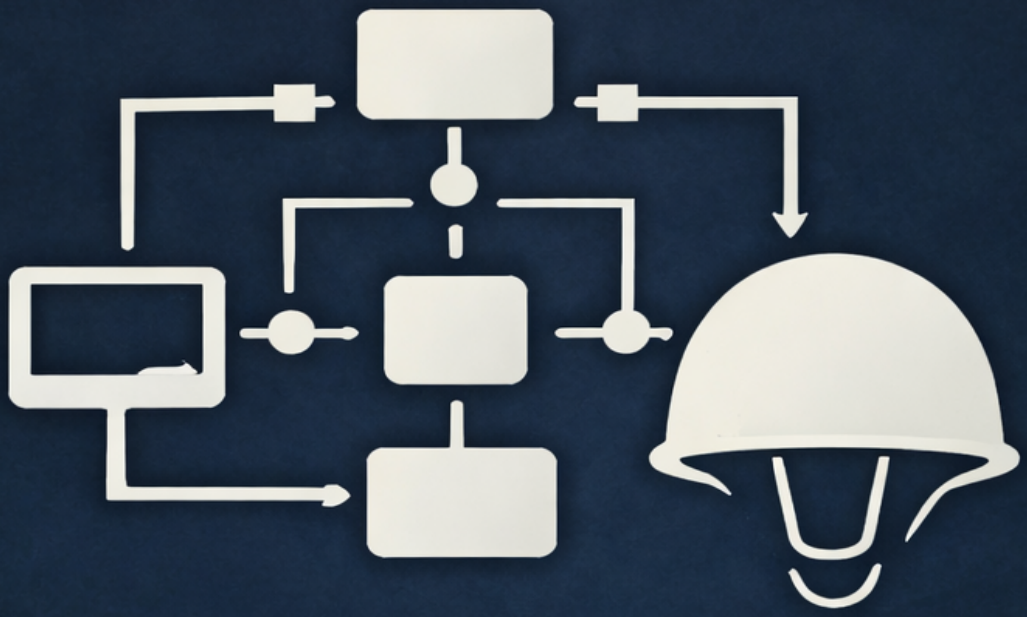

PHP Design Patterns:

From The Trenches

Practical Solutions for Real-World Code

PHP Design Patterns: From the Trenches

Real-World Experience and Practical Applications

Kayvan Alimohammadi

December 2025

A comprehensive guide to PHP Design Patterns focused on real-world experience and practical applications. This book covers 11 essential design patterns with 70% practical code examples and 30% theory, targeting intermediate PHP developers who write production code.

Features modern PHP 8.0+ syntax including constructor property promotion, match expressions, readonly properties, union types, and nullsafe operators.

Patterns covered: - Daily Toolkit: Adapter, Builder, Factory Method, Strategy - Flow & Behavior: Chain of Responsibility, Observer, Template Method - Structuring & Flexibility: Bridge, Composite, Decorator, Facade

Contents

Chapter 1

Introduction: How I Discovered Design Patterns

1.1 A Problem That Wouldn't Stay Fixed

Ten years ago, I was working in a development team on a project that needed user authentication through Yahoo accounts. Sounds straightforward, right? Well, not quite.

We'd installed a package to handle Yahoo login, but it had a quirk that drove us crazy. If a user had both a Yahoo account and a Gmail address, and they tried to sign in with their Gmail email, the system would extract and return the Gmail address instead of the Yahoo email. For our use case, this was completely wrong. We needed it to always return the Yahoo email, no matter what.

1.1.1 The Quick Fix That Wasn't

My colleagues came up with a solution—and I'll admit, it seemed reasonable at first. They found the class in the package responsible for extracting the email address, copied it into our project's source code, and then modified it to work the way we needed.

Problem solved. Or so we thought.

Then came the day someone ran `composer update`.

The package got updated to its new version, and... our changes were gone. The modified class reverted back to the original code from the package. So what did we do? We wrote a bash script that would automatically replace the modified file after every Composer update.

1.1.2 When The "Fix" Becomes Worse Than The Problem

Now we had created something fragile and dangerous:

- **The package could change at any moment** - breaking our logic

- **Copying and replacing files was risky** - what if the new version had a completely different structure?
- **The bash script added complexity** - one more thing that could break
- **We were fighting the package, not working with it** - this felt wrong

I remember the frustration. There had to be a better way.

