www.phparch.com

May 2023
Volume 22 - Issue 5

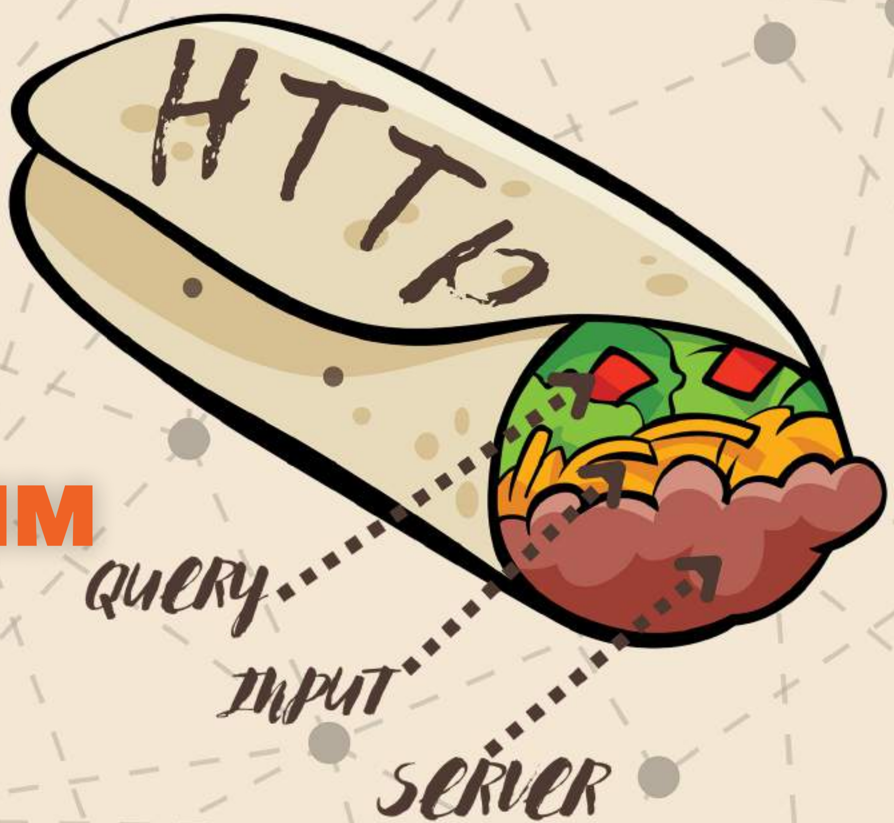
php[architect]

The Magazine For PHP Professionals

HTTP Burritos

HTTP Tortilla

Life After GIMM



ALSO INSIDE

The Workshop:
Minicli

DDD Alley:
First, Make it Easy

PHP Puzzles:
Maze Rats, Part 2

Security Corner:
Tabletop: Planning for Disaster

Education Station:
AI is not Coming For Your Job

PSR Pickup:
PSR-20: Clock

Artisan Way:
ADR vs MVC

finally{}:
CatAIstrophe

CONTENTS

MAY 2023
Volume 22 - Issue 05

Edited as last minute as possible

php[**architect**] is published twelve times a year by:

PHP Architect, LLC
9245 Twin Trails Dr #720503
San Diego, CA 92129, USA

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

The Future of Higher Ed

In today's rapidly evolving digital world, traditional education in the realm of computer programming, is being challenged like never before. It was once thought that a college degree was the only way to obtain a high-paying job. But is that still true today?

The field of computer programming is distinct from many other professions. It's an arena where your portfolio and skills often carry more weight than a degree. Programming is fundamentally about problem-solving, creativity, and a keen understanding of technologies being used to solve those problems. These abilities aren't exclusive to the college classroom; they can be learned in many ways today, including reading articles in PHP Architect, listening to podcasts, attending conferences such as php[tek], and talking to like-minded people that are willing to help.

PHP Architect has always tried to cater to beginners and experienced developers alike, with the added advantage of learning new skills that you don't even know you may need one day. This is a benefit that the rigid structure of a traditional college course seldom provides.

Coding boot camps offer another compelling alternative. These intensive, short-term programs focus on imparting practical skills swiftly, enabling want-to-be developers the opportunity to be job-ready within a few months. Many also provide career support and can help you land a job. With the shorter time commitment and typically lower cost compared to college, boot camps can be a viable alternative for many aspiring programmers.

There's also a significant trend of tech giants like Google, Apple, and IBM no longer requiring a college degree for many roles, including programming. They have recognized that skills and talent can come from non-traditional routes. This shift in focus from college credentials to a person's capability seems in line with today's world.

As a college graduate, I am conflicted about my stance on this. College was a huge expense, and I was lucky enough to stay focused and pay it off without deferral. Based on the cost and the fact that I didn't really learn anything that I use today, I am not sure it was worth it. But then I step back and think about all the other experiences I had, which I may not have had without college. Those stories and experiences have helped shape me and make me who I am today. So… with that, would I be where I am today without a college degree?

To be clear, this is not a call to dismiss college education altogether. For some, a college's structured environment and breadth of learning may be the best route. What's crucial, however, is acknowledging that it's not the only route. Aspiring programmers should have the flexibility to choose the pathway that best fits their learning style, career goals, and financial situation.

Moreover, the industry needs to continue evolving its hiring practices, placing even greater emphasis on skills and less on degrees. This change could help mitigate the perennial problem of "degree inflation" and open up the field to a broader range of talented individuals.

This month's feature articles start with an overview of an open-source package called HTTP Tortilla by Tim Lytle. While our second article is a little less technical, focusing on education with "Life After GIMM: Jumping into the Tech Industry After College" by Adam Giles.

Next, our columnists give us some great content. Eric Mann brings us "Tabletop: Planning for Disaster" over in Security Corner. Are you prepared for the case when things inevitably go sideways? Joe Ferguson is back in The Workshop, bringing us an article on "Minicli". Next, you will want to help your code adhere to standards by following along in Frank Wallen's PSR Pickup column, where he shares information about "PSR-20: Clock". Over in Artisan Way, Matt Lantz discusses "ADR vs MVC". Which one is right for you? The hot topic lately is all around AI, so Chris Tankersly has jumped into Education Station with "AI is not Coming For Your Job". Oscar Merida continues to expand our PHP Puzzles with "Maze Rats, Part Two". Ed Barnard wants us to "First, Make it Easy" in this month's DDD Alley. And finally{}, Beth Tucker Long leaves us with "CatAIstrophe".

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcments. Make sure you're connected with us.

- Subscribe to our list: https://phpa.me/sub-to-updates
- Mastodon: @editor@phparch.social
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download the Code Archive:

https://phpa.me/May2023_code

HTTP Tortilla

Tim Lytle

The PSR-7 interfaces for request and response messages are remarkably valuable. They establish a well defined interface for working with HTTP requests and responses. They introduce a reasonable amount of immutability (streams be streams, after all). Together, both lead to ease when testing and provide confidence and simplicity when integrating with existing libraries that wrangle, manipulate, and extract information from the HTTP life-cycle.

But everything in software development is a trade-off, right? The PSR interfaces define what might be considered a *minimum* set of methods needed to inspect and build up a response or request. While that's great for interfacing with code we don't own; when used internally, we can start to long for convenience methods—perhaps a syntactical sugar fix—tailored to our specific use cases.

Maybe instead of the simply *sufficient* `$request->getQueryParams()`, we'd prefer a more helpful `$request->getQueryParam($name, $default = null)`. Or maybe instead of checking if `$member = $request->getAttribute(Member::class)` is an instance of our member entity, we'd rather use `$request->isAuthenticated()` and `$request->getAuthenticatedMember()`.

Both of these improvements reduce a fair amount of boilerplate code and DRY things up for us. A simplistic approach could be creating our own 'request' and simply wrapping a PSR-7 request. Then we implement whatever domain-specific methods we'd like.

That works to a point. We're fine as long as it's only our code consuming our custom request object. But as soon as we want to integrate with library code expecting a PSR-7 request, we're in trouble unless our custom requests implement *all* of the PSR-7 methods. And once we see the value of something like PSR-15's middleware pattern, we will want to leverage that—which requires our requests to implement PSR-7 interfaces.

Want to know more about PSR-7 and the value of PSR-15? Take a look at this talk[1].

We could extend whatever library we're using, but we'd potentially run into issues with the `with*()` methods if the library doesn't use `static`. And even if it does, extending library code is problematic for a few reasons. Testing just our additions becomes difficult, and we risk using non-`public` methods and coupling our changes to code that may very well change.

So, the best solution—though tedious—seems to be still making our own request object, but this time implementing all the PSR-7 methods.

I was puzzling over this at the last [php[world]] during a panel discussion on frameworks where Ryan Weaver[2] mentioned this very problem. On one hand, wanting clean, simple, and common interfaces for interoperability. But also wanting methods that may not be any of those things, but add convenience for the consuming developer in a specific context.

I was puzzled over it because internally, we had started using PSR-7, but wrapped it in our own more developer-friendly object that exposed data in a more useful way—at least in our context. But as we moved towards implementing PSR-15 middleware, we encountered a problem. Our request wasn't a PSR-7 request, but we liked using it. And now I was faced with that tedious solution of implementing all the PSR-7 methods on our request object so we could keep the convenience and still start moving towards middleware.

We already just wrapped a PSR-7 instance and used that for our methods. After all, our goal was to remove the boilerplate code, not re-implement an entire request object. So satisfying the entire interface could be simply proxying the wrapped object's methods.

To clarify the behavior our request provided, the wrapping could be done in a trait. That moves all the boilerplate proxy methods into the trait and makes it easy to see what methods our request added. And when we want to change the behavior of one of the PSR-7 methods, it's easy to see which we updated. (See Listing 1 on the folling page)

With the trait, we can create a new custom request, add the query param method we'd like, and have all the other methods—including the one we call in our new method—proxied automatically. (See Listing 2 on the next page)

But since PSR-7 objects are not mutable, and we're just proxying the wrapped request, any `with*()` method returns a modified version of the wrapped request, not our version of the request. This leads to frustrating code where we mutate the request and then have to re-wrap it to guarantee we have our request.

```
$request = new ServerRequest(
    $request->withAttribute('member', $member)
);
$filter = $request->getQueryParam('filter', '');
```

A few modifications to our trait allow a wrapping 'factory' to be defined so `with*()` requests can pass the returned object to that factory for wrapping. Now evolving the request continues to return an instance of our request object, not the wrapped object. (See Listings 3 & 4 on the following pages)

1 this talk: https://www.youtube.com/watch?v=dutrUmKcTng

2 Ryan Weaver: https://twitter.com/weaverryan

Listing 1.

```
trait ServerRequestWrapper
{
    private ServerRequestInterface $wrapped;

    protected function setWrapped(
        ServerRequestInterface $request
    ): void {
        $this->wrapped = $request;
    }

    private function getWrapped():
        ServerRequestInterface
    {
        if (!
    ($this->wrapped instanceof ServerRequestInterface)
        ) {
            throw new \UnexpectedValueException(
                'must `setMessage` before using it'
            );
        }

        return $this->wrapped;
    }

    public function getServerParams(): array
    {
        return $this->getWrapped()->getServerParams();
    }

    // ...
}
```

Listing 2.

```
class ServerRequest implements ServerRequestInterface
{
  use ServerRequestWrapper;

  // a server request is usually constructed by
  // some factory, so we expected to be passed
  // the request to wrap
  public function __construct(
    ServerRequestInterface $request
  ) {
    // tell the trait what the wrapped request is
    $this->setWrapped($request);
  }

  public function getQueryParam(
    $name, $default = null
  ): ?string {
    return $this->getQueryParams()[$name] ?? $default;
  }
}
```

Listing 3.

```
trait ServerRequestWrapper
{
    private $wrapped;
    private $factory;

    // ...

    protected function setFactory(
        callable $factory
    ): void {
        $this->factory = $factory;
    }

    private function viaFactory(
        MessageInterface $message
    ): MessageInterface {
       if (!$this->factory) {
           return $message;
       }

       return call_user_func($this->factory, $message);
    }

    public function withCookieParams(
        array $cookies
    ): ServerRequestInterface {
      return $this->viaFactory(
        $this->getWrapped()->withCookieParams($cookies)
      );
    }

    // ...
}
```

If we're going to build a trait for the server request, it seems like we might as well make traits for the rest of the interfaces the PSR defines. That gives us the ability to make as many custom requests, responses, or URIs as we'd like. But why would we need more than one?

For one, if the library we're using doesn't provide concrete implementations for specific kinds of responses—for example, a JSON response, a text response, or an API Problem response—we can easily create those with a class that does little more than wrangle constructor arguments and set headers. In some cases, we may want to add access to the underlying data as well. Unlike wrapping a server request, we don't need to make the response and pass it to our new class; it can make the wrapped response itself. (See Listing 5)

But perhaps we should think about the question a little differently. Instead of what set of objects might be nice to have, we could ask what existing objects should also *be* a request, a response, or a URI.

Would it be nice to `throw` a response? Or, put another way, what if an `Exception` could also decide how it should be represented to the user? Would that be useful? It certainly seems like it would. We can follow the patterns we've already used, making the wrapped response internally. But instead of our

Listing 4.

```
class ServerRequest implements ServerRequestInterface
{
    use ServerRequestWrapper;

    private function __construct(
        ServerRequestInterface $request
    ) {
        // tell the trait what the wrapped request is
        $this->setWrapped($request);

        // any with method will now take the response
        // of the wrapped object, re-wrap it in
        // this class, and return the wrapping object
        // preserving our `CustomServerRequest`
        // through calls to with() methods
        $this->setFactory([self::class, 'instance']);
    }

    // a server request is usually constructed by
    // some factory, so we expected to be passed
    // the request to wrap
    public static function instance(
        ServerRequestInterface $request
    ): self {
        // only wrap if we have to
        if (!($request instanceof self)) {
            $request = new self($request);
        }

        return $request;
    }

    public function getQueryParam(
        string $name, string $default = null
    ): ?string {
        return $this->getQueryParams()[$name]
                    ?? $default;
    }
}
```

new class being primarily a response, it's now just *also* a response. (See Listing 6)

How about URIs? Do we have existing objects that could be considered a URI? If we're using some kind of request routing system, it's likely that our route definitions are very URI-like. I'm sure whatever represents a matched route has everything needed to make a URI. And like our other classes, we don't need to implement all the URI methods; we just provide the URI that we're wrapping. So if we already had something like `$route_definition->getUri()`, it's trivial to *be* the URI instead. (See Listing 7)

What's that you say? The library you're using already provides all the format specific responses you could ask for. Well, let's consider one final example use case. What if you had something that would be rendered as a response but would be useful if it wasn't rendered quite yet? We might call that a lazy response.

