RECTOR

The Power of Automated Refactoring

3rd Edition

Matthias Noback
Tomas Votruba

Rector - The Power of Automated Refactoring

Matthias Noback and Tomas Votruba

This book is available at
https://leanpub.com/rector-the-power-of-automated-refactoring

This version was published on 2025-11-18

Contents

Preface

By Tomas

Working with old code is painful. In any framework, in any company, for you, and for your employer. Nobody wants to ride a horse on the highway. But why are so many people doing it? Why are there so many legacy PHP projects?

A Trainer's Journey

I was asking myself this question 6 years ago, almost every day. I was doing trainings on Nette, Symfony, or Doctrine. My usual training looked like this: I came to a company, I gave a presentation about Symfony and its *cool new features*, we did hands-on coding, and I answered curious questions. I had a good feeling about this and feedback from participants was also positive. I gained a lot of experience from trainings in 30+ different PHP companies and used it in every next training. I thought: "That's what trainers do, I finally made it; changing the life of others in just one or two days." Gosh, I was proud of myself. Helping others is my life-long passion, so I was very happy, and also well paid. This feeling was soon to be changed.

I had one training about new features in PHP 7.0 for a project that was running PHP 5.4. After a year, the same company asked me for a fresh training about new features in the PHP world. I asked them, what version do they have now, so I wouldn't bore them with something they already know.

Their answer: "This year we'll definitely get to a new PHP version." I asked: "Oh, you're up to PHP 7.2 already? Wow!" They replied: "No, we're almost at the PHP 7.0. That's why we need you to help us!"

What? My mind was shocked. Suddenly, I remembered another training I had, where they asked for almost the same topic as a year before. No, that can't be right, can it?

The Other Side

A trainer's point of view, and one or two days of training, are just one side of the story. A demo is easy to code. Everybody is happy because they work on 200 lines of code and the progress is there. Then the developers have to go through a real code base and make the change happen there as well. They have to evolve the project from 1.000.000 to 5.000.000 lines of code, help the business to grow, and add new features. I realized that my trainings may expand knowledge by showing the hot news in the PHP world or a particular framework. But barely 2% of this knowledge will last for longer than a week after the training.

Who's Fault is It?

I started to blame myself. My teaching methods were likely not very effective. I tried to improve my presentation and learn more about the project beforehand, but the results didn't change much. Then I blamed the company business leaders. They were nice to hire a trainer, but after he or she is gone, they'd force programmers to work on that legacy code to add more features instead of refactoring it. Then I blamed programmers, who are not willing to spend four hours after work to educate themselves. They should be able to protest against legacy business and stop it, until the code is clean and fun to work with.

Later I've learned, none of these targets are right.

From Blame to Pain

I stopped giving training so often because I felt like a failure. I was part of a system that didn't make much change for the better. The system repeated the same action twice, with no gain. I couldn't do it anymore. I mean, I physically could not do it. Why not? Let me explain.

When I was 8 years old, I lost a cap while playing outside. In situations like these, my father used to punish me with tasks involving lots of repetition. I always had to write long sentences, a 100 times, like: "I will never go outside while playing and not returning back without the hat again, or I will have to go out and look for it and won't return until I find it."

At first it was an interesting exercise in writing. After the 30 sentences though, my hand started to hurt. The text was supposed to be clear to read and without any mistake, and there were 10 more sentences left to go. I remember a Saturday, when I spent almost 4 hours on this, unable to do anything else. My mom was helping me sometimes, adding 20 sentences herself.

This was repeated a couple times a month. I became terrified to do anything my father wouldn't like. These punishments came out of the blue. Once my brother broke my father's tool and told it was me. I said I don't know anything about it, so I got the-100-sentences punishment not to lie ever again.

Thanks to these punishments, I've hardcoded in my brain **that any repetition hurts me and I'll do anything to avoid it**. That's my personality trait. It took some time but I've put some healthy boundaries on this and learned to use it to my advantage.

P.S. Don't take me wrong: I love my father and we have a safe and warm relationship now. I'm very grateful for the patience, attention, focus, and resilience he taught me. Thanks to him I'm the man I want to be.

