packt

1ST EDITION

Clean Code in PHP

Expert tips and practices to write beautiful, human friendly, and maintainable PHP

CARSTEN WINDLER | ALEXANDRE DAUBOIS

Preface

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Clean-Code-in-PHP`. If there's an update to the code, it will be updated in the GitHub repository.

Chapter 3

Code

Code sample 3.1:

Here's an example of assigning a variable at the same time as testing its value:

```
if null === ($var = method())
```

Code sample 3.2:

Here's an example of a ternary comparison:

```
$var === null ? 'is null' : 'is not null'
```

Code sample 3.3:

Example of nested ternary comparisons becoming unreadable:

```
$var === null ? 'is null' : is_int($var) ? 'is int' : 'is not
null'
```

Chapter 4

Code

Code sample 4.1:

An example of an extract of the file that Composer uses to install dependencies:

```
{
    "require": {
        "php": ">=7.3",
        "symfony/dotenv": "3.4.*",
        "symfony/event-dispatcher-contracts": "~1.1",
        "symfony/http-client": "^4.2.2"
    }
}
```

Chapter 6

Code

Code sample 6.1:

Let's see what attributes look like:

```
<?php
namespace App\Controller;
class ExampleController
{
    public function home(#[CurrentUser] User $user)
    {
        // ...
    }
}
```

Code Sample 6.2:

Here is an example of the match syntax:

```
$foo = match($var) {
    <value 1> => Bar::myMethod1(),
    <value 2> => Bar::myMethod2(),
};
```

Code Sample 6.3:

Here is an example of the named arguments

```
$this->myMethodCall(needle: 'Bar', enabled: true);
```

Code Sample 6.4:

Here is how read-only properties are declared:

```
<?php
namespace App\Model;
class MyValueObject
{
    protected readonly string $foo;
    public function __construct(string $foo)
    {
        $this->foo = $foo; // First assignment, all good
        // Any further assignment of $this->foo will result
          in a fatal error
    }
}
```

Code Sample 6.5:

Using the `#[SensitiveParameter]` attribute:

```
<?php
namespace App\Controller;
class SecurityController
{
    public function authenticate(string $username,
      #[SensitiveParameter] string $password)
    {
        // In case of any exception occurring or var_dump
          being called in here, the value of $password will
          be hidden in the different outputs
    }
}
```

Chapter 7

Code Quality Tools

In the previous parts of this book, we learned the basics of clean code. Now, it is time to apply that knowledge to our everyday work. There are literally dozens of tools available for the PHP ecosystem that can help us detect flaws and potential bugs, apply the correct code styling, and generally inform us about quality issues.

To ensure a quick and easy start within the world of code quality tools, this section will introduce you to the most commonly used ones. For each, you will learn how to install, configure, and use it directly on your code. You will also learn about some useful extra features they provide.

We will look at the following groups of tools:

- Syntax checking and code styling
- Static code analysis
- IDE extensions

Technical requirements

For this chapter, you only need a bare minimum of tools to already be set up. The chances are high that you already have them installed if you have ever worked with PHP code before:

- A local installation of a recent PHP version (PHP 8.0 or higher is recommended).
- A code editor – often called an Integrated Development Environment (IDE).

- Composer, either installed as binary or globally. Please check `https://getcomposer.org/` if you are not familiar with Composer yet.

Please note that for the rest of this book, all examples are based on a Linux environment such as Ubuntu or macOS. If you are using Windows for development, you will most likely need to make some adjustments, as described here: https://www.php.net/manual/en/install.windows.commandline.php.

The code files for this chapter can be found here: `https://github.com/PacktPublishing/Clean-Code-in-PHP`

Syntax checking and code styling

The first group of tools we want to discuss helps us keep our code syntactically correct (i.e., it can be executed correctly by PHP) and formatted in a structured way. It seems to be obvious that the code needs to be written without errors, but it is always good to double-check, as some tools can actively change your code. Having a simple and fast way to ensure this will be essential when we automate the whole code quality process later in this book.

Having your code formatted following a common style guide reduces the effort required to read and understand your code, as well as the code of others. Especially when you are working in a team, an accepted style guide saves you hours of discussions about how to correctly format the code.

We will learn about the following tools:

- A PHP built-in linter
- The PHP CS Fixer tool

The PHP built-in linter

The first tool we want to look at is actually not a code quality tool of its own but rather an option built into the PHP binary itself: the Linter. It checks any code for syntax errors without executing it. This is particularly useful to ensure that the code works after refactoring sessions or when your code has been changed by an external tool.

Installation and usage

Since the Linter is already part of your PHP installation, we can immediately start using it by looking at an example. If you look closely, you will probably notice the error the author made in the following class example:

```
<?php
class Example
{
    public function doSomething() bool
    {
        return true;
    }
}
```

Do not worry if you do not spot the error immediately – that is precisely what the Linter is there for! Simply pass the full name and path of the file to be checked to the PHP binary, using the `-l` option. By adding the `-f` option, PHP will also check for fatal errors, which is something we want. Both options can be combined into `-lf`.

Let us assume the preceding class can be found in the `example.php` file in the current folder – then, all we need to type is the following:

```
$ php -lf example.php
```

We will get the following output:

```
PHP Parse error: syntax error, unexpected identifier
  "bool", expecting ";" or "{" in example.php on line 5
Errors parsing example.php
```

You can tell the linter to check a complete directory as well:

```
$ php -lf src/*
```

> **Note**
>
> The built-in PHP linter stops on the first error – as in, it will not give you a full list of all the detected errors. So, you better make sure to run the command again after resolving the issue.

A recap of the PHP built-in Linter

The built-in PHP linter is a handy tool for quick code checks but cannot do much more than that. There are other more sophisticated linters such as https://github.com/overtrue/phplint. Not only will this one return a full list of errors but it can also run multiple processes in parallel, which will be noticeably faster on large code bases. However, other code quality tools already include a linter, such as the tool that we will check in the next section.

PHP CS Fixer: a code sniffer

Another essential tool is a code sniffer. It scans PHP code for coding standard violations and other bad practices. *PHP CS Fixer* (https://github.com/FriendsOfPHP/PHP-CS-Fixer) is a viable choice to start with, since, as the name already implies, it not only reports the findings but also fixes them right away.

> **Other code sniffers**
>
> *PHP CS Fixer* is not the only available code sniffer. Another well-known one is the *PHP_CodeSniffer* (https://github.com/squizlabs/PHP_CodeSniffer), which we can fully recommend using as well.

Installation and usage

Using Composer, the installation is straightforward:

```
$ composer require friendsofphp/php-cs-fixer --dev
```

> **Alternatives to Composer**
>
> There are multiple ways to install the tools we will introduce in this book. We will also check more options out later in this book.

The typical use case for code sniffers is to take care of the placement of brackets and the number of indentations, whether they're whitespaces or tabs. Let's check out the following PHP file with its ugly format:

```
<?php
class Example
{
  public function doSomething(): bool { return true; }
}
```

If we run the code sniffer with its default settings, the command is nice and short:

```
$ vendor/bin/php-cs-fixer fix example.php
```

This will scan and fix the `example.php` file all in one go, leaving our code neat and shiny:

```
<?php
class Example
{
```

```
    public function doSomething(): bool
    {
        return true;
    }
}
```

If you do not want to fix the file immediately, you can use the `--dry-run` option to just scan for issues. Add the `-v` option as well, to display the findings:

```
$ vendor/bin/php-cs-fixer fix example.php --dry-run -v
```

As with all code quality tools, you can also run it on all the files in a folder. The following command will scan the `src` folder recursively, so all subfolders are scanned as well:

```
$ vendor/bin/php-cs-fixer fix src
```

Rules and rulesets

So far, we used *PHP CS Fixer* with its default settings. Before we can change these defaults, let us have a closer look at how it knows what to check and fix.

A common pattern within code quality tools is the organization of rules within rulesets. A rule is a simple instruction that tells *PHP CS Fixer* how our code should be formatted regarding a certain aspect. For example, if we want to make use of strict types in PHP, every PHP file should contain the `declare(strict_types=1);` instruction.

There is a rule in *PHP CS Fixer* that can be used to force this:

```
$ vendor/bin/php-cs-fixer fix src
  --rules=declare_strict_types
```

This command will check each file in `src` and add `declare(strict_types=1);` after the opening PHP tag.

Since a coding standard such as PSR-12 (`https://www.php-fig.org/psr/psr-12/`) includes many instructions on how the code should be formatted, it would be cumbersome to add all these rules to the preceding command. That is why rulesets have been introduced, which are simply a combination of rules, and even other rulesets.

If we want to format code following PSR-12 explicitly, we can just run this:

```
$ vendor/bin/php-cs-fixer fix src --rules=@PSR12
```

As you can see, a ruleset is indicated by the `@` symbol.

Rules and ruleset documentation

It is impossible to discuss every rule and ruleset for *PHP CS Fixer* within the scope of this book. If you are curious about what else it has to offer, please check out the official GitHub repository: https://github.com/FriendsOfPHP/PHP-CS-Fixer/tree/master/doc

Configuration

Executing commands manually is fine to start with, but at some point, we will not want to remember all the options every time. That is where configuration files come into play: most PHP code quality tools allow us to store the desired configuration in one or more files and in various formats, such as YAML, XML, or plain PHP.

For *PHP CS Fixer*, all the relevant settings can be controlled via the `.php-cs-fixer.dist.php` configuration file. Here, you will find an example:

```
<?php
$finder = PhpCsFixer\Finder::create()
    ->in(__DIR__)
    ->exclude('templates');
$config = new PhpCsFixer\Config();
return $config->setRules([
    '@PSR12' => true,
    'declare_strict_types' => true,
    'array_syntax' => ['syntax' => 'short'],
])
->setFinder($finder);
```

Numerous things are happening here. Firstly, an instance of `PhpCsFixer\Finder` is created, which is configured to use the same directory to look for PHP files where this configuration file is located. As the `root` folder of the application is usually located here, we may want to exclude certain subdirectories (such as `templates` in this example) from being scanned.

Secondly, an instance of `PhpCsFixer\Config` is created. Here, we tell *PHP CS Fixer* which rules and rulesets to apply. We already discussed the `@PSR-12` ruleset, as well as the `declare_strict_types` rule. The `array_syntax` rule forces the usage of the short array syntax.

You may have noticed that the name of the configuration file, `.php-cs-fixer.dist.php`, contains the abbreviation `dist`. This stands for distribution and usually indicates that this file is the one the project gets distributed with. In other words, this is the file that gets added to the Git repository and is immediately available after checkout.

If you want to use your own configuration on your local system, you can create a copy of it and rename it `.php-cs-fixer.php`. If this file exists, *PHP CS Fixer* will use it instead of `dist-file`. It is good practice to let Git ignore this file. Otherwise, you might accidentally add your local settings to the repository.

Advanced usage

The ability of *PHP CS Fixer* does not stop at automatically fixing coding standard violations. It can also be used to apply small refactoring tasks. One great use case, for example, is the automated migration to a higher PHP version: *PHP CS Fixer* ships with migration rulesets, which can introduce some new language features to your code base.

For example, with PHP 8.0, it is possible to use the `class` keyword instead of the `get_class()` function. *PHP CS Fixer* can scan your code and replace certain lines – for example, see the following:

```
$class = get_class($someClass);
```

It can replace the preceding line with this:

```
$class = $someClass::class;
```

The migration rulesets are separated into non-risky and risky ones. Risky rulesets can potentially cause side effects, while non-risky ones usually do not cause any problems. A good example of a risky change is the `declare_strict_types` rule we discussed previously. Be sure to test your application thoroughly after applying them.

The capabilities of these migrations are limited – your code will not suddenly include all new PHP version features.

Code fixers cannot fix syntax errors for us. For example, the `Example` class that we checked with PHP's built-in linter in the previous section would still require the developer to manually fix it first.

> **Linting**
>
> *PHP CS Fixer* checks the files that you want to have sniffed for syntax errors as the very first step and will not apply any changes in case it finds syntax errors. This means that you do not have to run the PHP built-in linter as an additional step.

A recap of PHP CS Fixer

A code sniffer such as *PHP CS Fixer* should be part of every serious PHP project. The ability to fix rule violations automatically will save you many hours of work. If you chose not to apply any risky fixes, it will hardly cause any problems at all.

We have now learned how to ensure that our code is well-formatted and syntactically correct. While this is the foundation of any high-quality code, it does not help us to avoid bugs or maintainability issues. At this point, Static code analysis tools come into play.

Static Code Analysis

Static Code Analysis means that the only source of information is the code itself. Just by scanning the source code, these tools will find issues and problems that even the most senior developer in your team would miss during a code review.

These are the tools we would like to introduce you to in the next sections:

- phpcpd
- PHPMD
- PHPStan
- Psalm

phpcpd – the copy and paste detector

Copy and paste programming can be anything from simply annoying to a real threat to your projects. Bugs, security issues, and bad practices will get copied around and thus become harder to fix. Think of it as though it were a plague spreading through your code.

This form of programming is quite common, especially among less experienced developers, or in projects where the deadlines are very tight. Luckily, our clean code toolkit offers a remedy – the PHP copy and paste detector (phpcpd).

Installation and usage

This tool can only be downloaded as *a self-containing PHP archive* (`phar`), so we will not use Composer to install it this time:

```
$ wget https://phar.phpunit.de/phpcpd.phar
```

> **Handling phar files**
>
> In *Chapter 9*, *Organizing PHP Quality Tools*, we will learn how to keep `phar` files organized. For now, it's enough to just download it.

Once downloaded, *phpcpd* can be used immediately without further configuration. It just requires the path of the target directory as a parameter. The following example shows how to scan the `src` directory for so-called "clones" (i.e., code that has been copied multiple times). Let's execute it with the default settings first:

```
$ php phpcpd.phar src
phpcpd 6.0.3 by Sebastian Bergmann.
No clones found.
Time: 00:00, Memory: 2.00 MB
```

If *phpcpd* does not detect any clones, it is worth checking the two options, `min-lines` and `min-tokens`, that control its "pickiness":

```
$ php phpcpd.phar --min-lines 4 --min-tokens 20 src
phpcpd 6.0.3 by Sebastian Bergmann.
Found 1 clones with 22 duplicated lines in 2 files:
- /src/example.php:12-23 (11 lines)
  /src/example.php:28-39
  /src/example3.php:7-18
32.35% duplicated lines out of 68 total lines of code.
Average size of duplication is 22 lines, largest clone has
  11 of lines
Time: 00:00.001, Memory: 2.00 MB
```

The `min-lines` option allows us to set the minimum number of lines a piece of code needs to have until it is considered a clone.

To understand the usage of `min-tokens`, we must clarify the meaning of a token in this context first: when you execute a script, PHP will internally use a so-called "tokenizer" to split the source code up into single tokens. A token is an independent component of your PHP program, such as a keyword, an operator, a constant, or a string. Think of them as words in human language. The `min-tokens` option therefore controls the number of instructions a piece of code contains before it is considered a clone.

You may want to play around with both parameters to find a good balance of "pickiness" for your code base. A certain amount of redundancy in your code is not automatically a problem and you also do not want to bother your fellow developers too much. Using the defaults to start with, therefore, is a good choice.

Further options

There are two more options you should be aware of:

- `--exclude <path>`: Excludes a path from the analysis. For example, unit tests often contain a lot of copy-and-paste code, so you want to exclude the `tests` folder. If you need to exclude multiple paths, the options can be given multiple times.
- `--fuzzy`: With this especially useful option, *phpcpd* will obfuscate the variable names when performing its check. This way, clones will be detected even if the variable names have been changed by a smart but lazy colleague.

Recap of phpcpd

Although *phpcpd* is easy to use, it is a significant help against the slow spread of copy and paste code in your projects. That is why we recommend adding it to your clean coder toolkit.

PHPMD: the PHP mess detector

A mess detector will scan code for potential issues, also known as "code smells" – parts of code that can introduce bugs, unexpected behavior, or are, in general, harder to maintain. As with the code style, there are certain rules that should be followed to avoid problems. The mess detector applies those rules to our code. The standard tool in the PHP ecosystem for this is *PHPMD*, which we will show you in this section.

Installation and usage

Before we take a closer look at what this tool has to offer for us, let us install it first using Composer:

```
$ composer require phpmd/phpmd --dev
```

After the installation is complete, we can run *PHPMD* already on the command line. It requires three arguments:

- The filename or path to scan (e.g., `src`). Multiple locations can be comma-separated.
- One of the following formats in which the report should be generated: `html`, `json`, `text`, or `xml`.
- One or more built-in rulesets or ruleset XML files (comma-separated).