Listing 5.

```
class HalResponse implements ResponseInterface
{
    use ResponseWrapper;
    private array $hal;
    public function __construct(
        array $hal,
        int $status = HttpStatus::OK,
        array $headers = [],
        int $options = self::DEFAULT_JSON_FLAGS
    ) {
        $json = json_seralize($hal, $options);
        $type = 'application/hal+json';
        $response = (new Response($json))
          ->withStatus($status)
          ->withHeader('Content-Type', $type);

        foreach($headers as $header => $value) {
            $response = $response
                        ->withHeader($header, $value);
        }
        $this->hal = $hal;
        $this->setWrapped($response);
    }

    public function getHal(): array {
        return $this->hal;
    }
}
```

Listing 6.

```
class MethodNotAllowedProblemResponse
        extends RuntimeException
        implements ResponseInterface
{
    use ResponseWrapper;

    private function __construct()
    {
        $status_code = HttpStatus::METHOD_NOT_ALLOWED;
        $title = 'Method Not Allowed';
        $detail = 'HTTP method used is not supported.';

        $this->setWrapped(
            ApiProblemFactory::make(
                $status_code,
                $title,
                ['detail' => $detail]
            )
        );
        parent::__construct(
            "{$title}: {$detail}", $status_code
        );
    }

    public static function make(): self {
        return new self();
    }
}
```

Listing 7.

```
class RouteDefinition {
    use UriWrapper;

    public function __construct(
        public readonly string $path,
        public readonly array $params
    )
    {
        // render a templated Uri that
        // matches our definition
        $this->setWrapped(new Uri(
            (string)
                (new UriTemplate($this->path))
                ->render($this->params)
        ));
    }
}
```

It could contain everything needed to realize the response, but unlike our other examples, it'd wait until the last possible moment instead of making the wrapped response early. And, to be useful, we could allow access to factory-like arguments that would be used to make up the response. (See Listing 8)

Considering that PSR-15 middleware deal in requests and responses, we'd have something we could pass up the middleware chain because it *is* a response. However, other middleware could look at what *really* makes up the response and potentially use those individual arguments to create another lazy response with even more detail.

When the response represents a collection, middleware could add paging constraints isolating the paging code to a single middleware instead of every API handler. If we use a hypermedia-like format, middleware could notice the resource the response represents and identify related resources that should be embedded into the response.

And we don't need to find an API library to do that because it's now really easy for us to introduce custom behavior in PSR-7 objects due to some simple yet powerful traits.

These traits do little more than wrap up all the wonderful ingredients needed to make the custom PSR-7 object you've been looking for. We call them `http-tortilla`[3] and we have open-sourced them so you can make your own HTTP burrito. Install them with `composer require phoneburner/http-tortilla`

If you'd like to see a bit more of example use, take a look at github.com/tjlytle/tortilla-example[4]. It's a simple API with a series of commits using `http-tortilla` in different ways.

Listing 8.

```
class LazyResponse implements ResponseInterface {
    use ResponseWrapper;
    protected ResponseInterface $wrapped;

    public function __construct(
        public readonly object $resource,
        public readonly Transformer $transformer,
        public readonly int $status = HttpStatus::OK
    ) { }

    public function withResource(
        object $resource
    ): self {
        return new self
          ($resource, $this->transformer, $this->status);
    }

    public function withTransformer(
        Transformer $transformer
    ): self {
        return new self(
            $this->resource,
            $transformer,
            $this->status
        );
    }

    private function getWrapped(): ResponseInterface
    {
        return $this->wrapped ??= (
          fn() => new HalResponse (
            $this->transformer->make($this->resource),
            $this->status
          )
        )();
    }
}
```

Tim likes making things, is a lover of well built APIs, and hates the top reply. A long time mercenary PHP developer, he's a fan of Laminas (you know, ZendFramework), Doctrine (2 of course), TDD, and practical design patterns. He's now working on things at PhoneBurner, tweeting sporadically from `@tjlytle`*, and hanging out with his wife and kids.*

3 https://phpa.me/github

4 http://github.com/tjlytle/tortilla-example

Life After GIMM: Jumping into the Tech Industry Post-College

Adam Giles

Games, Interactive Media, and Mobile, otherwise known as GIMM, is a program specific to Boise State University. On paper, it may look like an opportunity to play video games at school, but it goes far beyond that. The program introduces and trains students on the emerging technologies of the world. This means that professors focus on teaching students various topics related to game development and so much more. The curriculum allows students to learn about full-stack web development, 3D animation, and AR/VR development, to mention a couple. Through GIMM, students become prepared for life after college and learn how to jumpstart a career in technology. Do students wish to be generalists who can be flexible in a variety of positions, or do students aim to specialize in more focused subjects? Learn how academia is teaching the future of the tech industry. And how this knowledge influences individuals on ways to apply themselves post-college.

Introduction

In leading up to the final semester of college, countless questions rattle around in an individual's mind. The one that stands out the most is, "Where do I go from here?" Even if you choose not to attend college, this is still a pressing question when coming into the tech sector. The reason is that the initial competition is quite challenging. You're going against individuals who may be faster programmers or have more experience. With the overabundance of resources available via the Internet, learning how to program independently has grown more popular in recent years. So, this begs the question of whether a computer science degree or a similar field is worth it anymore. But no worries, these degrees are becoming far from obsolete. Instead, they are helping to teach the next generation the ways of technology in an innovative manner. In these new degrees, we see that students are provided with general information about technology that can be expanded upon if desired. This separates students into two roles, generalists and specialists. Generalists choose to know an array of subjects, allowing for more career flexibility. While specialists are individuals who hone their skills in a particular area, granting more career stability as roles are clearly defined. So, why are colleges teaching this mindset, and how can students decide which is best for their future? What are the advantages and disadvantages of being a generalist or a specialist? In this article, we will be breaking down an example of one of these new degrees, how students can market themselves to future employers, and what comes with being a generalist or a specialist.

What is Gimm?

As aforementioned, learning how to program independently has become more attainable than ever as individuals have access to a wealth of online resources that are highly informative. Examples of resources include Stack Overflow, Codecademy, and even Google. While this knowledge has become more accessible, it still requires motivation to learn. Without said motivation, individuals may face difficulties in learning as they can become quickly lost in the material. This is where college comes into play, as it provides students with a designated path in learning new subjects. Having a set curriculum can make it easier for new programmers to follow along and determine where to go next in their studies. Towards the end of the curriculum, students feel more confident in their abilities to teach themselves and strengthen their prior knowledge.

So where can students be taught these skills? One example would be an undergraduate program at Boise State University known as Games, Interactive Media, and Mobile[1] (GIMM). This degree helps prepare students for the competitive tech market. As a part of the College of Innovation and Design (COID) at Boise State University[2], we strive to be catalysts for innovation and foster learning via hands-on experimentation with technology. At first, when hearing the name GIMM, your immediate reaction may be that our primary focus is on video games. While games are a component of the degree, we also go beyond that and dive into emerging technologies. Examples of said technologies include augmented reality, virtual reality, the Internet of Things, and much more. In experimenting with these technologies, a general foundation of information is laid out and provided to

1 *Boise State University known as Games, Interactive Media, and Mobile: https://www.boisestate.edu/gimm/about/*

2 *College of Innovation and Design (COID) at Boise State University: https://www.boisestate.edu/cid/about/*

undergraduates. With this knowledge, undergraduates can expand upon it individually or stay focused on learning more subjects through the set curriculum.

When looking both in and outside the classroom, students are provided with experiences that result in portfolio artifacts that allow them to stand out from the crowd. The importance of gaining said artifacts is that it demonstrates the student's dedication and determination to a specific craft, such as art or programming. Without said demonstration of skills, employers may become skeptical of your abilities as an employee and may be reluctant to hire you. A primary goal of the GIMM program is for students to gain employment before or quickly after graduating. So let us learn more about this program by walking through my personal experiences and how it has helped prepare me from a career standpoint.

My College Experience

As a third-year student in the GIMM program, I have learned about numerous aspects of the tech industry and how my degree is unique in tech education. Throughout the curriculum, students are shown the essential roles art and programming have in emerging technology. This ideology is demonstrated through courses that address different topics, including full-stack development, AR/VR development, 3D animation, and more. Having a base foundation, I have expanded my knowledge in the areas I am passionate about and wish to pursue a career. To describe my college experience thoroughly, I will break down the components that make the GIMM program work and how GIMM can serve as a model for other institutions when preparing the future generation of tech employees.

Long-term Concepts

In analyzing the curriculum of GIMM[3], the first question that comes up is why is the educational scope so extensive and how does this benefit students? In each course, undergraduates are taught a general understanding of a topic that is expanded upon in future courses or by the student themselves. Gaining this allows students to establish foundational knowledge on multiple topics that can then be applied to one's skill sets. This is designed to facilitate learning of technology through a generalist mindset. This mindset allows students to gain experience and knowledge on a wide range of topics that are applied to different jobs in the tech market. In addition, this allows students to explore more of what they are interested in turning into a career. Even if students have particular preferences, such as being more proficient in programming, they will still have a general understanding of the art perspective in emerging tech. This is what makes GIMM unique as a degree. Third-year students in GIMM learn about the Internet of Things (IoT) while also practicing their abilities in 3D animation. Students will learn about both perspectives of technology throughout the curriculum, while other degrees focus on only one view. That is one of the main goals for GIMM, providing students with as much general knowledge as possible before joining the tech industry.

With all of the courses provided within this degree, one concept that is made very clear is that technology is always ever-changing. Something mainstream today could easily be replaced in the years to come. This statement isn't designed to scare students, but instead, it prepares them for change. To prevent a student's skill set from becoming obsolete, the curriculum not only thrives on teaching students tech but informing students on how to become quick and thorough troubleshooters. Talking with peers and professors is always an option, but independent problem-solving is highly advocated and can sometimes be where a student learns the most. One of the most valuable lessons expressed by current students and alums is the importance of understanding a problem and how to solve it. In addressing this lesson from a coding aspect, a developer can enter an error into Google and receive a snippet of code that can be molded to solve your issue. Google is never 100% accurate in providing coding solutions; instead, the programmer needs to modify said snippet to match. In my experience, there have been multiple instances in which I have received an error from a project, and my immediate reaction is to Google said error that appears in the console. In finding the solution, I will read through multiple documents on Stack Overflow or another documentation website. Once I feel confident in the solution, I will modify said snippets to my code and see if any errors come from the application. While the answer is mostly defined, the developer has to understand how to apply said code. Without this understanding, developers can become easily lost. This dilemma reinforces the importance of problem-solving and how to go about understanding both the issue and the solution, another key lesson that GIMM teaches. As indicated, this is taught in an assorted manner as students learn how to scour the Internet for documentation or keywords that replicate a coding error. Students who graduate from GIMM feel confident in their ability to understand errors and how to go about solving them efficiently. After learning different topics and skills through a set curriculum, how do students apply them post-graduation?

Application

One of the biggest struggles that college students face when finishing up school is where to go next, and one of those steps is figuring out how you want to apply yourself. This includes steps such as creating a resume, portfolio, etc. When looking at the tech industry, this is extremely important as you need to be prepared on how to sell yourself. When applying for jobs, you need to be able to demonstrate and discuss the skills you possess efficiently. One of the main appeals to this degree is that it encourages students to consistently apply themselves outside the classroom, whether in internships or partnering with fellow students

3 https://phpa.me/boisestate

to work on groundbreaking projects. When looking directly at GIMM, the program has multiple classes that teach students the importance of portfolio building and how to apply themselves post-graduation. This is demonstrated through courses that instruct students on making custom portfolio websites from scratch and utilizing network platforms such as LinkedIn. Partaking in these courses allows students to determine how they want to present themselves to employers based on the artifacts in their portfolios. Also, the benefit of these classes is that students are applying for positions and internships they are interested in pursuing. Once again highlighting the fact that this program prioritizes starting a career right away after college. In reflecting on these courses, the focus is to usher students into thinking about how to go through this process.

Community

Events within GIMM have proven to be a significant opportunity for students to push themselves and put what they have learned in the classroom to the test. The community with GIMM challenges individuals and fosters experiences that can apply to real-life work scenarios. As there is no grade attached to these events, students must be motivated to attend and compete. Without said motivation, students may miss the opportunity to challenge themselves and see how to apply the skills learned. In addition, these events also allow students to gain artifacts that can be applied to their portfolios. When looking at these activities, the GIMM Jam[4] stands out as it is an event held by fellow students. This event allows individuals to assemble their teams to make an interactive game using specific development engines in an allotted time. Some development engines used in the past include Unity, Unreal Engine, and CryENGINE. For this event, students communicate with others to determine how their team will operate. Each team member selects a specific role, such as a 3D artist or programmer. When designated a role, it becomes your focus on the project. Activities like these allow students to understand their specialties and what they are proficient in when surrounded by a team. This event reflects realistic work environments within the tech sector as individuals are assigned specific roles and responsibilities based on one's expertise. You not only support your team by choosing to work on specific components of the project, but you call attention to your specialties as a developer—highlighting to others where your passion lies, whether in programming or art.

Outside of working in a team setting, another great opportunity we see within this community is the communication that undergraduates have with alums and how they are willing to talk about their experiences in the tech sector after college. In GIMM's portfolio course, alums have previously spoken and interacted with current students through a question/answer format. These conversations have allowed individuals to hear about how skills learned within GIMM have been applied in interviews and current positions. Opportunities such as this provide insight for those beginning their careers. The takeaways can range from wanting to learn more about applying skills or how to market oneself to employers.

Generalist Vs. Specialist

As previously mentioned, we see that employees can be categorized into generalist and specialist categories. The current curriculum of GIMM allows students to learn various topics that can be expanded upon via personal projects and self-teaching. General knowledge is provided to individuals who then have to decide where to go next with this information. While this program does teach students a generalist mindset and how to be an individual who is capable of performing a variety of tasks, it does not outcast individuals who prefer to specialize in a more particular field. Instead, it advocates for it. While the curriculum hosts a variety of courses ranging from different topics, it presents students with something that can pique their interests. Doing this can open their minds to wanting to learn more about a particular field while still gaining a general understanding of other fields. That leads into our next section, which addresses the mindset of a generalist and a specialist within the tech industry through a student's perspective.

Generalist

The generalist's point of view advocates for individuals being a "Jack of all trades." This means an individual has a broad range of skills and knowledge on different topics. Employers are given someone versatile in different roles and responsibilities when hiring a generalist. Allowing individuals to have more flexibility in their career choices but limiting how qualified one may be on a specific topic. In having loosely defined responsibilities, a generalist risks career stability as employers may view them as employees with no set roles but instead work on the fly.

From personal experience, GIMM has provided students with general information on numerous topics, but the responsibility to expand said knowledge does fall back onto the student. Identifying where you are most proficient is essential as it can help you stand out from the competition and make you more appealing to employers. Being a generalist means you lose out on that expertise and are referred to as someone who dabbles in different areas. Generalizing your capabilities makes it difficult to separate yourself from others. But becoming a specialist can make you stand out from the competition in applying for a job.

Specialist

In dissecting specialists, it's clear that these individuals focus on demonstrating a specific set of skills instead of having general knowledge of different subjects. These skills can include individuals being more proficient in programming and web development while not being as competent from an art perspective. Being specifically skilled in an area makes you stand out as it demonstrates your expertise on

4 *GIMM Jam:* *https://itch.io/jam/gimm-jam-2023*

a subject, which employers are searching for in employees. Employers want individuals who are capable of handling tasks that are specific to a role. If you are applying for a job in back-end web development, the main focus is your capabilities as a developer in back-end programming languages. You want to personify these skills to employers as it will determine whether or not you get the job. Personifying these skills can look like an individual highlighting these capabilities in a resume, portfolio website, and LinkedIn profile. If you only have a general understanding of the qualifications, your competition may stand taller than you due to their level of expertise.

From experience, I am an individual who considers myself to be more on the artistic side of development which includes aspects such as 3D modeling and animation. I find myself to be more specialized in this field because I have a passion for it, as it is a subject I am always willing to learn more about. When comparing this to a generalist mindset, it may be difficult for individuals to identify what they are skilled in, as the focus is on having a wide range of skills. In not highlighting specific skills, competing with others when meeting particular job qualifications can be challenging. As a specialist, your skills are easily identifiable and make you stand out to employers, as those qualities are stressed in your application.