From Pain to Idea

Do you know when you look at the code, you feel it's wrong, but you don't know what it is? I had this feeling. I knew there must be some better way to do this that is cheaper, faster and re-usable. It won't be easy to find it, but there must be a way to solve this.

I started to share this pain and frustration in chats with my friends and I slowly built up a vision. A vision of a world where all programmers are only thinking about new algorithms, and where terms like "legacy code" and "technical debt" don't exists. Changing frameworks is a matter of a single click. Upgrades are as easy as `composer update`. And it's all open source, so anyone can use it.

From Idea to First Real Reconstructor

I had a weekend trip to Brno to visit a friend of mine, Petr Vacha. He encouraged me to just give it a try, when I didn't see the light yet. Thank you Peter. We went to a Happy Tearoom on Freedom Square, I think Saturday Jul 15, 2017 it was, and we spent 4 hours there. Petr doing his coding, and me, struggling with the abstract syntax tree and `nikic/php-parser`, without any IT University education. It was stressful as hell. That's how the first commits to *Reconstructor*[1] (before it was shortened to "Rector") came to be.

We've Just Started, Now You Get on Board

That was 2017. Today it's 2025 and legacy code is still a thing. Yes, we have the technology to change the PHP world. Rector is maturing and has over 630 rules that automate boring work. Hundreds of companies already use it to automate their upgrades and improve code quality in their CI. But that's not enough.

That's the reason we're writing this book. To empower you to both join the Rector community, and write rules that help you with your problems. If you're reading this book, you probably have the same feeling I had: "There must be a better way." There is. We'll show you the way, and we welcome you to get on board with us.

[1]https://github.com/rectorphp/rector/commits/v0.1.0?after=0cc75d58bee80233531e5c1d41cd49fba9ca81bb+1084&branch=v0.1.0

Introduction

What is Rector?

Rector is a tool that came into existence because Tomas wanted to help developers everywhere to:

- Prepare their code base for the latest PHP version
- Effortlessly upgrade to the next version of their current framework, or even
- Switch between frameworks if needed
- Improve their code quality without thinking about it
- Define their own automated refactoring procedures

The core abilities of a tool like this would be:

- Being able to understand PHP code
- Being able to manipulate PHP code without breaking it
- Being able to save the upgraded PHP code back to their locations on disk

Rector[2] can do all of this, and while it reads and parses your code, it will apply a number of *rules* to it. By doing so, Rector improves and upgrades your code, without human intervention. It won't make the typical mistakes that a human being will when they're typing and copy/pasting code in a text editor.

Some concrete examples of what Rector can do for you:

- Upgrade Symfony from 2.8 all the way to the latest version
- Turn property annotations into actual property types, add return types, etc.
- Change nested `if`s to early `return`s.
- Upgrade classic `switch` statements to PHP 8.0 `match` statements.
- And the list goes on and on[3]

[2] https://github.com/rectorphp/rector/
[3] https://getrector.com/find-rule

Who Should Read This Book?

This is a book for PHP developers who want to modernize their project. Whether the project consists of the worst case of legacy code, or is actually quite modern; knowing how to work with and extend Rector is going to help your project move forward.

We're assuming knowledge of object-oriented PHP programming and several years of experience maintaining PHP projects.

An Overview of the Contents

In the first chapter, Matthias explores the topic of modifying PHP code *with* PHP code, or Programmatically modifying PHP code. You'll learn about the *Abstract Syntax Tree*, and how to create and modify it with PHP-Parser. Tomas continues with a discussion of the various PHP tools in the game that can help you with modifying code in an automated fashion. Next, Matthias shows you how to migrate from low-level PHP-Parser node visitors to Rector rules. You'll be Creating your first Rector rule in this chapter.

Rector makes it easy to create rules, but the process will be challenging anyway. Matthias continues with a demonstration of test-driven rule development. There are several more testing techniques that need to be discussed, so Matthias adds one more chapter about TDD in the context of Rector.

Now that you know how Rector works, how you can use and extend it, Tomas helps you get your project to the next level. He shows you how you can not only use Rector incidentally, but apply Continuous Rectifying to your project. In the next chapter Tomas demonstrates how Rector combined with GitHub Actions can become the *Next Member of Your Team*. The last chapter is connecting some dots: with sophisticated tools Composer, PHP-Parser, and now Rector, we could get rid of wasteful Backward-Compatibility promises, and free the world from legacy code.