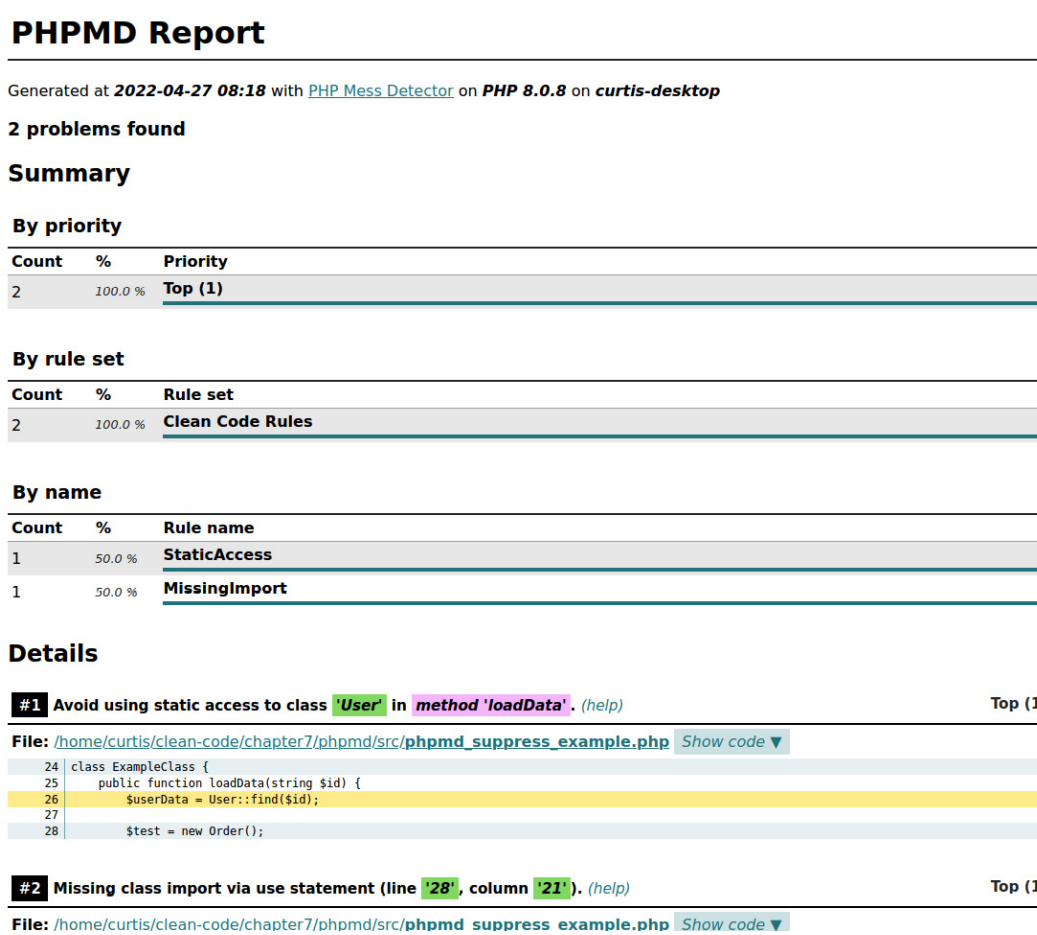
For a quick start, let's scan the `src` folder, create the output as text, and use the built-in `cleancode` and `codesize` rulesets. We can do this by running the following command:

```
$ vendor/bin/phpmd src text cleancode,codesize
```

PHPMD writes all output to the standard output (`stdout`), which is on the command line. However, all output formats except `text` are not meant to be read there. If you want to get a first overview, you may want to use the `html` output, as it generates a nicely formatted and interactive report. To store the output in a file, we will redirect it to a file using the `>` operator as follows:

```
$ vendor/bin/phpmd src html cleancode,codesize > phpmd_report.
html
```

Simply open the HTML file on your browser and you will see a report similar to the one shown in *Figure 7.1*:



PHPMD Report

Generated at ***2022-04-27 08:18*** with PHP Mess Detector on ***PHP 8.0.8*** on ***curtis-desktop***

2 problems found

Summary

By priority

Count	%	Priority
2	*100.0 %*	**Top (1)**

By rule set

Count	%	Rule set
2	*100.0 %*	**Clean Code Rules**

By name

Count	%	Rule name
1	*50.0 %*	**StaticAccess**
1	*50.0 %*	**MissingImport**

Details

#1 **Avoid using static access to class *'User'* in *method 'loadData'*.** *(help)* **Top (1)**

File: /home/curtis/clean-code/chapter7/phpmd/src/**phpmd_suppress_example.php** *Show code ▼*

```
24  class ExampleClass {
25      public function loadData(string $id) {
26          $userData = User::find($id);
27
28          $test = new Order();
```

#2 **Missing class import via use statement (line *'28'*, column *'21'*).** *(help)* **Top (1)**

File: /home/curtis/clean-code/chapter7/phpmd/src/**phpmd_suppress_example.php** *Show code ▼*

Figure 7.1: A PHPMD HTML report in a browser

The report is interactive, so make sure to click on buttons such as Show details or Show code to display all the information there is.

Rules and rulesets

In the preceding example, we used the built-in `cleancode` and `codesize` rulesets. Firstly, the rulesets are named according to the problem domain the rules check – as in, for the `cleancode` rule, you will only find rules that help to keep the code base clean. However, you can still end up with huge classes with many complex functions. To avoid this, adding the `codesize` ruleset is necessary.

The following table shows the available rulesets and their usage:

Ruleset	Short name	Description
Clean code rules	`cleancode`	Enforces clean code in general
Code size rules	`codesize`	Checks for long or complex code blocks
Controversial rules	`controversial`	Checks for best and bad practices where there are controversial opinions about them
Design rules	`design`	Helps find software design-related issues
Naming rules	`naming`	Avoids names that are too long or short
Unused code rules	`unused`	Detects unused code that can be deleted

Table 7.1: PHPMD rulesets

These built-in rules can simply be used by giving the aforementioned short names as arguments to the function call, as seen in the previous example.

If you are lucky enough to start a project on the green (i.e., from scratch), you can and should enforce as many rules from the beginning as you can. This will keep your code base clean right from the beginning. For existing projects, the effort is a bit greater, as we will see in the next section.

Using PHPMD in legacy projects

Often enough, you want to use *PHPMD* for an existing project, though. In this case, you will most likely be overwhelmed by the countless warnings that it will throw upon the first run. Do not give up – there are some options to help you!

Adjusting rulesets

If you plan to add *PHPMD* to an existing project, going all-in with the rulesets will surely lead to frustration because of how many issues are reported. You may want to concentrate on one or two rulesets at a time instead.

It is also highly likely that you will end up with rules that you find annoying or counter-productive at first – for example, the `ElseExpression` rule, which forbids the usage of `else` in an `if` expression. Leaving the discussion about the usefulness of this rule aside, the effort of rewriting countless statements that are working fine is not worth it. So, if you don't want to use that rule in your project, you need to create your own ruleset.

Rulesets are configured via XML files, which specify the rules that belong in them. Each rule is basically a PHP class that contains the rule logic. The following XML file defines a custom ruleset that just includes the `cleancode` and `codesize` rulesets:

```
<?xml version="1.0"?>
<ruleset name="Custom PHPMD rule set"
    xmlns=http://pmd.sf.net/ruleset/1.0.0
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation=http://pmd.sf.net/ruleset/1.0.0  http://
pmd.sf.net/ruleset_xml_schema.xsd
xsi:noNamespaceSchemaLocation=
  "http://pmd.sf.net/ruleset_xml_schema.xsd">
    <description>
        Rule set which contains all codesize and cleancode
        rules
    </description>
    <rule ref="rulesets/codesize.xml" />
    <rule ref="rulesets/cleancode.xml" />
</ruleset>
```

XML seems to be a bit out of fashion nowadays, but it still serves its purpose well. You usually do not need to worry about all the attributes of the `<ruleset>` tag – just make sure that they are present. The `<description>` tag can contain any text that you deem to be a good description for the ruleset.

The `<rule>` tag is important for us. In the preceding example, we referenced both the `codesize` and `cleancode` rules.

> **Tip**
>
> At this point, it is a good idea to dig through the built-in rule sets in the GitHub repository https://github.com/phpmd/phpmd/tree/master/src/main/resources/rulesets. Thanks to XML being a quite verbose file format, you will get familiar with it very quickly.

Imagine we want to remove the mentioned `ElseExpression` rule from our checks. To achieve this, you just need to add an `<exclude>` tag within the according `<rule>` tag as follows:

```
<rule ref="rulesets/cleancode.xml">
    <exclude name="ElseExpression" />
</rule>
```

This way, you can exclude as many rules from a ruleset as necessary. If you just want to pick certain rules from different rulesets, you can also go the other way round and reference the desired rules directly. If you want your custom ruleset to only include the `StaticAccess` and `UndefinedVariable` rules, your XML file should contain the following two tags:

```
<rule ref="rulesets/cleancode.xml/StaticAccess" />
<rule ref="rulesets/cleancode.xml/UndefinedVariable" />
```

One last important thing to know about the XML configuration files is how to change the individual properties of a rule. Again, a good way to figure out all the properties is to check out the actual ruleset file. Alternatively, you can check out the actual PHP classes of each rule at https://github.com/phpmd/phpmd/tree/master/src/main/php/PHPMD/Rule.

A typical example is to define exceptions for the `StaticAccess` rule. It is usually good practice to avoid static access, but often enough, you can't avoid it. Let us say your team agreed on allowing static access for the `DateTime` and `DateTimezone` objects – you can simply configure this as follows:

```
<rule ref="rulesets/cleancode.xml/StaticAccess">
    <properties>
        <property name="exceptions">
            <value>
                \DateTime,
                \DateTimezone
            </value>
        </property>
    </properties>
```

```
</rule>
```

To use this custom ruleset in the future, simply save the preceding XML in a file (usually called `phpmd.xml`) and pass it over to *PHPMD* upon the next run:

```
$ vendor/bin/phpmd src text phpmd.xml
```

> **Location of the configuration file**
>
> It is a common practice to place `phpmd.xml` with the rulesets you want to use in the `root` folder of your project and use it as the single source of configuration. If there are any modifications in the future, you only have to adjust one central file.

Suppressing warnings

Another useful tool for dealing with legacy code is the `@SuppressWarnings` DocBlock annotation. Let us assume one class in your project makes use of a static method call and that cannot be changed right now. By default, any static access will throw a warning. Since you do not want to use static access anywhere else in your code, but just in this class, removing the `StaticAccess` rule would be counterproductive.

In these cases, you can make use of the `@SuppressWarnings` annotation:

```
/**
* @SuppressWarnings(PHPMD.StaticAccess)
*/
class ExampleClass {
    public function getUser(int $id): User {
        return User::find($id);
    }
}
```

You can use multiple annotations in one DocBlock if required. Finally, if you want to suppress any warnings on a class, just use the `@SuppressWarnings(PHPMD)` annotation.

Be aware that using the `Suppress` annotations should be your last resort. It is very tempting to just add it everywhere. However, it will silence the output, but it will not solve the problems.

Accepting violations

Instead of suppressing warnings at the file level or excluding rules from rulesets, you can also decide to acknowledge existing violations. For example, when you want to use

PHPMD on a legacy project, you can decide to ignore all violations that are already in the code for now. However, if new violations are introduced by a new class, they will be reported. Luckily, *PHPMD* makes this task quite easy by providing a so-called baseline file, which it will generate for you automatically by running the following:

```
$ vendor/bin/phpmd src text phpmd.xml --generate-baseline
```

In the preceding command, we expect that a `phpmd.xml` file already exists in the project `root` folder. Using the preceding command, *PHPMD* will now create a file called `phpmd.baseline.xml`.

Now, you may run the following:

```
$ vendor/bin/phpmd src text phpmd.xml
```

The next time, *PHPMD* will automatically detect the previously generated baseline file and use it to suppress all warnings accordingly. However, if a new rule violation is introduced in a new location, it will still be detected and reported as a violation.

A word of warning: as with the `@SuppressWarning` annotation, the baseline feature is not a tool that can be used once and safely ignored in the future. The problematic code blocks are still part of your project as technical debt with all the negative effects. That is why if you decide to go with the baseline feature, you should make sure you don't forget about addressing these hidden problems in the future.

We will discuss how to deal with these problems later in the book. For now, it is only important for you how to update the baseline file from time to time. Again, *PHPMD* makes this an easy task. Simply run the following:

```
$ vendor/bin/phpmd src text phpmd.xml --update-baseline
```

All violations that no longer exist in your code will be removed from the baseline file.

A recap on PHPMD

Unless you are starting a project on the green, the configuration of *PHPMD* will require a bit more time. Especially if you are working within a team, you will spend more time arguing about which rules to use and which to exclude. Once this is done, though, you have a powerful tool at your disposal that will help developers write high-quality, maintainable code.

PHPStan – a static analyzer for PHP

You might have noticed that *PHPMD*, which we looked at in the previous section, was

not very PHP-specific but generally took care of the best coding practices. While this is, of course, very important, we want to use *PHPStan* to analyze our code with bad PHP practices in mind now.

As with every static analysis tool, *PHPStan* can only work with the information it can get out of the code. Therefore, it works better with modern object-oriented code. If, for example, the code makes strong use of strict typing, the analyzer has additional information to process, and will therefore return more results. But for older projects, it will be of immense help as well, as we will see in the following section.

Installation and usage

Installing *PHPStan* with Composer is just a one-liner again:

```
$ composer require phpstan/phpstan --dev
```

As with most code quality tools, *PHPStan* can be installed using PHAR. However, only when using Composer can you also install extensions. We will have a look at those a bit later in this section.

Let us use the following simplified example and store it inside the `src` folder:

```
<?php
class Vat
{
    private float $vat = 0.19;

    public function getVat(): int
    {
        return $this->vat;
    }
}
class OrderPosition
{
    public function getGrossPrice(float $netPrice): float
    {
        $vatModel = new Vat();
        $vat = $vatModel->getVat();
        return $netPrice * (1 + $vat);
    }
}
$orderPosition = new OrderPosition();
echo $orderPosition->getGrossPrice(100);
```

To execute a scan, you need to specify the `analyse` keyword, together with the path to scan, which is `src` in our case:

```
$ vendor/bin/phpstan analyse src
```

Figure 7.2 shows the output produced by *PHPStan*:

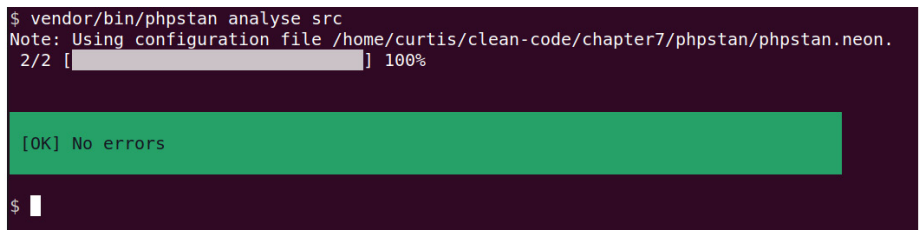


Figure 7.2: An example output of PHPStan

When we execute the PHP script, it will output `100`. Unfortunately, this is not correct because adding 19% of taxes to the net price should return 119, and not 100. So, there must be a bug somewhere. Let us see how *PHPStan* can help us here.

Rule levels

Unlike *PHPMD*, where you configure in detail which rules to apply, we will use different reporting levels here. These levels have been defined by the developers of *PHPStan*, starting from level 0 (just performing basic checks) to level 9 (being very strict on issues). To not overwhelm users with errors at first, *PHPStan* by default will use level 0, which only executes very few checks.

You can specify the level using the `level` (`-l|--level`) option. Let us try the next highest level:

```
$ vendor/bin/phpstan analyse --level 1 src
```

Using the level approach, you can effortlessly increase the quality of your code step by step, as we will demonstrate using the following, made-up example. Levels 1 and 2 will not return any errors either, though. As we eventually reach level 3, however, we will finally find a problem:

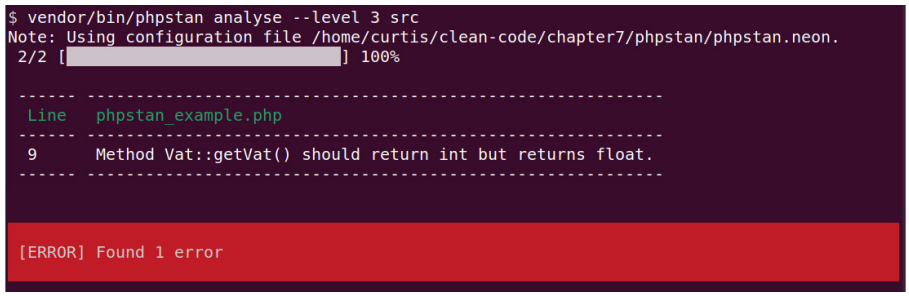


Figure 7.3: PHPStan reports one error with level 3

Checking our code again, we can spot the problem quickly: the `getVat()` method returns a float number (0.19) but using the `int` return type casts it to 0.