Overview

In my opinion, the specialized route is the front-runner for students getting ready to begin their careers. As mentioned above, being a generalist gives individuals more career flexibility regarding roles and responsibilities within a position but sacrifices job security as roles are loosely defined. When looking at specialists, skills can be easily defined, and an individual's expertise can be highlighted, which allows for better qualifications when pursuing a career. While specialists may lose out on the career flexibility that generalists have, specialists have stability in their careers as the roles and responsibilities are more clearly defined. Employers can see your worth as an employee and how you are more qualified than others in this field.

Conclusion

When looking at today's society, we see that a plethora of resources in learning programming and technology are readily available via the Internet. Knowing this fact can deter people from wanting to pursue an undergraduate degree in tech, but this should not steer individuals away. In undergraduate programs such as GIMM, we see that a curriculum can be structured to go beyond in-class activities and help students prepare for their careers in tech. Courses enlighten undergraduates about how technology is constantly evolving, how to address errors and much more. Students are also informed on how to structure resumes and are provided opportunities that build upon portfolios with artifacts. By implementing these learning practices, students can become more confident in their ability to apply themselves inside the tech sector, especially in how they want to display themselves. Once students near the completion of their degree, they will need to decide if they want to be more of a generalist or more of a specialist. By being a generalist, employees can demonstrate that they have a wide array of knowledge on different subjects, allowing for flexibility in one's career. But this mindset sacrifices career stability as roles and responsibilities are loosely defined, making employers question your overall value. When being a specialist, you are choosing to personify your expertise instead of displaying all of the knowledge you possess. In emphasizing these skills, it allows for better career stability as you are capable of demonstrating, to employers, your potential to handle specific responsibilities. Being a specialist, you also sacrifice your flexibility in your career. But in having these as your specialized skills, it is most likely that this is something an individual may enjoy over other elements. Overall, how you apply yourself to the tech sector post-college is determined by yourself and your skill set. Becoming either a generalist or a specialist has advantages and disadvantages. But programs such as GIMM can help students market themselves to employers while also pointing them in the right direction as to where to go next.

Adam Giles is currently a third-year student at Boise State University pursuing an undergraduate degree in Games, Interactive Media, and Mobile (GIMM). This program focuses on teaching students how to utilize modern technology. In studying GIMM, he has learned various programming techniques that can be reflected in both back and front-end development.

AI Is NOT Coming For Your Job

Chris Tankersley

As is tradition since the pandemic, the world has yet again changed. This time it is the explosion of AI or Artificial Intelligence. Programmers have been chasing AI since 1959[1] but the advances that the OpenAI[2] project has released to the general public have taken the world by storm. Programmers like you and I are now waiting for the day that AI takes our jobs.

Is that the reality, though? Why is it that AI has engrossed the world to the point that programmers, a knowledge-based profession, are now predicted to be completely replaced by what amounts to a chatbot? The reason is marketing.

Current AI products are Large Language Model-based applications or based on LLMs. Long story short, an LLM is an advanced text prediction model, like a souped-up version of the autofill feature of the keyboard on your phone. They are not "Artificial General Intelligence[3]" or systems that can replicate what a human does.

So why aren't we all going to be out of work in six months?

How Does an LLM Work?

Saying that an LLM is a text prediction system is a 100,000-foot explanation of what an LLM does, but it is not that far off the mark. Have you ever done one of those autofill games where you start a post with something like "Here lies [NAME]. [PRONOUN] was …" and you keep tapping the first autofill suggestion? Often, you will end up with a functional, if somewhat funny, sentence. In my case, I got:

"Here lies Chris. He was a great man and a great man who was a great leader and a great leader for the country."

Sure, it's a bit of a run-on sentence and repeats phrases, but the autofill did produce a sentence that is understandable. It does this by taking one or more words that have already been written and using a prediction algorithm to determine what it thinks the next most likely word is. A generic list of word suggestions is more than likely supplemented by common phrases you type into your device.

So given a short phrase like "He was," the prediction engine guesses that the word "a" is a likely candidate. The prediction engine returns a series of words rated on their probability of showing up after the word "was" or even the phrase "He was." When you select an autofill suggestion, the process starts all over.

If something like ChatGPT works similarly, why does ChatGPT give such human-sounding responses and autofill repeats "a great" many times? How well an LLM works is based on a few factors:

- Token size
- Amount of training data
- Raw horsepower of the device

Your mobile phone's autofill system was trained on a small amount of data. The model may be based on usage from other users, your own input, and available content like out-of-copyright books. The database that contains all this information must stay small so that it can be searched by low-powered (compared to massive clusters, anyway) devices in an almost instant amount of time.

To help cut down on the need for horsepower, autofill works on a limited number of "tokens," or the amount of existing data. You can think of a token as analogous to a word. A sentence like "I like turtles" has three tokens, and "The quick brown fox jumps over the lazy dog" has nine tokens. The more tokens a suggestion engine uses, the more "accurate" the suggestion will be.

Let's assume that the token size of autofill is four words. When we stopped typing, autofill gave us a suggestion based on "lies Chris. He was" and the probability matrix came up with "a." When we added "a", the token string became "Chris. He was a" and suggested that "great" was the next probable option. "He was a great" was the next considered string, and it suggested "man."

This is why we get sentences that can be a bit odd, as the token size is going to be small. The token size needs to be small as larger tokens need much more processing power and memory usage. Users would never wait for autofill to take ten seconds to offer a suggestion. No one wants their phone battery draining because a more accurate autofill suggestion would be helpful.

ChatGPT uses two different models. GPT3.5 has a token window of between 2,049 and 4,096 tokens, and GPT4 has a token window of 8,192 and 32,768 tokens[4]. These models can consider huge amounts of text to determine the next token, making them much more accurate than your phone keyboard's autofill system. These models have been trained on a massive amount of data.

1 *been chasing AI since 1959: https://phpa.me/aifils*

2 *OpenAI: https://openai.com*

3 *Artificial General Intelligence: https://phpa.me/ai_general_intel*

4 *8,192 and 32,768 tokens: https://en.wikipedia.org/wiki/GPT-4*

GPT3 had access to nearly 500 billion tokens[5] and various methods of training and reinforcement.

It is also estimated that to train this amount of data, it took $4.6 million. It is safe to say that the same quality and power are not linked to your keyboard autofill.

LLM AI Has No Intelligence

LLM-based artificial intelligence, especially ones like GPT3 and GPT4, are good at sounding intelligent, but as anyone knows, sounding intelligent and being intelligent are two different things. Have you ever talked to someone who has professed to have an interest in something you have a deep knowledge of, and it becomes apparent the other person is just stringing words together? That's most LLM-based AIs.

These types of systems are running in "Fake it until you make it" mode until they can learn from their experiences. Many of the LLMs that can be found online like, huggingface.co[6], fall into this category. They are raw large language models that can be good at stringing words together. GPT3 and higher do have some additional functionality that allows it to try and be better at some tasks, but their heart is still an LLM.

The intelligence displayed comes from GPT's ability to use huge token windows. The larger the token window, the more accurate the prediction of the next token is. GPT might also be chunking up individual words or combining common phrases into other tokens, increasing its ability to make more accurate predictions. There are also humans that help train the model by rating responses and coming up with manual responses for GPT to learn from, so it is not all "determine the next word."

If I was to estimate it roughly, I have written around 72 articles for php[architect] over my time working with the magazine (one year of "The Dev Lead Trenches", five and a half years of "Education Station", and a smattering of articles before my columns). At around 2,500 words an article, that is a rough 180,000 words. That is roughly five sets of token windows for GPT4. A single token window is roughly an entire year of article text.

If I ask GPT to write an article, it can fit ten months of back article text to reference into a single window. Given a normal English training set for probability and a single full token window of content, there is a good chance that it will generate coherent-sounding text that sounds correct under light scrutiny.

The Hallucination Problem

The issue is that the text is not created based on knowledge but on a series of weighted words compared to other, preceding words. Take the following prompt I put into ChatGPT using GPT4 and the response:

```
Prompt: "Please give me a list of all the articles
written by Chris Tankersley for php[architect]in 2020."

Response:
I am unable to provide real-time or post-2021
information as my knowledge was last updated in
September 2021. However, I can provide a list of
articles written by Chris Tankersley for php[architect]
up until 2021. Please keep in mind that the list may
not include any articles from 2020
or may be incomplete:

1. "Docker for PHP Developers" (February 2018)
2. "Debugging PHP With Xdebug" (April 2018)
3. "Using the Symfony Workflow Component" (Sept 2018)
4. "Understanding Middleware" (January 2019)
5. "Local Development Environments" (May 2019)

Please visit the php[architect] website or
contact them directly for the most up-to-date
information on articles written
by Chris Tankersley in 2020.
```

Without any further research, all of that sounds great! The problem is none of those articles are real. I did write articles in 2019, but all of them were part of this column, and "Understanding Middleware" nor "Local Development Environments" are real titles, let alone were published in January or May. 2018 was devoted to "The Dev Lead Trenches", which was not a technical column per se, but dealt with many issues team leads face. None of those 2018 titles are even close to the topics I wrote about in 2018.

One of the key selling points of ChatGPT and GPT4, especially, is the ability to converse with the "AI." I could tell it that none of those articles exist, and you know what it does? It generates a whole new list of non-existent titles. In some of my testing, it would even give me back some of the same article titles I told it was not real. There is no intelligence in the response, just a good prediction model.

When an AI responds like this, it is called a Hallucination. In reality, it is a lie. It is a very convincing lie as I have talked about these topics before, so GPT has basic knowledge about me, but the fact of the matter is, it is a lie. This is not unique to me. The Guardian, an independent news organization in London, England, has been battling fake articles and bylines[7] due to these "hallucinations."

GPT has been caught making up research papers[8] as well. These lies already sound convincing on their own, and when the AI backs up these lies with sources like The Guardian or articles published in well-known research portals, it is easy to take the responses at face value. As more people rely on AI based on LLMs and people use ChatGPT for answers to

5 *GPT3 had access to nearly 500 billion tokens:* https://phpa.me/gpt-3

6 *huggingface.co:* https://huggingface.co/

7 *has been battling fake articles and bylines:* https://phpa.me/fake-article

8 *has been caught making up research papers:* https://news.ycombinator.com/item?id=33841672

questions, disinformation begins to spread. These “hallucinations” start to get regurgitated like bad conspiracy theories, increasing their volume and then fed back into later training sets, reinforcing the bad information.

Artificial General Intelligence is NOT Here

The last few months have been drowning in new tools called AI Agents, which promise the golden age promised by the idea of Artificial General Intelligence. Artificial General Intelligence is the idea that an AI bot, or agent, can replicate an intellectual task like a human. LLMs lack intelligence due to their nature of being good text prediction models, much more than solving problems with actual intelligence.

What about things like AutoGPT[9], which purport to be agents that can utilize the amazing power of GPT to work toward a solution autonomously ? These projects claim to do things like research topics for you, summarize the information, and turn it into a blog post. Throw that into another agent that can then turn those blog posts into a site and another that will generate tweets on new posts. Now you have the makings for a blog content farm!

The main issue is that they do not work most of the time. I have been playing with AutoGPT and other tools for a few weeks now, and the only thing they have done is hit my OpenAI billing limit without generating anything useful.

I installed AutoGPT into a VM and gave it a simple task. Find the top five most popular PHP frameworks. Then summarize the main description of each framework. Finally, give me a list of pros and cons for each as a text document.

AutoGPT fired off a browser search. It found an article listing PHP frameworks, including Laravel, Symfony, Zend, Laminas, and CodeIgniter. For the next hour, it attempted to research Laravel by doing a Google search for “Why is Laravel popular?” and summarizing the same first search result. It claimed that the first article had no pros and cons for Laravel, tried “Top 5 PHP frameworks again,” got the same article listing the frameworks again, and got into a loop. No output was ever actually generated, and it never searched for anything other than Laravel.

I am sure that the next six to twelve months will see these tools mature; however, they are nowhere near being useful to the general public currently.

Legal Issues and “open” Models

Last but not least, there is the legal issue that many of these models may face. With a name like “OpenAI,” one would think that these language models are open-sourced in some way, correct? Open-source software allows us to fix things when we discover problems, so what can we do about this?

Nothing. Almost all of these systems are closed-source models, and systems like GPT do not publish the resulting models at all.

As LLMs grow in their training data sets, there becomes a question of where they get their data? Copilot, the coding AI developed by Microsoft, was accused in 2022 of training and returning copyrighted code[10]. Copilot was almost certainly trained on copyleft licensed code[11]. During the early beta periods of Copilot, I could get it to return code verbatim from Vonage’s code snippet repositories. I do not mean it gave me a similar block of code—it would give me the exact same code we published. Our code is Apache 2.0 licensed, and anyone using that “generated” code and not following the Apache 2.0 license is breaking the license.

Many companies refuse to allow GPL and other copyleft-licensed code into their codebases. What happens when Copilot returns the copyleft code and does not identify that the code might be derivative of the copyleft code? We do not know. Copilot today has some additional safeguards, but some are locked behind a business plan. That is on top of the base issue that sometimes the code it produces does not work.

This copyright issue is not unique to Copilot but is a basic uncertainty of OpenAI (which Copilot uses). Where exactly did OpenAI get all of its training data? We simply do not know. A huge chunk of it was from Common Crawl[12], a non-profit that crawls and archives the internet. This archive, petabytes in size, is available through various means and contains copyrighted material.

How is this legal? According to Forbes[13], Common Crawl can publish this data because its lawyers consider it fair use. This is akin to the book scanning that Google did years ago for Google Books. Common Crawl respects things like `noindex` and `robots.txt` to reduce what they crawl. At the end of the day, though, the argument is that the information they crawl is available to read on the web, and thus their usage of it is fair use.

Many other language models, like Vicuna[14], also propose to be open but do not share their exact training set.

At the moment, “Open” in the AI world means that the models can be used, not that they are open source. In my opinion, it feels like these models are getting a pass because groups like OpenAI have enough money and

9 *https://autogpt.net*

10 *https://phpa.me/copilot-licensed*

11 *https://phpa.me/legal-copilot*

12 *https://en.wikipedia.org/wiki/Common_Crawl*

13 *https://phpa.me/common-crawl*

14 *https://vicuna.lmsys.org*

lawyers to stop people from asking questions more than these systems being open.

Where AI Helps

AI is turning into a helpful and potentially world-altering tool, despite my previous two-thousand-word diatribe against it. It is important to understand *what* these AI tools are, and when you understand how they work, you can utilize them better.

AI tools like ChatGPT can be great for rubber ducking as long as you follow up any “facts” it gives you. There have been plenty of times I have asked it questions and the answers were good enough to break me out of an analysis paralysis situation or to provide a different side to a problem. I always follow up on its suggestions with a quick Google search, but the fact it sparks a new idea is great.

When it comes to code it is pretty spot on dealing with boilerplate things. Want to transform a huge list of items, like a list of countries, into an enum in a language? “Convert the following list into a PHP enum with uppercase, snake case keys, and lowercase values”, and it will generate the enum. Ask it to create a Symfony firewall and it falls flat on its face, but for boilerplate things, ChatGPT works well.

