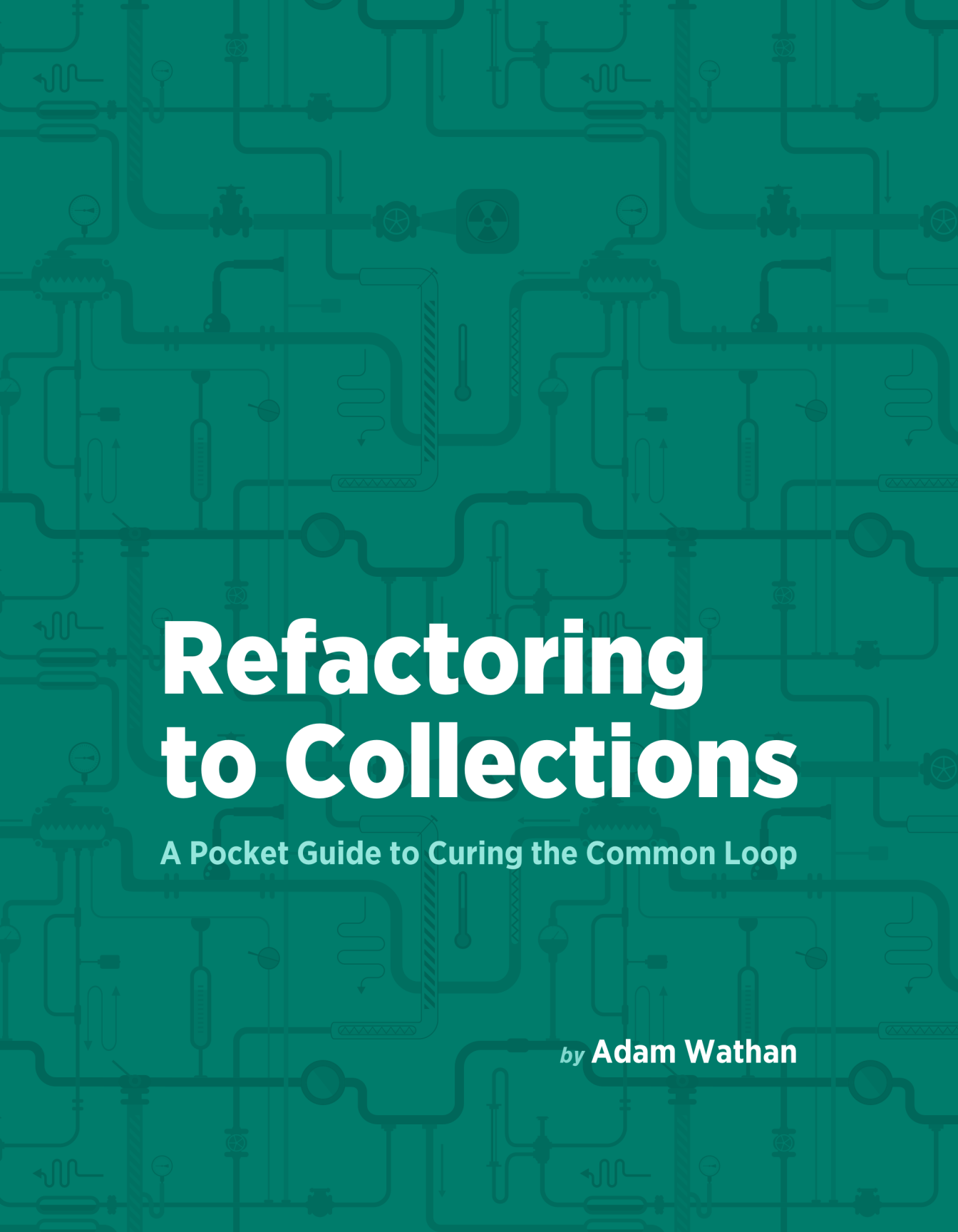

Refactoring to Collections

A Pocket Guide to Curing the Common Loop

by **Adam Wathan**

Contents

Higher Order Functions

A higher order function is a function that takes another function as a parameter, returns a function, or does both.

For example, here's a higher order function that retries a block of code up to `$maxAttempts` number of times:

```
function retry($maxAttempts, $func)
{
    $attempts = 0;

    while ($attempts < $maxAttempts)
        try {
            return $func();
        } catch (Exception $e) {
            $attempts++;
        }
    }

    throw $e;
}
```

And here's what it would look like to use:

```
try {
    $response = retry(5, function () {
        $this->httpClient->get('http://example.com/unreliable.json');
    });
} catch (ServerErrorException $e) {
    echo "Couldn't fetch data from server.";
}
```

Noticing Patterns

Higher order functions are powerful because they let us create abstractions around common programming patterns that couldn't otherwise be reused.

Say we have a list of customers and we need to get a list of their email addresses. We can implement that without any higher order functions like this:

```
$customerEmails = [];

foreach ($customers as $customer) {
    $customerEmails[] = $customer->email;
}

return $customerEmails;
```

Now say we also have a list of product inventory and we want to know the total value of our stock of each item. We might write something like this:

```
$stockTotals = [];

foreach ($inventoryItems as $item) {
    $stockTotals[] = [
        'product' => $item->productName,
        'total_value' => $item->quantity * $item->price,
    ];
}

return $stockTotals;
```

At first glance it might not look like there's much to abstract here, but if you look carefully you'll notice there's only one real difference between these two examples.

In both cases, we create an empty array, iterate over another list of items, create a new item by performing some operation on each item in that list, append each one of our new items to our new array, and then return that array. The only difference is the operation we are performing on each item.