We finish this book with the Epilogue, which is in the form of an interview with Tomas. We'll take a look at the vision for Rector and what we might expect from the project in the long run.

About the Code Samples

The code samples in this book aim to be as realistic as possible. In fact, we run the following tools on them to ensure their correctness now and in the future:

- PHPUnit, for verifying our own assumptions, and the correctness of the code samples.
- Rector, for automatic improvements, migrations to future PHP versions, libraries, etc.
- EasyCodingStandard, for applying a uniform coding style and to make all the lines fit within a single page width.

The latter isn't always possible, e.g. in the case of `use` statements which are sometimes quite long. In those cases we abbreviate the namespace and add a comment (`// (abbreviated)`). When copy/paste-ing code samples into your own project this means you sometimes have to manually expand these namespaces before you can run the code.

When showing the code, we sometimes remove somewhat irrelevant parts so they don't take up too much space on the page. However, you should always be able to figure out the full example based on the previous code samples. That way you can follow along with the tutorial chapters in your own project.

If something doesn't work in your project, please try the following options first:

- Upgrade Rector to its latest version (`composer require --dev rector/rector`).
- Run `composer dump-autoload` to let Composer regenerate its class loader.

About the Authors

Matthias Noback

Matthias Noback has been building web applications since 2003. He is the author of the *Object Design Style Guide* and *Advanced Web Application Architecture*. He's also a regular blogger, speaker and trainer[4].

Tomas Votruba

Tomas loves to combine open-source and innovations… yet he's super lazy. That's why he always spends dozens of hours to automate problems that would take an hour or two to do manually. Fortunately, he prefers those problems that can help the whole PHP community. Tomas is a regular blogger, speaker and trainer[5] as well.

Acknowledgements

Tomas

This book and its know-how is a team effort. That's why we would like to thank a few people - without them this book would never exist.

To Kerrial Beckett Newham for the tasty dinner where he made up the working and final title of the book. To Petr Vacha for one day in a Brno teahouse, where he supported my crazy idea of automated refactoring - there the first Rector commits were made, 4 years ago. To Jan Mikes for the fastest framework migration in a week. To Sebastian Schreiber for getting me back on track with a Rector vision and great laughs. To Matthias Noback, my hero blogger who helped me to overcome my scariest dream of writing a book like a walk in the park. Last but not least, to my dad for being my very first supporter on Patreon, although he never paid by card online before. It means the world to me.

Thank you, Tomas.

[4] https://matthiasnoback.nl
[5] https://tomasvotruba.com/

Matthias

Publishing a book is a scary business. You put in a lot of time without knowing how your work is going to be received. For *Rector - The Power of Automated Refactoring*, your support, reviews, and suggestions have been very helpful. You have successfully encouraged us to finish this project. Thanks a lot for that!

On a more personal note, thank you Tomas, for trusting me to write a book that would do your project justice. Along the way you have been able to stir up my old-fashioned ways of writing PHP. You've shown me so many new ways to modernize, and automate this project, and for making the tools that keep me on track. Thanks for being my personal Rector.

Programmatically Modifying PHP Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Introduction

In order to automatically refactor PHP code for you, Rector needs a safe way to manipulate code. As a developer, you know it's easy to break existing functionality when you're changing code manually. Rector has an advantage here: given that it's a program, when you give it the right instructions, it will not make the kind of human errors that we tend to make.

How can Rector be so sure it's making the right changes though? Part of this question will be answered in the chapter about test-driven rule development. We have to write tests to prove that Rector makes the right changes in any imaginable situation. Besides an extensive test suite, Rector also relies on low-level language analysis tools like PHP-Parser[6] and PHPStan[7]. Using these tools Rector can, for instance, distinguish a property name from a variable name, and derive its type.

This chapter is a discussion of the fundamentals of software that manipulates code. It covers the concepts that are at the core of Rector itself: parsing PHP code, visiting nodes of an abstract syntax tree, and manipulating them while visiting them.