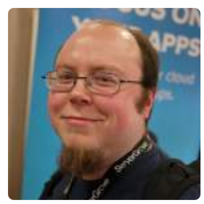
I also use it to help summarize long blocks of text or create outlines of things. I will also ask it to help suggest outlines or crucial information for business tasks. “Generate an outline for an incident response triage document” is generic enough that the AI cannot hallucinate, and since it is not asking for concrete facts there is almost nothing it will mess up.

If you realize that all ChatGPT is doing is stringing words together, and that you cannot trust any “fact” it returns without research, these tools can become good tools to help speed up mundane work.

AI Will Not Replace You

There are jokes about programmers becoming prompt engineers, but how different is that from most of our days being good at searching Google? Current AI lacks the intelligence part of its name, which is what will continue to make programmers, system architects, designers, and the like useful. Our brains can learn from mistakes and successes, and AI cannot match the years of experience we gain.

AI may weed out developers that skirt by on raw output or move from simple project to simple project. Still, no modern AI replaces a programmer who is good at problem-solving and can leverage a big squishy brain with a real ability to learn and adapt.



Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with companies and developers for integrating containers into their workflows. @dragonmantank

Tabletop: Planning for Disaster

Eric Mann

Roughly twice a year, I take time to play a game with my team. To those who play Dungeons & Dragons, this might sound familiar. I spend time planning a particular campaign, then each team member picks a role and plays through it. Except we're not fighting monsters or casting spells. Instead, I take the role of Dungeon Master for a simulated cybersecurity incident. To those in the industry, this is commonly referred to as a tabletop exercise.

Tabletop Objectives

The overall goal of a tabletop exercise isn't to test your team. It's not an opportunity to plan with trick questions or mysterious challenges. The actual objective is to test the incident response process your team leverages to tackle real issues on the job.

Your responsibility is to provide enough information so the team can talk through how they would respond in a real-world incident. They'll walk through incident response, cross-department communication, and mitigation planning as if a breach existed.

It's also important that the exercise include representatives from the stakeholder groups with whom you work. Your legal department, finance team, infrastructure managers, and even executive staff should all be invited. Everyone who would respond to a real incident at your company should have a seat at the table so they can exercise their response procedures outside of the stress of an actual attack.

At the end of the simulation, your team should have a clear set of action items for any further policy or process refinement needed among the team. Everyone involved will also have a better understanding of their own role should an incident arise.

Laying Out the Scenario

Any quality simulation should pull from actual events your company and team have faced. Since you've handled those events already and put new policies in place, the tabletop event is a great opportunity to test the efficacy of those policies. Take care to sanitize any participant names when leveraging previous events. The goal of the tabletop is to test your response procedures, not to perform a retrospective on past incidents.

A tabletop exercise is also an opportunity to walk your stakeholders through a more comprehensive threat modeling workshop—instead of having them brainstorm everything that could go wrong with an application or system, write up a fictional scenario detailing what events would've happened anyway.

Perhaps an infrastructure engineer mistakenly downloaded a malicious document to their machine and leaked their SSH keys to an attacker. Or a customer service representative installed a keylogger into their browser while playing Candy Crush. You may want to simulate an executive who loaned their work phone to their kids on vacation and subsequently installed malware on the device.

Whatever the scenario you choose, make sure you:

- Clearly identify the what, who, and when of the event - makeup dates and names but otherwise make the simulation believable.
- Since you know what the team will be doing when they respond to the event, plan for further information you can provide during the simulation. If you know a first action would be to audit network access logs, plan to provide information about what they would've found during that investigation.
- At times your team will surprise you by investigating something you *haven't* planned for. In those situations, feel free to make something up that's otherwise consistent with he scenario so they can keep moving forward.

Running the Tabletop

While the team is running the simulation, take a back seat and let them lead. You're not an active participant—you're the source of information they can use to diagnose what's gone wrong. Deputize a member of your team to act as Incident Commander—the individual responsible for delegating investigation, controlling communications, and otherwise running the show until the incident is resolved.

As the team moves through their faux investigation, offer them details in response to their actions. But resist the urge to offer guidance on what they should investigate. If at any point you think they've missed a detail, take notes and follow up later. In other words, run the simulation as if you're on vacation on a deserted island and inaccessible to the team—they can't ask for your help.

That said, if the team veers too far down a rabbit hole, please step in and guide them back to the task at hand. When starting this process, the team might rely heavily on you for guidance; it will take time to feel comfortable running the incident response playbook in your absence.

Plan, Simulate, Polish, and Repeat

Once the fictional incident is resolved, give everyone a break to breathe. Then immediately step into the retrospective phase.

- What went well? Where did the process adequately help to resolve the incident? What insights did the team offer that will be useful in the future?
- What went wrong? Where did process get in the way of progress? Where were gaps found that need to be resolved down the line?
- What needs to change? If the process had failings, how can it be improved? What other processes or policies can the team implement to improve incident response and prevent this kind of incident in the future?

The advantage of a tabletop exercise is that you can test your response plan while the team is calm and collected. No servers are down, no data has been leaked, and no barbarians have breached the gates. You can ensure both that your response plan will handle potential incidents in the future and that your team is well enough versed in the plan to run through things smoothly when it counts.

If you need to change the plan—or implement additional controls within the company—take the time to do so. This added polish of your incident response procedures will come in handy later and help the team be more effective. Just make sure you repeat this entire workflow regularly.

Plan a *new* scenario in 3-6 months and start over from scratch. Test a different attack vector. Read through an incident report from elsewhere in the industry and have your team treat it like an attack on your own company. Repetition will help strengthen both the response plan and your team's ability to use it.

Related Reading

- *Security Corner: The Risks of Free Conference Internet* by Eric Mann, April 2023. https://phpa.me/security-apr-2023
- *Security Corner: InfoSec 102: Phishing* by Eric Mann, March 2023. https://phpa.me/security-mar-2023
- *Security Corner: Infosec 101: The Confused Deputy* by Eric Mann, February 2023. https://phpa.me/security-feb-2023

Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: @EricMann

The Workshop: Minicli

Joe Ferguson

Should you use Minicli? It's definitely worth a look. Let's review what it has to offer.

Building console applications with PHP is nothing new; in fact, we'd often reach for Symfony's Console Component[1], if not Laravel's Commands[2]. This month we're going to explore Minicli[3], which self-proclaims to be a "minimalist framework for building CLI-centric PHP applications". The minimalist is quite apparent as you need a PHP version 8.0 or higher and the `ext-readline` extension to read user input. Could we really build useful console applications with this one framework?

Starting a fresh application using PHP 8.2, we'll run the following to create our demo project: (See Listing 1)

Listing 1.

```
$ mkdir demo; cd demo
$ touch example
$ chmod +x example
$ composer require minicli/minicli
Info from https://repo.packagist.org: #StandWithUKR
./composer.json has been created
Running composer update minicli/minicli
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking minicli/minicli (4.0.0)
Writing lock file
Installing dependencies from lock file (including dev)
Package operations: 1 install, 0 updates, 0 removals
  - Downloading minicli/minicli (4.0.0)
  - Installing minicli/minicli (4.0.0): Extracting ...
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found
Using version ^4.0 for minicli/minicli
```

We'll create our `demo` folder and install Minicli with `composer require minicli/minicli`. We also create a file named `example` with the `touch` command to create the empty file and we set it executable on our system via `chmod +x example` so that we can run the file via `./example`.

Open our empty `example` file in your code editor of choice and paste in the following: (See Listing 2)

Listing 2.

```
#!/opt/homebrew/bin/php
<?php
if(php_sapi_name() !== 'cli') { exit;}
require __DIR__ . '/vendor/autoload.php';
use Minicli\App;

$app = new App([
    'app_path' => [__DIR__ . '/app/Command'],
    'theme' => '\\Unicorn', 'debug' => false,
]);

$app->registerCommand('demo', function () use ($app) {
    $app->getPrinter()
        ->success('Hello php[architect] :D' , false);
    $app->getPrinter()
        ->info('Info message with background' , true);
    $app->getPrinter()
        ->error('Error Message :(', false);
});

$app->runCommand($argv);
```

You'll notice the first line of our file is the typical shebang `#!` but it's pointing to `/opt/homebrew/bin/php`, which the macOS users will quickly identify as the Homebrew symbolic link for `php` which is currently referencing PHP 8.2.5. You should change this line to point to where PHP may be installed on *your* system. It might be something like `/usr/bin/php` or `/usr/local/bin/php`, To locate where PHP is currently on your system, you could use `which php` should report the path, if not you may not have installed PHP correctly or it may not be linked in your environment. This shebang line allows us to inform the shell when executing that the commands should be sent to the PHP binary in our path. Alternatively, we could name the file `example.php`, but you would still want the shebang line pointing to PHP for the best experience. Our application continues by checking to ensure we're running PHP from the command line, then setting up auto-loading and importing the Minicli classes and creating a new app configured to use a Unicorn theme and debug set to false. The next block of code creates a `demo` command that prints three messages via `$app->getPrinter()`. The `success` and `info` messages are sent to standard out while `error` is sent to standard error, as you would expect from a console application. We can run our command in the terminal via `./example demo` to see the output. (See Figure 1 on the next page)

When running our example demo command, we can see the three messages being output to the user: "Hello php[architect] :D", "Info message with background", and "Error

1 Component: *https://phpa.me/sym-console-component*

2 Commands: *https://phpa.me/write-commands*

3 Minicli: *https://minicli.dev*

Figure 1.

Message :(". While our example demo command demonstrates the absolute basics, we can see that this isn't a scalable solution. You might have caught the `'app_path' => [__DIR__ . '/app/Command'],` line in our `example` script. This hints at the better way to structure our Minicli-based application. We'll create our path with `mkdir -p app/Command/Demo` and `touch app/Command/Demo/DefaultController.php`. We need to add the following block to our `composer.json` to tell Composer to look in our `app` folder and load any classes there according to PSR-4[4].

```
"autoload": {
    "psr-4": {
        "App\\\\": "app/"
    }
}
```

Since we have introduced our PSR-4 folder, we need to run `composer dump-auto` to regenerate our updated class maps to discover our `App\\Command\\Demo\\DefaultController.php` class. The contents of `DefaultController` are: (See Listing 3)

Listing 3.

```php
<?php
namespace App\Command\Demo;
use Minicli\Command\CommandController;
class DefaultController extends CommandController
{
  public function handle(): void
  {
    $this->getPrinter()
        ->success('Hello php[architect] :D' , false);
    $this->getPrinter()
        ->info('Info message with background' , true);
    $this->getPrinter()
        ->error('Error Message :(', false);
  }
}
```

We've moved our code to output text from our `example` command to our controller. Now we need to update our `example` code to run our commands if we've created them; otherwise, return an error. This use case is perfect for `try` / `catch` blocks: (See Listing 4)

Listing 4.

```php
#!/usr/bin/php
<?php
if(php_sapi_name() !== 'cli') { exit;}
require __DIR__ . '/vendor/autoload.php';
use Minicli\App;
use Minicli\Exception\CommandNotFoundException;

$app = new App([
    'app_path' => [__DIR__ . '/app/Command'],
    'theme' => '\\Unicorn', 'debug' => false,
]);

$app = new App([
    'app_path' => [
        __DIR__ . '/app/Command'
    ],
    'theme' => '\\Unicorn',
    'debug' => true
]);

try {
    $app->runCommand($argv);
} catch (CommandNotFoundException $exception) {
    $app->getPrinter()->error("Command Not Found.");
    return 1;
} catch (Exception $exception) {
    if ($app->config->debug) {
        $printer = $app->getPrinter();
        $printer->error("An error occurred:");
        $printer->error($exception->getMessage());
    }
    return 1;
}

return 0;
```

With our `DefaultController.php` updated, the output of our command `./example demo` is the same three lines we saw before. To recap what we've accomplished, we've moved to a structured file layout for our Command Controllers[5], where our commands will be auto-loaded and run following the conventions of Minicli. It is also important to note that if we do catch an exception, we `return 1` instead of `return 0` to signal to the terminal that an exception or error has occurred. When a program exits normally, the exit code is 0, when an error happens, it should exit with code 1. For a deeper dive

4 PSR-4: https://www.php-fig.org/psr/psr-4/

5 https://phpa.me/command-controller

into exit codes, see https://tldp.org/LDP/abs/html/exitcodes.html[6].

Minicli also supports sub-commands which is why we named our controller `DefaultController.php`. Since we didn't specify a sub-command, it is assumed that we want `default`. This means running our example demo command via `./example demo` and `./example demo default` will show the same output. To create a sub-command to accompany our default controller by creating `app/Command/Demo/TpsController.php` with the following contents: (See Listing 5)

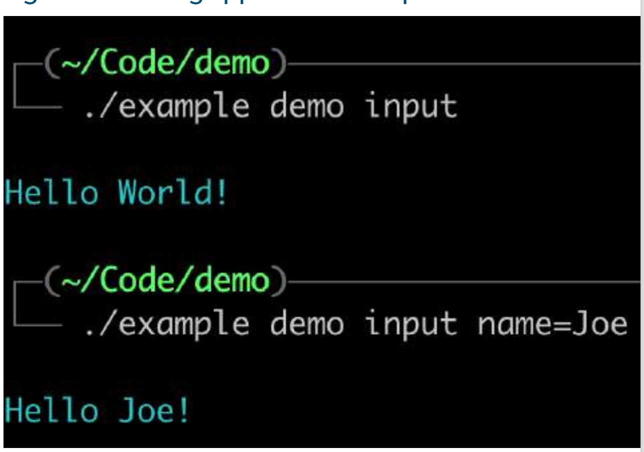
Listing 5.

```php
<?php
namespace App\Command\Demo;
use Minicli\Command\CommandController;
class TpsController extends CommandController
{
    public function handle(): void
    {

    }
}
```

While displaying pretty colors in the output is nice, it isn't often the primary task when building console applications. Our console applications will need to handle input and support flags that can change the behavior of a given command, such as a test mode flag where any write operations would be ignored if the test flag was used. We can pass parameters and flags to our console application. Parameters would be input, such as a name, email, or company where a flag would be used to change the application's behavior. If you do not pass any parameters, we can see a basic parameter check in the Minicli docs for hello world. If you pass a `name` parameter, it will return the name in the string:

Figure 2. Running app with a name param and without



6 https://tldp.org/LDP/abs/html/exitcodes.html

```php
# app/Command/Demo/TpsController.php handle()
$name = "World";
if ($this->hasParam('name')) {
    $name = $this->getParam('name');
}
$this->getPrinter()->display("Hello $name!");
```

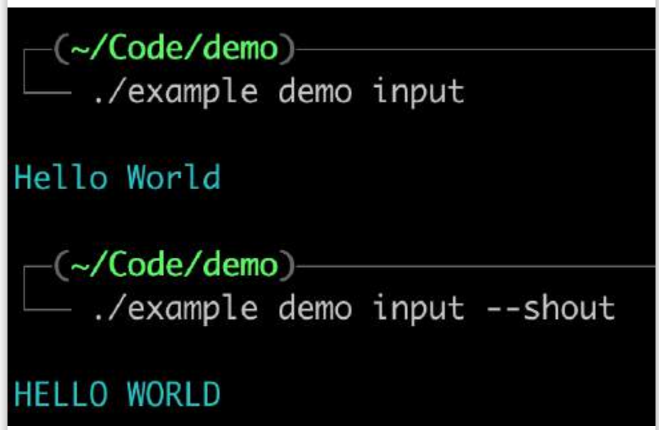
We are able to check for flags in a very similar way by using `hasFlag` to check if a flag has been passed to our command:

```php
# app/Command/Demo/TpsController.php handle()
$message = "Hello World";
if ($this->hasFlag('shout')) {
    $message = strtoupper($message);
}

$this->getPrinter()->display($message);
```

If we run our command without any flags, we get "Hello World", and we can use `--shout` to have our application pass the string to `strtoupper()`, resulting in the shouting `HELLO WORLD` output. (See Figure 3)

Figure 3. example using `--shout` flag in our command.