1.2 The Moment Everything Changed

I started researching. Deep research. How did professional teams solve this kind of problem? There had to be a pattern for handling incompatible code.

That's when I discovered the **Adapter pattern**.

Instead of copying and modifying the original class, I created a new class in our source code that acted as an adapter. This adapter wrapped the incompatible Yahoo class and implemented our desired logic on top of it. Now, instead of using the original class directly, we used our adapter.

The results were immediate:

- **No more copied code to maintain**
- **Composer updates didn't break anything** - we were using an adapter, not a modified original
- **Our code was explicit** - it was clear we were adapting a third-party service
- **The solution was elegant** - the problem just... went away

That was my first real encounter with design patterns. Not as abstract theory, but as a practical tool that solved a real problem in production code.

1.3 Why This Book Exists

From that moment on, I started looking at design patterns differently. They weren't academic concepts or ivory tower abstractions. They were solutions to problems I'd already faced or would face in the future.

Over the years, I've used design patterns to:

- Create flexible object creation systems (Factory patterns)
- Build event-driven architectures (Observer)
- Manage complex workflows (Chain of Responsibility)
- Keep my code maintainable as projects grew (Decorator, Strategy)

But I noticed something when I read most design pattern books and articles: they were heavy on theory and light on practice. Lots of diagrams and explanations, but few real-world examples. They answered "what" and "why" but not enough "when" and "how in the real world."

This book is different. It's based on ten years of actually *using* these patterns in production code. In e-commerce systems, content management systems, APIs, authentication systems, caching layers—all the places where patterns matter.

1.4 What You'll Find Here

This isn't a reference book you memorize. It's a practical guide.

70% of this book is code and real-world examples. You'll see how patterns actually work in projects you'd recognize. You'll see common mistakes I've made so you don't have to. You'll see trade-offs and gotchas that you won't find in theoretical books.

30% is the theory you actually need. Enough to understand what the pattern is, why it exists, and when to use it. But not so much that your eyes glaze over.

Each pattern chapter follows the same practical structure:

1. A real problem that made me reach for this pattern
2. How the pattern solves it
3. Complete, production-ready code examples
4. Common mistakes and how I avoided them
5. When to use it, and just as importantly, when *not* to use it

1.5 Who This Book Is For

This book is for you if you're a PHP developer who:

- **Knows the basics** - you can write functions and classes, you understand OOP
- **Builds real projects** - you're not just learning, you're shipping code
- **Wants to write better code** - you've felt the pain of unmaintainable systems and want to do better
- **Values practical knowledge** - you'd rather learn from experience than theory
- **Works in teams** - you need to communicate your designs with other developers

You don't need to be an expert. You don't need a degree. You just need to care about writing code that works, that your team understands, and that won't fall apart when requirements change.

1.6 A Word About This Approach

Everything in this book is based on actual experience. Not theory. Not assumptions. Real problems from real projects.

When I share a design pattern, I'm not just explaining how it works in the abstract. I'm telling you where I used it, why it mattered, and what would have happened if I hadn't. I'm showing you the code that actually runs in production.

This approach has one consequence: you won't learn every single design pattern in existence. This book doesn't cover every pattern from Gang of Four. Instead, it covers the patterns that matter in practical PHP development. The ones that solve real problems.

1.7 How to Read This Book

You don't have to start at chapter one and read straight through. Each pattern chapter is designed to stand alone. If you're dealing with object creation, jump to the Creational Patterns section. If you're building an event system, go to Behavioral Patterns.

But I recommend at least reading the introduction to each part. They'll give you context about why these patterns exist and when you might need them.

When you see code examples, don't just read them. Think about your own projects. Have you faced the same problem? How did you solve it? Would this pattern help?

1.8 Let's Begin

Over the next several chapters, I want to share what ten years of experience has taught me about design patterns. Not as abstract ideas, but as tools that make your code better, more maintainable, and more elegant.

The journey that started with a bash script replacing a modified Yahoo class will show you how to think about design patterns in your own work.

Let's go.

