Crafting Quality Software

Gems From The Qafoo Blog

Benjamin Eberlei, Kore Nordmann,
Manuel Pichler & Tobias Schlitt

Published by Qafoo GmbH

Published by Qafoo GmbH
Version 2.0.0; Published June 29, 2017
https://qafoo.com/ | <contact@qafoo.com>

Contents

1. Introduction

1.1 About This Book

This book is a curated collection of blog posts from the Qafoo Team Blog[1]. Over the time we created many blog posts focussing on the topics of Clean Code, Object Oriented Design, Testing, Refactoring and Software Architecture. To make it easier for you to consume those blog posts we re-structured them and collected them in this book.

Many of the topics in this book are covered in our trainings and you can use this book as a guide after those trainings to read more about topics you particularly care about or topics not covered in your training – since we tailor our trainings to the concrete requirements each customer has.

If you are interested in further information on any of the topics in this book get in contact[2] with us or directly request one of our workshops[3] to be held on-site, just for you.

When reading through this book remember that you read blog posts. For every topic you find a link to the original post where you can leave remarks, the author and the original publishing date. We'd love to hear back from you on our blog.

[1] https://qafoo.com/blog
[2] https://qafoo.com/contact
[3] https://qafoo.com/services/workshops

1.2 About Qafoo

Qafoo combines deep knowledge with passion for software quality. It is our goal to impart this passion to our customers through consulting, coaching and workshops. This enables our customers to create amazing software with sustainable quality.

All experts from Qafoo do not only have a solid training in their field but also extensive experience from practical application of their knowledge. All of us successfully develop open source and commercial software since years.

If you like to get more information about Qafoo we suggest you to visit the following pages:

- Our Customers[4]
- Possible Workshops[5]
- Presentations at conference[6]

You are, of course, always welcome to get in contact with us[7].

[4] https://qafoo.com/clients
[5] https://qafoo.com/services/workshops/workshops
[6] https://qafoo.com/resources/presentations
[7] https://qafoo.com/contact

1.3 The Authors

The team members from left to right:

1.3.1 Kore Nordmann

Kore Nordmann has a a university degree in computer science and extraordinary broad experience as a software developer in the professional and open-source PHP projects. Based on this unique mix of theoretical and practical knowledge, he supports teams to take the right path in critical software design and architecture decisions. Kore is one of the founders of Qafoo GmbH where he works as an expert consultant and trainer.

1.3.2 Tobias Schlitt

Tobias Schlitt has a degree in computer science and works in professional PHP-based web projects since 1999. As an open source enthusiast, he contributed to various community projects. Tobias is co-founder of Qafoo GmbH which helps development teams to produce high-quality web application in terms of expert consulting and individual training. Tobias main focus is on software architecture, object oriented design and automated testing.

1.3.3 Benjamin Eberlei

Benjamin works at Qafoo GmbH where he is teaching and consulting on software-quality, refactoring and architecture. Recently he has also founded the Tideways company, which offers performance monitoring, profiling and exception tracking solutions for PHP. As a project leader of Doctrine he is mentoring the next generation of ORM developers.

1.3.4 Manuel Pichler

Manuel Pichler is a software architect with extraordinary practical experience basis. As the creator of some of the most widely used PHP quality assurance tools (PHPMD, PHP_Depend and Ant Build Commons) he has a vivid passion on static code analysis, software metrics and other quality assurance related technologies. Manuel works as an expert consultant and trainer at Qafoo GmbH, of which he is also a co-founder.

2. Clean Code

2.1 Developers Life is a Trade-Off

Kore Nordmann at 27. May, 2015[1]

At Qafoo, we train a lot of people on topics like object oriented software design, automated testing and more. It happens quite often that an attendee asks questions like "Which is the best solution for problem class $x?", "What is the optimal tool for task $y" or "There is a new technology $z, is that the future of web development?". Some are disappointed when I reply "It depends" or "That does not exist", but that's the truth.

There is no silver bullet and one of the most important skills every developer needs to hone is to assess possibilities and to find the best trade-off for the current challenge.

To make that point clear I'm giving three examples from my personal experience, some where it went well and some where it did not.