> **Strict typing**
>
> If we had used strict mode by adding the `declare(strict_types=1);` statement at the top of the example code, PHP would have thrown an error instead of silently casting the return value to `int`.

This demonstrates the beauty and power of Static Code Analysis: fixing this little bug will make our code work as expected and it takes us just a couple of seconds to do since we are still in our development environment. However, if this bug had reached the production environment, it would have taken us much longer to fix and left some angry customers behind.

Configuration

You can use configuration files to make sure that the same level and the same folders are always checked. The configuration is written in NEON (https://ne-on.org/), a file format that is very similar to YAML; if you can read and write YAML, it will work out just fine.

The basic configuration only contains the level and the folders to be scanned:

```
parameters:
    level: 4
    paths:
        - src
```

It is a good practice to save this configuration in a file named `phpstan.neon` in the `root` folder of your project. That is the location where *PHPStan* expects it to be by default. If you follow this convention, the next time you want to run it, you only need to

specify the desired action:

```
$ vendor/bin/phpstan analyse
```

If you used the above example configuration, *PHPStan* will now scan the `src` folder, using all rules from level 0 to level 4.

That is not everything you can configure here. In the next section, we will learn about some additional parameters.

Using PHPStan in legacy projects

If you want to use *PHPStan* in existing projects of a certain age, you will most likely end up with hundreds if not thousands of errors, depending on the chosen level. Of course, you can decide to keep using a lower level; but that also means that the analyzer will miss more bugs, not only existing ones but also in new or modified code.

In an ideal world, you would start with level 0, solve all errors, then continue with level 1, solve all new errors, and so on. This requires a lot of time, though, and, if no automated tests are available, a complete manual test run at the end would be necessary. You probably won't have that much time, so let us see what other options we have.

There are two ways that *PHPStan* can be told to ignore errors: firstly, using *PHPDocs* annotations, and secondly, using a special parameter in the configuration file.

Using PHPDocs annotations

To ignore a line of code, simply add a comment before or on the affected line, using the special `@phpstan-ignore-next-line` and `@phpstan-ignore-line` *PHPDocs* annotations:

```
// @phpstan-ignore-next-line
$exampleClass->foo();
$exampleClass->bar(); // @phpstan-ignore-line
```

Both lines of code will not be scanned for errors anymore. It is up to you to choose the way you prefer. It is not possible to ignore bigger code blocks or even entire functions or classes, though (unless you want to add a comment to every line, that is).

Using ignoreErrors parameters

The *PHPDocs* annotations are perfect for quick fixes in only a few locations, but you will need to touch many files if you wish to ignore numerous errors. Using the `ignoreErrors` parameter in the configuration file is not very comfortable, though, as you have to write a regular expression for every error you would like to ignore.

The following example will explain how it works. Let's assume we keep getting an error as follows:

```
Method OrderPosition::getGrossPrice() has no return type
specified.
```

Although theoretically, this would be easy to fix, the team decides against adding a type hint so as not to risk any side effects. The `OrderPosition` class is awfully written and not covered with tests, yet still works as expected. Since it will be replaced soon anyway, we are not willing to take the risk and touch it.

To ignore this error, we need to add the `ignoreErrors` parameter to our `phpstan.neon` configuration file:

```
parameters:
    level: 6
    paths:
        - src
    ignoreErrors:
        - '#^Method OrderPosition\:\:getGrossPrice\(\) has no
return type specified\.$#'
```

Instead of defining a rule or ruleset to ignore, we need to provide a regular expression here that matches the message of the error that should be ignored.

> **Tip**
>
> Writing regular expressions can be challenging. Luckily, the *PHPStan* website offers a very useful little tool to generate the necessary `phpstan.neon` part from the error message: https://phpstan.org/user-guide/ignoring-errors#generate-an-ignoreerrors-entry.

Upon the next run, the error will no longer be displayed regardless of where it occurs, as it matches the regular expression here.

PHPStan does not inform you about the fact that errors are ignored. Do not forget to fix them at some point! However, if you improve your code further over time, *PHPStan* will let you know when errors that are set to be ignored are no longer matched. You can safely remove them from the list then.

If you want to ignore certain errors completely, but just in one or more files or paths, you can do so by using a slightly different notation:

```
ignoreErrors:
```

```
    -
        message: '#^Method
          OrderPosition\:\:getGrossPrice\(\) has no return
          type specified\.$#'
        path: src/OrderPosition.php
```

The path needs to be relative to the location of the `phpstan.neon` configuration file. When given, the error will only be ignored if it occurs in `OrderPosition.php`.

Baseline

As we just saw in the previous section, adding errors you want to be ignored manually to your configuration file is a cumbersome task. But there is an easier way: similar to *PHPMD*, it is possible to automatically add all the current errors to the list of ignored errors at once by executing the following command with the `--generate-baseline` option:

```
$ vendor/bin/phpstan analyse --generate-baseline
```

The newly generated file, `phpstan-baseline.neon`, is in the same directory as the configuration file. PHPStan will not make use of it automatically, though. You have to include it manually in the `phpstan.neon` file as follows:

```
includes:
    - phpstan-baseline.neon
parameters:
    …
```

The next time you run PHPStan now, any previously reported errors should not be reported anymore.

Internally, the baseline file is nothing more than an automatically created list of the `ignoreErrors` parameters. Feel free to modify it to your needs. You can always regenerate it by executing `phpstan` using the `--generate-baseline` option again.

Extensions

It is possible to extend the functionality of *PHPStan*. The vivid community has already created a respectable number of useful extensions. For example, frameworks such as Symfony, Laminas, or Laravel often make use of magic methods (such as `__get()` and `__set()`), which cannot be analyzed automatically. There are extensions for these frameworks that provide the necessary information to *PHPStan*.

While we cannot cover these extensions in this book, we encourage you to check out the

extension library: https://phpstan.org/user-guide/extension-library. There are also extensions for PHPUnit, phpspec, and WordPress.

A recap of PHPStan

PHPStan is a powerful tool. We cannot cover all its functionality in just a few pages but we have given you a good idea of how to start using it. Once you are familiar with its basic usage, check out https://phpstan.org to learn more!

Psalm: A PHP static analysis linting machine

The next and last static code analyzer we want to introduce is *Psalm*. It will check our code base for so-called issues and report any violations. Furthermore, it can resolve some of these issues automatically. So, let us have a closer look.

Installation and usage

Once again, installing *Psalm* with Composer is just a matter of a few keystrokes:

```
$ composer require --dev vimeo/psalm
```

It is available as a `phar` file as well.

After installation, we cannot just start, though – rather, we need to set up a configuration file for the current project first. We can use the comfortable `--init` option to create it:

```
$ vendor/bin/psalm --init
```

This command will write a configuration file called `psalm.xml` in the current directory, which should be the project root. During its creation, *Psalm* checks whether it can find any PHP code and decides which error level is suitable, to begin with. Running *Psalm* doesn't require any more options:

```
$ vendor/bin/psalm
```

Configuration

The configuration file was already created during the installation process and could, for example, look similar to this:

```
<?xml version="1.0"?>
<psalm
    errorLevel="7"
```

```
    resolveFromConfigFile="true"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns=https://getpsalm.org/schema/config
    xsi:schemaLocation=https://getpsalm.org/schema/config
vendor/vimeo/psalm/config.xsd
>
    <projectFiles>
        <directory name="src" />
        <ignoreFiles>
            <directory name="vendor" />
        </ignoreFiles>
    </projectFiles>
</psalm>
```

Let us have a look at the attributes of the `<psalm>` node. You do not need to worry about the schema- and namespace-related information, only about the following two things:

- `errorLevel`: The levels go from 8 (basic checks) to 1 (very strict). In other words, the lower the level, the more rules will be applied.
- `resolveFromConfigFile`: Setting this to `true` lets *Psalm* resolve all relative paths (such as `src` and `vendor`) from the location of the configuration file – so usually, from the project root.

> **Psalm documentation**
>
> Psalm offers many more configuration options that we cannot cover in this book. As always, we recommend checking the documentation (https://psalm.dev/docs) to learn more about this tool.

Inside the `<psalm>` node, you will find more settings. In the previous example, *Psalm* is told to only scan the `src` folder and ignore all the files in the `vendor` folder. Ignoring the `vendor` folder is important, as we don't want to scan any third-party code.

Using Psalm in legacy projects

We will now have a look at how we can adjust *Psalm* to deal with existing projects better. As with the previous tools, there are basically two ways to ignore issues: using the configuration file or docblock annotations.

There are three code issue levels: `info`, `error`, and `suppress`. While `info` will just print info messages if minor issues have been found, issues that are at the level of an

`error` type, on the other hand, require you to get active. An issue of the `suppress` type will not be shown at all.

> **Continuous Integration**
>
> The difference between `info` and `error` becomes more important when building a Continuous Integration pipeline. `info` issues would let the build pass, while `error` issues would break it. We will have a closer look at this topic later.

Docblock suppression

The `@psalm-suppress` annotation can be used either in a function docblock or a comment for the next line. The `Vat` class from the previous examples could look as follows:

```
class Vat
{
    private float $vat = 0.19;
    /**
     * @psalm-suppress InvalidReturnType
     */
    public function getVat(): int
    {
        /**
         * @psalm-suppress InvalidReturnStatement
         */
        return $this->vat;
    }
}
```

Configuration file suppression

If we want to suppress issues, we need to configure `issueHandler` for them, where we can set the type to suppress manually. This is done in the configuration file by adding an `<issueHandler>` node inside the `<psalm>` node:

```
<issueHandlers>
    <InvalidReturnType errorLevel="suppress" />
    <InvalidReturnStatement errorLevel="suppress" />
</issueHandlers>
```

The preceding configuration would suppress all the `InvalidReturnType` and `InvalidReturnStatement` issues in the whole project. We can make this a bit more

specific, though:

```
<issueHandlers>
    <InvalidReturnType>
        <errorLevel type="suppress">
            <file name="Vat.php" />
        </errorLevel>
    </InvalidReturnType>
    <InvalidReturnStatement>
        <errorLevel type="suppress">
            <dir name="src/Vat" />
        </errorLevel>
    </InvalidReturnStatement>
</issueHandlers>
```

In the documentation (https://psalm.dev/docs/running_psalm/dealing_with_code_issues/), you will find even more ways to suppress issues – for example, by the variable name.

Baseline

As with the previous static code analyzers we discussed, *Psalm* also provides a feature to generate a baseline file, which will include all the current errors so that they will be ignored during the next run. Please note that the baseline feature only works for `error` issues, but not `info` issues. Let us create the file first:

```
$ vendor/bin/psalm --set-baseline=psalm-baseline.xml
```

Psalm has no default name for this file, so you need to pass it as an option to the command:

```
$ vendor/bin/psalm --use-baseline=psalm-baseline.xml
```

You can also add it as an additional attribute to the `<psalm>` node in the configuration file:

```
<psalm
    ...
    errorBaseline="./psalm-baseline.xml"
>
```

Finally, you can update the baseline file – for example, after you have made some

improvements to the code:

```
$ vendor/bin/psalm --update-baseline
```

Fixing issues automatically

Psalm will not only find the issue but it can also fix many of them automatically. It will let you know when this is the case and you can use the `--alter` option:

```
Psalm can automatically fix 1 issues.
Run Psalm again with
--alter --issues=InvalidReturnType --dry-run
to see what it can fix.
```

Let's execute the command as *Psalm* suggests:

```
$ vendor/bin/psalm --alter --issues=InvalidReturnType --dry-run
```

The `--dry-run` option tells *Psalm* to only show you what it would change as `diff`, but not to apply the changes. This way, you can check whether the change is correct:

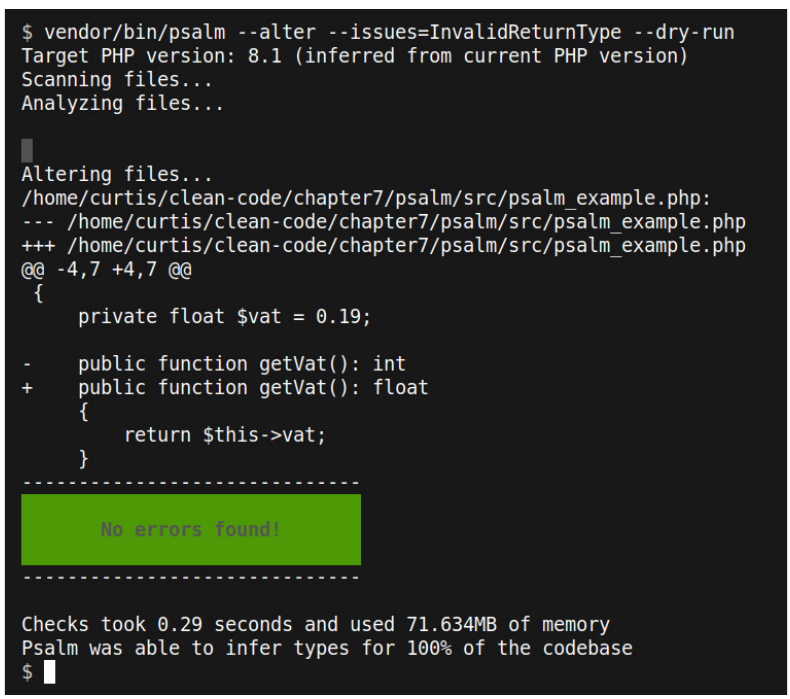


Fig 7.4: Psalm showing proposed changes

If you remove the `--dry-run` option, the changes will be applied.

A recap on Psalm

Psalm is a standard tool in the clean coder's toolkit for good reason. It is fast, easy to use, and powerful. Additionally, the code manipulation feature will save you a lot of time. Of course, there are numerous similarities with *PHPStan*, but often enough, you will find both tools working together on the same code base without problems. At least, you should consider giving it a try.

IDE extensions

The tools we looked at so far share something in common: they need to be applied to our code after we have written it. Of course, this is much better than nothing, but wouldn't it be great if the tools gave us their feedback immediately at the time that we wrote the code?

That is what many other developers thought as well, so they created extensions for the most popular IDEs, which are currently Visual Studio Code (VS Code) and PhpStorm:

- *PhpStorm* is an established, commercial IDE from JetBrains with several PHP-specific tools, checks, and built-in integrations for many of the code quality tools we discussed in this chapter. There are many useful extensions available for it as well. You can try it out for 30 days for free.
- *VS Code* is a highly flexible code editor from Microsoft with tons of third-party (partly commercial) extensions that can turn these tools into an IDE for virtually every relevant programming language today. Because the code editor itself is free, is it becoming more and more popular.

> **Alternative PHP IDEs**
>
> *PhpStorm* and *VS Code* are not the only IDEs that exist for PHP. Other alternatives are *NetBeans* (`https://netbeans.apache.org`), *Eclipse PDT* (https://www.eclipse.org), or *CodeLobster* (`https://www.codelobster.com`).

In this section, we will introduce you to three extensions for these two IDEs:

- PHP Inspections (EA Extended) for PhpStorm
- Intelephense for VS Code

Code quality tool integration in PhpStorm

PhpStorm offers seamless integration for the following tools that we have discussed: *PHP CS Fixer*, PHPMD, PHPStan, and Psalm. More information can be found here: https://www.jetbrains.com/help/phpstorm/php-code-quality-tools.html.

PHP Inspections (EA Extended)

This plugin (https://github.com/kalessil/phpinspectionsea) is for PhpStorm. It will add even more types of inspections to the pool of already existing ones, covering topics such as code style, architecture, or possible bugs.

IDE Inspections

Modern IDEs are already equipped with a lot of useful code checks. In PHPStorm, they are called *Inspections*. Some are already enabled by default – more can be activated manually (https://www.jetbrains.com/help/phpstorm/code-inspection.html#access-inspections-and-settings). For VS Code, you need to install an extension first. Check out the documentation (https://code.visualstudio.com/docs/languages/php) for more information.

Installation

As with every PhpStorm plugin, the installation is done via the File -> Settings -> Plugins dialog. You will find detailed information on how to install a plugin on the vendor's website (https://www.jetbrains.com/help/phpstorm/managing-plugins.html). Simply search for `EA Extended`. Please note that there is a second version of this plugin, EA Ultimate, which you have to pay for. We will not cover it in this book.