Primitive Ways of Modifying Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

[6]https://github.com/nikic/PHP-Parser
[7]https://phpstan.org/

Tokenizing PHP Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Parsing PHP Tokens: the Abstract Syntax Tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Converting the AST Back to PHP Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Manipulating the AST

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Node Visitors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Manipulating the AST with a Node Visitor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Built-in Node Visitors

Resolving Fully-qualified Names

Finding Nodes

Summary

PHP Tools in the Game

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Introduction

Before we start to explore Rector internals and its powerful features, we'll take a broader look at the wider range of tools that can analyze and change your code. Why do we talk about other tools in a book about Rector? Well, have you heard about the Unix philosophy[8]? It's nothing about free software or independence, but rather about co-dependence:

> Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".

Rector is just one of the tools in the chain. It's the most powerful one, but without its friends it would be an utterly useless empty shell, like our body would be without lungs.

Working Together with Giants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

2007 - Now Timeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

[8]https://en.wikipedia.org/wiki/Unix_philosophy

The Primary Feature

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

1. Coding Standard Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

PHP_CodeSniffer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

PHP CS Fixer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Easy Coding Standard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

2. Static Analyzers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

PHP-Parser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

PHPStan

Psalm

3. Instant Upgrade Tools

Symfony-Upgrade-Fixer

Rector

When to Use Which Tool?

Is Your Project Bare Without Any Tools?

Run Rector First, Then Polish with Coding Standards

Recommended Tools

Summary

Creating Your First Rector Rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Introduction

In the chapter Programmatically modifying PHP code we used PHP-Parser to load a single PHP file, parse it, and manipulate some of its nodes if they matched certain criteria. In this chapter we'll take the next step: using Rector to load *all* PHP files in a given project directory and modify the code according to certain *rules*.

What's a Rector Rule?

A rule is an automated refactoring. It represents one particular thing that Rector could change for all the PHP files in your project at once. Here are several examples of rules that are already included in the Rector package:

- Rename method; e.g. call `newMethod()` instead of `oldMethod()` on `SomeExampleClass`
- Encapsed strings to sprintf; e.g. replace `return "Unsupported format {$format}";` with `return sprintf('Unsupported format %s', $format);`
- Finalize classes without children; if a class has no subclasses, make it `final`.

It's always smart to scan the list of existing rules[9] before you start implementing your own rule. Chances are that what you're looking for already exists, and in that case it's likely that the existing rule can deal with many edge cases you didn't think of yet.

[9] https://getrector.com/find-rule

Creating a Custom Rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Extending AbstractRector

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Finding the Right Node Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Expr vs Stmt

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Running a Single Rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Refactoring the Method Call Node

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

What Is the Type of a Variable?

What if We Run Rector Twice?

Effectivity Beats Perfection

Summary

Test-driven Rule Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Introduction

In the previous chapter we've created our first Rector rule. We concluded that it was a throw-away rule because it wasn't generalized enough to use it in other situations. It was also a single-use rule because we didn't write tests for it. When you want to maintain a rule for more than a couple of days, it won't be enough to rely on *exploratory testing* only. Running your rule on the code base of a real project gives you great feedback about the quality of the rule; it gives you ideas for edge cases you didn't think of. But verifying the correctness of your rule manually, using only visual verification, will be time-consuming and unreliable as well. Automated testing is the easiest way to solve this problem.

Rector ships with some testing tools based on PHPUnit. There's an `AbstractRectorTestCase` class you can extend your test class from. In general each Rector rule you create will have its own test class. This class should extend `AbstractRectorTestCase` and it will be called `[SomeName]RectorTest`. When creating a rule you add several before and after code snippets to demonstrate that your rule makes the correct transformations. You also add examples of code snippets that should be left untouched by your rule.

By adding many examples you can shape your rule gradually to match those examples. Any edge case you encounter later on can be added as yet another example. Doing so will save you a lot of stress: once an example is added as an automated test, it will keep working forever. If not, you'll know it because the test fails.

Although you can write tests for a rule that has already been created, the best time to write tests is *before* you write your first line of code in the rule class. Doing so brings you into the right mindset. You'll think about the problem, while not worrying about the possible complexity of the solution. Just describe what you want, and have faith in your ability to *eventually* come up with the solution.