Chapter 2

Adapter Pattern

Structural Pattern | **Daily Toolkit**

2.1 Quick Summary

- **What it solves**: Makes incompatible interfaces work together by creating an adapter that translates between them. Lets you use existing code with interfaces it wasn't designed for.
- **When to use**: Integrating third-party libraries with different interfaces, legacy code that can't be modified, want to use multiple implementations with the same interface
- **When to avoid**: Can add unnecessary indirection, if original interface is flexible enough
- **Complexity**: Low/Medium
- **Real-world examples**: PDF generators, payment gateways, API clients, database drivers, legacy system integration

---

2.2 The Problem

This is where my journey with design patterns actually began, as I mentioned in the introduction. But let me dive deeper into how Adapter saved me from a serious architectural problem.

I was building a system that needed to generate PDF invoices and reports. After researching libraries, I settled on a popular package for converting HTML to PDF. It had excellent documentation and seemed perfect for the job.

I designed my application around a clean interface:

```php
interface InvoiceGeneratorInterface
{
    public function generate(Invoice $invoice): string; // Returns PDF content
    public function save(string $pdfContent, string $filename): void;
}
```

My code relied on this interface throughout the application—in the invoice controller, in scheduled jobs, in email services. Everything expected to work with this contract.

Then I integrated the PDF package. I quickly realized the problem: the package's API looked nothing like my interface.

```php
// The library's actual interface
$pdfGenerator = new \Vendor\Pdf\Generator();
$html = $invoice->renderAsHtml();
$pdfGenerator->fromString($html);
$pdfGenerator->setOption('orientation', 'portrait');
$pdfGenerator->setOption('pageSize', 'A4');
$pdfGenerator->setOption('margin', '10mm');
$pdf = $pdfGenerator->render(); // Returns raw PDF bytes, not a string
$pdfGenerator->output($filename, 'file'); // Saves separately
```

My application expected one thing, the library provided something completely different. I had a few choices:

1. **Rewrite my entire application** to match the library's API (terrible idea)
2. **Modify the library code** to match my interface (breaks when library updates)
3. **Create an adapter** that translates between the two (the right choice)

The third option—creating an adapter—meant I could keep my clean interface, use the external library without modification, and swap to a different PDF library later without changing application code.

This was the real power of the Adapter pattern.

2.3 Understanding the Pattern

2.3.1 What It Is

The Adapter pattern converts the interface of a class into another interface that clients expect. It lets classes work together that couldn't otherwise because of incompatible interfaces.

Think of it like an electrical adapter. You have a device with a US plug, and you're in Europe where outlets are different. An adapter translates between the two, letting your device work without modification.

Key insight: Adapter doesn't change *what* the code does—it translates *how* it does it. The PDF library still generates PDFs. The adapter just wraps it in the interface your application expects.

2.3.2 How It Differs From Other Patterns

- **Decorator**: Adds new behavior to existing objects (decorates them)
- **Adapter**: Translates between incompatible interfaces (bridges the gap)
- **Facade**: Simplifies a complex subsystem (hides complexity)
- **Bridge**: Decouples abstraction from implementation (prevents tight coupling)

2.3.3 Key Components

- **Target Interface**: What your application expects
- **Adapter**: Translates calls from target interface to library's actual interface
- **Adaptee**: The existing library with incompatible interface
- **Client Code**: Your application that uses the Target interface

---

2.4 Basic Implementation

Let me show you the simplest version of an adapter.

```php
<?php

namespace App\Invoices;

// Your application's expected interface
interface InvoiceGeneratorInterface
{
    public function generate(Invoice $invoice): string;

    public function save(string $pdfContent, string $filename): void;
}

// The external library (you can't modify this)
namespace Vendor\Pdf;

class PdfLibrary
{
    public function fromString(string $html): self
    {
        // Sets HTML content
        return $this;
    }

    public function setOption(string $key, string $value): self
    {
```

```
        // Sets options
        return $this;
    }

    public function render(): string
    {
        // Returns PDF as string
        return "PDF content...";
    }

    public function output(string $filename, string $type): void
    {
        // Saves to file
    }
}

// The Adapter - bridges your interface and the library
namespace App\Invoices\Adapters;

use App\Invoices\InvoiceGeneratorInterface;
use Vendor\Pdf\PdfLibrary;

class PdfLibraryAdapter implements InvoiceGeneratorInterface
{
    private PdfLibrary $pdfLibrary;

    public function __construct(private readonly PdfLibrary $pdfLibrary) {}

    public function generate(Invoice $invoice): string
    {
        // Translate your interface to library interface
        $html = $invoice->renderAsHtml();

        return $this->pdfLibrary
            ->fromString($html)
            ->setOption(key: 'orientation', value: 'portrait')
            ->setOption(key: 'pageSize', value: 'A4')
            ->render();
    }

    public function save(string $pdfContent, string $filename): void
    {
        // Library saves differently, we need to adapt
        // Save the content we got from generate()
        file_put_contents(filename: $filename, data: $pdfContent);
    }
}

// Usage - application code never knows about the library
$adapter = new PdfLibraryAdapter(new PdfLibrary());
$pdf = $adapter->generate($invoice);
$adapter->save($pdf, 'invoice.pdf');
```