After installation, not all the inspections are immediately active. Let us have a look at the PhpStorm inspections configuration, as shown in *Figure 7.4*:

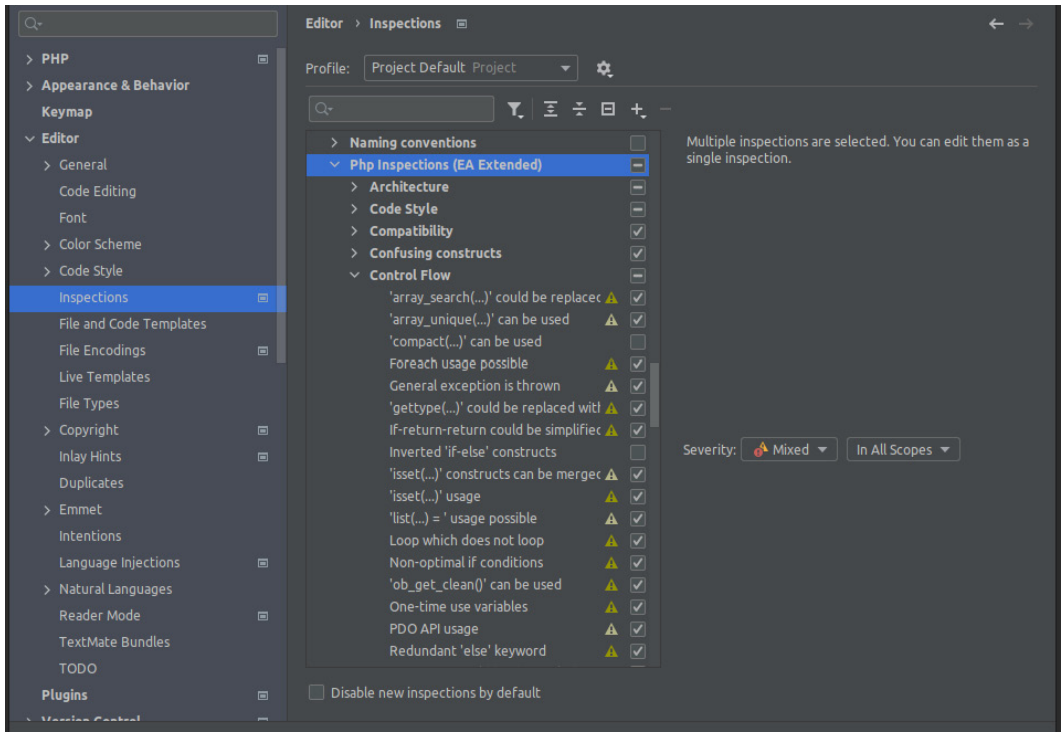


Figure 7.5: The Inspections configuration dialog in PhpStorm

All the inspections of this plugin can be found in the Php Inspections (EA Extended) section. The inspections that are not active by default can easily be activated by checking the checkbox next to them. We recommend reading the documentation (https://github.com/kalessil/phpinspectionsea/tree/master/docs) before activating any further inspections – otherwise, you might end up with too many rules. You can revisit them later.

Usage

PHP Inspections (EA Extended) not only warns you about problems but often also offers so-called Quick-Fixes, which let the IDE do the work for you. Here, you will find an example. Note the highlighted `if` clause on line 7:

Figure 7.6: Example code with an issue found by PHP Inspections (EA Extended)

When you hover your mouse pointer over the highlighted area, PhpStorm will show a pop-up window with further instructions about the suggested improvement:

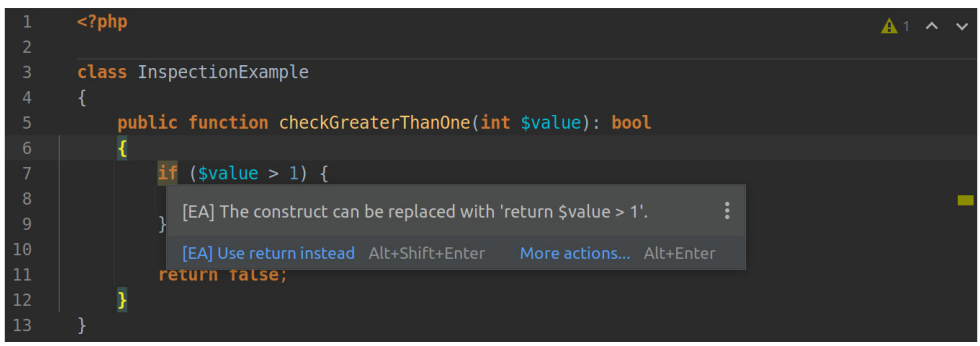


Figure 7.7: PHP Inspections (EA Extended) suggesting a code improvement

You can choose to fix the issue directly by pressing *Alt* + *Shift* + *Enter* at the same time, or you can click on the highlighted area to show the Quick-Fix bubble. If you click on the bubble, you will see a menu with some more options.

You can also invoke the following dialog by pressing *Alt + Enter*:

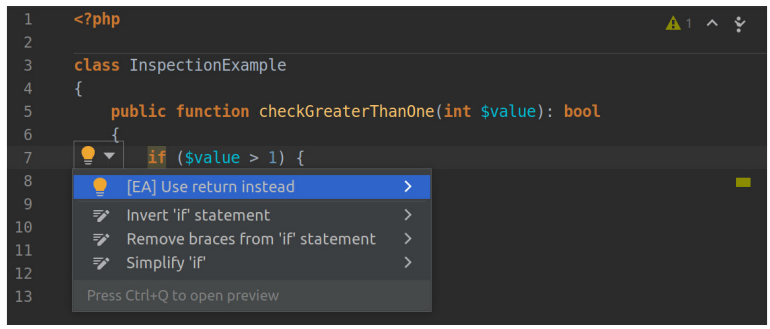


Figure 7.8: The Quick-Fix options menu

PhpStorm offers you several fixes now. The first one, marked with `[EA]`, is a suggestion by the plugin. Another click will apply the fix:

Figure 7.9: The code after applying a Quick-Fix

That's it! Within just a few seconds, you made your code shorter and easier to read. PHP Inspections (EA Extended) is a great addition to PhpStorm, as it offers sensible inspections and integrates them seamlessly. If you are using this IDE, you should not hesitate to install it.

Inspections when working in a team

These inspections are a great way to improve your code and educate yourself on best practices. However, there is a huge drawback: how do you ensure that every developer working on your project has the same inspections activated? We will cover this topic in *Working in a Team*.

Intelephense

The second extension we want to introduce is *Intelephense* for VS Code. It is the most frequently downloaded PHP extension for this editor and provides a lot of functionality

(such as code completion and formatting), which turns VS Code into a fully capable PHP IDE. There is also a commercial, premium version of this extension that offers even more functionality. To install it, please follow the instructions on the Marketplace website for this plugin (https://marketplace.visualstudio.com/items?itemName=bmewburn.vscode-intelephense-client).

Intelephense does not have the range of functionality that a full-grown, commercial IDE would offer by any means, yet for a free service, it is a perfect choice. It offers so-called Diagnostics (which are similar to Inspections in PhpStorm) that can be configured in the plugin settings screen, as shown in *Figure 7.9*:

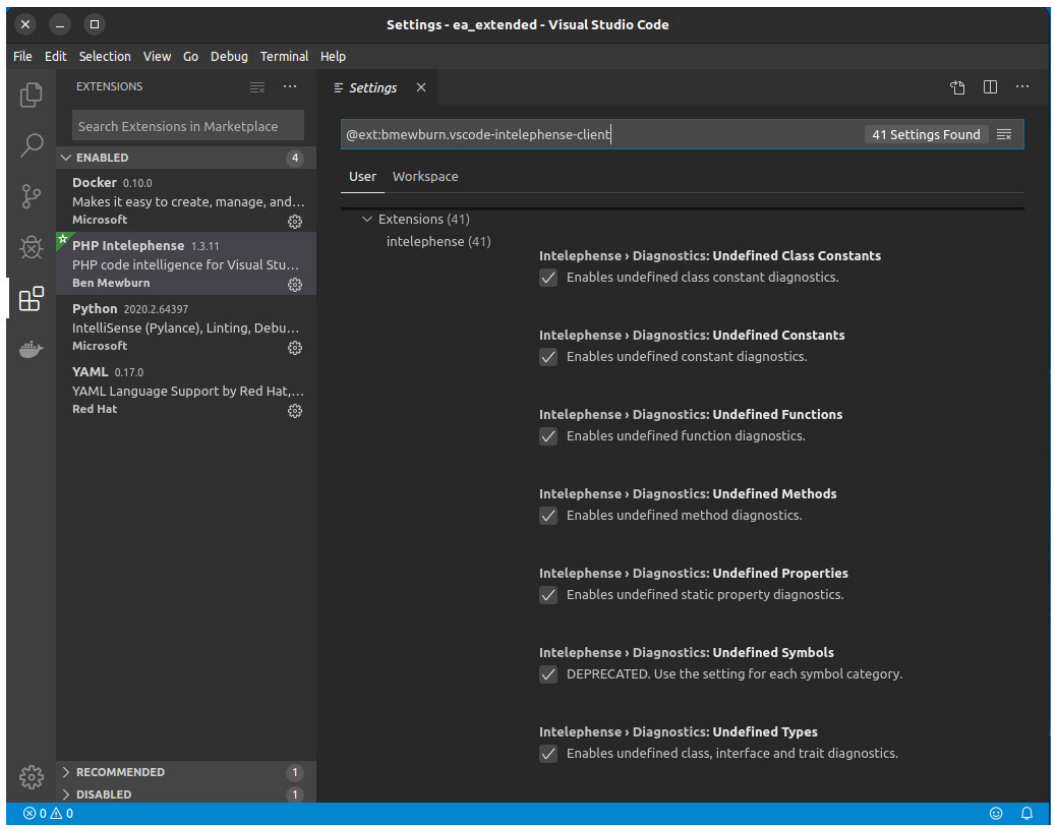


Figure 7.10: The Intelephense settings screen

Usage

The following figure shows Diagnostics in Intelephense in action:

Figure 7.11: A sample class showing how Intelephense highlights issues

Two things can be seen here. Firstly, and more obviously, is the red line underneath `TestClass`. Hovering the mouse pointer over `TestClass` will show a pop-up window with an explanation: Undefined type TestClass. This makes sense since this class does not exist.

Secondly, and more subtly, you will notice that `$ununsedAttribute` and `$testInstance` have a slightly darker color than the other variables. This indicates another issue, which can be revealed by hovering the mouse over one of the variables:

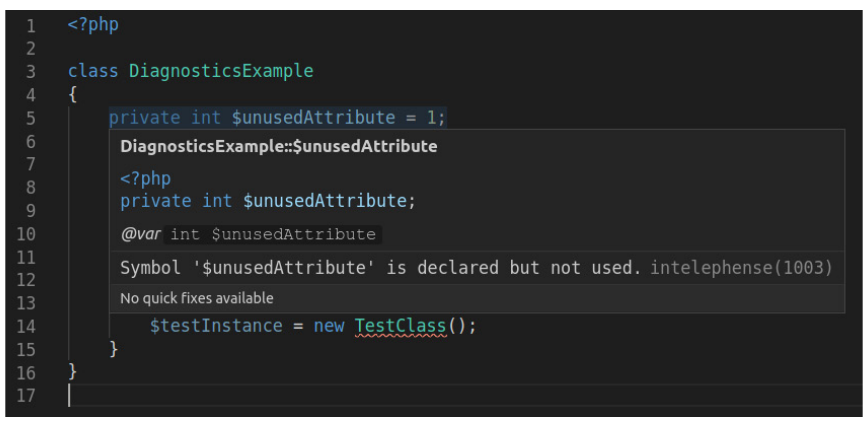


Figure 7.12: An info popup in Intelephense

The popup tells us that `$unsuserAttribute` is not used elsewhere in the code. The same applies to `$testInstance` as well.

Although it provides some basic issue detection rules and code formatting, it clearly can be said that, at the time of writing, the focus of this plugin is not on clean code. However,

given the fact that VS Code and this plugin are freely available, you already have a decent PHP IDE on hand to start coding.

Code quality tool integration in VS Code

As in PhpStorm, it is possible to integrate some common code quality tools into VS Code using plugins, such as for PHPStan (`https://marketplace.visualstudio.com/items?itemName=calsmurf2904.vscode-phpstan`), *PHP CS Fixer* (`https://marketplace.visualstudio.com/items?itemName=junstyle.php-cs-fixer`), and PHPMD (`https://marketplace.visualstudio.com/items?itemName=ecodes.vscode-phpmd`). So, if you want to code with VS Code, be sure to check Marketplace for new plugins every now and then.

Summary

In this chapter, we learned about state-of-the-art tools to assist you in creating high-quality PHP code. They will help you spot issues early in the Software Development Life Cycle (SDLC), which saves you vast amounts of time. The PHP community is still vivid and very productive, and we were not able to cover all the fantastic software that exists out there in this book. However, with the tools we introduced in this chapter, you are now well equipped for your journey towards clean code.

In the next chapter, you will learn about how to evaluate code quality by using the established metrics and, of course, the necessary tools to gather them. See you there!

Further reading

If you want to try out even more code quality tools, consider the following projects:

- *Exakat* (`https://www.exakat.io`) – A tool that also covers security issues and performance, for example. It can fix issues automatically, too.
- *Phan* (`https://github.com/phan/phan`) – A static code analyzer that you can try out immediately in your browser
- *PHP Insights* (`https://phpinsights.com/`) – Another analyzer, yet with easy-to-use metrics in terms of the code, architecture, complexity, and style

Chapter 8

Links

The code files for this chapter can be found here: https://github.com/PacktPublishing/Clean-Code-in-php

If you are interested in maintainability, you can look it up here: https://www.verifysoft.com/en_maintainability.html.

Pdepend online documentation: https://pdepend.org/documentation/software-metrics/index.html

dePHPend (https://dephpend.com/)

Command and Codes

Code Sample 8.1:

The following code snippet illustrates how it is supposed to work. Consider the following line of code:

```
while($i < 5) { echo "test"; /* Increment by one */
  $i++; }
```

Here, the LOC would be 1. Since we have three executable statements in this line, LLOC would count this as 3, as the code can also be written with each statement in one line:

```
while($i < 5) {
    echo "test";
    /* Increment by one */
    $i++;
}
```

In the preceding example, we highlighted the executable statements. Comments, empty lines, and syntactical elements such as brackets are not executable statements – that is why the full-line comment and the closing brace at the end of the loop are not counted as a logical line.

Code Sample 8.2:

```
// first decision point
function someExample($a, $b)
{
    // second decision point
    if ($a < $b) {
        echo "1";
    } else {
        echo "2";
    }
    // third decision point
    if ($a > $b) {
        echo "3";
    } else {
        echo "4";
    }
}
```

Code Sample 8.3:

For now, let us just follow the author's advice and download `phar` directly:

```
$ wget https://phar.phpunit.de/phploc.phar
```

This will download the latest version of `phploc` into the current directory. After downloading it, we can directly use it to scan a project:

```
$ php phploc.phar src
```

Code Sample 8.4:

We will go with the Composer-based installation for now:

```
$ composer require pdepend/pdepend --dev
```

If there were no unpleasant surprises, you can execute it directly:

```
$ vendor/bin/pdepend --summary-xml=pdepend_summary.xml src
```

Code Sample 8.4:

`PDepend` does output some numbers, though, as can be seen in the following example code:

```
PDepend 2.10.3
```

```
Parsing source files:
.........................................          47
Calculating Cyclomatic Complexity metrics:
................                                   355
Calculating Node Loc metrics:
............                                       279
Calculating NPath Complexity metrics:
................                                   355
Calculating Inheritance metrics:
.....                                              101
```

Code Sample 8.5:

Structure of the XML report:

```
<?xml version="1.0" encoding="UTF-8"?>
<metrics>
  <files>
    <file name="/path/to/Namespace/Classname.php"/>
    <!-- ... -->
  </files>
  <package name="Namespace">
    <class name="Classname" fqname="Namespace\Classname">
      <file name="/path/to/Namespace/Classname.php"/>
      <method name="methodName"/>
      <!-- ... -->
    </class>
    <!-- ... -->
  </package>
</metrics>
```

Code Sample 8.6:

This is an example node from a XML report that was generated on the source code of *PDepend* itself:

```
<method name="setConfigurationFile" start="80" end="89"
  ccn="2" ccn2="2" loc="10" cloc="0" eloc="8" lloc="3"
  ncloc="10" npath="2" hnt="15" hnd="21"
  hv="65.884761341681" hd="7.3125" hl="0.13675213675214"
  he="481.78231731105" ht="26.765684295058"
  hb="0.020485472371812" hi="9.0098818928795"
  mi="67.295865328327"/>
```

Code Sample 8.7:

- The syntax for `–exclude` and `–ignore` is:

```
$ vendor/bin/pdepend --summary-xml=pdepend_summary.xml
--exclude="Your\Namespace,Another\Namespace" src
```

```
$ vendor/bin/pdepend --summary-xml=pdepend_summary.xml
--ignore="path/to/folder,path/to/other/folder" src
```

Code sample 8.8:

Let us add *PhpMetrics* to your project using Composer.

```
$ composer require phpmetrics/phpmetrics --dev
```

After all the files have been downloaded, you can immediately start generating your first report:

```
$ vendor/bin/phpmetrics --report-html=phpmetrics_report src
```

