Introduction to the Builder Pattern in PHP

Background

The Builder pattern is one of the "Gang of Four" (GoF) design patterns. It is classified as a *creational design pattern*—i.e. it's intended to address some brittle historical practices (i.e. anti-patterns) for creating object instances.

When using the Builder pattern, we create an object of one type by creating an object of a second, supporting type, invoking a number of methods on the second object, ending with a method that returns a fully constructed object instance (often immutable) of the first type.

Scenario

Requirements

A client has asked us to develop a random password generator component. In joint design conversations, we agree on the following minimal technical and functional requirements:

1. The generator functionality will be abstracted and encapsulated into a class; instances of the class will be created on demand, and used to generate 1 or more passwords in a single request.

2. The generator must be configurable, to allow inclusion/exclusion of the following:

    - Upper-case letters from the Latin alphabet;
    - Lower-case letters from the Latin alphabet;
    - The digits 0-9;
    - Punctuation and (printable) symbols.

3. Against our advice, the client insisted (and we accepted) that for each of the above character sets, the generator must be configurable with a minimum count, to enforce policies such as "A password must include at least 1 upper-case letter, 1 lower-case letter, and 1 special character."

4. In general, the generator should support all of the punctuation and symbol characters in the Basic Latin Unicode block, except for the space character (i.e. valid characters are in the ranges `\u0021–\u002F`, `\u003A–\u0040`, `\u005B–\u0060`, `\u007B–\u007E`). However, on exploring the contexts in which the generated passwords might be used, we agreed with the client that the generator should support some constraints on punctuation and symbols—specifically, the generator must allow, on initialization, the exclusion of a subset of punctuation and symbol characters. (More generally, we may opt to provide a mechanism for excluding *any* characters that would otherwise be considered valid.)

5. The generator must allow optional exclusion of the mutually ambiguous character pairs, “1” & “l” (lower-case “L”), and “0” (zero) & upper-case “O”. This option must be enabled by default.

6. The generator class should have no dependencies on external configuration files. *All* of the above configuration should be specifiable by the class’s consumer.

7. At our insistence, the lifecycle of the generator object will be one-way: initialize/configure a generator instance, then use it; changing configuration options on a generator instance after we use it to generate passwords will not be supported.

Technical Specifications

Our next task is to propose an API for consuming the class—that is, for instantiating, initializing, and invoking methods on instances of the class. Very quickly, we come to the realization that, with the wide variety of configuration options, initializing the generator could become pretty complicated. In many (maybe most) use cases, we won’t need to change more than one or two of the configuration options from their default values—but in a few cases, we’ll need to change most of those options. We need to come up with an initialization approach that not only makes the generator component easy to work with for the simple use cases, but flexible enough for the tricky ones.

Approach 1: Constructor tricks

- **Constructor overloads**

  Can we have multiple constructor methods, with different numbers/types of parameters, to support the range of configuration scenarios we anticipate?

  No. PHP doesn’t support constructor method overloading (or overloading of any other method or function, for that matter). In PHP, the *signature* of a function or method—which must be unique within the scope of the function or method—consists only of the method name. (This is also the case for JavaScript, Python, and C—though not for C++, Java, C#, and VB.NET.)

  Even if we could use overloaded constructors, such an approach would arguably be a poor fit for this situation. A small number of constructors might be acceptable, but we could easily end up with many more for our generator—following a practice we sometimes call the *telescoping constructor anti-pattern*.

- **Constructor with several parameters**

  Can’t we can define a constructor with parameters for all of the configuration options?

  We certainly could—and we’d end up with a constructor that’s much more complicated to work with than we’d like. It would have the necessary flexibility, but it wouldn’t be simple to use even in the most basic use cases.

- **Constructor with default parameter values**

Can we define a constructor with default values for one or more parameters?

Yes. We might do something like this:

```
class PasswordGenerator
{

    private $upperIncluded;
    private $lowerIncluded;
    private $digitIncluded;
    private $punctuationIncluded;
    private $ambiguousExcluded;
    private $minUpper;
    private $minLower;
    private $minDigits;
    private $minPunctuation;
    // More fields here ...

    public __construct(
        bool $upperIncluded = true,
        bool $lowerIncluded = true,
        bool $digitIncluded = true,
        bool $punctuationIncluded = true,
        bool $ambiguousExcluded = true;
        int $minUpper = 0,
        int $minLower = 0,
        int $minDigits = 0,
        int $minPunctuation = 0,
        // And so on ...
    ) {
        $this->upperIncluded = $upperIncluded;
        $this->lowerIncluded = $lowerIncluded;
        $this->digitIncluded = $digitIncluded;
        $this->punctuationIncluded = $punctuationIncluded;
        $this->ambiguousExcluded = $ambiguousExcluded;
        $this->minUpper = $minUpper;
        $this->minLower = $minLower;
        $this->minDigits = $minDigits;
        $this->minPunctuation = $minPunctuation;
        // More initialization code here ...
    }

    // More methods here ...

}
```