Migrating from DateTime to DateTimeImmutable

In this chapter we'll *test-drive* the development of a Rector rule. We start with a problem, describe in code what we want, and then we develop a rule (and eventually a set of rules) that provide the solution to the problem. The problem I'm thinking about is this: at some point PHP added the `DateTimeImmutable` class, which is an immutable variant[10] of the already existing `DateTime` class. First, let's take a look at the differences between `DateTime` and `DateTimeImmutable`.

```php
use DateTime;

function nextMonday(DateTime $dt): DateTime
{
    $modified = $dt->modify('next monday');

    // The return value of `modify()` could be `false`
    assert($modified instanceof DateTime);

    return $modified;
}

$today = new DateTime();
$nextMonday = nextMonday($today);

echo $today->format('Y-m-d'), "\n";
echo $nextMonday->format('Y-m-d'), "\n";
```

Although the function `nextMonday()` seems to return a new instance of `DateTime`, but modified to represent the next Monday, in reality this function also updates the

[10] https://derickrethans.nl/immutable-datetime.html

original `DateTime` object passed to it. So running this script will just output the same date twice. In code bases that use `DateTime`, this is a common problem that often goes unnoticed, until one day it leads to hard-to-debug problems. The proposed solution has always been to `clone` the provided instance before modifying it:

```php
use DateTime;

function nextMonday(DateTime $dt): DateTime
{
    $dt = clone $dt;

    $dt->modify('next monday');

    return $dt;
}
```

Doing so keeps the original object from being modified.

If we had used the “new” `DateTimeImmutable` class, we wouldn’t have had this problem, and we wouldn’t have to `clone` the object:

```php
use DateTimeImmutable;

function nextMonday(DateTimeImmutable $dt): DateTimeImmutable
{
    $modified = $dt->modify('next monday');

    assert($modified instanceof DateTimeImmutable);

    return $modified;
}
```

If `nextMonday()` is the only function you’d need to adapt, it wouldn’t be a big deal. But what if your code base has thousands of usages of `DateTime` and you want to migrate to `DateTimeImmutable`? Rector is the perfect tool for this job. Let’s create a Rector rule that can do at least part of the migration for us.

What applies to other programming tasks, applies to creating Rector rules as well: a problem that looks quite complex from the start should be subdivided into smaller and simpler problems first. For instance, the seemingly complex migration from `DateTime` to `DateTimeImmutable` consists of several smaller (and simpler) tasks:

1. Where the code uses `DateTime` as a type (e.g. as a parameter, return, or property type) we should change it to `DateTimeImmutable`.
2. Where the code has a `use`-statement for `DateTime`, we should also change it to `DateTimeImmutable`.
3. Where the code calls `modify()` on a `DateTime` instance we should capture the result in a variable, e.g. `$dt = $dt->modify('next monday');`
4. Where the code `clone`s a `DateTime` instance, we should no longer clone it.

It's likely that we'll come up with other tasks, but at least this seems like a list of things we can solve with the knowledge acquired from previous chapters. This attitude is what makes test-driven development so great: we can start small, and cycle between adding another test and writing a bit of code to make that test pass. Eventually we'll be able to work ourselves up from the simple, somewhat uninteresting tasks, accumulating smaller bits of behavior into bigger chunks of more complex and powerful behavior. In order to be successful, don't try to build too much at once.

Creating a Test Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Adding First Test Fixture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Making the First Test Pass

Narrowing the Scope of the Refactoring

Capturing the Return Value of modify()

Skipping Calls on Classes That Are Not DateTime

Summary

More Testing Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Rules Should Be Idempotent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Removing the Clone Step

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Low-Hanging Fruit First

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Create a Configurable Generic Rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Making Rule Behavior Configurable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Configurable Rule Makes Hidden Assumptions Explicit

Using Supporting Classes in Fixtures

Summary

Continuous Rectifying

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Introduction

PHPUnit is a tool that runs test on your code base; it makes sure the code still works after you change it. PHPStan is a tool that detects the types of your variables, method calls, arguments and so on; it makes sure your code does not make obvious mistakes like calling a method on `null`. ECS is a tool that makes your code look standardized; all of the spaces and brackets in every class and method will look the same.