2.1.1 The NoSQL Dilemma

Choosing a storage system is probably the most common decision to be made in any web project. For one project we already knew that there will be a large amount of data, so we started RnD on recent storage solutions and NoSQL was the big

[1] https://qafoo.com/blog/075_developers_life_trade_off.html

topic then. Cassandra appeared to suite our needs perfectly from its description. So we gave it a try and prototyped against it, using a thin class layer that hid the database detail.

It was a wise decision to create that layer whereas going with Cassandra from the start was not a very good one. While the database met our expectations, it crashed multiple times with loosing data in our development VMs and none of us had a clue, why, or more importantly, how to fix it. So we revised that decision and eventually went with MySQL and a denormalized schema, where only important attributes became dedicated table fields and the main data structure was serialized JSON in blob. That worked fine for several years.

What did we do? We found the solution which promised to fit our requirements quite well. Luckily, we did not jump blindly on it, but kept it a bit away from the system and evaluated further. Constraints like "knowledge on maintenance" were eventually considered and so we made a trade-off between these requirements that pushed us in contrary directions.

2.1.2 Overengineered State Machines

You don't need to jump on such a high-level bandwagon to find examples on where you need to make trade-offs. In one project I created a component to match our business model against a 3rd party one to use their data in our system. While the models were similar, a fundamental difference was how it was determined when data was published and deleted.

That problem appeared perfect for applying a state machine to it to trigger changes on our side when the 3rd party model changed. So I went the extra mile by creating an abstract state machine with `Node` and `Transition` classes. A transition was triggered through an abstract `Condition` and a node fired an `Event` which eventually triggered that change in our model. It was beautiful code in my eyes, with really small classes (1-2 lines executable code). To "ease" configuration and since I expected frequent changes, I made it configurable in XML.

Some time later, a co-worker needed to check if the implemented logic still met new requirements. Digging through that, starting from the XML configuration, running through so many classes to get the complete picture of what it did, took

him a large amount of time. It turned out that the logic was fine, but the amount of time spent re-occured some more times after that experience.

A year later, we needed to finally adjust the logic. But the model did not fit the new requirement well, so that we needed to implement many additional classes and the XML configuration became even more complex.

What went wrong? I did not evaluate all available options and pick a good trade-off for the situation. As a minimal solution, I could have hidden the state processing behind a class and code it straight with nested `if` conditions. Maybe that would have been 40 lines of code, but a good bunch of unit tests could have covered that. Maybe a solution in between, abstracting only the states and hard-coding the trigger and transition logic, would have been the optimal trade-off.

So, what's the moral of the story? Whenever you take a software design decision in your project, there is a ton of possible solutions. Just picking a random, interesting, clean, ... one is most probably not the right choice. Instead, you need to check your actual constraints and then find the best trade-off between the possibilities. Which one that is can vary greatly.

2.1.3 Hack Hack Hack

There are times in a project, where the option of just hacking a problem solution into the core of the system looks viable. Be warned explicitly: only do that with great care and foresight. In many environments that can lead to stacked up technical debt and possibly result in a really hard to maintain project.

In that project we had an import workflow which performed time-costly analysis on the imported data. It worked well as the incoming chunks of data were small. But then a party registered which provided huge chunks of data and the system became unresponsive. Not only that their imports were delayed, but it affected all other involved parties, too. On the other hand, that big-chunk-party did not even need all the analysis we performed on their data.

A clean solution would have been to create a configuration flag for the kinds of analysis. Another one would have introduced sharding for the import process, that was already in the backlog with low priority. Both solutions would have required too much time, we needed to fix the actual problem immediately. So we hacked a

condition into a prominent place in the core, skipping the expensive analysis for the big-chunk-party.

I had an eye on that code quite for a long time, being prepared to refactor it whenever viable. But there was no need. The code worked fine, nobody needed to touch it again and the problem was solved. Eventually, the expensive analysis was not necessary at all anymore, so we removed it and with it the ugly hack.

What did we actually do? We made a trade-off. That one was very much in one of the possible directions, but because we knew of the potential problems and were prepared to revise the decision, we could take the risk.