That's it. The adapter translates between what your application expects and what the library provides.

2.5 Real-World Implementation

Now let me show you a complete system with multiple PDF generators, where adapters let you swap implementations without changing application code.

2.5.1 Complete Example: Multi-Provider PDF Generation System

Context: An invoicing system that supports multiple PDF generation libraries. Different libraries have different APIs, but the application code needs a consistent interface. An adapter allows swapping libraries or supporting multiple simultaneously without modifying application code.

The Code:

```php
<?php

namespace App\Invoicing;

use App\Models\Invoice;
use DateTime;

// The interface your application uses
interface PdfGeneratorInterface
{
    /**
     * Generate PDF content from invoice.
     */
    public function generatePdf(Invoice $invoice): string;

    /**
     * Save PDF content to file.
     */
    public function savePdf(string $pdfContent, string $filepath): void;

    /**
     * Get the name of the PDF generator being used.
     */
    public function getName(): string;
}

// Value objects for PDF generation options
final class PdfGenerationOptions
{
    public function __construct(
        public readonly string $orientation = 'portrait',
```

```
        public readonly string $pageSize = 'A4',
        public readonly string $margin = '10mm',
        public readonly bool $includeFooter = true,
    ) {}
}

// The Invoice model (simplified for example)
final class Invoice
{
    public function __construct(
        public readonly string $id,
        public readonly string $customerName,
        public readonly float $amount,
        public readonly DateTime $issueDate,
    ) {}

    public function renderAsHtml(): string
    {
        return sprintf(
            '<html><body><h1>Invoice %s</h1><p>Customer: %s</p><p>Amount:
            ↪  $%.2f</p></body></html>',
            $this->id,
            $this->customerName,
            $this->amount
        );
    }
}

// ================================================================
// ADAPTER 1: For a popular PDF library (like MPDF or TCPDF)
// ================================================================

namespace App\Invoicing\Adapters;

use Vendor\Pdf\MpdfLibrary;

final class MpdfAdapter implements PdfGeneratorInterface
{
    public function __construct(
        private readonly MpdfLibrary $mpdf,
        private readonly PdfGenerationOptions $options = new PdfGenerationOptions(),
    ) {}

    public function generatePdf(Invoice $invoice): string
    {
        try {
            // Translate to Mpdf's interface
            $html = $invoice->renderAsHtml();

            // Mpdf constructor takes format
            $format = $this->mapPageSize($this->options->pageSize);
```

```
            $this->mpdf->setPageSize(format: $format);
            $this->mpdf->setMargins(
                left: $this->parseMargin($this->options->margin),
                top: $this->parseMargin($this->options->margin),
                right: $this->parseMargin($this->options->margin),
                bottom: $this->parseMargin($this->options->margin),
            );

            $this->mpdf->writeHTML(html: $html);

            return $this->mpdf->output('', 'S'); // S = return as string
        } catch (\Exception $e) {
            throw new PdfGenerationException("Mpdf generation failed: " .
            ↪  $e->getMessage());
        }
    }

    public function savePdf(string $pdfContent, string $filepath): void
    {
        if (!file_put_contents($filepath, $pdfContent)) {
            throw new PdfGenerationException("Failed to save PDF to: $filepath");
        }
    }

    public function getName(): string
    {
        return 'Mpdf';
    }

    private function mapPageSize(string $pageSize): string
    {
        return match($pageSize) {
            'A4' => 'A4',
            'A3' => 'A3',
            'Letter' => 'Letter',