It's very nice to run these tools on the command-line and get instant feedback about what is wrong and where. But their biggest added value is going to be somewhere else in 2025…

Imagine a new developer will join the team. Every time it happens you need to give them training about company standards: "No, we use tabs, not spaces. No, we don't use static service calls" etc. With all the right tools in place, instead of training we can give the person access to our repository and just let them make a pull request. Then continuous integration (CI) will become their trainer. They will get instant feedback from a failed CI job. PHPUnit failed? They should fix the logic. PHPStan failed? They should improve the types. ECS failed? There is a difference in code style. Just fix it and push again.

So how does all this CI feedback relate to Rector? We're glad you asked.

> rectify *verb*
>
> **rectify something** to put right something that is wrong
>
> – Oxford Advanced Learner's Dictionary[11]

[11]https://www.oxfordlearnersdictionaries.com/definition/english/rectify?q=rectify

We already know how to use Rector to change our code, how to write our own rule and how to test it. You can also use Rector to upgrade to a new PHP version in just a fraction of the time it would take when doing this work manually. You can switch between frameworks over a weekend, or refactor from static service calls to constructor dependency injection. That sounds exciting, right?

We bet you have ideas how Rector can help you get your project in shape again.

Certainly, some teams are using Rector this way: install it, handle a PHPUnit upgrade and a Symfony upgrade, then remove it again. That's quite sad, really. It's like using PhpStorm to open your project once and then switch back to Notepad again. The real power of Rector is not in quick jobs. Its full potential can be unlocked if you make it part of your project in the long run. When you work with Rector every day, it will continue helping you out, shares its wisdom, and taking care of the project.

The Next Member of Your Team

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Turning a Junior Into a Senior on Day 1

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Who's to Blame?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

What Does it Look Like to Work with Rector in CI?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

How Rector is Rectifying Itself

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Removing Boring Work Opens Your Creativity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Rector + CI = Next Member of Your Team

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Introduction

GitHub Actions[12] is a CI system from GitHub. How is it different from other systems?

- It's blazing fast
- It's free for open-source
- It has prepared "recipes" for all the common operations you can think of

These recipes are called *GitHub Actions*. They wrap complex functionality and only ask for a few configuration values. Do you need to push to remote SSH servers? Just provide your email address and a public key and that's it. Do you need to run `composer install` with cache for parallel jobs? You only have to change one line in your configuration file.

It's really handy that actions are stored in public repositories on GitHub. If we need to understand any action in depth, or want to learn how to use it, we just open the repository, e.g. ramsey/composer-install[13] and find out.

This whole architecture allows us to create combos with various workflows that we stack upon each other: clone repository, generate the website, deploy the website, and tweet about the new post. Boom!

Let's add Rector to your project using GitHub Actions and see what it can do for us.

[12]https://github.com/features/actions
[13]https://github.com/ramsey/composer-install

What are The Steps for a Rector Run?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

How to Add Rector to GitHub Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

1. Generic PHP Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

2. Rector Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Allow Rector to contribute the Pull Request

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

The Future of Instant Upgrades

Introduction

It takes time to fully grasp the power of new tools. We have to include them in our workflow and see where they help.

It took Composer three years to get 66 millions downloads a month. In 2021, the same number applies to daily downloads[14].

When php-parser was introduced, no one really noticed. 3 years later it had only 14.000 daily downloads. Now in 2025 it's 450.000+ downloads[15].

It's not about the tools themselves. It's about the PHP community that finds new ways to use these tools. Whether it's a plugin or a tool built on top of it.

This is not a book on how to use Xth version of already established language or framework. You have a unique advantage in your hands. We're on the verge of automated refactoring epoque. Only very few dozen people in the whole PHP community have the power to reshape whole projects in minutes like you do now.

In this chapter we explore a few futuristic paths where building on top of Rector could lead us.

Removing Legacy

[14]https://packagist.org/statistics
[15]https://packagist.org/packages/nikic/php-parser/stats

How Can We Be So Confident?

A Member of your Team That Sends Pull-Requests Daily

PHP Stands Out

Epilogue

Node Type and Refactor Examples

Book Revisions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Update 2025-11-18

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Update 2025-03-07

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Update 2024-12-09 and Rector 2.0

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Update 2024-02-07 and Rector 1.0

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.

Update 2024-01-30 and Rector 0.19.3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/rector-the-power-of-automated-refactoring.