If you're looking at all of the boilerplate `hasParam()` calls and want a better alternative, look no further than `$this->getArgs()`, which will output everything to the command in an array in the order they were supplied. (See Figure 4 on the next page)

Using `getArgs()` allows you to be a bit more fast and loose if you *know* you're going to have the same fields every time and don't want to bother with the overhead of checking or in the case where a script fails due to a bad input isn't the end of the world. We could write our command to get all of the arguments and then print out the order in which we need them:

```php
# app/Command/Demo/TpsController.php handle()
$args = $this->getArgs();
$this->getPrinter()
    ->display("REPORT {$args[2]}");
$this->getPrinter()
    ->display("Company: {$args[3]}  Mgr: {$args[4]}");
$this->getPrinter()
    ->display("Employee: {$args[5]}");
```

Figure 4.

```
╭─(~/Code/demo)
╰── ./example demo tps
array(3) {
  [0]=>
  string(9) "./example"
  [1]=>
  string(4) "demo"
  [2]=>
  string(3) "tps"
}

╭─(~/Code/demo)
╰── ./example demo tps foo bar bazz
array(6) {
  [0]=>
  string(9) "./example"
  [1]=>
  string(4) "demo"
  [2]=>
  string(3) "tps"
  [3]=>
  string(3) "foo"
  [4]=>
  string(3) "bar"
  [5]=>
  string(4) "bazz"
}
```

Running our application and carefully placing our values so that our report comes out as intended. (See Figure 5)

Instead of having several references to `$args[]`, we can instead get our parameters by name and update our command to pass them by name. (See Listing 6)

We still need to get our Report title from `$args[2]` since we're using the sub command name for the reported value. The other three values which make up our report can be accessed via `getParam()`. This allows us to improve the readability of our code, which helps convey the context and content we're attempting to build with our command. We need to update our command to use our named parameters when we execute our command. Using named parameters also negates the need to keep them in any particular order since we now access them by named value instead of index position in the array. (See Figure 6)

Figure 5.

```
╭─(~/Code/demo)
╰── ./example demo tps foo bar bazz

REPORT tps

Company: foo  Manager: bar

Employee: bazz
```

Figure 6.

```
╭─(~/Code/demo)
╰── ./example demo tps company=foo manager=bar employee=bazz

REPORT tps

Company: foo  Manager: bar

Employee: bazz
```

Listing 6.

```
# app/Command/Demo/TpsController.php handle()
$args = $this->getArgs();
$company = $this->getParam('company');
$manager = $this->getParam('manager');
$employee = $this->getParam('employee');

$this->getPrinter()
    ->display("REPORT {$args[2]}");
$this->getPrinter()
    ->display("Company: {$company}  Mgr: {$manager}");
$this->getPrinter()
    ->display("Employee: {$employee}");
```

Listing 7.

```
$test = false;
if ($this->hasFlag('test')) {
    $this->getPrinter()
        ->display("Operating in test mode!");
    $test = true;
}
// Process data and need to update a DB
if (!$test) { # if we're not using test flag
    // update database!
}
```

The last thing I want to show you when building command line applications is to use a test flag if the application is doing any write operations, such as creating files or database entries—or even updating a database entry. This allows us to provide a safety net of sorts to allow users to run our command without fear of any changes being saved. Coupled with proper risk management, this is a solid solution to accidental command running in production. What I build into most commands is similar to the below: (See Listing 7 on the previous page)

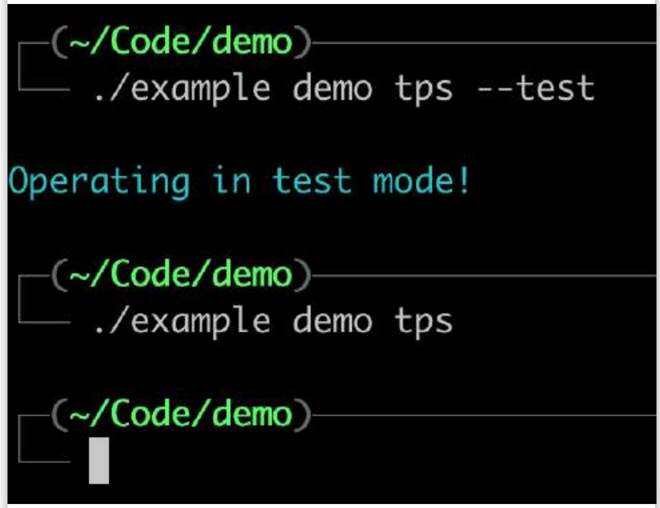
Running our command with and without our test flag: (See Figure 7)

Test mode flags can save you a lot of headaches, especially if you're in an environment where not all users running console commands *should* be running them. This is a way to add some guardrails to your applications.

Should you use Minicli? You should certainly check it out, if nothing else. It's a nice alternative to Symfony Console, and it feels less wasteful than spinning up a full Laravel stack for a single Console command application. Minicli is good at what it does, and it gets out of your way. I'll definitely be using it in the future.

Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. [@JoePFerguson]

Figure 7.



Maze Rats, Part Two

Oscar Merida

Last month, we solved how to represent a maze by storing the configuration of walls for each grid square within. Now, we've been tasked with generating a solvable maze at random.

Recap

> *Archlin studies your notes on how you propose to record mazes in the PHP-o-nomicon. "This might do after all, apprentice," he mutters as one of his great ears twitches excitedly. "This might keep the treklins busy and out of trouble. Come, we have little time left for the next step."*
>
> *He strides out of his study in the great tower of Sharadun towards the Great Courtyard. He points his trunk to the ground below. "I will cast a spell to build a maze below, and each hour, the maze will change. You will design the walls of each maze on the fly."*
>
> *The maze will be a square grid. Each square in the grid has four walls (north, south, east, and west), and each wall can either be solid or an opening. There should be a single entrance in the north edge, a single exit in the south edge, and no dead ends in the maze.*

Algorithms and Perfect Mazes

Working with mazes programs has a long history that we touched on last month. There's no reason for us to throw all that work away and try to write a wholly new algorithm from scratch. In fact, that's likely a waste of time. We'll run into the same problems others have solved. Let's build on the work already done on this topic—only we'll do it in PHP this time.

It shouldn't surprise you to discover that there are also many algorithms for generating them. When researching a solution, I found Jamis Buck had summarized many of them at Maze Generation: Algorithm Recap[1]. Some of these algorithms include:

1. The **recursive backtracker**, which generates long and winding passages. You need to have enough memory to store the entire maze state.
2. **Eller's algorithm**,** **which can generate huge mazes in linear time by cleverly only working with one row at a time.
3. **Wilson's algorithm**, which is one of many that uses uniform spanning trees to generate mazes.

> *A perfect maze is one where all the cells are connected to one another and there are no cyclic paths. Most (but not all) algorithms you may find in your research are for generating perfect mazes.*

After reviewing his pseudocode and looking at the demonstration animations of each maze-generating technique in action, I chose the recursive backtracker. I kind of understood what it was doing under the hood—enough to start, at least. You might choose to implement a different one or more than one.

Rendering First

Before figuring out how to generate the maze, I wanted to better present my mazes by saving them as PNG images. Last month's ASCII maze isn't going to cut it. We can use the same 2-dimensional array of integers to represent our maze. From there, it was straightforward to use the GD library (since it's bundled with PHP) to loop through the array and draw each cell. I again used some bitwise operators to test which walls I had to draw for each cell. To add a little flair, I then drew the walls offset in a different color. Listing 1 is the Rendering class. I've begun to appreciate named arguments for very long lists of function parameters.

With the code below, we get an image like Figure 1.

```
$maze = new \OMerida\Maze\Renderer(5, 5, 50);
$maze->setCells([
    [0x9, 0xC, 0xA, 0xD, 0xA],
    [0x7, 0x9, 0x0, 0xA, 0x3],
    [0xB, 0x7, 0x7, 0x1, 0x2],
    [0x5, 0xC, 0xA, 0x3, 0x7],
    [0xD, 0xC, 0x4, 0x6, 0xF]
]);
$maze->draw();
$maze->save(__DIR__ . '/maze.png');
```

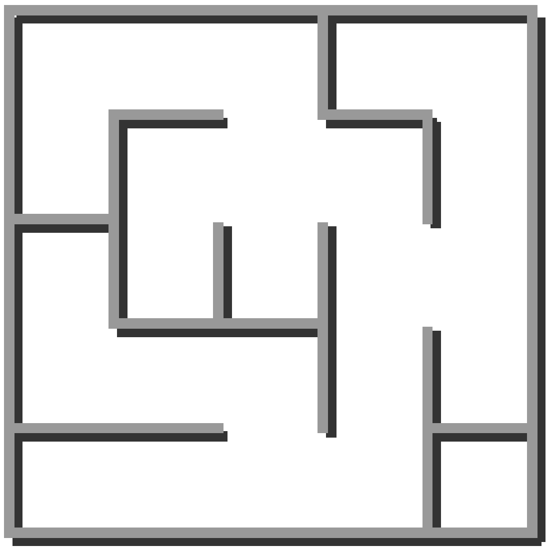
A Messy Procedural First-pass

Given the sample code in Buck's blog post about the recursive backtracker[2], I started writing code. I knew I had to pick a starting cell, possibly keep track of what cell I was in, and have a way to "breakthrough" one of the cell walls at random.

1 Maze Generation: Algorithm Recap: https://phpa.me/weblog

2 recursive backtracker: https://phpa.me/weblog-recursive

Figure 1.



Refer to last month's article to refresh your memory about how we represent the state of a maze cell's walls using integer values.

After some trial and error, I had a working procedural solution, as shown at the end of this article in Listing 3. First, I initialized all the cells to have all four walls for any given maze by setting them to `15`. Any cell with all four walls is one we haven't visited while generating the maze.

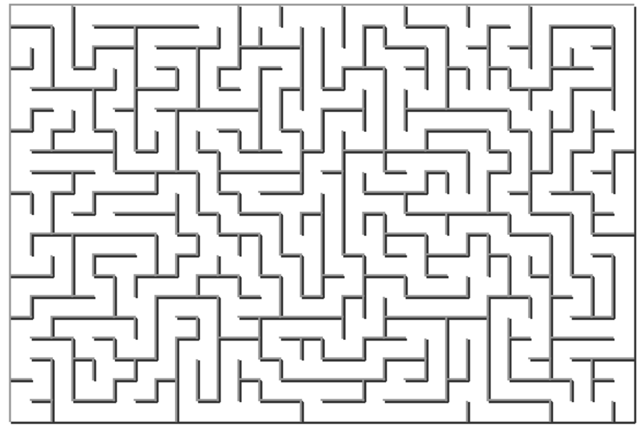
I had a function for choosing which wall to "bust through." After building an array of the available walls in a cell, I use `shuffle()`[3] to randomize the order and return the array. Given that set, I'd check each one to see if I could break the wall as long as it is not on the edge of the map and the cell on the other side was "unvisited." If we don't shuffle the walls, our maze will always carve paths in the same direction. Try returning an unshuffled array to see the output.

For a while, I was stuck on how to keep track of which cell I was inspecting, backtrack if I was at a dead end, and continue building the maze. After re-reading how the algorithm was supposed to work, a light bulb went off. I understood why it was recursive. After shuffling the walls, we will try to break through all of them, but only if it's allowed. Each break only happens after the previous one has meandered into a dead-end. If it does, there's nothing left to do, and the "current cell" rewinds to the starting cell and looks for other walls to bust.

The procedural approach worked! I saved a 20x30 maze like the one shown in Figure 2. But we're not done yet.

3 `shuffle()`: *https://php.net*

Figure 2.



Generating with Class

The procedural approach was helpful for understanding how the algorithm worked, but it's not terribly reusable. There are constants and functions in the global scope and a big chunk of copy-pasted code that we shouldn't need to repeat. A bit of cleanup gives us Listing 5, located at the end of this article.

The maze generation code is functionally the same, though it now all lives in the `visitCell()` method. The wizard also asked us to add an entrance in the north, exit to the south, and remove any dead ends. `addEntranceExit()` does this by picking one of the cells along the north or south edge of the map and removing the north or south wall, respectively. `removeDeadEnds()` scans the maze for any of the four possible dead-end states and busts through one of the walls randomly.

Now, given the code below, we can generate an endless variety of mazes to make Archlin happy. Figures 3 and 4 are two such mazes.

```
$rows = 25; $cols = 35; $size = 75;

$maze = new \OMerida\Maze\RecursiveGenerator(
$rows, $cols
);
$maze->generate();
$maze->removeDeadEnds();
$maze->addEntranceExit();

$render = new \OMerida\Maze\Renderer(
$rows, $cols, $size
);
$render->setCells($maze->getCells());
$render->draw();

$render->save(__DIR__ . '/maze-23x35.png');
```

How large can we make our labyrinth? I was able to generate a 200x300 maze in a few seconds. The main constraint is memory for the GDImage. If you're a fan of solving mazes, have a crack at the 80x60 maze in Figure 5.

Listing 1.

```php
<?php namespace OMerida\Maze;
class Renderer
{
  private \GdImage $image;
  private array $cells;
  private int $color1;
  private int $color2;
  private int $lineWidth;

  const WEST = 0x1;
  const EAST = 0x2;
  const SOUTH = 0x4;
  const NORTH = 0x8;

  public function __construct(
    private int $height,
    private int $width,
    private int $cellSize,
  ) { }

  public function setCells(array $cells): void {
    $this->cells = $cells;
  }

  public function draw(): bool
  {
    $padding = ceil($this->cellSize * 0.6);

    $this->image = imagecreatetruecolor(
      width: $this->width*$this->cellSize+$padding*2,
      height: $this->height*$this->cellSize+$padding*2,
    );

    $this->lineWidth = $this->cellSize * 0.1;

    // first call sets the background color
    $bg = imagecolorallocate(
      $this->image, 0xff, 0xff, 0xff
    );
    $this->color1 = imagecolorallocate(
      $this->image, 0x33, 0x33, 0x33
    );
    $this->color2 = imagecolorallocate(
      $this->image, 0x99, 0x99, 0x99
    );
    imagefill($this->image, 0, 0, $bg);
    imagesetthickness($this->image, $this->lineWidth);

    // starting top-left point
    $x = $y = $padding;
    foreach ($this->cells as $row) {
      foreach ($row as $cell) {
        $this->drawCell($cell, $x, $y);
        $x += $this->cellSize;
      }

      $x = $padding;
      $y += $this->cellSize;
    }
    return true;
  }
```

Listing 1 (continued).

```php
  private function drawCell(
    int $cell, int $x, int $y
  ): void {
    // draw north wall?
    $lineOff = floor($this->lineWidth * 0.5);
    if (($cell & self::NORTH) === self::NORTH) {
      imageline($this->image,
        x1: $x - 2 * $lineOff,
        y1: $y - $lineOff,
        x2: $x + $this->cellSize,
        y2: $y - $lineOff,
        color: $this->color2
      );
    }

    // draw south wall?
    if (($cell & self::SOUTH) === self::SOUTH) {
      imageline($this->image,
        x1: $x,
        y1: $y + $this->cellSize + $lineOff,
        x2: $x + $this->cellSize + $lineOff,
        y2: $y + $this->cellSize + $lineOff,
        color: $this->color1
      );
    }

    // draw west wall?
    if (($cell & self::WEST) === self::WEST) {
      imageline($this->image,
        x1: $x - $lineOff, y1: $y,
        x2: $x - $lineOff, y2: $y + $this->cellSize,
        color: $this->color2
      );
    }

    // draw east wall?
    if (($cell & self::EAST) === self::EAST) {
      imageline($this->image,
        x1: $x + $this->cellSize + $lineOff,
        y1: $y + $lineOff,
        x2: $x + $this->cellSize + $lineOff,
        y2: $y + $this->cellSize + $lineOff,
        color: $this->color1
      );
    }
  }

  public function save(string $file)
  {
    imagepng($this->image, $file);
  }
}
```

Figure 3.