Figures

```
$ php phploc.phar src
phploc 7.0.2 by Sebastian Bergmann.

Directories                                8
Files                                     47

Size
  Lines of Code (LOC)                   3349
  Comment Lines of Code (CLOC)           199 (5.94%)
  Non-Comment Lines of Code (NCLOC)     3150 (94.06%)
  Logical Lines of Code (LLOC)           897 (26.78%)
    Classes                              868 (96.77%)
      Average Class Length                18
        Minimum Class Length               0
        Maximum Class Length             117
      Average Method Length                3
        Minimum Method Length              0
        Maximum Method Length             40
      Average Methods Per Class            3
        Minimum Methods Per Class          1
        Maximum Methods Per Class         23
    Functions                            29 (3.23%)
      Average Function Length             1
    Not in classes or functions           0 (0.00%)

Cyclomatic Complexity
  Average Complexity per LLOC          0.22
  Average Complexity per Class         4.89
    Minimum Class Complexity           1.00
    Maximum Class Complexity          51.00
  Average Complexity per Method        2.10
    Minimum Method Complexity          1.00
    Maximum Method Complexity         26.00

Dependencies
  Global Accesses                         0
    Global Constants                      0 (0.00%)
    Global Variables                      0 (0.00%)
    Super-Global Variables                0 (0.00%)
  Attribute Accesses                    805
    Non-Static                          805 (100.00%)
    Static                                0 (0.00%)
  Method Calls                          364
    Non-Static                          349 (95.88%)
    Static                               15 (4.12%)
```

Figure 8.1: An example output of phploc (an excerpt)

```
Executing system analyzes...



Executing composer analyzes, requesting https://packagist.org...



LOC


    Lines of code                           876
    Logical lines of code                   762
    Comment lines of code                   114
    Average volume                          172.78
    Average comment weight                  14.48
    Average intelligent content             14.48
    Logical lines of code by class          35
    Logical lines of code by method         11
Object oriented programming
    Classes                                 22
    Interface                               14
    Methods                                 69
    Methods by class                        3.14
    Lack of cohesion of methods             0.86

Coupling
    Average afferent coupling               1.09
    Average efferent coupling               2.95
    Average instability                     0.8
    Depth of Inheritance Tree               1.47

Package
    Packages                                7
    Average classes per package             5.14
    Average distance                        0.18
```

Figure 8.2: The PhpMetrics console output (an excerpt)

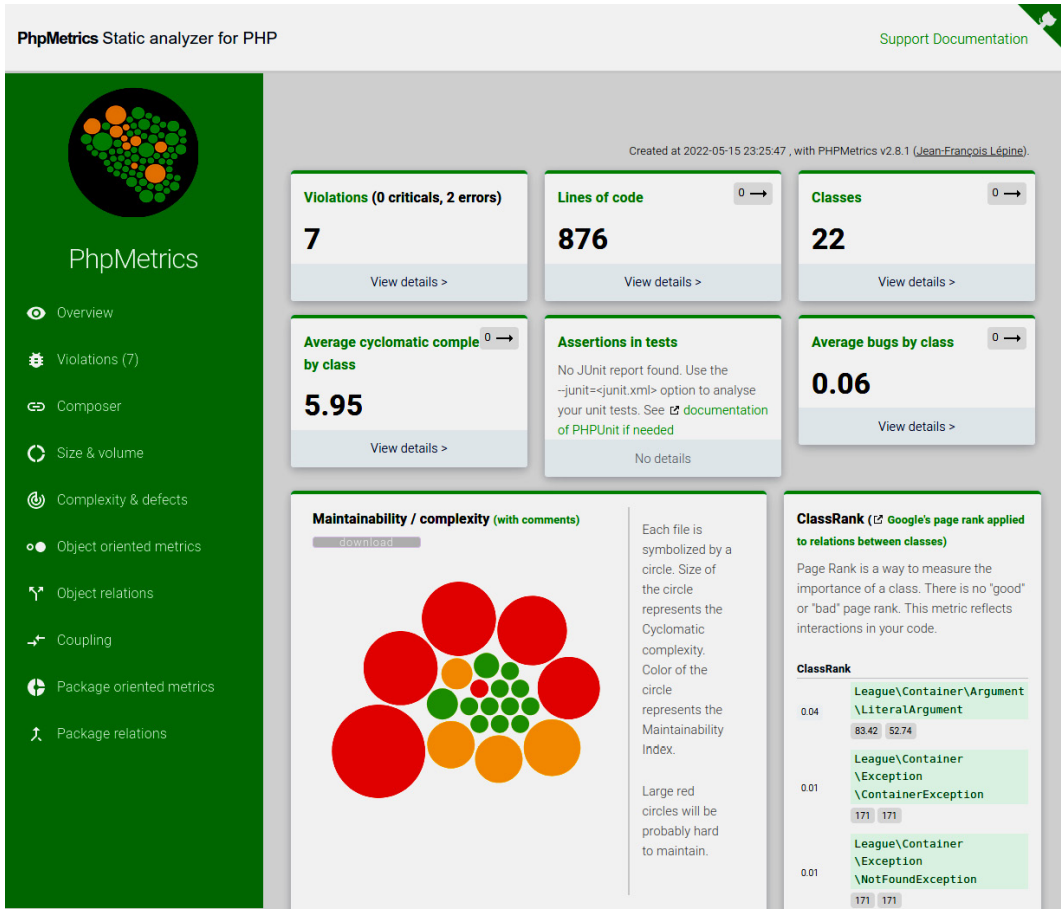


Figure 8.3: A PhpMetrics report overview

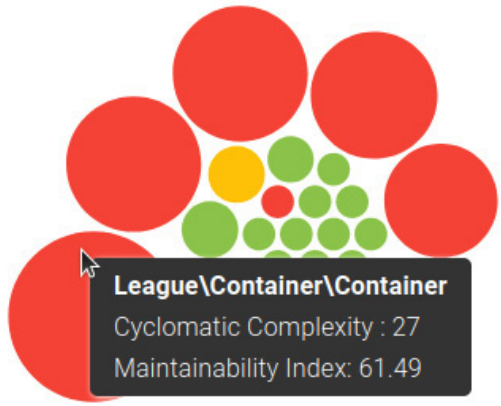


Figure 8.4: The Maintainability / complexity graph with a popup

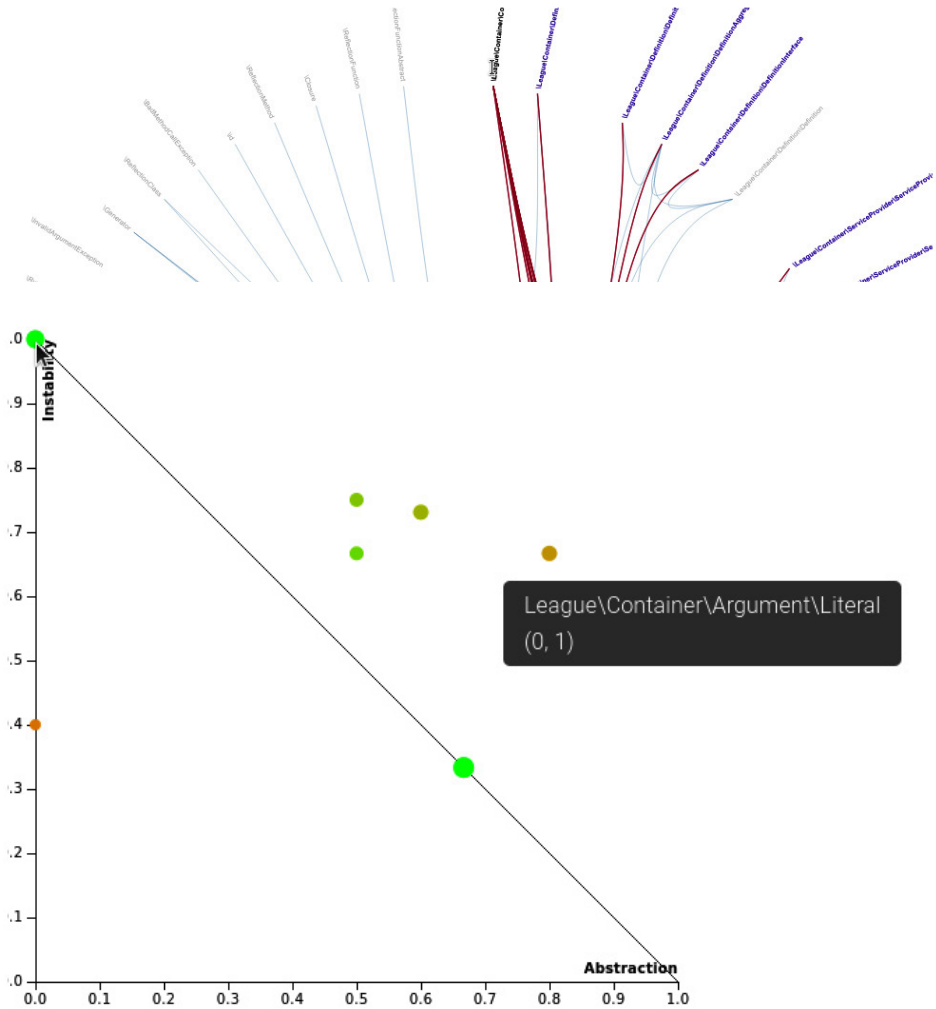


Figure 8.6: An Abstractness vs. Instability graph

Chapter 9

Links:

All code samples can be found in our GitHub repository: https://github.com/PacktPublishing/Clean-Code-in-PHP.

Command and Codes:

Code sample 9.1:

If you want to add PhpMetrics, you can do so by running the following command:

```
$ composer require phpmetrics/phpmetrics -dev
```

Code Sample 9.2:

Here, you can see an excerpt of a typical `composer.json` file:

```
{
  "name": "vendor/package",
  ...
  "require": {
    "doctrine/dbal": "^2.10",
    "monolog/monolog": "^2.2",
    ...
  },
  "require-dev": {
    "phpunit/phpunit": "^9.5",
    "phpmetrics/phpmetrics": "^2.8",
    ...
  },
  ...
}
```

Code Sample 9.3:

Installing packages globally simply requires adding the `global` modifier, like so:

```
$ composer global require phpmetrics/phpmetrics
```

Likewise, updating all global packages is effortless too, as demonstrated here:

```
$ composer global update
```

After global installation, tools such as the PHP Coding Standards Fixer (PHP-CS-Fixer) can simply be executed without having to specify the path, like so:

```
$ php-cs-fixer fix src
```

Code Sample 9.4:

```
{
  ...
  "scripts": {
    "analyze": [
      "vendor/bin/php-cs-fixer fix src",
      "vendor/bin/phpstan analyse --level 1 src"
    ]
  }
}
```

Code Sample 9.5:

```
{
  ...
    "scripts": {
        ...
    },
    "scripts-descriptions": {
        "analyze": "Perform code cleanup and analysis"
    }
}
```

Code Sample 9.6:

```
$ wget https://phar.phpunit.de/phploc.phar -O phploc
```

Code Sample 9.7

Once you download phploc, you can immediately run it using your local PHP installation, like so:

```
$ php phploc src
```

If you do not want to type `php` every time, you need to make the `phar` file executable, which on Linux—for example—would look like this:

```
$ chmod +x phploc
```

Afterward, you just need to run the following command to execute `phploc`:

```
$ ./phploc src
```

Command 9.8

```
$ wget https://github.com/phar-io/phive/releases/
download/0.15.1/phive-0.15.1.phar -O phive
$ chmod +x phive
```

Chapter 10

Links

The code files for this chapter can be found here: https://github.com/PacktPublishing/Clean-Code-in-php

If you are interested in creating maintainable E2E tests, you should check out the concept of page objects (https://www.martinfowler.com/bliki/PageObject.html).

Please note that you can use other extensions for this, such as PCOV (https://github.com/krakjoe/pcov).

Xdebug profiling capabilities: https://xdebug.org/docs/profiler

Infection: https://infection.github.io

BackstopJS: https://github.com/garris/BackstopJS

Codeception: https://codeception.com

Behat: https://github.com/Behat/Behat

Figures

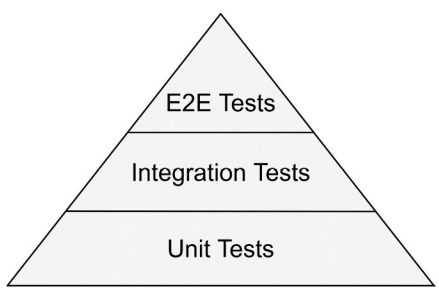


Figure 10.1: Testing pyramid

```
$ cat reports/coverage.txt


Code Coverage Report:
  2022-06-21 21:02:07

 Summary:
  Classes: 50.00% (2/4)
  Methods: 50.00% (3/6)
  Lines:   50.00% (6/12)

CodeCoverageExample\MyApp
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% (  3/  3)
CodeCoverageExample\MyOtherClass
  Methods:   0.00% ( 0/ 2)   Lines:  33.33% (  2/  6)
CodeCoverageExample\MyRepository
  Methods: 100.00% ( 1/ 1)   Lines: 100.00% (  1/  1)
$
```

Figure 10.2: Text code coverage report

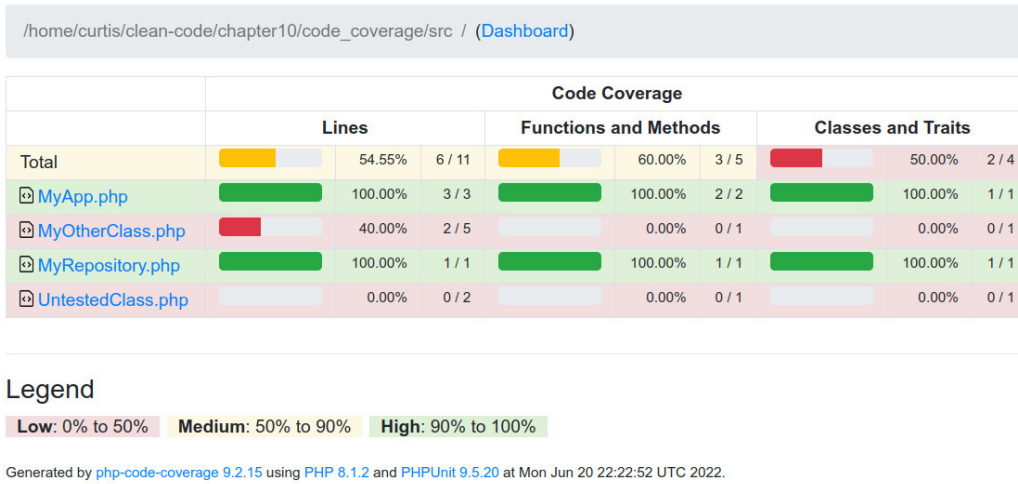


Figure 10.3: HTML code coverage report

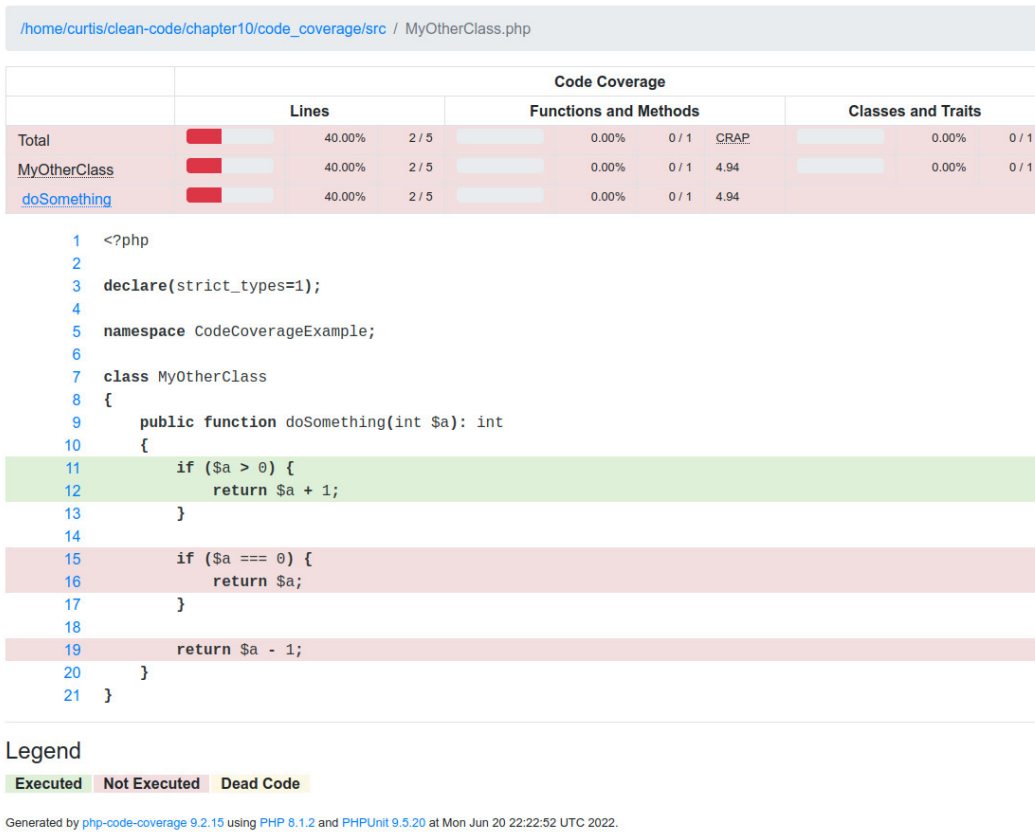


Figure 10.4: HTML code coverage report – class view

Code

Code sample 10.1

```
<?php
class MyApp
{
    public function __construct(
        private myRepository $myRepository
    ) {
    }
    public function run(): string
```

```
    {
        $dataArray = $this->myRepository->getData();
        return $dataArray['value_1'] .
          $dataArray['value_2'];
    }
}
```

