianer event_dispatcher
```

And, fortunately, this class *does* implement an interface. Over on GitHub... on the `symfony/symfony` repository, hit `t` and open `EventDispatcher.php`.

And... yup! This implements `EventDispatcherInterface`. Decoration *will* work!

Creating the Decorator Class

Let's go make our decorator class. I'll create a new `Decorator/` directory... and inside, a new PHP class called... how about `DebugEventDispatcherDecorator`.

Step one, is always to implement the interface: `EventDispatcherInterface`... though this is a little tricky because there are *three* of them! There's `Psr`, which is the smallest... one from `Contract`, and *this* one from `Component`. The one from `Component` extends the one from `Contract`... which extends the one from `Psr`.

Which do we want? The "biggest" one: the one from `Symfony\Component`:

14 lines | src/Decorator/DebugEventDispatcherDecorator.php

```
... lines 1 - 2
namespace App\Decorator;

use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class DebugEventDispatcherDecorator implements EventDispatcherInterface
{
... lines 9 - 12
}
```

The *reason* is that, if our `EventDispatcher` decorator is going to be passed around the system in place of the *real* one, it needs to implement the *strongest* interface: the interface that has the *most* methods on it.

Go to "Code"->"Generate" - or `Command + N` on a Mac - and select "Implement methods" to add the *bunch* we needed. Whew... there we go!

55 lines | src/Decorator/DebugEventDispatcherDecorator.php

```
... lines 1 - 7
class DebugEventDispatcherDecorator implements EventDispatcherInterface
{
... lines 10 - 14
    public function dispatch(object $event, string $eventName = null): object
    {
... line 17
    }

    public function addListener(string $eventName, $listener, int $priority = 0)
    {
... line 22
    }

    public function addSubscriber(EventSubscriberInterface $subscriber)
    {
... line 27
    }

    public function removeListener(string $eventName, $listener)
    {
... line 32
    }

    public function removeSubscriber(EventSubscriberInterface $subscriber)
    {
... line 37
    }

    public function getListeners(string $eventName = null): array
    {
... line 42
    }

    public function getListenerPriority(string $eventName, $listener): ?int
    {
... line 47
    }

    public function hasListeners(string $eventName = null): bool
    {
... line 52
    }
}
```

The other thing we need to do is add a constructor where the *inner* `EventDispatcherInterface` will be passed to us... and make that a property with `private readonly`:

55 lines | src/Decorator/DebugEventDispatcherDecorator.php

```
... lines 1 - 7
class DebugEventDispatcherDecorator implements EventDispatcherInterface
{
    public function __construct(
        private readonly EventDispatcherInterface $eventDispatcher
    ) {
    }
... lines 14 - 53
}
```

Now that we have this, we need to *call* the inner dispatcher in *all* of these methods. This part is simple.... but boring. Say `$this->eventDispatcher->addListener($eventName, $listener, $priority)`:

55 lines | src/Decorator/DebugEventDispatcherDecorator.php

```
... lines 1 - 7
class DebugEventDispatcherDecorator implements EventDispatcherInterface
{
... lines 10 - 19
    public function addListener(string $eventName, $listener, int $priority = 0)
    {
        $this->eventDispatcher->addListener($eventName, $listener, $priority);
    }
... lines 24 - 53
}
```

We also need to check whether or not the method should return a value. We don't need to return in this method... but there *are* methods down here that *do* have return values, like `getListeners()`.

To avoid spending the next 3 minutes repeating what I just did 8 more times and putting you to sleep... bam! I'll just paste in the finished version:

55 lines | src/Decorator/DebugEventDispatcherDecorator.php

```
... lines 1 - 7
class DebugEventDispatcherDecorator implements EventDispatcherInterface
{
... lines 10 - 14
    public function dispatch(object $event, string $eventName = null): object
    {
        return $this->eventDispatcher->dispatch($event, $eventName);
    }

    public function addListener(string $eventName, $listener, int $priority = 0)
    {
        $this->eventDispatcher->addListener($eventName, $listener, $priority);
    }

    public function addSubscriber(EventSubscriberInterface $subscriber)
    {
        $this->eventDispatcher->addSubscriber($subscriber);
    }

    public function removeListener(string $eventName, $listener)
    {
        $this->eventDispatcher->removeListener($eventName, $listener);
    }

    public function removeSubscriber(EventSubscriberInterface $subscriber)
    {
        $this->eventDispatcher->removeSubscriber($subscriber);
    }

    public function getListeners(string $eventName = null): array
    {
        return $this->eventDispatcher->getListeners($eventName);
    }

    public function getListenerPriority(string $eventName, $listener): ?int
    {
        return $this->eventDispatcher->getListenerPriority($eventName, $listener);
    }

    public function hasListeners(string $eventName = null): bool
    {
        return $this->eventDispatcher->hasListeners($eventName);
    }
}
```

You can copy this from the code block on this page. We're simply calling the inner dispatcher in every method.

Finally, now that our decorator is doing all the things it *must* do, we can add our custom stuff. Right before the inner `dispatch()` method is called, I'll paste in two `dump()` lines and also dump `Dispatching event`, `$event::class`:

59 lines | src/Decorator/DebugEventDispatcherDecorator.php