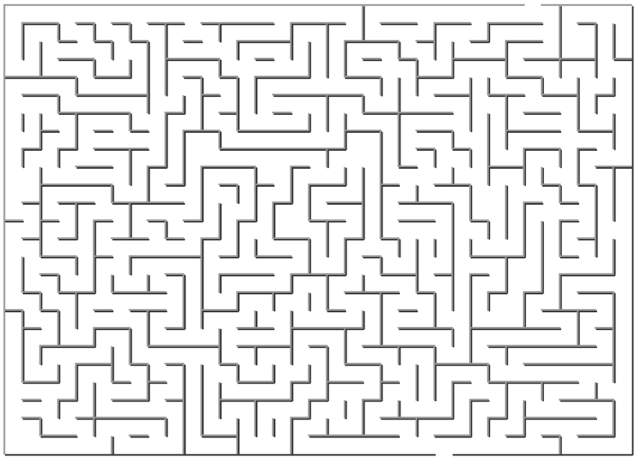


Figure 4.

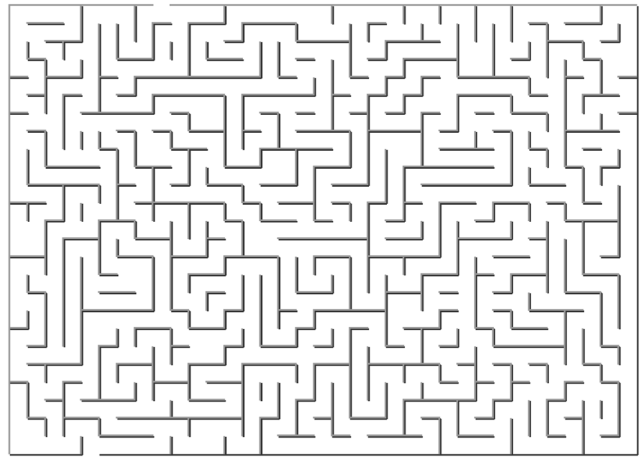
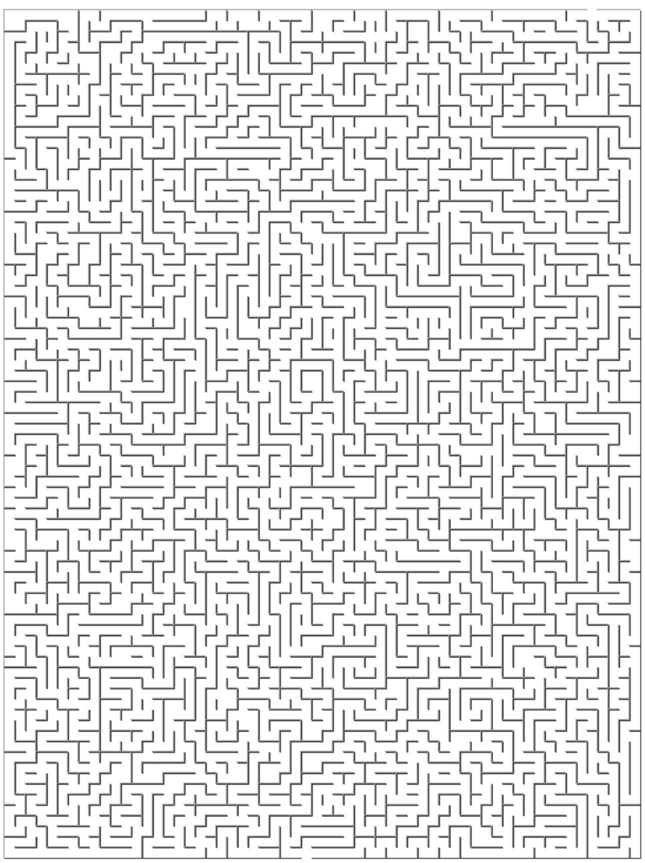


Figure 5.



Daniel Stori {turnoff.us}

Listing 2.

```php
<?php

use \OMerida\Maze\Renderer;
// now make a big random map
$maxRows = 40; $maxCols = 60;

// initialize them to all closed
$cells = range(0, $maxRows - 1);
$cells = array_map(function() use ($maxCols) {
   $row = range(0, $maxCols - 1);
   return array_map(fn() => 15, $row);
}, $cells);

// start at top left
$row = $col = 0;

// constants for the walls of a cell
const WEST = 0x1;
const EAST = 0x2;
const SOUTH = 0x4;
const NORTH = 0x8;

// break direction helpers
const BASH = [
   WEST =>
['opposite' => EAST, 'xOffset' => -1, 'yOffset' => 0],
   EAST =>
['opposite' => WEST, 'xOffset' => 1, 'yOffset' => 0],
   NORTH =>
['opposite' => SOUTH, 'xOffset' => 0, 'yOffset' => -1],
   SOUTH =>
['opposite' => NORTH, 'xOffset' => 0, 'yOffset' => +1],
];

// get Valid Directions from current cell
$possible = shuffleWalls($cells[$row][$col]);
if ($possible) {
   foreach ($possible as $direction) {
      $opp = BASH[$direction];
      $newX = $col + $opp['xOffset'];
      $newY = $row + $opp['yOffset'];
      if ($newX < 0 || $newY < 0
         || $newX >= $maxRows || $newY >= $maxCols) {
         continue;
      }

      $neighbor = $cells[$newX][$newY];
      if ($neighbor === 15) {
         // haven't broken any walls/visited it yet
         breakWall($cells, $row, $col, $direction);
      }
   }
}

$maze2 = new Renderer($maxRows, $maxCols, 100);
$maze2->setCells($cells);
$maze2->draw();
$maze2->save(__DIR__ . '/maze2.png');
```

Listing 2 (continued).

```php
function shuffleWalls($cell)
{
   $possible = [];
   // which of the four walls should we break?
   if ($cell & WEST) {
      $possible[] = WEST;
   }
   if ($cell & EAST) {
      $possible[] = EAST;
   }
   if ($cell & NORTH) {
      $possible[] = NORTH;
   }
   if ($cell & SOUTH) {
      $possible[] = SOUTH;
   }

   if (count($possible)) {
      // pick one of the remaining ones
      shuffle($possible);
   }

   return $possible;
}

function breakWall(
    array &$cells, int $row, int $col, int $direction
): void {
   global $maxCols, $maxRows;

   // figure out my neighbor
   $opp = BASH[$direction];
   $newRow = $row + $opp['yOffset'];
   $newCol = $col + $opp['xOffset'];
   if ($newRow < 0 || $newCol < 0
      || $newCol >= $maxCols || $newRow >= $maxRows) {
      return;
   }

   if ($cells[$newRow][$newCol] !== 15) {
      return;
   }
   // get rid of the indicated wall in our current cell
   $cells[$row][$col] -= $direction;

   // go to my neighbor in this direction
   // and bust opposite wall
   $cells[$newRow][$newCol] -= $opp['opposite'];

   // get my neightbor's possible walls
   $possible = shuffleWalls($cells[$newRow][$newCol]);
   if ($possible) {
      foreach ($possible as $newDir) {
         // haven't broken any walls/visited it yet
         breakWall($cells, $newRow, $newCol, $newDir);
      }
   }
}
```

Listing 3.

```php
<?php
namespace OMerida\Maze;

class RecursiveGenerator
{
  // constants for the walls of a cell
  private const WEST = 0x1;
  private const EAST = 0x2;
  private const SOUTH = 0x4;
  private const NORTH = 0x8;
  private const CLOSED = 15;

  // break direction helpers
  private const BASH = [
    self::WEST  => [
      'opposite' => self:: EAST,
      'xOffset' => -1,
      'yOffset' => 0
    ],
    self::EAST  => [
      'opposite' => self:: WEST,
      'xOffset' => 1,
      'yOffset' => 0
    ],
    self::NORTH => [
      'opposite' => self:: SOUTH,
      'xOffset' => 0,
      'yOffset' => -1
    ],
    self::SOUTH => [
      'opposite' => self:: NORTH,
      'xOffset' => 0,
      'yOffset' => +1
    ],
  ];

  private array $cells;

  public function __construct(
    private int $maxRows,
    private int $maxCols
  ) {
    // initialize them to all closed
    $this->cells = range(0, $maxRows - 1);
    $this->cells = array_map(
      function() use ($maxCols) {
        $row = range(0, $maxCols - 1);
        return array_map(fn() => self::CLOSED, $row);
    }, $this->cells);
  }
```

Listing 3 (continued).

```php
  /**
   * @return array|\int[][]
   */
  public function getCells(): array
  {
    return $this->cells;
  }

  public function generate(): void
  {
    $startX = rand(0, $this->maxRows - 1);
    $startY = rand(0, $this->maxCols - 1);

    $this->visitCell($startX, $startY);
  }

  private function visitCell(int $x, int $y)
  {
    if ($possible = $this->shuffleWalls($x, $y)) {
      foreach ($possible as $direction) {
        $coords = $this->getNeighborCoords($x, $y,
                          $direction);
        if (!$coords) {
          continue;
        }

        [$newX, $newY] = $coords;
        $neighbor = $this->cells[$newX][$newY];

        if ($neighbor !== self::CLOSED) {
          continue;
        }

        // get rid of the wall in our current cell
        $this->cells[$x][$y] -= $direction;

        // go to my neighbor in this direction and
        // bust the opposite wall
        $opp = self::BASH[$direction];
        $this->cells[$newX][$newY] -= $opp['opposite'];

        // visit neighbor to continue maze generation
        $this->visitCell($newX, $newY);
      }
    }
  }
```

Listing 3 (continued).

```
private function getNeighborCoords($x, $y, $direction)
{
  $opp = self::BASH[$direction];
  $newX = $x + $opp['yOffset'];
  $newY = $y + $opp['xOffset'];

  if ($newX < 0 || $newY < 0
    || $newX >= $this->maxRows
    || $newY >= $this->maxCols
  ) {
      return [];
  }

  return [$newX, $newY];
}

private function shuffleWalls(int $x, int $y)
{
  $cell = $this->cells[$x][$y];

  $possible = [];
  // which of the four walls should we break?
  if ($cell & self::WEST) {
    $possible[] = self::WEST;
  }
  if ($cell & self::EAST) {
    $possible[] = self::EAST;
  }
  if ($cell & self::NORTH) {
    $possible[] = self::NORTH;
  }
  if ($cell & self::SOUTH) {
    $possible[] = self::SOUTH;
  }

  if (count($possible)) {
    // pick one of the remaining ones
    shuffle($possible);
  }

  return $possible;
}
```

Listing 3 (continued).

```
public function removeDeadEnds(): void
{
  foreach ($this->cells as $x => $row) {
    foreach ($row as $y => $cell) {
      // is it a dead end
      // 1+2+4 or 1+2+8 or 2+4+8 or 1+4+8
      if (in_array($cell, [7, 11, 13, 14])) {
        $possible = $this->shuffleWalls($x, $y);
        $direction = $possible[0];
        $coords = $this->getNeighborCoords($x, $y,
                      $direction);
        if (!$coords) { continue; }

        [$newX, $newY] = $coords;

        // get rid of the wall in our current cell
        $this->cells[$x][$y] -= $direction;

        // go to my neighbor in this direction
        // and bust the opposite wall
        $opp = self::BASH[$direction];
        $this->cells[$newX][$newY] -=
                          $opp['opposite'];
      }
    }
  }
}

public function addEntranceExit() {
  // make an entrance
  [$entX, $entY] = $this->pickFromSide(self::NORTH);
  $this->cells[$entX][$entY] -= self::NORTH;

  // make an exit but on another side
  [$exitX, $exitY] =$this->pickFromSide(self::SOUTH);
  $this->cells[$exitX][$exitY] -= self::SOUTH;
}

private function pickFromSide(int $side): array {
  // we avoid picking corners for any case
  if ($side === self::EAST) {
    $row = rand(1, $this->maxRows - 2);
    $col = $this->maxCols - 1;
  }
  if ($side=== self::WEST) {
    $row = rand(1, $this->maxRows - 2);
    $col = 0;
  }
  if ($side === self::NORTH) {
    $row = 0;
    $col = rand(1, $this->maxCols - 2);
  }
  if ($side === self::SOUTH) {
    $row = $this->maxRows - 1;
    $col = rand(1, $this->maxCols - 2);
  }

  return [$row, $col];
}
}
```

Get Out!

"What if the treklins trap us in our maze," muses Archlin.

He grabs a parchment and tosses it toward a gold pen which springs to life with a wave of the wizard's hand. "I have your next assignment, apprentice."

The pen scrawls the following instruction: "For any maze you generate, create a map of the maze that traces the path from the entrance to the exit."

Some Guidelines And Tips

- *The puzzles can be solved with pure PHP. No frameworks or libraries are required.*
- *Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.*
- *You're encouraged to make your first attempt at solving the problem without using the web for clues.*
- *Refactoring is encouraged.*
- *I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.*
- *Go ahead and try many solutions if you like.*
- *PHP's interactive shell (`php -a` at the command line) or 3rd party tools like PsySH[4] can be helpful when working on your solution.*
- *To keep solutions brief, we'll omit the handling of out-of-range inputs or exception checking.*

4 PsySH: https://psysh.org

Related Reading

- *PHP Puzzles: Maze Rats* by Oscar Merida, April 2023. https://phpa.me/puzzles-apr-2023
- *PHP Puzzles: Grade Deviations* by Oscar Merida, March 2023. https://phpa.me/puzzles-march-2023
- *PHP Puzzles: Stats 101 Grade Book* by Oscar Merida, February 2023. https://phpa.me/puzzles-feb-2023
- *PHP Puzzles: Any Two Birthdays* by Oscar Merida, January 2023. https://phpa.me/puzzles-jan-2023

Oscar Merida has been working with PHP since version 4 was released and is constantly learning new things about it and still remembers installing an early version on Apache. When not coding or writing, he enjoys RPGs, soccer, and drawing. @omerida

First, Make it Easy

Edward Barnard

Like any software, PHP code can become rigid to the point that it's difficult to change. When the change is needed by tomorrow, for example, it's clear that a proper (multi-day) rewrite is not an option. With few automated tests in place, and none covering the area in question, adding tests to protect the modifications is probably not an option either.