2.1.4 Bottom Line

One of the most important tasks of a developer is to make trade-offs. They occur wherever you look in your every day life. It is a highly important step to realize and accept this. And it is important to hone that skill. You need to open your mind for new technology and techniques, learn and try them wherever you can. But then you need to step back, analyze the current situation and then find the best trade-off between all possible approaches. In addition, you need to be able to reflect your decisions and be prepared to revise them. The plus side: by doing so, you will surely learn something for all the upcoming trade-offs.

2.2 Never Use `null`

Kore Nordmann at 3. May, 2016[2]

When doing code reviews together with our customers we see a pattern regularly which I consider problematic in multiple regards – the usage of `null` as a valid property or return value. We can do better than this.

Let's go into common use cases first and then discuss how we can improve the code to make it more resilient against errors and make it simpler to use. Most issues highlighted are especially problematic when others are using your source code. As long as you are the only user (which hopefully is not the case) those patterns might be fine.

2.2.1 What it is Used For

One common use case is setter injection for optional dependencies, like:

```
class SomeLoggingService {
    private $logger = null;

    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }
}
```

The logger will be set in most cases but somebody will forget this when using your service. Now a second developer enters the scene and writes a new method in this class using the property `$logger`. The property is always set in the use cases they test during development, thus they forget to check for `null` – obviously this will be problematic in other circumstances. You rely on methods called in a certain order which is really hard to document. An internal `getLogger()` method constructing a default null logger might solve this problem, but it still might not be used because the second developer wasn't aware of this method and just used the property.

In PHP versions < 7 a call to `$this->logger->notice(...)` will result in a `Fatal Error` which is particularly bad since the application can't handle this kind of errors in a sane way. In PHP 7 those errors are finally catchable but still nothing you'd expect in this situation.

[2] https://qafoo.com/blog/083_never_use_null.html

What is even worse is *debugging* this kind of initialization. This is often even used together with aggregated objects which are *required* by the aggregating class. (You should not use setter injection for mandatory aggregates, but it is still used this way.) Let's consider the following code now:

```
class SomeService {
    public function someMethod() {
        $this->mandatoryAggregate->someOtherMethod(/* ... */);
    }
}
```

When calling `someMethod()` and the property `$mandatoryAggregate` is not initialized we get a fatal error, as mentioned. Even if we get a backtrace through XDebug[3] or change the code to throw an exception and get a backtrace it is still **really hard** to understand why this property is not initialized since the construction of `SomeService` usually happens outside of the current callstack but inside the Dependency Injection Container or during application initialization.

The debugging developer is now left with finding all occurrences where `SomeService` is constructed, check if the `$mandatoryAggregate` is properly initialized and fix it, if not.

The solution

All mandatory aggregates **must always** be initialized during construction. If you want a slim constructor consider a pattern like the following:

```
class SomeService {
    public function __construct(Aggregate $aggregate, Logger $logger = null) {
        $this->aggregate = $aggregate;
        $this->logger = $logger ?: new Logger\NullLogger();
    }
}
```

The parameter `$aggregate` now is really mandatory, while the logger is optional – but it will still always be initialized. The `Logger\NullLogger` now can be logger which just throws all log messages away. This way there is no need to care about checking the logger every time you want to use it.

Use a so called null object if you need a default instance which does nothing. Other examples for this could be a null-mailer (not sending mails) or a null-cache

[3] https://xdebug.org/

(does not cache). Those null objects are usually really trivial to implement. Even it costs time to implement those you'll safe a lot time in the long run because you will not run in `Fatal Errors` and have to debug them.

2.2.2 `null` as Return Value

A similar situation is the usage of `null` as a return value for methods which are documented to return something else. It is still commonly used in error conditions instead of throwing an exception.