            default => 'A4',
        };
    }

    private function parseMargin(string $margin): float
    {
        // Convert "10mm" to millimeters
        return (float)str_replace(['mm', 'cm', 'in'], '', $margin);
    }
}

// ============================================================
// ADAPTER 2: For another PDF library (like DomPDF)
// ============================================================

namespace App\Invoicing\Adapters;
```

```php
use Vendor\Pdf\DompdfLibrary;

final class DompdfAdapter implements PdfGeneratorInterface
{
    public function __construct(
        private readonly DompdfLibrary $dompdf,
        private readonly PdfGenerationOptions $options = new PdfGenerationOptions(),
    ) {}

    public function generatePdf(Invoice $invoice): string
    {
        try {
            $html = $invoice->renderAsHtml();

            // Dompdf has completely different interface
            $this->dompdf->loadHtml(html: $html);
            $this->dompdf->setPaper(
                size: $this->options->pageSize,
                orientation: $this->options->orientation,
            );
            $this->dompdf->render();

            return $this->dompdf->output(); // Dompdf returns PDF directly
        } catch (\Exception $e) {
            throw new PdfGenerationException("Dompdf generation failed: " .
            ↪  $e->getMessage());
        }
    }

    public function savePdf(string $pdfContent, string $filepath): void
    {
        if (!file_put_contents($filepath, $pdfContent)) {
            throw new PdfGenerationException("Failed to save PDF to: $filepath");
        }
    }

    public function getName(): string
    {
        return 'Dompdf';
    }
}

// ============================================================
// ADAPTER 3: For TCPDF library
// ============================================================

namespace App\Invoicing\Adapters;

use Vendor\Pdf\TcpdfLibrary;

final class TcpdfAdapter implements PdfGeneratorInterface
{
```

```
    public function __construct(
        private readonly TcpdfLibrary $tcpdf,
        private readonly PdfGenerationOptions $options = new PdfGenerationOptions(),
    ) {}

    public function generatePdf(Invoice $invoice): string
    {
        try {
            $html = $invoice->renderAsHtml();

            // TCPDF uses yet another interface
            $this->tcpdf->setPageSize($this->options->pageSize);
            $this->tcpdf->setOrientation($this->options->orientation);
            $this->tcpdf->addPage();
            $this->tcpdf->writeHTML($html);

            return $this->tcpdf->output('', 'S'); // S = return as string
        } catch (\Exception $e) {
            throw new PdfGenerationException("TCPDF generation failed: " .
            ↪  $e->getMessage());
        }
    }

    public function savePdf(string $pdfContent, string $filepath): void
    {
        if (!file_put_contents($filepath, $pdfContent)) {
            throw new PdfGenerationException("Failed to save PDF to: $filepath");
        }
    }

    public function getName(): string
    {
        return 'TCPDF';
    }
}

// ============================================================
// EXCEPTION CLASS
// ============================================================

namespace App\Invoicing;

final class PdfGenerationException extends \Exception {}

// ============================================================
// PDF GENERATOR FACTORY
// ============================================================

namespace App\Invoicing;

use App\Invoicing\Adapters\{
    DompdfAdapter,
```

```
    MpdfAdapter,
    TcpdfAdapter,
};

final class PdfGeneratorFactory
{
    public static function create(
        string $library,
        PdfGenerationOptions $options = new PdfGenerationOptions()
    ): PdfGeneratorInterface {
        return match($library) {
            'mpdf' => new MpdfAdapter(
                new \Vendor\Pdf\MpdfLibrary(),
                $options
            ),
            'dompdf' => new DompdfAdapter(