```
... lines 1 - 7
class DebugEventDispatcherDecorator implements EventDispatcherInterface
{
... lines 10 - 14
    public function dispatch(object $event, string $eventName = null): object
    {
        dump('--------------------');
        dump('Dispatching event: ' . $event::class);
        dump('--------------------');

        return $this->eventDispatcher->dispatch($event, $eventName);
    }
... lines 23 - 57
}
```

AsDecorator: Making Symfony use OUR Service

Ok! Our decorator class is done! But, there are *many* places in Symfony that rely on the service whose ID is `event_dispatcher`. So here's the million dollar question: how can we replace *that* service with our *own* service... but still get the original event dispatcher passed to *us*?

Whelp, Symfony has a feature built *specifically* for this and you're going to love it! Go to the top of our decorator class, add a PHP 8 attribute called: `#[AsDecorator()]` and pass the ID of the service that we want to decorate: `event_dispatcher`:

61 lines | src/Decorator/DebugEventDispatcherDecorator.php

```
... lines 1 - 4
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;
... lines 6 - 8
#[AsDecorator('event_dispatcher')]
class DebugEventDispatcherDecorator implements EventDispatcherInterface
{
... lines 12 - 59
}
```

That's *it*. Seriously! This says:

> Hey Symfony! Thanks for being so cool! Also, please make *me* the *real* `event_dispatcher` service, but still autowire the *original* `event_dispatcher` service *into* me.

Let's try it! Run our app:

```
php ./bin/console app:game:play
```

And... it works! Look! You can see the event being dumped out! And there's our custom event too. And when I exit... another event at the bottom! We just replaced the core `event_dispatcher` service with our *own* by creating a *single* class. That's bananas!

Using AsDecorator with OutputtingXpCalculator

Could we have used this `AsDecorator` trick earlier for our own `XpCalculator` decoration situation? Yep! Here's how: In `config/services.yaml`, remove the manual arguments:

31 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 27
    App\Service\OutputtingXpCalculator:
        arguments:
            $innerCalculator: '@App\Service\XpCalculator'
```

And change the interface to point to the original, undecorated service: `XpCalculator`:

27 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 25
    App\Service\XpCalculatorInterface: '@App\Service\XpCalculator'
```

Basically, in the service config, we want to set things up the "normal" way, as if there were *no* decorators.

If we tried our app now, it *would* work, but it wouldn't be using our decorator. But *now*, go into `OutputtingXpCalculator` add `#[AsDecorator()]` and pass it `XpCalculatorInterface::class`, since that's the ID of the service we want to replace:

34 lines | src/Service/OutputtingXpCalculator.php

```
... lines 1 - 6
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;

#[AsDecorator(XpCalculatorInterface::class)]
class OutputtingXpCalculator implements XpCalculatorInterface
{
... lines 12 - 32
}
```

Donezo! If we try this now:

```
php ./bin/console app:game:play
```

No errors. An even faster way to prove this is working is by running:

```
php ./bin/console debug:container XpCalculatorInterface --show-arguments
```

And... check it out! It says that this is an *alias* for the service `OutputtingXpCalculator`. So anyone that's autowiring this interface will actually get the `OutputtingXpCalculator` service. And if you look down here at the arguments, the first argument passed to `OutputtingXpCalculator` is the *real* `XpCalculator`. That's amazing!

Multiple Decoration

All right, the decorator pattern is *done*. What a cool pattern! One feature of the decorator pattern that we only mentioned is that you can decorate a service as many times as you want. Yep! If we created *another* class that implemented `XpCalculatorInterface` and gave it this `#AsDecorator()` attribute, there would now be *two* services decorating it. Which service would be on the outside? If you care enough, you could set a `priority` option on one of the attributes to control that.

Decoration in the Wild?

Where do we see decoration in the wild? The answer to that is... sort of all over! In API Platform, it's common to

use decoration to extend core services like the `ContextBuilder`. And Symfony *itself* uses decoration pretty commonly to add debugging features while we're in the `dev` environment. For example, we know that this `EventDispatcher` class would be used in the `prod` environment. But in the *dev* environment - I'll hit `t` to search for a "TraceableEventDispatcher" - assuming that you have some debugging tools installed, *this* is the actual class that represents the `event_dispatcher` service. It *decorates* the real one!

I can prove it. Head back to your terminal and run:

```
php ./bin/console debug:container event_dispatcher --show-arguments
```

Scroll to the top and... check it out! The `event_dispatcher` service is an *alias* to `debug.event_dispatcher` ... whose class is `TraceableEventDispatcher`! And if you scroll down to its arguments, ha! *It's* passed our `DebugEventDispatcherDecorator` as an argument. Yup, there are *3* event dispatchers in this case: Symfony's core `TraceableEventDispatcher` is on the outside, it calls into *our* `DebugEventDispatcherDecorator` ... and then *that* ultimately calls the *real* event dispatcher. Inception!

Problems Solved by Decorator

And what problems does the decorator pattern solve? Simple: it allows us to extend the behavior of an *existing* class - like `XpCalculator` - even if that class does *not* contain any other extension points. This means we can use it to override vendor services when all else fails. The only downside to the decorator pattern is that we can only run code *before* or *after* the core method. And the service we want to decorate *must* implement an interface.

Okay, team. We're *done*! There are *many* more patterns out there in the wild: this was a collection of some of our favorites. If we skipped one or several that you really want to hear about, let us know! Until then, see if you can spot these patterns in the wild and figure out where *you* can apply them to clean up your own code... and impress your friends.

Thanks for coding with me, and I'll see you next time!