It is, again, a lot harder to debug if this occurs in a software you use but you are not entirely familiar with. The `null` return might pass through multiple call layers until it reaches your code which makes debugging that kind of code a journey through layers of foreign and undiscovered code – sometimes this can be fun but almost never what you want to do when in a hurry:

```
class BrokenInnerClass {
    public function innerMethod() {
        // ...
        if ($error) {
            return null;
        }
        // ...
    }
}

class DispatchingClass {
    public function dispatchingMethod() {
        return $this->brokenInnerClass->innerMethod();
    }
}

class MyUsingClass {
    public function myBeautifulMethod() {
        $value = $this->dispatchingClass->dispatchingMethod();
        $value->getSomeProperty(); // Fatal Error
    }
}
```

Usually there are even more levels of indirection, of course. We live in the age of frameworks after all.

The solution

If a value could not be found do not return `null` but throw an exception – there are even built in exceptions for such cases like the `OutOfBoundsException`, for example.

In the callstack I can see immediately where something fails. In the optimal case the exception message even adds meaning and gives some hints of what I have to fix.

2.2.3 Summary

Using `null` can be valid inside of value objects and sometimes you just want to show nothing is there. In most cases `null` should be either replaced by throwing an exception or providing a null object which fulfills the API but does nothing. Those null objects are trivial and fast to develop. The return on investment will be **huge** due to saved debugging hours.

2.3 Struct classes in PHP

Kore Nordmann at 24. January, 2011[4]

PHP arrays are a wonderful tool and one of the reasons I like PHP. Their versatility makes it possible to easily set up proof of concepts (POC), either used as hash maps storing multiple keys, or as lists, stacks, trees or whatever you like.

But once you are past the phase of the initial POC, the excessive usage of arrays and exactly their versatility has some drawbacks: If you see an `array` type hint or return documentation, you know nearly nothing about the data structure. Using arrays as key-value hash maps for storing configuration keys or data sets you also know nearly nothing about the expected contents of the array.

This is no problem during the initial implementation, but can become a problem during maintenance - it might not be trivial to find out what the array contains or is supposed to contain (without dumping it). There are no common ways to document such array structures nor you get auto-completion from common IDEs. If such a hash map is filled with data in different locations in your application it even gets worse. Also, mistyping a key - wether on read or write - creates a serious debugging hell.

In Apache Zeta Components and in several of my own projects we are using - so called - struct classes to solve this issue: The struct classes do not define any methods but just contain documented properties. They just deal as a data container, similar to a hash map.

There are several benefits and one drawback using this approach. The benefits:

- Struct classes are far easier to document
- Your IDE can provide you with correct auto-completion
- Your IDE even knows the type of each child in a struct allowing you to create and process deeply nested structures correctly
- You can be sure which properties a passed struct has - no need to check the availability of each property on access
- Structs can throw exceptions access to non-existent properties

The drawback:

[4] https://qafoo.com/blog/016_struct_classes_in_php.html

- The structs are objects, which means they are passed by reference. This can be an issue if you are operating on those structs. I will show an example later.

2.3.1 Implementation

To see what I am talking about let's take a look at a example base class for structs:

```
<?php

abstract class Struct
{
    public function __get( $property )
    {
        throw new RuntimeException( 'Trying to get non-existing property ' .
            $property );
    }

    public function __set( $property, $value )
    {
        throw new RuntimeException( 'Trying to set non-existing property ' .
            $property );
    }
}
```

In a struct base class you can implement `__get()` and `__set()` so they throw an exception if an unknown property is accessed. For me PHPs behavior of silently creating public properties on property write access caused quite some irritations over time. A typo in a property name and your code does strange things. I like to get a warning or (even better) an exception for that. Now, let's take a look at a concrete struct:

```
<?php

class LocationStruct extends Struct
{
    /**
     * @var string
     */
    public $city;

    /**
     * @var string
     */
    public $country;
```

```
    public function __construct( $city = null, $country = null )
    {
        $this->city    = $city;
        $this->country = $country;
    }
}
```

The `LocationStruct` has two documented, public properties. Each one, of course, could be a struct again. If the `LocationStruct` is used as a type hint somewhere in your application or library you now know exactly what data is expected and can create a it comfortable, supported by your favorite IDE. The definition of a constructor is really helpful to easily create new struct instances.