In the first example we're just extracting the `email` field from the item:

```
-   $customerEmails = [];
-
-   foreach ($customers as $customer) {
+       $email = $customer->email;
-
-       $customerEmails[] = $email;
-   }
-
-   return $customerEmails;
```

In the second example, we create a new associative array from several of the item's fields:

```
-   $stockTotals = [];
-
-   foreach ($inventoryItems as $item) {
+       $stockTotal = [
+           'product' => $item->productName,
+           'total_value' => $item->quantity * $item->price,
+       ];
-
-       $stockTotals[] = $stockTotal;
-   }
-
-   return $stockTotals;
```

If we generalize the names of everything except the two chunks of code that are different, we get this:

```
+  $results = [];
+
+  foreach ($items as $item) {
-      $result = $item->email;
-
+      $results[] = $result;
+  }
+
+  return $results;
```

```
+  $results = [];
+
+  foreach ($items as $item) {
-      $result = [
-          'product' => $item->productName,
-          'total_value' => $item->quantity * $item->price,
-      ];
-
+      $results[] = $result;
+  }
+
+  return $results;
```

We're close to an abstraction here, but those two pesky chunks of code in the middle are preventing us from getting there. We need to get those pieces out and replace them with something that can stay the same for both examples.

We can do that by extracting those chunks of code into anonymous functions:

```
+   $func = function ($customer) {
+       return $customer->email;
+   };
-
-   $results = [];
-
-   foreach ($items as $item) {
+       $result = $func($item);
-
-       $results[] = $result;
-   }
-
-   return $results;
```

```
+   $func = function ($inventoryItem) {
+       return [
+           'product' => $item->productName,
+           'total_value' => $item->quantity * $item->price,
+       ];
+   };
-
-   $results = [];
-
-   foreach ($items as $item) {
+       $result = $func($item);
-
-       $results[] = $result;
-   }
-
-   return $results;
```

Now there's a big block of code in both examples that is exactly the same and ready to be extracted. If we bundle that up into its own function, we've implemented a higher order function called `map`!

```
function map($items, $func)
{
    $results = [];

    foreach ($items as $item) {
        $results[] = $func($item);
    }

    return $results;
}

$customerEmails = map($customers, function ($customer) {
    return $customer->email;
});

$stockTotals = map($inventoryItems, function ($item) {
    return [
        'product' => $item->productName,
        'total_value' => $item->quantity * $item->price,
    ];
});
```

Functional Building Blocks

Map is just one of dozens of powerful higher order functions for working with arrays. We'll talk about a lot of them in later examples, but let's cover some of the fundamental ones in depth first.

Each

Each is no more than a `foreach` loop wrapped inside a higher order function:

```
function each($items, $func)
{
    foreach ($items as $item) {
        $func($item);
    }
}
```

You're probably asking yourself, *"why would anyone bother to do this?"* Well for one, it hides the implementation details of the loop *(and we hate loops.)*

Imagine a world where PHP didn't have a `foreach` loop. Our implementation of `each` would look something like this:

```
function each($items, $func)
{
    for ($i = 0; $i < count($items); $i++) {
        $func($items[$i]);
    }
}
```

In that world, having an abstraction around "do this with every item in the array" seems pretty reasonable. It would let us take code that looks like this:

```
for ($i = 0; $i < count($productsToDelete); $i++) {
    $productsToDelete[$i]->delete();
}
```

...and rewrite it like this, which is a bit more expressive:

```
each($productsToDelete, function ($product) {
    $product->delete();
});
```

Each also becomes an obvious improvement over using `foreach` directly as soon as you get into chaining functional operations, which we'll cover later in the book.

A couple of things to remember about `each`:

- If you're tempted to `use` any sort of collecting variable, `each` is not the function you should be using.

```
// Bad! Use `map` instead.
each($customers, function ($customer) use (&$emails) {
    $emails[] = $customer->email;
});
```

- Unlike the other basic array operations, `each` doesn't return anything. That's a clue that it should be reserved for *performing actions*, like deleting products, shipping orders, sending emails, etc.

```
each($orders, function ($order) {
    $order->markAsShipped();
});
```

Map

We've talked about map a bit already, but it's an important one and deserves its own reference.

Map is used to *transform* each item in an array into something else. Given some array of items and a function, map will apply that function to every item and spit out a new array of the same size.

Here's what `map` looks like as a loop:

```
function map($items, $func)
{
    $result = [];

    foreach ($items as $item) {
        $result[] = $func($item);
    }

    return $result;
}
```

Remember, every item in the new array has a relationship with the corresponding item in the original array. A good way to remember how `map` works is to think of there being a *mapping* between each item in the old array and the new array.

Map is a great tool for jobs like:

- Extracting a field from an array of objects, such as mapping customers into their email addresses

```
$emails = map($customers, function ($customer) {
    return $customer->email;
});
```

- Populating an array of objects from raw data, like mapping an array of JSON results into an array of domain objects

```
$products = map($productJson, function ($productData) {
    return new Product($productData);
});
```

- Converting items into a new format, for example mapping an array of prices in cents into a displayable format

```
$displayPrices = map($prices, function ($price) {
    return '$' . number_format($price * 100, 2);
});
```

Map vs. Each

A common mistake I see people make is using `map` when they should have used `each`.

Consider our `each` example from before where we were deleting products. You could implement the same thing using `map` and it would technically have the same effect:

```
map($productsToDelete, function ($product) {
    $product->delete();
});
```

Although this code works, it's semantically incorrect. We didn't `map` anything here. This code is going to go through all the trouble of creating a new array for us where every element is `null` and we aren't going to do anything with it.

Map is about transforming one array into another array. If you aren't transforming anything, you shouldn't be using `map`.

As a general rule, you should be using `each` instead of `map` if any of the following are true:

1. Your callback doesn't return anything.
2. You don't do anything with the return value of `map`.
3. You're just trying to perform some action with every element in an array.