Unfortunately, this doesn't improve matters much. When we initialize `PasswordGenerator` using this constructor, we can only omit arguments at the right end of the parameter list; we can't skip some in the middle and specify others after that. We might work around this by using `null` default values, skipping the types on the parameter declarations, or using *nullable* types (where a question mark

precedes the type in the parameter declaration); any of those options would let us specify `null` arguments in an invocation to indicate that the corresponding parameters should take their default values. But we'd still have a long list of parameters, and the arguments we pass on invocation would have to match the parameter order exactly. Producing effective documentation—not just for the generator API, but also for any code that consumes the generator component—would be a challenge.

- **Combining multiple Boolean values into a bit field**

  Rather than use several `bool` parameters, can we combine them into a single `int`, treated as *bit field*?

  Indeed, this would reduce the number of parameters significantly in a situation like this one. Consider the following:

```
class Generator
{

    public const UPPER_INCLUDED = 1;
    public const LOWER_INCLUDED = 2;
    public const DIGIT_INCLUDED = 4;
    public const PUNCTUATION_INCLUDED = 8;
    public const AMBIGUOUS_EXCLUDED = 16;
    // More option flags here ...
    public const DEFAULT_OPTIONS =
        UPPER_INCLUDED
        | LOWER_INCLUDED
        | DIGIT_INCLUDED
        | PUNCTUATION_INCLUDED
        | AMBIGUOUS_EXCLUDED; // Maybe more ...

    private $upperIncluded;
    private $lowerIncluded;
    private $digitIncluded;
    private $punctuationIncluded;
    private $ambiguousExcluded;
    private $minUpper;
    private $minLower;
    private $minDigits;
    private $minPunctuation;
    // More fields here ...

    public __construct(
        int $options = DEFAULT_OPTIONS,
        int $minUpper = 0,
        int $minLower = 0,
        int $minDigits = 0,
        int $minPunctuation = 0,
        // And so on ...
    ) {
        // Unpack the option values from the bit field.
        $this->upperIncluded = (($options & UPPER_INCLUDED) > 0);
        $this->lowerIncluded = (($options & LOWER_INCLUDED) > 0);
        $this->digitIncluded = (($options & DIGIT_INCLUDED) > 0);
        $this->punctuationIncluded = (($options & PUNCTUATION_INCLUDED) > 0);
        $this->ambiguousExcluded = (($options & AMBIGUOUS_EXCLUDED) > 0);
        // And so on ...
        // More initialization code here ...
    }

    // More methods here ...

}
```

Now we might create an instance of `PasswordGenerator` that uses upper-case letters and digits (for

example) this way:

```
new Generator(
    Generator::DIGIT_INCLUDED | Generator::UPPER_INCLUDED | Generator::EXCLUDE_AMBIGOUS,
    0, 0, 0, 0,
    // And so on ...
);
```

We've reduced the number of parameters, and we no longer have to remember the order of our option flags. Also, since those flags are all powers of 2, we can combine them for the invocation using the bitwise operators (recommended), or just addition—for example, note that `1 + 2 + 8` gives the same value (11) as `1 | 2 | 8`.

On the other hand, only Boolean parameters can be packed into bit fields in this fashion. Also, while many "old school" programmers are used to bit fields, a less experienced programmer may find them quite confusing, and will need to rely more on external documentation, rather than the API itself being more effectively self-documenting.

- **Using an associative array for configuration options**

  Can we simply specify all the configuration option flags and settings as elements in an associative array, or properties of an object?