Extending the base struct

There are some sensible extension you probably want to use for the base struct: As mentioned before the structs are passed by reference, which is not always what you want. You therefore probably want to implement `__clone()` in a sensible way, generically for all your structs:

```
<?php

abstract class Struct
{
    // ...

    public function __clone()
    {
        foreach ( $this as $property => $value )
        {
            if ( is_object( $value ) )
            {
                $this->$property = clone $value;
            }
        }
    }
}
```

Another functionality you might want to implement, and a good use case of late static binding[5] (LSB) in PHP 5.3, is the `__set_state()` method, so you can export your struct using `var_export()` just like arrays:

[5] https://qa.fo/book-language_oop5_late-static-bindings

```
<?php

abstract class Struct
{
    // ...

    public static function __set_state( array $properties )
    {
        $struct = new static();

        foreach ( $properties as $property => $value )
        {
            $this->$property = $value;
        }

        return $struct;
    }
}
```

If you are using `__set_state()` to ex- and import structs in your application, this is a good reason to define sensible default values for all constructor arguments.

2.3.2 Copy on write

As mentioned before, one problem with this usage of struct classes is that they are always passed by reference. It is not entirely obvious why this would be a problem, but it already caught me some times, so here is a example.

In the Graph component from the Apache Zeta Components[6] we, for example, use a struct class to represent coordinates (`ezcGraphCoordinate`). Obviously there are quite some calculations to perform when rendering (beautiful) charts.

Now imagine you want to draw a set of circles at increasing offsets:

```
$offset = new ezcGraphCoordinate( 42, 23 );
for ( $i = 0; $i < $shapeCount; ++$i )
{
    $driver->drawCircle( $coordinate, 10 );

    $offset->x += 15;
}
```

The `drawCircle()` method now might perform additional calculation on the passed coordinate, for example, because the currently used driver does not use the cen-

[6] http://zetac.org/Graph

ter point, but the top left edge of the circle as a drawing offset. In this case the method might internally modify the coordinate and thus the offset in the shown loop would also be modified. Hopefully you got tests for this in place and therefor add a `$offset = clone $offset` in the `drawCircle()` method. This hit me **very** seldomly until now, but it might be an issue you should be aware of when using struct classes.

2.3.3 Summary

Even requiring slightly more work when writing software, the benefit of struct classes during the maintenance phase of projects makes them a true winner - in my personal opinion.

For POCs I tend to still use arrays for structs, but once the software reaches production quality I tend to convert array structs into struct classes since some time in the software I write / maintain.

In C#, for example, such struct classes are a language element and differ from common object exactly in the copy-on-write vs. pass-by-reference behaviour mentioned in this post. I would love to see that in PHP but my knowledge of the Zend Engine is limited and maybe I should bribe a more experienced PHP internals developer...

Final note

There are other ways to implement struct classes, like using a properties array instead of public properties, which enable you to perform type checks on property write access. Those might be discussed in another blog post but would exceed the purpose of this blog post.

3. Object Oriented Design

3.1 Learn OOD - to unlearn it again

Tobias Schlitt at 11. February, 2014[1]

One topic we regularly teach in workshops for our customers is *object oriented design* (ODD), i.e. the art of crafting classes/interfaces in a way that the result is an easy-to-understand, maintainable and flexible code base. With the agile focus on shipping working software, some might consider the skill of OOD less important. One popular argument is that quick reaction to change is more important than designing objects and their interaction carefully. I personally look at it the exact other way around. This blog post summarizes why you need to learn OOD first, in order to avoid it to some degree again.

3.1.1 Learning OOD the classical way

Back in the days where the waterfall project methodology was most wide spread, object oriented design was a primary skill for software developers: After analyzing the tremendous functional specification, lots of class diagrams were created, itemizing each and every object in order to provide a full way navigation for the actual implementors.

[1] https://qafoo.com/blog/064_learn_ood_to_unlearn_it.html

Developers needed to understand interaction between objects in perfect detail, modeling business processes to their long tail and infrastructure alike. Applying a design pattern wherever possible in a UML diagram was considered the holy grail in many software development projects back then.

Junior developers those days were left to do the actual implementation of the upfront design, occasionally leaving a small component unspecified to allow them to try out the skills they learned from watching the seniors' design.