Now is a great time to consider Kent Beck's advice[1]:

> *For each desired change, make the change easy (warning: this may be hard), then make the easy change.*

The experts, including Beck himself, will advise you to get some tests in place to ensure:

- Nothing else breaks in the process (no regressions)
- The new feature works as intended

Problem

In my case, I'm faced with a 1300-line PHP class. I've worked with it before (adding nearly half of those lines). It now has 28 methods, with many of those methods running 100 lines with nested `if` blocks, loops, and conditional logic. I had already broken out parts of the logic by splitting about six long methods into those 28 methods.

However, telling you the numbers has not explained the problem, has it? The problem is *not* the code itself.

Here's the problem. This class, `StripePaymentController`, supports athlete registration fees via Stripe as our transaction processor. The difficulty of change comes in because we have four different types of registration:

- Regular season registration, taking place before the competition season begins
- League tournament registration, taking place near the end of the competition season
- National championship registration, immediately after competition season ends
- A special-case Minnesota tournament, with Minnesota's league tournament being the qualifier

The different types of registration, with different business workflows, all run through the same `StripePaymentController`: that places registration, confirmation email, roster updates, etc., all in one place.

The difficulty arises (in part) because we're evolving our underlying database structure. Table changes require corresponding code changes. But we don't want to risk making any changes to `StripePaymentController` during active registration. When a financial record gets messed up, it's hard to detect, let alone resolve, and our reputation potentially suffers.

Our safest approach is to make no changes to `StripePaymentController` when any sort of registration is open. The problem is that some form of registration is open, or about to open, at pretty much any point in time!

What about creating a separate repository branch, making the change, and only merging that branch once active registration closes? That is, have feature changes ready to go, but wait until it's safe to merge.

That approach went spectacularly poorly.

`StripePaymentController` is so difficult to work with that we keep restructuring it, trying to make it a bit easier for next time. We also try to isolate the change to a separate `if` block, or whatever. The result is that unmerged code becomes hopelessly out of date. When the time came, my best option was to rewrite that "waiting in the wings" feature all over again… from scratch.

The "write the feature and leave it to sit" approach did not work. It would have been better to merge right away. When I rewrote the feature, we *did* merge that right away. There's something to learn here. There was an unknown risk in waiting, that is, leaving the feature unmerged until later.

Leaving that feature unmerged, in this case, forced `StripePaymentController` to become even more rigid and resistant to change. We now realize we can't touch `StripePaymentController` with an unmerged feature that touches most of the thousand-line file. In our case, we had to touch it anyway, which made the unmerged feature useless.

Robert C. Martin, in "The Clean Code Blog", elaborated on his own explanation of The Single Responsibility Principle[2]:

> *And this gets to the crux of the Single Responsibility Principle. This principle is about people.*

Our real issue here isn't the code itself. The code works. It's the context.

In our particular environment, those different types of registrations support different sets of business rules. Those rules evolve independently, at different times, for different reasons. For example, adjusting how we do the National Tournament has nothing to do with regular season registration.

1 *advice: https://phpa.me/kentbeck-0032*

2 *The Single Responsibility Principle: https://phpa.me/cleancoder*

We have separated out our data store. That is, season registrations are in different tables from tournament registration, and the different types of tournaments (league tournaments and the national tournament) are in different tables from each other. Our concern for each registration is at a different point in time. In other words, we don't think about details of National Tournament registration during season registration, before the season has even started.

Our difficulty comes from the fact that we've collected all those differently-timed concerns into the same file. It's a relatively high risk to touch the file for a *future* concern (such as National Tournament registration) while that file is actively handling cash flow for a *current* concern (such as active League Tournament registration).

Solution

Kent Beck advises, "make the change easy." "Uncle Bob" Martin advises, "separate the concerns." My project manager advises, "have this ready by tomorrow."

The result will *not* be pretty, but we'll make this change easy *and* future changes easier, as well, by making `StripeController` less rigid. What can we do *quickly* that's relatively safe?

My first step was to move methods into separate files. Martin Fowler's guest author Clare Sudbery explains the thinking, with an excellent walkthrough, at Refactoring: This class is too large[3]. However, keep in mind that I have a different purpose. My objective is to separate-out the workflows from each other. When League Tournament registration is open, specifically, I need to be able to safely make changes to National Tournament registration because we have made changes to the underlying National Tournament database tables.

I decided to start at the top of the `StripePaymentController` file, taking the first method past the constructor:

```
protected function captureFlow($post): string
{
    $flow = RegistrationWorkflow::captureFlow();
    $_SESSION['registration_flow'] = $flow;
    // Save form POST data for recovery
    // after round trip to Stripe
    $_SESSION['stripe_post'] = $post;
    return $flow;
}
```

The first thing I noticed is the qualifier `protected`. Class `StripePaymentController` does not extend any classes, nor do any classes extend `StripePaymentController`. My first step was to change all `protected` methods to `private`. At this point I should run all unit tests to ensure nothing is broken. However, there are no tests for `StripePaymentController`, so I moved on to the next step.

I extracted method `captureFlow()` into a class of its own (`use` and `namespace` not shown): (See Listing 1)

3 https://phpa.me/martinfowler

Listing 1.

```
class CaptureFlow
{
    public function captureFlow(array $post): string
    {
        $flow = RegistrationWorkflow::captureFlow();
        $_SESSION['registration_flow'] = $flow;
        // Save form POST data for recovery after
        // round trip to Stripe
        $_SESSION['stripe_post'] = $post;
        return $flow;
    }
}
```

Next, in `StripePaymentController`, I created a new method `initialize()` and called it from the constructor like this:

```
public function __construct(ViewService $view)
{
    $tempDir = dirname(__DIR__, 4) .
                    '/views/Payment/Stripe';
    $this->view = $view->setView($tempDir);
    $this->initialize();
}
```

Note that this is *not* necessarily the best practice concerning class initialization. We are doing a rapid refactoring aimed at reducing the code's rigidity due to overlapping registration periods. We're doing just enough to make the upcoming change easier.

Initialize instantiates the new class:

```
    private CaptureFlow $captureFlow;
    public function initialize(): void
    {
        $this->captureFlow = new CaptureFlow();
    }
```

The method call then changes from this:

```
$flow = $this->captureFlow($args['post']);
```

To this:

```
$flow = $this->captureFlow->captureFlow($args['post']);
```

Note the class name is the same as the method name (except for the first letter becoming upper case). The property `$this->captureFlow` has the same name as the method. Thus, the code edit becomes quite easy. Since I marked the original method as private, once I switch all references to the new class, my editor flags the original method as unused and should be deleted. That's the next step: Delete the original method.

This is a small step with no change in observable functionality. If I had automated tests to run, I would run them at this point, and they should pass.

I continued down this same path, extracting each method into a new class and using it. Here are the new properties: (See Listing 2 on the next page)

Listing 2.

```
private CaptureFlow $captureFlow;
private CaptureMode $captureMode;
private CaptureApiKey $captureApiKey;
private RestorePost $restorePost;
private restoreStripeSession $restoreStripeSession;
private RetrieveSeasonStripeSession
                    $retrieveSeasonStripeSession;
private RetrieveLeagueStripeSession
                    $retrieveLeagueStripeSession;
private RetrieveMshslStripeSession
                    $retrieveMshslStripeSession;
private RetrieveNationalStripeSession
                    $retrieveNationalStripeSession;
private ConfirmSeason $confirmSeason;
private ConfirmLeague $confirmLeague;
private ConfirmMshsl $confirmMshsl;
private ConfirmNational $confirmNational;
private CheckoutSessionSeason $checkoutSessionSeason;
private CheckoutSessionLeague $checkoutSessionLeague;
private CheckoutSessionMshsl $checkoutSessionMshsl;
private CheckoutSessionNational
                    $checkoutSessionNational;
```

My new `initialize()` method looks like this. Note that some methods have dependencies on other methods. I simply pass those dependencies into the new class constructor: (See Listing 3 on the next page)

Take, for example, the last class created, `CheckoutSessionNational`. Its constructor looks like this: (See Listing 4)

The new class, `CheckoutSessionNational`, can call the dependent class as you would expect:

```
$this->nationalLogSessionBegan
     ->nationalLogSessionBegan($args);
```

Listing 4.

```
class CheckoutSessionNational
{
  private RunStripeCheckout $runStripeCheckout;
  private NationalLogSessionBegan
                      $nationalLogSessionBegan;
  public function __construct(
    RunStripeCheckout $runStripeCheckout,
    NationalLogSessionBegan $nationalLogSessionBegan
  ) {
    $this->runStripeCheckout = $runStripeCheckout;
    $this->nationalLogSessionBegan =
                        $nationalLogSessionBegan;
  }
}
```

Listing 3.

```
public function initialize(): void
{
  $this->captureFlow = new CaptureFlow();
  $this->captureMode = new CaptureMode();
  $this->captureApiKey = new CaptureApiKey();
  $runStripeCheckout = new RunStripeCheckout();
  $this->restorePost = new RestorePost();
  $this->restoreStripeSession =
        new RestoreStripeSession($this->captureApiKey);
  $this->retrieveSeasonStripeSession =
        new RetrieveSeasonStripeSession();
  $this->retrieveLeagueStripeSession =
        new RetrieveLeagueStripeSession();
  $this->retrieveMshslStripeSession =
        new RetrieveMshslStripeSession();
  $this->retrieveNationalStripeSession =
        new RetrieveNationalStripeSession();
  $this->confirmSeason = new ConfirmSeason();
  $this->confirmLeague = new ConfirmLeague();
  $this->confirmMshsl = new ConfirmMshsl();
  $this->confirmNational = new ConfirmNational();
  $this->checkoutSessionSeason =
    new CheckoutSessionSeason(
        new CollectSeasonTeamInfo(),
        new CollectSeasonRosterIds(),
        new CollectSeasonLineItems(),
        $runStripeCheckout
    );
  $this->checkoutSessionLeague =
    new CheckoutSessionLeague(
        new CollectLeagueTeamInfo(),
        new CollectLeagueRosterIds(),
        new CollectLeagueLineItems(),
        $runStripeCheckout
    );
  $this->checkoutSessionMshsl =
    new CheckoutSessionMshsl(
        new CollectMshslRosterId(),
        $runStripeCheckout,
        new MshslLogSessionBegan()
    );
  $this->checkoutSessionNational =
    new CheckoutSessionNational(
        $runStripeCheckout,
        new NationalLogSessionBegan()
    );
}
```

The Result

Did we actually accomplish anything? We scattered code from one large file into many very small files. We reduced `StripePaymentController` from 1,300 lines to 300 lines. The business flow, as a result, is far easier to follow. Thus, there is a win, but at what cost?

This restructuring went extremely quickly, even double-checking every step (because I have no automated tests to run).

Remember Kent Beck's advice:

> *For each desired change, make the change easy (warning: this may be hard), then make the easy change.*

Our objective *here* was to make the change easy. It now is! My task is to change the National Tournament registration to work with the new tables. Now I need to edit a few small files, each of which is specific to National Registration. I have reduced the risk of breaking something else.

The Mess

I carefully scattered a large mess into two dozen small messes. Is that actually an improvement? Yes, it is! Remember our analysis of the problem? The real problem to solve was that we dare not touch inactive registration code when a different type of registration is active because they all touch the same code paths in the same large file.

The mess is still there – and so is the new feature I was assigned to write concerning National Registration. Mission accomplished.

However, this new state of the codebase allows us to tackle one small mess at a time with much lower risk. That's because the messes are separate from each other.

Summary

When faced with time constraints, we're often forced into writing "quick and dirty" features. The result is ugly. However, I rarely think about "quick and dirty" refactoring. That's because the objective of refactoring is usually to improve code quality. In my mind, that means it should look cleaner and prettier, easier to edit in the future.

Instead, I made use of a deeper insight. This particular piece of code is dangerous to change because we have gathered all the registration workflows into a single file. We dare not change one workflow when another is active.

Thus, my objective in the "ugly" refactoring was to separate the workflows. This brings us closer to the Single Responsibility Principle's ideal: "gather together the things that change at the same time and for the same reason, and separate the things that change at different times for different reasons."

Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others. *@ewbarnard*

PSR-20: Clock

Frank Wallen

Let's take a closer look at some of the features of the Clock interface. We'll see how wrapping date/time in an object can simplify the process of making future changes.

This month we are reviewing PSR-20[1], the Clock interface. Interoperability is critical for implementing libraries across projects and applications, swapping libraries without impacting behavior, and (importantly) testing. Why use an interface or a class when it's as simple as calling `time()` or `date()`? Libraries that wrap these functions offer many usable tools and improved handling of the date/time, especially consistency when working across time zones. Instead of applying timezone math to your timestamp, many modern libraries allow this with a simple method, such as `setTimezone('America/Los_Angeles')`.

Let's take a look at the `Clock` interface:

```
namespace Psr\Clock;

interface ClockInterface
{
    public function now(): \DateTimeImmutable;
}
```

That's it. A straightforward interface with only one required method. Note the return type: `DateTimeImmutable`. This maintains consistency. When you create your time object, you can't change it. Calling the `modify()` method on it will return a *new* `DateTimeImmutable` (with the modified date), leaving the original unchanged. If your model's date/time properties use `DateTime`, they are at risk of having their data changed by code performing date comparisons or date-math. For example: (See Listing 1)

Listing 1.

```
class MyModel
{
    //...other properties
    public DateTime $createdDate;

    public function __construct()
    {
        $this->createdDate = new DateTime();
    }
    //...methods, etc.
}

$myModel  = new MyModel();
$tomorrow = (new DateTime())->modify('+1 day');
$nextDay  = $myModel->createdDate->modify('+1 day');
if ($nextDay->getTimestamp() ===
                $tomorrow->getTimestamp()) {
    //...
}
```

In this example, our entity `MyModel` sets its created date using `DateTime()`. Later, when comparing the `createdDate` property to our `$tomorrow DateTime` (it's a bit of a silly comparison, but consider the impact), we actually modified the created date of our entity by advancing it one day. If we had persisted this entity to the database, it would have stored the incorrect date.

Besides the goal of interoperability between implementations, the other goal of this interface is to provide a simple, mockable way to get and use the time when testing. When testing, tests rarely need to know the current time; they need to work within the confines of the scenario and use cases that are being tested. Let's review an example using a mocked Clock.

The class we are testing is the `Appointment` class below. It's instantiated with an appointment clock and the appointment date/time. (See Listing 2 on the next page)

Because we will be testing using date/time, we need to mock our appointment clock to freeze the time. When `now()` is called on `MockAppointmentClock` it will always return the same `DateTimeImmutable`. (See Listing 3 on the next page)

And our test: (See Listing 4 on the next page)

Out test wants to assert that we should send a follow up after 14 days have passed since the appointment finished. We instantiate our appointment clock with the date/time 1 hour in the past so we can use that as our baseline. A new `Appointment` is created at the 'current' time (which is actually 1 hour before 'now'). We set the finished date/time for the appointment using our appointment clock, advanced by 1 hour. The `shouldSendFollowUp()` method expects the date/time to check against (a `DateTimeImmutable` object), so we get our frozen time from the `MockAppointmentClock`, add the 14 days of follow-up time, and 1 hour for the length of the appointment, which results in a successful test assertion.

The `ClockInterface` does not include specifications for handling time zones, `sleep`, or modifying date/time. Its only purpose is to present a simple method for retrieving the time. Implementing time zones is up to the application or the implementor, as it usually has a lot to do with, or is controlled by, the application's business logic.