Code sample 10.2

```
<?php
class MyRepository
{
    public function getData(): array
    {
        return [
            'value_1' => 'some data...',
            'value_2' => 'and some more data'
        ];
    }
}
```

Code Sample 10.3

```
public function testRun(): void
{
    // Arrange
    $repositoryMock =
      $this->createMock(MyRepository::class);
    $repositoryMock
        ->expects($this->once())
        ->method('getData')
        ->willReturn([
            'value_1' => 'a',
            'value_2' => 'b'
        ]);
    // Act
    $appTest = new MyApp($repositoryMock);
    $result = $appTest->run();
    // Assert
    $this->assertEquals('ab', $result);
}
public function testGetDataReturnsAnArray(): void
{
    // Arrange
    $repositoryTest = new MyRepository();
```

```
    // Act
    $result = $repositoryTest->getData();
    // Assert
    $this->assertIsArray($result);
    $this->assertCount(2, $result);
}
```

Code Sample 10.4:

If we now execute our application, it will of course break as shown below:

```
$ php index.php
PHP Warning:  Undefined array key "value_1" in
  /home/curtis/clean-
  code/chapter10/unit_tests_fail/src/MyApp.php on line 18
PHP Warning:  Undefined array key "value_2" in
  /home/curtis/clean-
  code/chapter10/unit_tests_fail/src/MyApp.php on line 18
```

But if you execute the tests, they will still pass as shown below:

```
$ vendor/bin/phpunit tests
PHPUnit 9.5.20 #StandWithUkraine

..                                                  2 / 2 (100%)

Time: 00:00.008, Memory: 6.00 MB

OK (2 tests, 4 assertions)
```

Code Sample 10.5:

```
public function productIsSaved(Tester $tester)
{
    $product = new Product();
    $product->setId(123);
    $product->setName('USB Coffee Maker');
    $product->save();

   $this->tester->seeInDatabase(
        'products',
        ['id' => 123, 'name' => 'USB Coffee Maker' ]
    );
}
```

Code Sample 10.6:

```
describe('Application Login', function () {
    it('successfully logs in', function () {
        cy.visit('http://localhost:8000/login')
        cy.get('#username')
            .type('test@test.com')
        cy.get('#password')
            .type('supersecret')
        cy.get('#submit')
            .click()
        cy.url()
            .should('contain',
              'http://localhost:8000/home')
    })
})
```

Code Sample 10.7

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="vendor/autoload.php">
    <testsuites>
        <testsuite name="default">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
    <coverage>
        <include>
            <directory suffix=".php">src</directory>
        </include>
        <report>
            <html outputDirectory="reports/coverage" />
            <text outputFile="reports/coverage.txt" />
        </report>
    </coverage>
</phpunit>
```

Code Sample 10.8:

```
/**
* @covers MyRepository
*/
class MyRepositoryTest extends TestCase
{
```

```
    public function testGetDataReturnsAnArray(): void
    {
        // ...
    }
}
```

Code Sample 10.9:

```
public function testUselessTestCase(): void
{
    $repositoryMock =
      $this->createMock(MyRepository::class);
    $repositoryMock
        ->method('getData')
        ->willReturn([
            'value_1' => 'a',
            'value_2' => 'b'
        ]);
    $this->assertEquals(
        [
           'value_1' => 'a',
           'value_2' => 'b'
        ],
        $repositoryMock->getData()
    );
}
```

Chapter 11

Links:

- The example application that we will use in this chapter can be downloaded from the GitHub repository to this book: https://github.com/PacktPublishing/Clean-Code-in-PHP/tree/main/ch11/example-application.
- PhpUnit webserver warning: `https://phpunit.readthedocs.io/en/9.5/installation.html#webserver`
- Docker Overview: https://docs.docker.com/get-started/overview
- CaptainHook documentation: https://captainhookphp.github.io/captainhook
- *GitHub Actions* documentation: https://docs.github.com/en/actions
- `setup-php` is not only very useful for PHP developers, but also offers a lot of useful information—for example, about the *matrix setup* (how to test code against several PHP versions) or *caching Composer dependencies* to speed up the build: https://github.com/marketplace/actions/setup-php-action
- A good overview of CD: https://www.atlassian.com/continuous-delivery
- Logging and monitoring explained: https://www.vaadata.com/blog/logging-monitoring-definitions-and-best-practices/
- A great introduction to advanced deployment methods: https://www.techtarget.com/searchitoperations/answer/When-to-use-canary-vs-blue-green-vs-rolling-deployment
- Tools and links regarding your local pipeline:
- More insights on Git hooks: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

- GrumPHP is a local CI pipeline "out of the box": https://github.com/phpro/grumphp

Figures

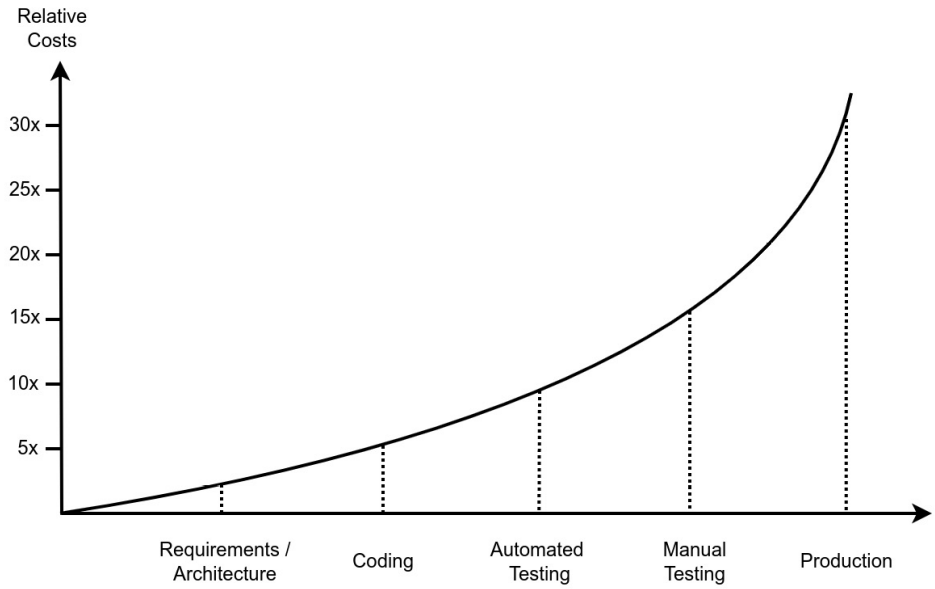


Figure 11.1: Estimated relative costs of fixing a bug based on the time of its detection

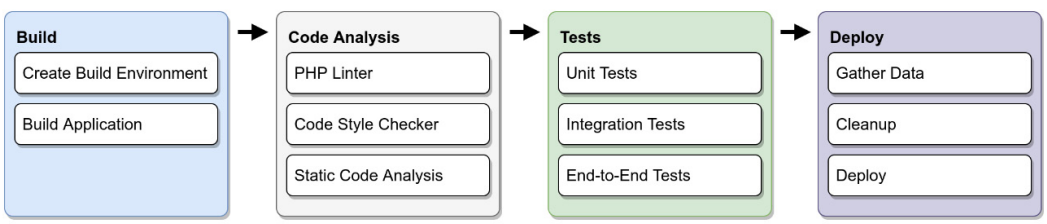


Figure 11.2: Schema of a CI pipeline

```
$ git add src/Controller/ProductController.php
$ git commit -m "Some arbitrary changes"
pre-commit:
 - vendor/bin/php-cs-fixer fix --dry-run                              : failed

failed to execute: vendor/bin/php-cs-fixer fix --dry-run

   1) src/Controller/ProductController.php

Checked all files in 0.007 seconds, 14.000 MB memory used

Loaded config default from "/home/curtis/dev/github-actions-test/.php-cs-fixer.dist.php".
Using cache file ".php-cs-fixer.cache".

$
```

Figure 11.6: The pre-commit hook fails

```
$ vendor/bin/php-cs-fixer fix
Loaded config default from "/home/curtis/dev/github-actions-test/.php-cs-fixer.dist.php".
Using cache file ".php-cs-fixer.cache".
   1) src/Controller/ProductController.php

Fixed all files in 0.007 seconds, 14.000 MB memory used
$
```

Figure 11.7: Using PHP-CS-Fixer to automatically fix code style issues

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   src/Controller/ProductController.php

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/Controller/ProductController.php

$
```

Figure 11.8: Unstaged changes

```
$ git add src/Controller/ProductController.php
$ git commit -m "Some arbitrary changes"
pre-commit:
 - vendor/bin/php-cs-fixer fix --dry-run                    : done
 - vendor/bin/phpstan                                       : done
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
$
```

Figure 11.9: pre-commit hook passes

Code and Commands

Command 11.1:

```
$ composer require --dev captainhook/captainhook
```

Command 11.2:

```
$ vendor/bin/captainhook configure
```

Code Sample 11.3

```
{
    "config": {
        "fail-on-first-error": true
    },
    "pre-commit": {
        "enabled": true,
        "actions": [
            {
             "action": "vendor/bin/php-cs-fixer
               fix --dry-run"
            },
            {
                "action": "vendor/bin/phpstan"
            }
        ]
    }
}
```

Command 11.4:

```
$ vendor/bin/captainhook install -f
```

Command 11.5

```
$ vendor/bin/captainhook hook:pre-commit
```

Code Sample 11.6:

```
"post-autoload-dump": [
    "if [ -e vendor/bin/captainhook ]; then
      vendor/bin/captainhook install -f -s; fi"
]
```

Code Sample 11.7

```
{
    "pre-commit": {
        "enabled": true,
        "actions": [
            {
                "action": "vendor/bin/php-cs-fixer fix
                  {$STAGED_FILES|of-type:php} --dry-run"
            },
            {
                "action": "vendor/bin/phpstan analyse
                  {$STAGED_FILES|of-type:php}"
            }
        ]
    }
}
```

Building a pipeline with GitHub Actions

After learning about all the stages of CI, it is time to practice. Introducing you to all the features of one or more CI/CD tools is out of scope for this book; however, we still want to show you how easy it can be to set up a working build pipeline. To keep the barrier to entry as low as possible for you and to avoid any costs, we have decided to use GitHub Actions.

GitHub Actions is not a classic CI tool like Jenkins or CircleCI, but rather a way to build

workflows around GitHub repositories. With a bit of creativity, you can do much more than "just" a classical CI/CD pipeline. We will only focus on that aspect, of course.

You probably already have a GitHub account, and if not, getting one will not cost you anything. You can use GitHub Actions for free up to 2,000 minutes per month at the time of writing for public repositories, which makes it a great playground or a useful tool for your open source projects.

> **Example project**
>
> We created a small demo application to use during this chapter. You will find it here: https://github.com/PacktPublishing/Clean-Code-in-PHP/tree/main/ch11/example-application. Please note that it does not serve any other purpose than demonstrating the basic use of GitHub Actions and Git hooks.

GitHub Actions in a nutshell

GitHub Actions offers no fancy user interface where you can configure all stages. Instead, everything is configured via YAML Ain't Markup Language (YAML) files that are stored directly in the repository. As a PHP developer, you are most likely experienced with using YAML files for all sorts of configurations—if not, do not worry, as they are easy to understand and use.

GitHub actions are organized around workflows. A workflow gets triggered by certain events and contains one or more jobs to be executed if the event occurred. A job consists of one or more steps that execute one action each.

The files have to be stored in the `.github/workflows` folder of a repository. Let us have a look at the first lines of the `ci.yml` file, which will be our CI workflow:

```
name: Continuous Integration
on:
  workflow_dispatch:
  push:
    branches:
      - main
  pull_request:
jobs:
  pipeline:
    runs-on: ubuntu-latest
    steps:
      - name: ...
        uses: ...
```

That is quite some information already. Let us go through it line by line:

- `name` defines how the workflow will be labeled within GitHub and can be any string
- `on` states which events should trigger this workflow; these comprise the following:
- `workflow_dispatch` allows us to manually trigger the workflow from the GitHub website, which is great for creating and testing a workflow. Otherwise, we would need to push a commit to main, or create a PR every time.
- `push` tells GitHub to execute this workflow whenever a push happens. We narrow it down to pushes on the `main` branch only.
- `pull_request` will additionally trigger the workflow on every new PR. The configuration might look a bit incomplete because there is no more information after the colon.
- `jobs` contains a list of jobs to be executed for this workflow, as detailed here:
- `pipeline` is the identifier (ID) of the only job in this YAML. We chose the word *pipeline* to illustrate that we can use GitHub Actions to build our CI/CD pipeline. Note that an ID must consist of one word or several words, concatenated by an underscore (_) or a dash (-).
- `runs-on` tells GitHub to use the latest Ubuntu version as a runner (that is, as a platform) for this job. Other available platforms are Windows and macOS.
- `steps` marks a list of steps to be executed for this job. In the next section, we will have a closer look at this.

We have the basics of workflow configured now, so we can begin adding the build stages.

Stage 1 – Build project

The steps are what makes GitHub Actions so powerful: here, you can choose from a vast amount of already existing *actions* to use in your workflow. They are organized in *GitHub Marketplace* (`https://github.com/marketplace`). Let us add some steps to the workflow YAML, as follows:

```
steps:
  ####################
  # Stage 1 - Build #
  ####################
  - name: Checkout latest revision
    uses: actions/checkout@v3
  - name: Install PHP
    uses: shivammathur/setup-php@v2
```

```
      with:
        php-version: '8.1'
        coverage: pcov
```

Actions maintained by GitHub are to be found in the `actions` namespace. In our example, this is `actions/checkout`, which is used to check out the repository. We do not need to specify any parameters for now, as this action will automatically use the repository in which this workflow file is located.

The `@V3` annotation is used to specify the *major* version to use. For `actions/checkout`, this would be version `3`. Please note that the latest *minor* version is always used, which at the time of writing would be version `3.0.2`.

The other action, `shivammathur/setup-php`, is provided by one of the many great people who make their work available as open source. For this step, we are using the `with` keyword to specify further parameters. In this example, we use the `php-version` option to have PHP `8.1` installed on the previously selected *Ubuntu* machine. Using the `coverage` parameter, we can tell `setup-php` to enable the `pcov` extension for generating code coverage reports.

> **Action parameters**
>
> Both actions introduced previously offer far more parameters than we can describe here. You can find more information about their functionality by looking them up in *Marketplace*.

Regarding the formatting, we used comments and blank lines between the steps to make the file more readable. There is no convention, and it is completely up to you how to format your YAML files later.

The next step is the installation of the project dependencies. For PHP, this usually means running `composer install`. Please note that we *do not* use the `--no-dev` option because we need to install the `dev` dependencies to perform all the quality checks. We will remove them at the end of the pipeline again.

> **Dependency management**
>
> We use the Composer workflow in this chapter as an example to manage our code quality tools, because this is the most common way. However, both other ways of organizing the code quality tools we introduced in *Chapter 9, Organizing PHP Quality Tools,* would work with GitHub Actions as well. In that chapter, we also explained the `--no-dev` option in detail.

This is what the next steps could look like:

```
- name: Get composer cache directory
  id: composer-cache
  run: echo "::set-output name=dir::$(composer config cache
    files-dir)"
- name: Cache dependencies
  uses: actions/cache@v2
  with:
    path: ${{ steps.composer-cache.outputs.dir }}
    key: ${{ runner.os }}-composer-${{
      hashFiles('**/composer.lock') }}
    restore-keys: ${{ runner.os }}-composer-
- name: Install composer dependencies
  run: composer install
```

GitHub actions require some manual work to make caching of Composer dependencies possible. In the first step, we store the location of the Composer cache directory, which we get from Composer using the `config cache-files-dir` command, in an output variable called `dir`. Note `id: composer-cache` here—we will need this to reference the variable in the next step.