One of the results of this way to handle projects was a massive risk to over-engineer the system, taking away the flexibility for spontaneous change requests and leading to a long time to market. These are two of the reasons that led to the agile movement, from a technical perspective - leaving the even more important social aspects aside here.

3.1.2 OOD in fast pace and agile

With the rise of agile, the way we handle projects changed drastically: Instead of creating a full system design upfront, teams start with a small set of features, rolling these out to the user as fast as possible and iteratively refining the software together with the customer. Shipping working software is one of the values in the agile manifesto, while clean code is not.

While not directly connected, agility and time to market is also where PHP and JavaScript have their strengths as a development platform. Both languages do not enforce a compelling clean code structure, but leave you the option to implement a hack in order build prototypes quickly. This is both their biggest strength and weakness at the same time.

As can be seen in many projects, working at the fast feature pace often led to *big ball of mud* (BBOM) code bases, which reduced the agility of the project day by day, eventually leading to the stage where changes are costly and developers demanded yet another re-write to free themselves of legacy hell. This is what can be described as *under-engineering*.

Luckily, the area of automatic testing and clean code rose alike with the demand for agility, restraining us from just hacking away, in the best case. And especially unit testing requires you to slice your code carefully into fine-grained objects, which in turn requires a deep understanding of OOD.

3.1.3 Refactoring is the key

The solution to the mismatch of fast change and clean code is the art of refactoring. Instead of creating a full blown object design upfront, we nowadays let the code flow for some time until we realize that there is a need for cleanup. We then hopefully take the time to rip the created code apart carefully and to restructure it into a nicely crafted object structure, without moving into the direction of over-engineering.

But being able to refactor requires again a deep knowledge of how structures can be crafted in a clean and maintainable way, how objects should interact to stay flexible. In addition, a particularly good test coverage is needed to enable that process of changing the underlying structure of a system without breaking its business processes (refactoring).

3.1.4 Learning OOD to unlearn it

Today's software projects require us to react flexible to emerging changes. This can be achieved by letting the object design of a system emerge instead of attempting to craft it upfront. Automatic testing and continuous refactoring are the key methodologies to master this balance act. Both topics require a deep knowledge about how objects relate to each other, about how to craft them in a sensible way and how to slice the code to extract dependencies. But this is essentially what we call *object oriented design*.

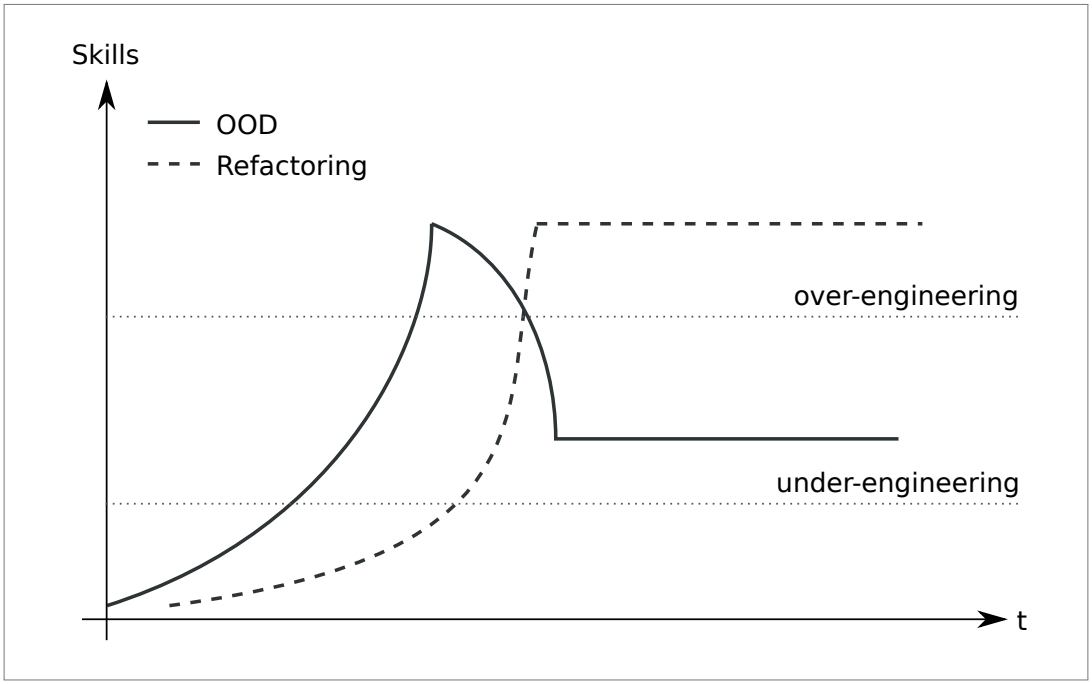
This comprehension leads to what we experienced with many development teams during the past years: in order to enable a team to follow the described development path, knowledge about clean code and various topics object oriented design (like SOLID and design patterns) is an essential basis. Otherwise, the developers will not be able to test responsibly and to perform refactorings successfully.