                new \Vendor\Pdf\DompdfLibrary(),
                $options
            ),
            'tcpdf' => new TcpdfAdapter(
                new \Vendor\Pdf\TcpdfLibrary(),
                $options
            ),
            default => throw new \InvalidArgumentException("Unknown PDF library:
            ↪  $library"),
        };
    }
}

// ============================================================
// APPLICATION CODE - NO LIBRARY-SPECIFIC LOGIC!
// ============================================================

namespace App\Invoicing;

final class InvoiceService
{
    public function __construct(private readonly PdfGeneratorInterface $pdfGenerator) {}

    /**
     * Generate PDF and save to disk.
     * This code is IDENTICAL regardless of which PDF library we use!
     */
    public function generateAndSaveInvoicePdf(
        Invoice $invoice,
        string $outputPath,
    ): void {
        try {
            $pdfContent = $this->pdfGenerator->generatePdf(invoice: $invoice);
           $this->pdfGenerator->savePdf(pdfContent: $pdfContent, filepath: $outputPath);
        } catch (PdfGenerationException $e) {
            throw new \RuntimeException("Failed to generate invoice PDF: " .
            ↪  $e->getMessage());
```

```
        }
    }

    /**
     * Generate and return PDF for streaming to browser.
     */
    public function generateInvoicePdfForDownload(Invoice $invoice): string
    {
        return $this->pdfGenerator->generatePdf(invoice: $invoice);
    }

    /**
     * Generate multiple invoices in batch.
     */
    public function generateBatchInvoicePdfs(array $invoices, string $outputDir): void
    {
        foreach ($invoices as $invoice) {
            $filename = sprintf('%s/invoice_%s.pdf', $outputDir, $invoice->id);
            $this->generateAndSaveInvoicePdf(invoice: $invoice, outputPath: $filename);
        }
    }
}

// ============================================================
// USAGE IN CONTROLLERS
// ============================================================

namespace App\Http\Controllers;

use App\Invoicing\{
    Invoice,
    InvoiceService,
    PdfGeneratorFactory,
};

class InvoiceController
{
    private InvoiceService $invoiceService;

    public function __construct()
    {
        // Get the configured PDF library from config
        $pdfLibrary = config('pdf.library', 'dompdf');

        // Factory creates the right adapter
        $pdfGenerator = PdfGeneratorFactory::create($pdfLibrary);

        // Service uses the adapter - but doesn't know which library it is
        $this->invoiceService = new InvoiceService($pdfGenerator);
    }

    public function downloadInvoice(string $invoiceId)
```

```
    {
        $invoice = Invoice::find($invoiceId);

        // Service generates PDF using whatever library is configured
        // No if/else statements about which library to use!
        $pdfContent = $this->invoiceService->generateInvoicePdfForDownload($invoice);

        return response($pdfContent, 200)
            ->header('Content-Type', 'application/pdf')
            ->header('Content-Disposition', 'attachment; filename="invoice.pdf"');
    }

    public function emailInvoice(string $invoiceId)
    {
        $invoice = Invoice::find($invoiceId);
        $tempFile = '/tmp/invoice_' . $invoiceId . '.pdf';

        // Generate and save
        $this->invoiceService->generateAndSaveInvoicePdf($invoice, $tempFile);

        // Email it (no library-specific code!)
        Mail::send(new InvoiceMailable($invoice, $tempFile));

        // Cleanup
        unlink($tempFile);
    }
}

// ============================================================
// CONFIGURATION - CHOOSE YOUR PDF LIBRARY HERE
// ============================================================

// config/pdf.php
return [
    'library' => env('PDF_LIBRARY', 'dompdf'), // Change this and everything works!
    'options' => [
        'orientation' => 'portrait',
        'pageSize' => 'A4',
        'margin' => '10mm',
    ],
];
```