  In fact, this is an approach used by many PHP, JavaScript, and Python libraries: rather than initializing a complex object (or invoking a complex method) by passing a long list of arguments, we pass a much shorter argument list, where one or more of the arguments is itself an object or associative array.

```php
class Generator
{

    public const UPPER_INCLUDED_KEY = 'upperIncluded';
    public const LOWER_INCLUDED_KEY = 'lowerIncluded';
    public const DIGIT_INCLUDED_KEY = 'digitIncluded';
    public const PUNCTUATION_INCLUDED_KEY = 'punctuationIncluded';
    public const AMBIGUOUS_EXCLUDED_KEY = 'ambiguousExcluded';
    public const MIN_UPPER_KEY = 'minUpper';
    public const MIN_LOWER_KEY = 'minLower';
    public const MIN_DIGITS_KEY = 'minDigits';
    public const MIN_PUNCTUATION_KEY = 'minPunctuation';
    // More keys here ...

    private $upperIncluded;
    private $lowerIncluded;
    private $digitIncluded;
    private $punctuationIncluded;
    private $ambiguousExcluded;
    private $minUpper;
    private $minLower;
    private $minDigits;
    private $minPunctuation;
    // More fields here ...

    public __construct(array $options = [])
    {
        // Extract the options from the array.
        $this->upperIncluded = getDefault($options, UPPER_INCLUDED_KEY, true);
        $this->lowerIncluded = getDefault($options, LOWER_INCLUDED_KEY, true);
        $this->digitIncluded = getDefault($options, DIGIT_INCLUDED_KEY, true);
        $this->punctuationIncluded = getDefault($options, PUNCTUATION_INCLUDED_KEY, true)
        $this->ambiguousExcluded = getDefault($options, AMBIGUOUS_EXCLUDED_KEY, true);
        // And so on ...
        // More initialization code here ...
    }

    private function getDefault(array $array, string $key, $defaultValue)
    {
        return (isset($array[$key]) || array_key_exists($key, $array)) ?
            $array[$key] : $defaultValue;
    }

    // More methods here ...

}
```

Potentially, we could use this approach to collapse all of our constructor parameters to a single associative array. Values could be specified (or not) in any order in the array, and the resulting

constructor logic wouldn't be affected. Of all of the "constructor tricks" approaches described, this is arguably the best.

However, the API (at least the constructor portion) is less self-documenting than ever. We'd have to write a lot of additional documentation (probably as phpDocumentor comments) to explain how it works.

Approach 2: Using accessors and mutators (getters and setters)

Rather than write a constructor that's complicated in its invocation—or in the implementation code required to make the invocation less complicated—we might instead write a very simple constructor, and use mutators to set the generator options. As is often the case when we use accessors and mutators, this gives us a certain level of encapsulation (generally a good thing), at the expense of boilerplate code.

(Note that the example here doesn't make use of the PHP "magic methods" `__set` and `__get`. These methods can be very useful—though they have serious shortcomings if we have an aim of writing self-documenting code—but they're outside the scope of this introduction.)

```
class Generator
{

    private $upperIncluded;
    private $lowerIncluded;
    private $digitIncluded;
    private $punctuationIncluded;
    private $ambiguousExcluded;
    private $minUpper;
    private $minLower;
    private $minDigits;
    private $minPunctuation;
    // More fields here ...

    public __construct()
    {
        // General initialization code here ...
    }

    public isUpperIncluded(): boolean
    {
        return $this->upperIncluded;
    }

    public setUpperIncluded(boolean $upperIncluded)
    {
        $this->upperIncluded = $upperIncluded;
    }

    public isLowerIncluded(): boolean
    {
        return $this->lowerIncluded;
    }

    public setLowerIncluded(boolean $lowerIncluded)
    {
        $this->lowerIncluded = $lowerIncluded;
    }

    // More getters and setters here ...

    // More methods here ...

}
```

This certainly looks like a good approach. Among other benefits, we could include new configuration options in the future, without modifying the approach. Further, if we ever decide to load configuration options from files, there are libraries that will infer property names from a settings file, and automatically invoke the appropriate mutators. This approach is also much more self-documenting than any of the

constructor-oriented options described above.

On the other hand, there's no guarantee that after some sequence of mutator invocations, the generator is in a suitable state to begin generating passwords; ensuring that would mean making the mutators much more aware of the entire state of the generator than we'd normally like them to be. Further, there's nothing preventing modifications of a generator instance via a mutator, even after we've begun using it to generate passwords. To satisfy the requirements that we and the client agreed to, the generator objects should really be *immutable*—that is, after instantiation and initialization by a constructor, the object state shouldn't be allowed to change. Since the point of a mutator is to change the state of an object, this approach may be a dead end for us.

Approach 3: Constructing immutable objects with a Builder

Let's examine a different approach altogether. Instead of one class with a complicated constructor (which we could use to create an immutable object, at the cost of code that's difficult to document, maintain, and use), or one class with simple constructor with mutators for every configuration option (more self-documenting, but won't produce immutable objects), we'll implement the Builder pattern with 2 classes:

- `PasswordGeneratorBuilder`

  Instances of this class will be mutable objects that aren't themselves password generators, but rather *builders* for password generator objects. This class will have a number of methods for setting the configuration options, e.g.