Honing OOD skills requires a pretty good deal of training and practice. Experimenting with object relations, trying to get it right from scratch and iterating learning cycles are absolutely necessary. But once a developer moves in this direction, evidence shows that he automatically begins to over-engineer his code. It is natural and important to undergo this phase of learning, because it is the only way to build up experience and finally acquire a intuitive understanding of how objects relate to each other.

Once this level of perception is reached, the developer needs to restrain himself again. He needs to *unlearn* parts of what he learned before, in order to let the code flow in a controlled manner, without planning each and every detail and without creating a large object oriented design upfront.

"Unlearn" is of course not the ideal term here, because forgetting the knowledge would be a disaster. Furthermore, refactoring the code requires exactly that know-how and experience. Assessing which code can safely be let slide until refactoring is required and encapsulating that code behind a thin and safe abstraction is essential in order to not end up in a BBOM. Especially for being able to apply a sensible test mix (which is another complex topic in itself) the latter point is highly important.

So there is no way around that learning process, which is illustrated in the following graphic:



Of course, undergoing this process takes time, and failures are required to learn properly. Therefore, junior developers need proper guidance from good seniors -

on technical, social and mentoring levels - to go through that process on the job. Pair programming, coding dojos, code retreats and code review can support this process and speed it up to some degree.

3.1.5 Conclusion / TL;DR

In order to master agility, fast feature pace and high quality code at the same time, every developer needs to learn the art of refactoring and automated testing. This in turn requires a high skill level in object oriented design, which can only be reached through a good deal of training and practice. Undergoing a phase of over-engineering during this learning process is natural and even desired to enable the developer to deliver his full potential.

Qafoo experts can help you with a sound training concept[2] to guide your developers safely on their individual way, giving them a well designed mixture of theoretical knowledge, participation and practice. We can furthermore provide sustainable mentoring for your team, providing them with expert advice and constructive feedback, enabling them to take the shortest possible path to becoming a professional.

[2] https://qafoo.com/services/training.html

3.2 Object lifecycle control

Kore Nordmann at 5. April, 2011[3]

From time to time I trip over APIs, which do not allow me to control the object lifecycle of the used objects myself. This is a bad thing, because it breaks with several concepts of object oriented programming, and forces you back into the dark ages of class oriented programming[4].

The problem I am talking about is that the API expects a class name instead of an instance (object). The PHP stream wrapper API, for example, let you register a class for a certain schema:

```
stream_wrapper_register( 'virtual', 'myVirtualFileSystemHandler' );
```

In the example above a new object of the class `myVirtualFileSystemHandler` would be created for each file system operation, by the internal stream wrapper implementation.

Such class based plugin mechanisms are not uncommon, but do have several drawbacks I want to outline here - and also provide you with solutions to this very problem.

3.2.1 Why is this bad?

The main problem with all class-name based APIs is, that it is not possible to inject further dependencies into the objects resulting from the class name dependency.

Let's examine this in further detail: When we only pass a class name for dependency injection, there are two ways the component in question is using our class:

1. It only calls static methods (this is bad[5]).
2. It creates the object internally, itself.

If the object is created internally, the user of the component is not able to inject additional dependencies into the object just created.