Then, we access this variable in the next step by using the `steps.composer-cache.outputs.dir` reference (a combination of the `id` value we set in the previous step, and the variable name) to define the directory that should be cached by the `actions/cache` action. `key` and `restore-key` are used to generate unique caching keys—that is, the cache entries where our Composer dependencies are stored.

Lastly, we use the `run` parameter to directly execute `composer install`, as if we would execute it locally on an Ubuntu machine. This is important to keep in mind: you can, but you do not have to use existing GitHub actions for every step—you can just execute pure shell commands (or equivalent commands on Windows runners) as well.

There are also actions in *Marketplace* that take over the writing of commands, such as `php-actions/composer`. We do not have a preferred solution here; both will work fine.

Because we want to run integration tests on the API of our example application, we need to have a web server running. For our simple use case, it is totally enough to use the PHP built-in web server, which we can start using in the following step:

```
- name: Start PHP built-in webserver
  run: php -S localhost:8000 -t public &
```

The `-S` option tells the PHP binary to start a web server that is listening on the

`localhost` address and port `8000`. Since we start in the `root` folder of our project, we need to define a *document root* folder (the folder where the web server looks for files to execute) using the `-t` option. Here, we want to use the public folder, which only contains the `index.php` file. It is good practice to not store any other code in the document root folder since this makes it harder for attackers to hack our application.

> **PHP built-in web server**
>
> Please note that the built-in web server of PHP is only to be used for *development* purposes. It should never be used in *production*, since it was not built with performance or security in mind.

You surely noticed the ampersand (`&`) at the end of the command. This tells Linux to execute the command, but not wait for its termination. Without it, our workflow would get stuck at this point because the web server does not terminate by itself, as it needs to keep listening for requests until we run our *integration tests* at a later stage.

The setup of our build environment is complete. Now, it is time to run the first code quality checks on our sample application.

Stage 2 – Code analysis

In the first build stage, we created our build environment and checked out our application code. At this point, the application should be completely functional and ready to be tested. Now, we want to do some static code analysis.

The standard approach is to use dedicated GitHub actions for each tool. The benefit is that we keep the development tools away from the build environment, as they will be executed in separate Docker containers that will be discarded right after use. There are some drawbacks to this approach, though.

Firstly, with each action, we introduce yet another dependency, and we rely on the author to keep it up to date and not lose interest in maintaining it after a while. Additionally, we add some overhead, since Docker images are usually many times bigger than the actual tool. And lastly, when our application setup gets more complicated, running the code quality tools in separate Docker containers can cause issues, simply because it is not the same environment as the build environment. Sometimes, already tiny differences can cause problems that keep you engaged for hours or days in solving them.

As we saw in the previous section, we can simply execute Linux shell commands in the build environment, so nothing speaks against executing our code quality tools directly on the build environment—we just need to make sure to remove them afterward so that they do not get released into production.

In our example application, we added PHP-CS-Fixer and PHPStan to the `require-dev` section of the `composer.json` file. By adding the following lines of code to our workflow YAML, we will let them execute as the next steps:

```
############################
# Stage 2 - Code Analysis #
############################
- name: Code Style Fixer
  run: vendor/bin/php-cs-fixer fix --dry-run
- name: Static Code Analysis
  run: vendor/bin/phpstan
```

We do not need many parameters or options here, since our example application provides both the `.php-cs-fixer.dist.php` and `phpstan.neon` configuration files, which both tools will look up by default. Only for PHP-CS-Fixer will we use the `--dry-run` option because we only want to check for issues during the CI/CD pipeline, and not solve them.

> **Setting the scope of checks**
>
> For our small example application, it is OK to run the preceding checks on all files because they will execute quickly. If our application grows, however, or we wish to introduce CI/CD to an existing application (which we will discuss further on in this chapter), it is sufficient to run these checks on those files that have only changed in the latest commit. The following action could be helpful for you in this case: `https://github.com/marketplace/actions/changed-files`.

If neither PHP-CS-Fixer nor PHPStan reports any issues, we can safely execute the automated tests in the next stage: the tests.

Stage 3 – Tests

Our code has been thoroughly analyzed and checked for bugs and syntax errors, yet we need to check for logical errors in our code. Luckily, we have some automated tests to ensure that we did not inadvertently introduce any bugs.

For the same reasons as for the code quality tools in *Stage 2*, we do not want to use a dedicated action for running our PHPUnit test suites. We simply execute PHPUnit as we would on our local development system. Using the `phpunit.xml` file clearly proves useful here since we do not need to remember all the many options to use here. Let us have a look at the *workflow YAML* first, as follows:

```
####################
# Stage 3 - Tests #
####################
- name: Unit Tests
  run: vendor/bin/phpunit --testsuite Unit
- name: Integration Tests
  run: vendor/bin/phpunit --testsuite Api
```

The only thing worth noting here is that we do not just run all tests, but we split them up in two test suites: `Unit` and `Api`. Since our unit tests should execute the fastest, we want to run them (and fail) first, then followed by the slower integration tests. Please note that we did not add any E2E tests as our application does not run in the browser but is a mere web service.

We split the tests up by using the `phpunit.xml` configuration file. The following code fragment shows you its `<testsuites>` node, where we separate the suites by their directory (`Api` and `Unit`):

```
<testsuites>
    <testsuite name="Api">
        <directory>tests/Api</directory>
    </testsuite>
    <testsuite name="Unit">
        <directory>tests/Unit</directory>
    </testsuite>
</testsuites>
```

We also configured PHPUnit to create code coverage reports, as illustrated here:

```
<coverage processUncoveredFiles="false">
    <include>
        <directory suffix=".php">src</directory>
    </include>
    <report>
        <html outputDirectory="reports/coverage" />
        <text outputFile="reports/coverage.txt" />
    </report>
</coverage>
```

To create these reports, PHPUnit will automatically use the `pcov` extension, which we configured in *Stage 1*. They will be written into the `reports` folder, which we will take care of in the next stage.

That is already everything that needs to be done for the tests stage. If our tests did not discover any errors, we are good to go into the last stage of our pipeline and wrap

everything up.

Stage 4 – Deploy

Our application is now thoroughly checked and tested. Before we are ready to deploy it into whichever environment we envisioned, we need to take care of removing the `dev` dependencies first. Luckily, this is very easy, as we can see here:

```
####################
# Stage 4 - Deploy #
####################
- name: Remove dev dependencies
  run: composer install --no-dev --optimize-autoloader
```

Running `composer install --no-dev` will simply delete all the `dev` dependencies from the `vendor` folder. Another noteworthy feature is the `--optimize-autoloader` option of Composer: since in production, we will not add or change any classes or namespaces as we would do in development, the Composer autoloader can be optimized by not checking for any changes, and thus disk access, speeding it up a bit.

As the very last step, we want to create build artifacts: one artifact is the deliverable—that is the code we intend to deploy. The other artifact is the code coverage reports we created in *Stage 3*. GitHub Actions will not keep any additional data than the logging information displayed on the GitHub website after the workflow YAML has been executed, so we need to make sure they are stored away at the end. The code is illustrated in the following snippet:

```
- name: Create release artifact
  uses: actions/upload-artifact@v2
  with:
    name: release
    path: |
      public/
      src/
      vendor/
- name: Create reports artifact
  uses: actions/upload-artifact@v2
  with:
    name: reports
    path: reports/
```

We use the `actions/upload-artifacts` action to create two ZIP archives (called *artifacts* here): `release` and `reports`. The first contains all files and directories we need to run our application on production, and nothing more. We omit all the configuration

files in the root folder of our project, even the `composer.json` and `composer.lock` files. We do not need them anymore, since our `vendor` folder already exists.

The `reports` artifact will just contain the `reports` folder. After the build, you can simply download both ZIP archives separately on GitHub. More about this in the next section.

Integrating the pipeline into your workflow

After adding the workflow YAML to the `.github/workflows` folder (for example, `.github/workflows/ci.yml`), you only need to commit and push it to the repository. We configured our pipeline to run upon every opened PR or whenever someone pushes a commit to the `main` branch.

When you open `https://github.com` and go to your repository page, you will find an overview of your last workflow runs on the Actions tab, as shown in the following screenshot:

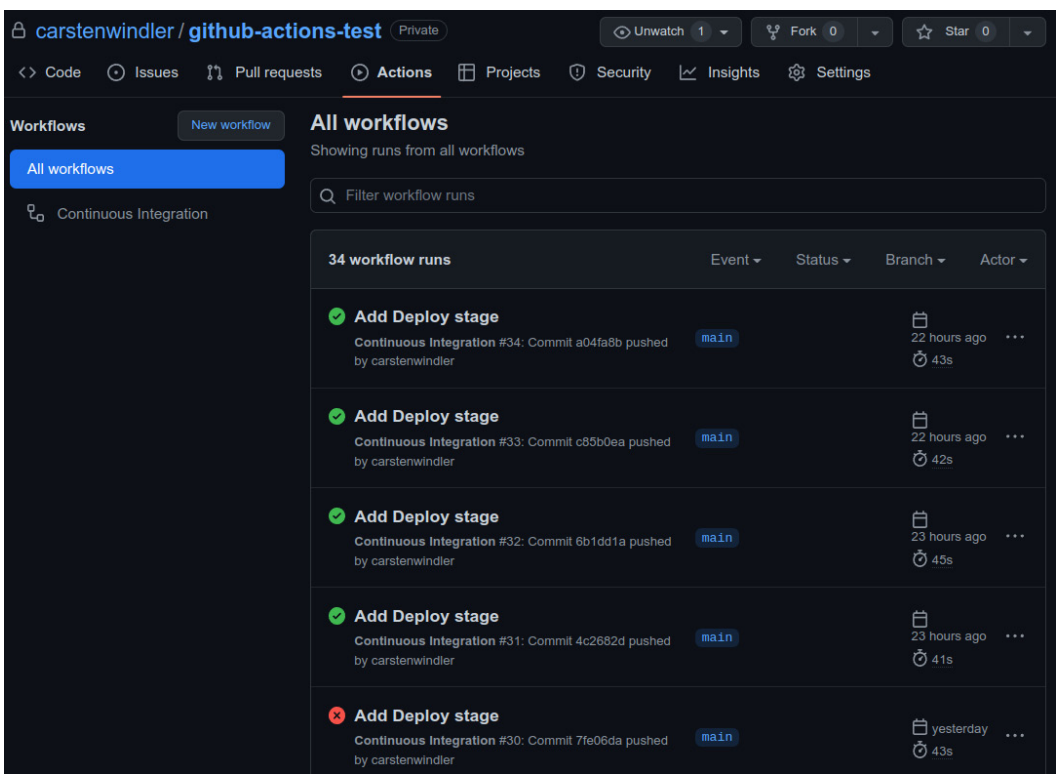


Figure 11.3: The repository page on github.com

The green checkmark marks the successful runs, while the red cross—of course—marks the failed ones. You can also see when they were executed and how long this took. By clicking on the three dots on the right side of each entry, you will find more options—for example, where you can delete the workflow run. Clicking on the title of the run, which is the corresponding commit message of the run, you will enter the Summary page, as shown in the following screenshot:

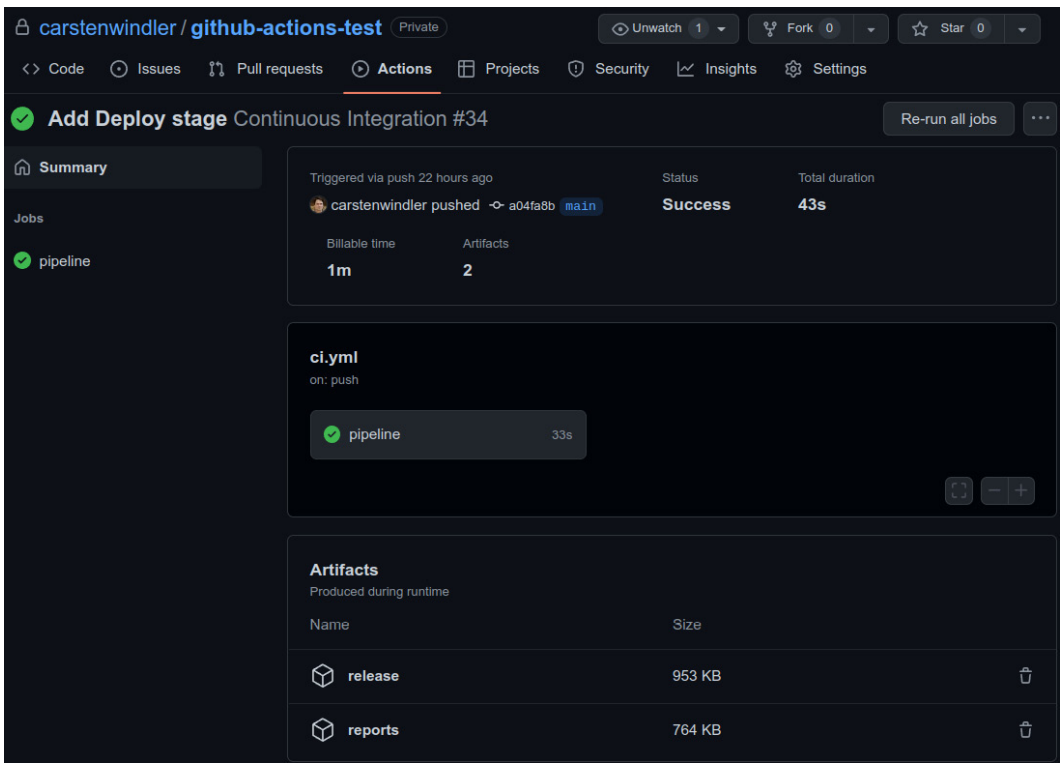


Figure 11.4: The workflow run summary page

Here, you can see all jobs of the workflow. Since our example only consists of one job, `pipeline`, you will only see one. On this page, you can also find any generated artifacts (such as our release and reports artifacts) and download or delete them. GitHub offers only limited disk space for free, so make sure to delete them when you are running out of space.

Another important piece of information is the billed time. Although our job only ran for 43 seconds in total, GitHub will deduct 1 minute from your monthly usage. GitHub offers a generous free plan, but you should have a look at your usage from time to time. You can find more information about this on your user settings page in the Billing and plans section (`https://github.com/settings/billing`).

If you want to see what exactly happens during the workflow run—for example, if something goes wrong—you can click on the `pipeline` job to get a detailed overview of all of its steps, as illustrated in the following screenshot:

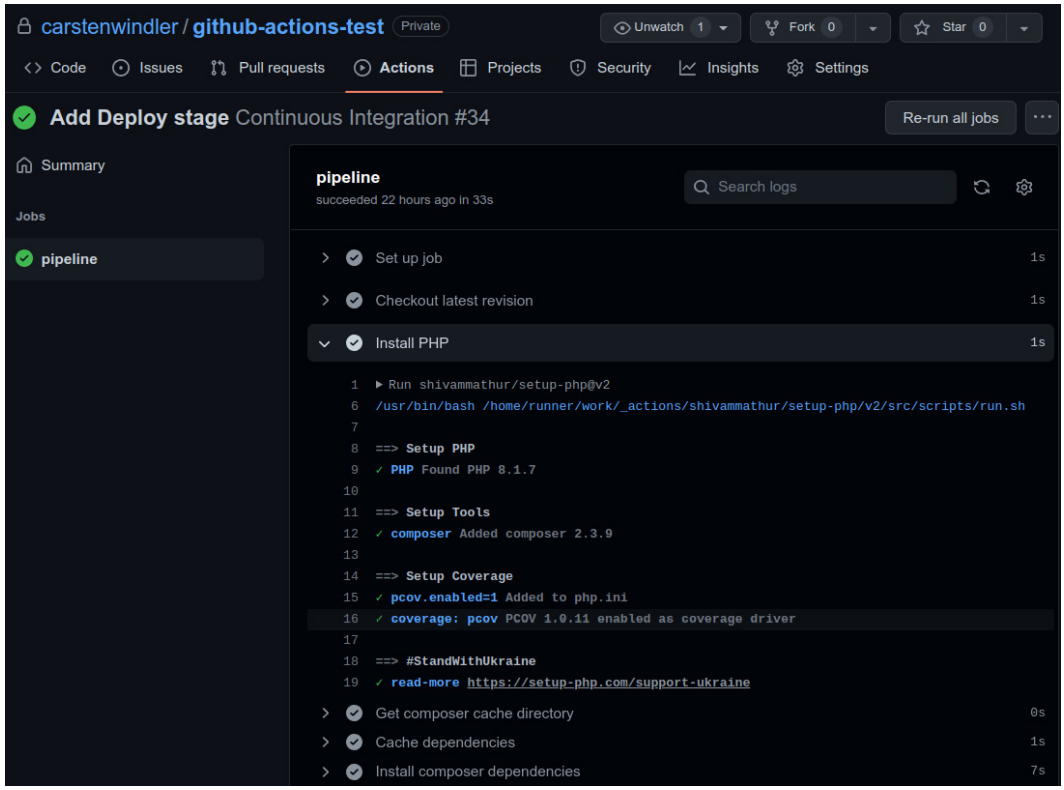


Figure 11.5: Job details page

Each step can be expanded and collapsed to get additional information about what exactly happened during its execution. In the preceding screenshot, we expanded the Install PHP step to see what the action did in detail.