How It Works:

Each PDF library has a different API. Instead of embedding library-specific code throughout the application, we create an adapter for each library. The adapter translates the application's expected interface to whatever the library requires.

When you want to switch PDF libraries—maybe Dompdf is slow and you want to try Mpdf—you just change one config value. No application code changes.

Why This Approach:

1. **Isolation** - Library-specific code is isolated in adapters
2. **Swappability** - Switch implementations by changing configuration
3. **Testability** - Mock the interface, not the library
4. **Reusability** - Same application code works with any PDF library
5. **No Modification** - Use external libraries without modifying them

---

2.6 Variations & Trade-offs

2.6.1 Variation 1: Class Adapter (Inheritance)

Instead of wrapping (composition), inherit from the library:

```
// Use when the library is a class you can extend
final class AdapterViaInheritance extends VendorPdfLibrary implements
↪  PdfGeneratorInterface
{
    public function generatePdf(Invoice $invoice): string
    {
        $html = $invoice->renderAsHtml();
        $this->fromString($html);
        return $this->render();
    }

    public function savePdf(string $content, string $filepath): void
    {
        file_put_contents($filepath, $content);
    }
}
```

- **Pros**: Simpler, no wrapping needed
- **Cons**: Tight coupling to library, can't extend multiple classes
- **When to choose**: Simple adapters, single inheritance needed

2.6.2 Variation 2: Two-Way Adapter

Adapts in both directions:

```
final class BiDirectionalAdapter implements PdfGeneratorInterface, VendorInterface
{
    public function generatePdf(Invoice $invoice): string
    {
        // Translate from application to vendor
        return $this->toApplicationFormat(
            $this->vendorGenerate($invoice->renderAsHtml())
        );
    }

    public function vendorMethod($param): void
```

```
    {
        // Translate vendor calls back to application format
        // Allows vendor code to call application interface
    }
}
```

- **Pros**: Both sides can use each other
- **Cons**: More complex, harder to test
- **When to choose**: Bidirectional integration needed

2.6.3 Framework Integration: Laravel

In Laravel, adapters work well with service container binding:

```php
// In service provider
$this->app->bind(PdfGeneratorInterface::class, function($app) {
    $library = config('pdf.library');
    return PdfGeneratorFactory::create($library);
});

// In controller - automatic injection
public function generatePdf(PdfGeneratorInterface $generator, Invoice $invoice)
{
    $pdf = $generator->generatePdf($invoice);
}
```

2.7 Common Mistakes

2.7.1 Mistake 1: Adapter Too Smart

Problem: Adapter does more than translate - it adds business logic.

```php
// WRONG - Adapter is doing too much
final class PdfAdapter implements PdfGeneratorInterface
{
    public function generatePdf(Invoice $invoice): string
    {
        // Just translation is fine, but this adds logic:
        if ($invoice->isPaid()) {
            $html = $invoice->renderAsHtmlWithWatermark('PAID');
        } else {
            $html = $invoice->renderAsHtml();
        }

        // And this:
        $discount = $invoice->getApplicableDiscount();

        // These should be in Invoice or a service, not the adapter!
```

```
        $this->library->applyDiscount($discount);

        return $this->library->render();
    }
}
```

Solution: Keep adapter focused on translation:

```
// Correct - Adapter only translates
final class PdfAdapter implements PdfGeneratorInterface
{
    public function generatePdf(Invoice $invoice): string
    {
        // Get prepared HTML from Invoice - invoice handles its own logic
        $html = $invoice->renderAsHtml();

        // Translate to library interface
        return $this->library
            ->fromString($html)
            ->render();
    }
}
```

2.7.2 Mistake 2: Leaking Library Details

Problem: Adapter returns library-specific objects to client code.

```
// WRONG - Returns library object
interface PdfGeneratorInterface
{
    public function generatePdf(Invoice $invoice): LibraryPdfObject;
}

// Now client code knows about the library!
$pdf = $generator->generatePdf($invoice);
$pdf->watermark('DRAFT'); // This is library-specific
```

Solution: Keep library details hidden:

```
// Correct - Adapts all the way
interface PdfGeneratorInterface
{
    public function generatePdf(Invoice $invoice): string; // Just PDF content
    public function watermark(string $text, string $content): string; // Adapted method
}

// Client code never knows about library
$pdf = $generator->generatePdf($invoice);
$pdf = $generator->watermark('DRAFT', $pdf); // Generic interface
```

2.7.3 Mistake 3: Adapter That's Too Generic

Problem: Trying to adapt everything, losing the specific interface your app needs.

```
// WRONG - Too generic
interface LibraryAdapterInterface
{
    public function call(string $method, array $params): mixed;
}

// This defeats the purpose - might as well use library directly!
$result = $adapter->call('generatePdf', [$invoice]);
```

Solution: Keep adapter specific to what you need:

```
// Correct - Focused on application needs
interface PdfGeneratorInterface
{
    public function generatePdf(Invoice $invoice): string;
    public function savePdf(string $content, string $filepath): void;
    public function getName(): string;
}

// This forces adapters to be explicit about what they support
```