  - `includeUpper(bool $include = true)`
  - `requireUpper(int $min = 1)`
  - `includeLower(bool $include = true)`
  - `requireLower(int $min = 1)`
  - …

  Each of these option-setting methods will be written to support a *fluent interface*, where the return value of each method on which the method was invoked, so that as many options can be set as needed, in a very direct fashion, via *method chaining*. (The fluent interface for method chaining isn't an intrinsic element of the Builder pattern, but it is often employed as part of the pattern.)

  Most importantly, `PasswordGeneratorBuilder` will have a method to create and return a password generator, with the current set of applied options:

  - `build()`

- `PasswordGenerator`

  This class is the type returned by the `build` method of `PasswordGeneratorBuilder`. The API for this class will be extremely simple, including just 2 public methods:

  - `generate(int $length, $int count = 1)`

    This method will generate and return passwords.

- `builder()`

  This will be a `static` method, creating and returning an instance of **`PasswordGeneratorBuilder`** (or of a subclass of that).

To summarize how these classes will be used: the consumer code will invoke **`PasswordGenerator::builder`** to get an instance of **`PasswordGeneratorBuilder`**; then, after setting options via methods of the latter class, the consumer will use the **`build`** method of that class to get a **`PasswordGenerator`**, and then use the `generate` method of *that* class to generate passwords.

At first glance, this usage choreography probably seems convoluted—and in fact, the Builder pattern doesn't require that we do things exactly this way. But this aspect of the implementation lets us define both of these classes as `abstract` classes, with `protected` constructors. There will be *no way* for consumer code to create an instances of either of these two classes using the **`new`** keyword with a constructor; it will have to use the methods mentioned above—which is exactly what we wanted. Further, since these are both `abstract` classes, some of the more specialized aspects of the processing will be performed by overridden methods in subclasses. Implementing in this fashion should give us a lot of flexibility for further subclassing, as necessary (e.g. for specialized password generation requirements we haven't anticipated yet).

Implementation

The accompanying PHP files contain the implementation of the above classes, along with an example script that demonstrates their use.

- `PasswordGenerator.php`

  Commented source code for the **`PasswordGeneratorBuilder`** and **`PasswordGenerator`** classes.

- `generator_demo.php`

  Script with 3 usage examples.

Summary

When do we use the Builder pattern?

The Builder pattern is useful in a variety of situations, including (but not limited to) any of the following:

- We need to initialize/configure instances of a class that uses composition extensively—i.e. an instance of our class is composed of instances of a number of other classes.
- Initialization of an instance involves several steps performed/invoked by the consumer, and the instance may not be reliably in a "ready" state in the intermediate points along the way.
- Once an instance of our class is fully initialized, we want it to be immutable—but it may take several

steps to inititalize it, and the details of those steps may vary from instance to instance.

- A class has a number of attributes that affect the critical behaviors of the class in non-linear/interacting ways.

How do we implement it?

In general, we need at least 2 abstract classes or interfaces—1 for the builder type, and one for the target object type—and at least 2 concrete classes, extending or implementing the abstract classes or interfaces. (For compactness in the scenario above, our implementing classes are anonymous classes, but it's more common to use named classes for this purpose.) We might also include a "director" class, which invokes the necessary operations on the builder to build an instance of the target class; in many real-world cases, the consumer's code performs this role.

In relatively simple cases, we might take advantage of some of the strengths of the Builder pattern without going to the lengths of defining abstract classes or interfaces—that is, we might simply use 2 concrete classes: 1 for the builder, and 1 for the target object. In fact, this is essentially how the Builder pattern is implemented for the Java `StringBuilder` and `String` class. Keep in mind, however, that using abstract classes or interfaces to declare the API of our builder and target types gives us more flexibility for the future.

The builder class will typically have several methods that configure the components and behavior of the (eventual) target object. (These methods are good candidates for a *fluent interface*, where each method returns the object instance on which the method was invoked, so that we can easily chain multiple invocations of these methods. As noted previously, however, the Builder pattern does not necessarily imply a fluent interface.) It will also have at least 1 method that constructs and returns an instance of the target object type (e.g. our **`PasswordGeneratorBuilder::build`** method). In practice, this method (and typically the entire concrete builder class) will have privileged knowledge of, and access to, the implementation details of the associated concrete target class. (In part, this is why we used anonymous classes for our scenario: by including the the concrete subclass of the target class within the builder class, the latter had privileged access to the former. In PHP, at least as of v7.1, this is the only way to nest the definition of one class within another.)

For creating a builder instance, we'll usually provide either a constructor in the concrete builder class, or a static factory method of the abstract target class (e.g. **`PasswordGenerator::builder`**).