Congratulations—you now have a working CI pipeline for your project! This ends our little tour through GitHub Actions. Of course, you can extend the pipeline as you like—for example, by uploading the release artifact to an SSH File Transfer Protocol (SFTP) server or an AWS Simple Storage Service (S3) bucket. There is a lot more than can be done, so make sure to experiment with it.

Chapter 12

Links

- The code files for this chapter can be found here: https://github.com/PacktPublishing/Clean-Code-in-php
- *Symfony* guidelines website: https://symfony.com/doc/current/contributing/code/standards.html
- You will find more information about PER Coding style here: https://www.php-fig.org/per/coding-style
- PHP Documentor project (https://www.phpdoc.org/)
- You will find a complete list on the official PHP-fig website: https://www.php-fig.org.
- Functionality of a modern DI container: https://container.thephpleague.com
- PSR-11 – Container interface: https://www.php-fig.org/psr/psr-11.
- https://google.github.io/eng-practices/review provides you with more information about the *code review* process at Google

Useful sources about *design patterns* in PHP:

- https://refactoring.guru/design-patterns/adapter/php/example
- https://sourcemaking.com/design_patterns/adapter/php

Code

Code Sample 12.1: Redundant Docblock

```
// Redundant DocBlock
/**
 * @param int $property
 * @return void
 */
public function setProperty(int $property): void {
    // ...
}
```

Code Sample 12.2: Useless and wrong DocBlock

```
// Useless DocBlock
/**
 * @param $property
 */
public function setProperty(int $property): void {
    // ...
}
// Wrong DocBlock
/**
 * @param string $property
 */
public function setProperty(int $property): void {
    // ...
}
```

Code Sample 12.3: Useful DocBlock to specify content in an array and mark a deprecated function

```
// Useful DocBlock
/**
 * @return string[]
 */
public function getList(): array {
    return [
       'foo',
       'bar',
    ];
}
/**
 * @deprecated use function fooBar() instead
```

```
 */
public function foo(): bool {
    // ...
}
```

Code Sample 12.4: Exceptions for very short statements

```
// Example for short statement
$isFoo ? 'foo' : 'bar';
// Usual notation
$isLongerVariable
    ? 'longerFoo'
    : 'longerBar';
```

Code Sample 12.5: Example for nested operators that are hard to read and debug.

```
// Example for nested operators
$number > 0 ? 'Positive' : ($number < 0 ? 'Negative' :
'Zero');
```

Code Sample 12.6: Using the constructor property

```
// Before PHP 8+
class ExampleDTO
{
    public string $name;
    public function __construct(
        string $name
    ) {
        $this->name = $name;
    }
}
// Since PHP 8+
class ExampleDTO
{
    public function __construct(
        public string $name,
    ) {}
}
```

Code Sample 12.7: Using the Short array notation

```
// Old notation
$myArray = array(
    'first entry',
```

```
    'second entry'
);
// Short array notation
$myArray = [
    'first entry',
    'second entry',
];
```

Code Sample 12.8: Using Brackets

```
// Bad
if ($statement === true)
    do_something();
// Good
if ($statement === true) {
    do_something();
}
```

Code Sample 12.9: Avoiding else statements

```
// Bad
if ($statement) {
    // Statement was successful
    return;
} else {
    // Statement was not successful
    return;
}
// Good
if (!$statement) {
    // Statement was not successful
    return;
}
// Statement was successful
return;
```

Code Sample 12.10: Avoiding empty `catch` blocks

```
// Bad
try {
    $this->someUnstableCode();
} catch (Exception $exception) {}
// Good
try {
    someUnstableCode();
```

```
} catch (Exception $exception) {
    $this->logError($exception->getMessage());
}
```

Text Sample 12.11:

```
# Definition of Done
## Reviewer
[ ] Code changes reviewed
    1. Coding Guidelines kept
    2. Functionality considered
    3. Code is well-designed
    4. Readability and Complexity considered
    5. No Security issues found
    6. Coding standard and guidelines kept
[ ] Change tested manually
## Developer
[ ] Acceptance Criteria met
[ ] Automated Tests written or updated
[ ] Documentation written or updated
```

Code Sample 12.12:

```
abstract class AbstractWriter
{
    public function write(array $data): void
    {
        $encoder = $this->createEncoder();
        // Apply some filtering which should always happen,
        // regardless of the output format.
        array_walk(
            $data,
            function (&$value) {
                $value = str_replace('data', '', $value);
            }
        );
        // For demonstration purposes, we echo the result
        // here, instead of writing it into a file
        echo $encoder->encode($data);
    }
    abstract protected function createEncoder(): Encoder;
}
```

Code 12.13: Using the Encoder class

Note the `createEncoder` method – this is the factory method that gave the pattern the name, since it acts, in a sense, as a factory for new instances. It is defined as an abstract function, so it needs to be implemented by one or more subclasses.

To be flexible enough for future formats, we intend to use separate `Encoder` classes for each format. But first, we define an interface for these classes so that they are easily exchangeable.

```
interface Encoder
{
    public function encode(array $data): string;
}
```

Then, we create an `Encoder` class for each format that implements the `Encoder` interface; first, we create `JsonEncoder`:

```
class JsonEncoder implements Encoder
{
    public function encode(array $data): string
    {
        // the actual encoding happens here
        // ...
        return $encodedString;
    }
}
```

Then we create `CsvEncoder`:

```
class CsvEncoder implements Encoder
{
    public function encode(array $data): string
    {
        // the actual encoding happens here
        // ...
        return $encodedString;
    }
}
```

Now, we need to create one subclass of the `AbstractWriter` class for each format we want to support. In our case, that is `CsvWriter` first:

```
class CsvWriter extends AbstractWriter
{
```

```
    public function createEncoder(): Encoder
    {
        $encoder = new CsvEncoder();
        // here, more configuration work would take place
        // e.g. setting the delimiter
        return $encoder;
    }
}
```

And second, it is `JsonWriter`:

```
class JsonWriter extends AbstractWriter
{
    public function createEncoder(): Encoder
    {
        return new JsonEncoder();
    }
}
```

Please note that both subclasses only overwrite the Factory Method `createEncoder`. Also, the `new` operators occur only in the subclasses. The `write` method remains unchanged, as it gets inherited from `AbstractWriter`.

Finally, let us put this all together in an example script:

```
function factoryMethodExample(AbstractWriter $writer)
{
    $exampleData = [
        'set1' => ['data1', 'data2'],
        'set2' => ['data3', 'data4'],
    ];
    $writer->write($exampleData);
}
echo "Output using the CsvWriter: ";
factoryMethodExample(new CsvWriter());
echo "Output using the JsonWriter: ";
factoryMethodExample(new JsonWriter());
```

The `factoryMethodExample` function first receives `CsvWriter` and, for the second run, `JsonWriter` as parameters. The output will look like this:

```
Output using the CsvWriter:
3,4
1,2
```

```
Output using the JsonWriter:
[["3","4"],["1","2"]]
```

Code Sample 12.14:

```
class InstantiationExample
{
    private Logger $logger;
    public function __construct()
    {
        $this->logger = new FileLogger();
    }
}
```

Code Sample 12.15:

```
class ConstructorInjection
{
    private Logger $logger;
    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
    }
}
```

Code Sample 12.16:

```
$constructorInjection = new ConstructorInjection(
     new FileLogger()
);
```

Code Sample 12.17: Setter Injection

```
class SetterInjection
{
    private Logger $logger;
    public function __construct()
    {
        // ....
    }
    public function setLogger(Logger $logger): void
    {
        $this->logger = $logger;
    }
}
```

Code Sample 12.17:

```
class SetterInjectionFactory
{
    public function createInstance(): SetterInjection
    {
        $setterInjection = new SetterInjection();
        $setterInjection->setLogger(new FileLogger());
        return $setterInjection;
    }
}
```

Code Sample 12.18:

```
class CustomerAccount
{
    public function __construct(
        private MailService $mailService
    ) {}
    public function cancelSubscription(): void
    {
        // Required code for the actual cancellation
        // ...
        $this->mailService->sendEmail(
            'sales@example.com',
            'Account xy has cancelled the subscription'
        );
    }
}
```

Code Sample 12.19:

```
use SplSubject;
use SplObjectStorage;
use SplObserver;
class CustomerAccount implements SplSubject
{
    private SplObjectStorage $observers;
    public function __construct()
    {
        $this->observers = new SplObjectStorage();
    }
    public function attach(SplObserver $observer): void
    {
        $this->observers->attach($observer);
```

```
    }
    public function detach(SplObserver $observer): void
    {
        $this->observers->detach($observer);
    }
    public function notify(): void
    {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }
    public function cancelSubscription(): void
    {
        // Required code for the actual cancellation
        // ...
        $this->notify();
    }
}
```

Code Sample: 12.20

```
class CustomerAccountObserver implements SplObserver
{
    public function __construct(
        private MailService $mailService
    ) {}
    public function update(CustomerAccount|SplSubject
      $splSubject): void
    {
        $this->mailService->sendEmail(
            'sales@example.com',
            'Account ' . $splSubject->id . ' has cancelled
              the subscription'
        );
    }
}
```

Code Sample: 12.21

```
$mailService = new MailService();
$observer = new CustomerAccountObserver($mailService);
$customerAccount = new CustomerAccount();
$customerAccount->attach($observer);
```

Code Sample 12.22:

```
$instance = Singleton::getInstance();
```

Code sample 12.23

```
class Singleton
{
    private static ?Singleton $instance = null;
    public static function getInstance(): Singleton
    {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }
}
```

Code sample 12.24:

```
class ServiceLocatorExample
{
    public function __construct(
        private ServiceLocator $serviceLocator
    ) {}
    public function fooBar(): void
    {
        $someService = $this->serviceLocator
          ->get(SomeService::class);
        $someService->doSomething();
    }
}
```

Chapter 13

Links

The code files for this chapter can be found here: https://github.com/PacktPublishing/Clean-Code-in-php

Mermaid Live Editor (https://mermaid.live)

PlantUML (https://plantuml.com

diagrams.net (https://www.diagrams.net

Diagrams (https://diagrams.mingrammer.com),

PlantUML (https://plantuml.com)

O.A.S specification with Swagger UI (https://github.com/swagger-api/swagger-ui

OpenAPI documentation: https://oai.github.io/Documentation

`swagger-php` (https://github.com/zircote/swagger-php

PHP Swagger Test (https://github.com/byjg/php-swagger-test)

Spectator (https://github.com/hotmeteor/spectator)

Swagger UI (https://github.com/swagger-api/swagger-ui)

API Blueprint (https://apiblueprint.org

RESTful API Modeling Language (R.A.M.L) (https://raml.org

Figure

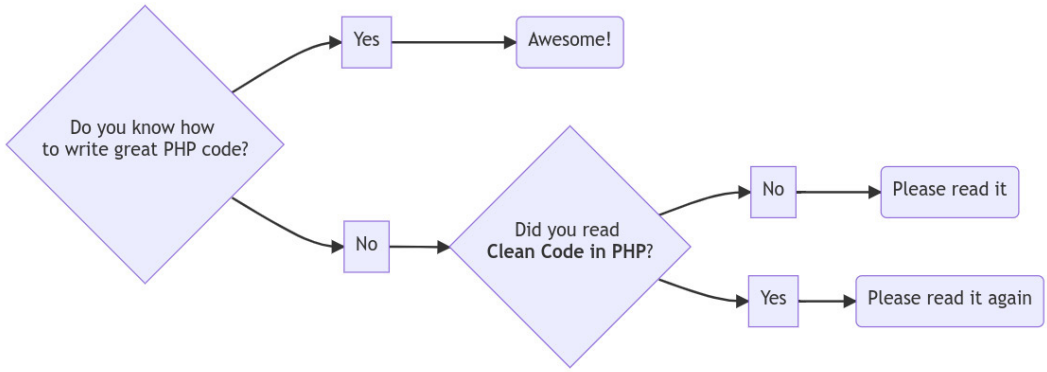


Figure 13.1: Mermaid diagram example

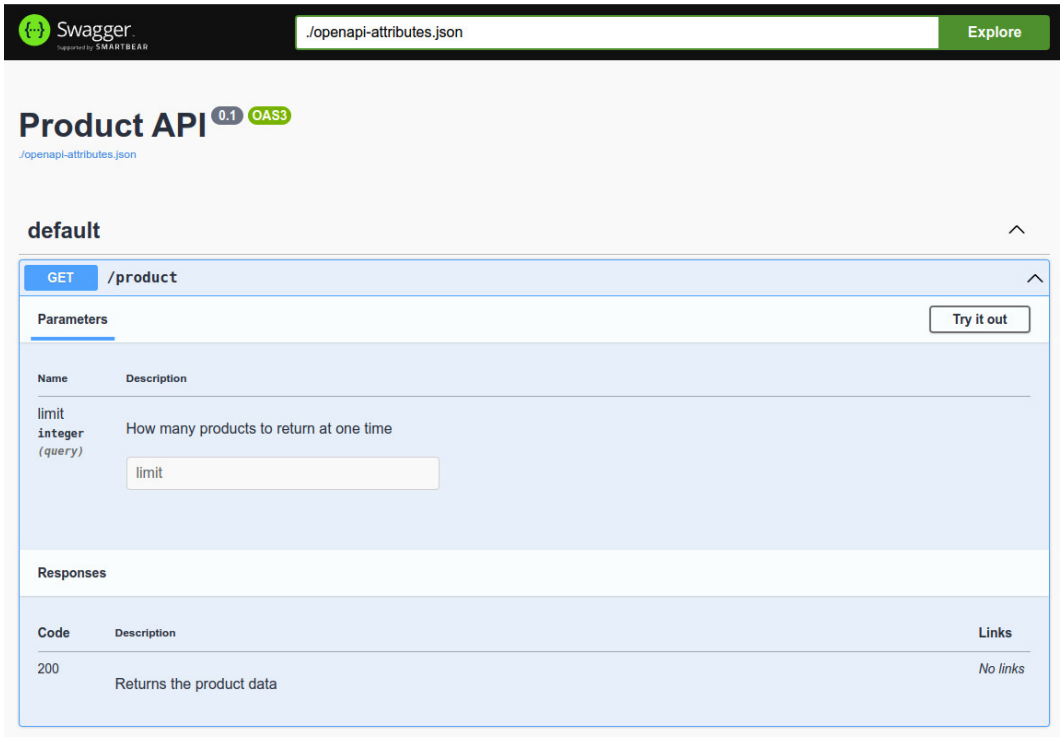


Figure 13.2: Swagger UI

Code:

Code Sample 13.1:

```
graph LR
    A{Do you know how to write great PHP code?} --> B[No]
    A --> C[Yes]
    C --> E(Awesome!)
    B --> D{Did you read Clean Code in PHP?} --> F[No]
    D --> G[Yes]
    G --> H(Please read it again)
    F --> I(Please read it)
```

Code Sample 13.2:

```
openapi: 3.0.0
info:
  title: 'Product API'
  version: '0.1'
paths:
  /product:
    get:
      operationId: getProductsUsingAnnotations
      parameters:
        -
          name: limit
          in: query
          description: 'How many products to return'
          required: false
          schema:
            type: integer
      responses:
        '200':
          description: 'Returns the product data'
```

Code Sample 13.3:

```
/**
 * @OA\Info(
 *     title="Product API",
 *     version="0.1"
 * )
 */
class ProductController
{
```

```
    /**
     * @OA\Get(
     *     path="/product",
     *     operationId="getProducts",
     *     @OA\Parameter(
     *         name="limit",
     *         in="query",
     *         description="How many products to return",
     *         required=false,
     *         @OA\Schema(
     *             type="integer"
     *         )
     *     ),
     *     @OA\Response(
     *         response="200",
     *         description="Returns the product data"
     *     )
     * )
     */
    public function getProducts(): array
    {
        // ...
    }
}
```

Code Sample 13.4:

```
use OpenApi\Attributes as OAT;
#[OAT\Info(
    version: '0.1',
    title: 'Product API',
)]
class ProductController
{
    #[OAT\Get(
        path: '/v2/product',
        operationId: 'getProducts',
        parameters: [
            new OAT\Parameter(
                name: 'limit',
                description: 'How many products to return',
                in: 'query',
                required: false,
                schema: new OAT\Schema(
                    type: 'integer'
```

```
                ),
            ),
        ],
        responses: [
            new OAT\Response(
                response: 200,
                description: 'Returns the product data',
            ),
        ]
    )]
    public function getProducts(): array
    {
        // ...
    }
}
```