2.8 Testing the Pattern

Testing adapters is straightforward - test that they correctly translate between interfaces:

```
<?php

namespace Tests\Unit\Invoicing;

use PHPUnit\Framework\TestCase;
use App\Invoicing\{
    Invoice,
    PdfGeneratorInterface,
};
use App\Invoicing\Adapters\DompdfAdapter;
use DateTime;

class PdfAdapterTest extends TestCase
{
    private PdfGeneratorInterface $adapter;
    private Invoice $invoice;

    protected function setUp(): void
    {
        // Use a mock or stub of the actual library
        $mockDompdf = $this->createMockDompdf();
```

```
        $this->adapter = new DompdfAdapter($mockDompdf);

        $this->invoice = new Invoice(
            id: 'INV-001',
            customerName: 'John Doe',
            amount: 150.00,
            issueDate: new DateTime('2024-01-15')
        );
    }

    public function testGeneratePdfReturnsPdfContent(): void
    {
        $pdf = $this->adapter->generatePdf($this->invoice);

        $this->assertIsString($pdf);
        $this->assertStringContainsString('PDF', $pdf); // PDF files start with %PDF
    }

    public function testAdapterTranslatesInvoiceToLibraryFormat(): void
    {
        // Verify the adapter correctly calls library methods
        $mockDompdf = $this->createMockDompdfWithExpectations();
        $adapter = new DompdfAdapter($mockDompdf);

        $adapter->generatePdf($this->invoice);

        // Verify loadHtml was called with rendered HTML
        // Verify setPaper was called with correct options
        // Etc...
    }

    public function testSavePdfCreatesFile(): void
    {
        $testFile = tempnam(sys_get_temp_dir(), 'pdf_');
        unlink($testFile); // Delete so we can test creation

        $pdfContent = "mock PDF content";
        $this->adapter->savePdf($pdfContent, $testFile);

        $this->assertFileExists($testFile);
        $this->assertEquals($pdfContent, file_get_contents($testFile));

        unlink($testFile); // Cleanup
    }

    public function testGetNameReturnsAdapterName(): void
    {
        $name = $this->adapter->getName();

        $this->assertEquals('Dompdf', $name);
    }
```

```
    private function createMockDompdf()
    {
        // Return a mock that behaves like Dompdf
        $mock = \Mockery::mock(\Vendor\Pdf\DompdfLibrary::class);
        $mock->shouldReceive('loadHtml')->andReturnSelf();
        $mock->shouldReceive('setPaper')->andReturnSelf();
        $mock->shouldReceive('render')->andReturnSelf();
        $mock->shouldReceive('output')->andReturn('mock pdf content');

        return $mock;
    }
}

// Test that any adapter can be swapped
class PdfGeneratorFactoryTest extends TestCase
{
    public function testFactoryCreatesCorrectAdapter(): void
    {
        $dompdfAdapter = PdfGeneratorFactory::create('dompdf');
        $this->assertInstanceOf(DompdfAdapter::class, $dompdfAdapter);

        $mpdfAdapter = PdfGeneratorFactory::create('mpdf');
        $this->assertInstanceOf(MpdfAdapter::class, $mpdfAdapter);
    }

    public function testAllAdaptersImplementInterface(): void
    {
        foreach (['dompdf', 'mpdf', 'tcpdf'] as $library) {
            $adapter = PdfGeneratorFactory::create($library);
            $this->assertInstanceOf(PdfGeneratorInterface::class, $adapter);
        }
    }
}
```

2.9 Should You Use This Pattern?

Scenario	Use?	Why
Integrating third-party library with different interface	✅	This is Adapter's primary use case
Want to swap implementations without changing code	✅	Adapter isolates library-specific code
Library API doesn't match your domain	✅	Adapter bridges the gap
Need to support multiple libraries	✅	Create an adapter for each

Scenario	Use?	Why
Library interface matches your needs exactly	□	No need to add indirection
One-time integration, never swapping	~	Consider complexity vs. benefit

---

2.10 Further Reading

- **Facade**: Simplifies complex subsystems (similar structure, different intent)
- **Decorator**: Adds behavior without changing interface (different from Adapter)
- **Bridge**: Separates abstraction from implementation (proactive vs. reactive)
- **Factory Method**: Often paired with Adapter to create adapters