The phpunit-bridge[2] package for symfony provides a `ClockMock` class and it mocks the system time functions like `time()`, `microtime()`, `sleep()`, `date()`, `gmdate()`, etc., which is very useable, but it does not mock using `DateTimeImmutable`.

1 PSR-20: *https://www.php-fig.org/psr/psr-20/*

2 phpunit-bridge: *https://phpa.me/clock-mock*

Listing 2.

```php
<?php namespace Appointments;
use Psr\Clock\ClockInterface;

class Appointment
{
  protected ?DateTimeImmutable $finishedDateTime;

  public function__construct(
    protected ClockInterface $appointmentClock,
    protected \DateTimeImmutable $appointmentDateTime
  ) { }

  public function setFinishedDateTime(
    \DateTimeImmutable $dateTime
  ) {
    $this->finishedDateTime = $dateTime;
  }

  public function getFinishedDateTime(
  ): ?DateTimeImmutable {
    return $this->finishedDateTime;
  }

  public function shouldSendFollowUp(
    \DateTimeImmutable $currentDateTime
  ): bool {
    if(is_null($this->getFinishedDateTime())) {
        return false;
    }

    $finishedDate = $this->getFinishedDateTime();
    $followUpDate = $finishedDate->getTimestamp();
    $now          = $currentDateTime->getTimestamp();

    return $now >= $followUpDate;
  }
  //more functionality here
}
```

Listing 3.

```php
namespace Appointments;

use Psr\Clock\ClockInterface;

class MockAppointmentClock implements ClockInterface
{
    public function __construct(
        protected \DateTimeImmutable $dateTime
    ) { }

    /**
     * @inheritDoc
     */
    public function now(): \DateTimeImmutable
    {
        return $this->dateTime;
    }
}
```

Listing 4.

```php
use Appointments\Appointment;
use Appointments\MockAppointmentClock;
use PHPUnit\Framework\TestCase;

class SendFollowUpNotificationTest extends TestCase
{
    /**
     * @test
     */
    public function testItShouldSendAFollowUp()
    {
        $appointmentClock = new MockAppointmentClock(
          (new \DateTimeImmutable())->modify('-1 hour')
        );

        $appointment = new Appointment(
                        $appointmentClock,
                        $appointmentClock->now()
        );

        $appointment->setFinishedDateTime(
          $appointmentClock->now()->modify('+ 1 hour')
        );

        $this->assertTrue(
            $appointment->shouldSendFollowUp(
                $appointmentClock->now()
                                 ->modify('+14 days')
                                 ->modify('+1 hour')
            )
        );
    }
}
```

Conclusion

Out of the PSRs we have reviewed in this column, the `ClockInterface` has only a few requirements, but it shows the value of having your objects be testable and mockable. Doing so is good practice for clean abstractions, especially flexibility, and utility. If the application is accessing time from various system routines and calls and then newly introduced business rules dictate a change in how date/time is handled, one would have to visit all those locations to update the code. By wrapping date/time in an object, it will be easier to modify that code to support the new business rules with less effort.

Frank Wallen is a PHP developer, musician, and tabletop gaming geek (really a gamer geek in general). He is also the proud father of two amazing young men and grandfather to two beautiful children who light up his life. He is from Southern and Baja California and dreams of having his own Rancherito where he can live with his family, have a dog, and, of course, a cat. @frank_wallen

ADR vs MVC

Matt Lantz

In ADR or Action-Domain-Response, we maintain a three-piece pattern that lets us split our responsibilities. The classic MVC structure or Model-View-Controller is prevalent across all languages in the web development industry. Let's take a closer look at both.

> *A refreshing and exciting feeling wafts over each developer when they first write that line in what I suspect is iTerm2.*

```
$ laravel new kickass-site
```

or, more classically

```
$ composer create-project laravel/laravel kickass-site
```

More often than not, the next step is to contemplate which front-end framework to use and begin coding. Most developers will use customized artisan commands, generate their tests and resources, and map complex business entities to elegant models with transparent object layers. Most Laravel developers will build their applications in a classic MVC structure. However, some will explore working with single-responsibility controllers; some will integrate Actions throughout the application. Some will explore event-driven systems, and some will dive deep into domain-driven design.

The classic MVC structure or Model-View-Controller is prevalent across all languages in the web development industry. More often than not, Laravel developers will begin by creating a route in the `web.php` file, pointing it to a Controller, and start tinkering with their view or their Model. Beyond that, in many cases, developers will implement Middleware elements as well as Jobs, Events, and Listeners. What often happens in this classic MVC structure with sprinkles of enhancements is that business logic begins to creep through the various layers making it harder to maintain consistency or enable teams to work on code without impacting multiple layers. This dilemma is why enterprise teams and scaling organizations often begin implementing SPA solutions and breaking code into further and further micro-services, all the while hoping that the breakdown of the monolith is what will resolve the cross-layer contamination of the business logic.

Views often contain permission checks, and Models become somewhat bloated with complex queries. Controllers 10 years ago were bloated with code logic, but these days it's now sprinkled across Services or Models, and in the last few years, we've seen an active rise in the use of Actions. To avoid repetition, developers can use these self-contained code snippets in Commands, Controllers, and Services alike. The value gain is apparent; it enables developers to centralize where the logic is contained. This same pattern has become popular across front-end systems in how Vue and Livewire enable developers to have Components. In these cases, you have a component that can contain all logic of that entity in one

file or a small set of files. However, these Components can quickly become bloated and handle all an entity's interactions, thereby becoming spaghetti-like. Each element aims to help developers move logic out of Blade templates and Controllers into reusable spaces.

Laravel itself has enabled developers to start with a very barren starter structure, and they can set things accordingly within their structure of choice. This benefit is that it allows developers to choose their own architecture. However, as mentioned above, most developers will often build things as an MVC structure and then sprinkle in other features as needed without documenting or designing the architecture as a whole. It becomes an MVC with random add-on patterns.

Paul M. Jones, a well-recognized member of the PHP community, proposed an alternative architecture in 2014 titled ADR or Action-Domain-Response. Its overall structure helps developers remove the repetition of business logic in their applications by drawing much deeper lines in the sand. Implementing the ADR pattern in Laravel would likely increase the total development time while reducing mental

Listing 1.

```
resources/
    templates/
        blog/
            index.php
            create.php
            read.php
            update.php
            delete.php
            _comments.php
src/
    Domain/
        Blog/
            BlogModel.php
                        BlogService.php
    Ui/
        Web/
            Blog/
                Index/
                    BlogIndexAction.php
                    BlogIndexResponder.php
                Create/
                    BlogCreateAction.php
                    BlogCreateResponder.php
                Read/
                    BlogReadAction.php
                    BlogReadResponder.php
                Update/
                    BlogUpdateAction.php
                    BlogUpdateResponder.php
                Delete/
                    BlogDeleteAction.php
                    BlogDeleteResponder.php
```

Listing 2.

```
class BlogCreateAction
{
    public function __construct(
        Request $request,
        BlogCreateResponder $responder,
        BlogService $domain
    ) {
        // ...
    }

    public function __invoke()
    {
        if ($this->request->isPost()) {
            $data = $this->request->getPost('blog');
            $blog = $this->domain->create($data);
        } else {
            $blog = $this->domain->newInstance();
        }

        return $this->responder->response($blog);
    }
}
```

Listing 3.

```
class BlogCreateResponder
{
    public function __construct(
        Response $response,
        TemplateView $view
    ) {
        // ...
    }

    public function response(BlogModel $blog)
    {
        // is there an ID on the blog instance?
        if ($blog->id) {
            // yes, which means it was saved already.
            // redirect to editing.
            $this->response->setHeader(
                'Location',
                '/blog/edit/{$blog->id}'
            );
        } else {
            // no, which means it has not been
            // saved yet. show the creation form with
            // the current data.
            $html = $this->view->render(
                'create.php',
                ['blog' => $blog]
            );
            $this->response->setContent($html);
        }

        return $this->response;
    }
}
```

strain and code complexity, thus enabling less complex maintenance.

Paul has some examples on GitHub of how the ADR pattern can be implemented and provides an example of refactoring an MVC to an ADR approach. Below are his examples: (See Listing 1 on the next page) https://phpa.me/refactoring[1].

In ADR, we maintain a three-piece pattern that lets us split our responsibilities. Actions are simple single tasks. The actions would follow the pattern of `Ui/Web/Blog/Create/CreateBlogAction.php`. Furthermore, you would have the Domain/Blog directory containing the Blog model and Service type classes interacting with the Model. Within the `Ui/Web/Blog`, you would have classes that build the entity's listing or provide a single view of a Form for editing the entity with a Responder. In some cases, we see how Responders can work much like Components since they can contain logic and use templates where needed. We can quickly see how we can also handle something like `Api/Blog/Create/CreateBlogResponder.php` or `Ui/Web/Blog/Create/CreateBlogResponder.php`; each of these can be defined in our action based on the request and

1 https://phpa.me/refactoring

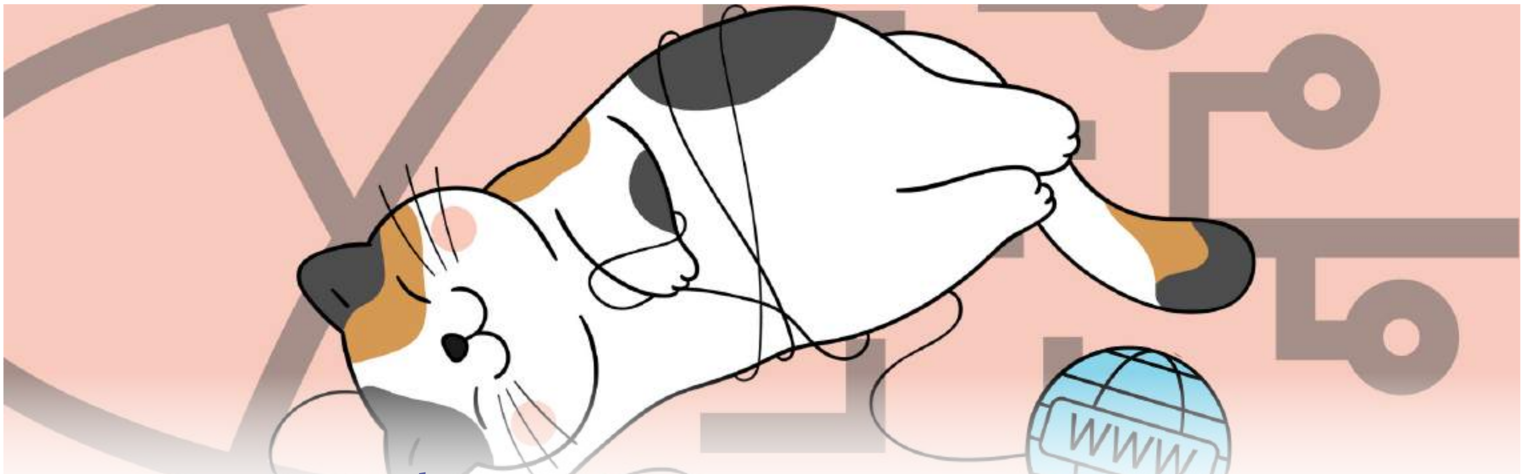
its expected platform, which falls into better alignment with Bob Martin's Clean Architecture. (See Listing 2)

Within this example, we can see clearly the various cases where business logic can be injected into the domain and detached from the Responder and Action. (See Listing 3)

Similarly, we can see here how the response is mitigated, and the template is rendered and injected into the response.

We are seeing more and more systems arise in the Laravel community, which are helping to decouple the business logic from the presentation layer. Things like Livewire Components enable developers to centralize logic into single files. It also allows developers to remove JavaScript coding requirements, resulting in less maintenance. However, it doesn't resolve the issue of business logic being in both the presentation layer and the Component itself. Systems like Inertia enable developers to build applications with a more formal REST backend and have their VueJS components contain all the presentation layers and the corresponding logic. In either of these cases, we still see the persistence of logic crossing multiple application layers. There are some valid critiques of ADR, which Paul M. Jones has, in some cases, addressed in his GitHub repo detailing the pattern. Building applications with this pattern can become verbose with numerous files, many of which are very small and somewhat repetitive. There are also concerns about logic being placed poorly across these layers as well. We can see that we gain the option of more granular testing and overall improvements to the readability of the code within the ADR pattern, but ultimately, it comes at a cost. In the outlined case above, we can see some easy ways to split up our view handling and reduce the probability of injecting logic into the presentation layer.

Matt has been developing software for over 13 years. He started his career as a developer working for a small marketing firm, but has since worked for a few Fortune 500 companies, lead a couple teams of developers and is currently working as a Cloud Architect. He's contributed to the open source community on projects such as Cordova and Laravel. He also made numerous packages and helped maintain a few. He's worked with Start Ups and sub-teams of big teams within large divisions of companies. He's a passionate developer who often fills his weekend with extra freelance projects, and code experiments. @MattyLantz



CatAIstrophe

Beth Tucker Long

You would be hard-pressed to find a news source that is not overflowing with dire warnings and speculations about recent advancements in AI. I have even gotten messages from people wondering if I know what career I will do next, seeing as AI is eliminating my job. Is "Judgment Day" coming?

Advancements in technology have long been met with skepticism and fear, especially when portrayed in the media. As machines became a staple in manufacturing, we watched the big screen in horror as the working class of Metropolis was banished underground, basically enslaved by the owners of the city's massive machines. As advancements in surgical technology began curing more ailments, Frankenstein's scary monster lurched across the screen. As technological advances in flight fostered explorations of space, body-snatching aliens invaded the theaters. As nuclear technology developed, Godzilla stormed across the screen, destroying cities. As computers began to spread into homes, Terminators powered by AI horrified the masses.

This time, though, instead of the movies, it's the news that is scaring us. Headlines like "Could AI Destroy Humanity? Experts Warn of Catastrophic Consequences" and "New Research Shows the Robots are Coming for Jobs—But Stealthily" are not designed to make us feel better about AI, but is the fear justified?

Every time technology has taken a big leap forward, it has eliminated jobs. The printing press eliminated scribes. Trains eliminated Pony Express riders. Cars eliminated stagecoach drivers. Modern telephones eliminated switchboard operators. Automation eliminated assembly line workers.

Each time an advancement has eliminated jobs, though, it has also created jobs. The printing press created a new need for machinists to create parts and repair presses. Trains created a new need for people to build tracks and trains. Cars created a need for mechanics and design engineers. Modern telephones created a need for electrical engineers. Automation created a need for mechanical engineers and programmers.

Each advancement eliminated familiar jobs but created new jobs we didn't know we would need. Yes, change is scary, but it is inevitable. We need to be mindful of making change in a thoughtful way, but we cannot allow the potential loss of "the way things have always been done" to block progress.

It's scary to move out of our comfort zone, try something new, and wonder if we will lose our jobs, but we humans are made of tough stuff. We survive progress. Just look at all the good we have done with our previous advancements, and now, let's lay in the grass while AI answers our emails and we dream about the amazing things we can do with our next advancement.

Beth Tucker Long is a developer and owner at Treeline Design[1], a web development company, and runs Exploricon[2], a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development[3] and Full Stack Madison[4] user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT

1 *Treeline Design: http://www.treelinedesign.com*

2 *Exploricon: http://www.exploricon.com*

3 *Madison Web Design & Development: http://madwebdev.com*

4 *Full Stack Madison: http://www.fullstackmadison.com*

PhpStorm

Enjoy productive PHP

jetbrains.com/phpstorm