Depending on the API we are interfacing with, the object to be created could for example need a database connection or something alike. The user of the API now only has one choice: Introduce a global state, so that the class can access

[3] https://qafoo.com/blog/020_object_lifecycle_control.html
[4] https://qa.fo/book-0103_static_considered_harmful
[5] https://qa.fo/book-0103_static_considered_harmful

this global state to fetch a database connection. Since the object is not provided explicitely with the dependency, there is no way but fetching it from some globally known place.

The global state can either be a static property, singleton, registry or something alike - all are global states, and introduce various problems, which are discussed elsewhere[6].

To summarize: Class based APIs force the user to create a global state in his / her application, if he / she wants to do something non-trivial with your API. It breaks dependency inversion.

3.2.2 How can I solve this?

In most cases it should be sufficient to let the user pass an object instead of a class name. If you call methods on that object which require data generated by your code, you can pass this data as an argument to the called method.

Here we get to something different - a sometimes difficult descision: When should I pass something as an argument to a method, and when should I pass dependencies to the constructor of an object? A good rule of thumb is: If an object requires something to exist and cannot live without it, then use the constructor. For everything else use method arguments.

This should solve most issues, but for example the PHP stream API does not just use one object, but wants to create new objects for all kind of use cases. Everytime you access a stream URL somehow, a new object needs to be created for that path. (The constructor is only called sometimes, btw. For example a stat() call on URL does not trigger the constructor, but you get a new object.)

There we have the situation that a parameter is vital to the object (the URL). The stream wrapper API needs to construct multiple objects and cannot cope with a single instance we can inject upfront. Therefore you currently specify a class name and the stream-wrapper implementation creates needed instances of this class for you. If you now want to maintain connections (HTTP, FTP) or maybe emulate a file system in memory (PHP variables), you need to introduce a global state, because there is no way to inject any state into those internally constructed objects.

[6] https://qa.fo/book-0103_static_considered_harmful

The better way for the stream-wrapper-API would be to require an **instance** of a factory to be registered for some URL schema. The API can then request a new object for a given path from the factory. While the factory needs to follow a certain interface, the implementation is left entirely to the user, who can decide to pass additional other dependencies to the created object - like a connection handler, or a data storage.

The user can even decide to re-use objects, if this is really desired. A typical use for this would be some kind of *identity management* inside of the factory: Reusing the same object for the same path. (Note that this might not make sense in case of the stream wrapper.)

3.2.3 Conclusion

From this elaboration you can learn one very important rule for your API design: *Do not create class based APIs, since they force everybody using your API to create a global state, sooner or later.* Allowing to pass an object or a factory keeps the user of your API in control of the lifecycle of his objects. He / she can act far more flexible then.

3.3 Ducks Do Not Type

Kore Nordmann at 11. July, 2013[7]

Even in ecosystems which generally follow a high standard of code quality I tend to find public methods in classes which do not originate from an interface or an abstract class. I think that this is a really bad habit for several reasons I will explain below. To give this some context, let's start with a simple example:

```
abstract class HttpClient
{
    public function request($method, $path, $body = null, array $headers = array()
        );
}

class MyHttpClient extends HttpClient
{
    public function request($method, $path, $body = null, array $headers = array()
        )
    {
        // ...
    }

    public function setConfiguration($key, $value)
    {
        // ...
    }
}
```

We have an abstract base type for HTTP clients (which could also be an interface (See: Abstract Classes vs. Interfaces)) and one implementation. Implementations then sometimes tend to define additional public methods, like `setConfiguration()` in this case, for various different reasons. I want to explain why **I consider this a code smell**.

Semantically, the main point behind abstract classes and interfaces is to build an abstraction other components can implement against. In practice, abstract classes are also often used for code-re-use, but this is also a bad practice and might be a topic of a different blog post.

To stay with the example above, any other component in the application, which wants to use an HTTP client, defines a dependency on the abstract type `HttpC`

[7] https://qafoo.com/blog/050_ducks_do_not_type.html

`lient`. The actually implementation used would then depend on the application configuration. This is the basic idea of *Dependency Inversion* (S.O.L.I.D.), or to say it with the words of Martin Fowler:

> A. High-level modules should not depend on low level modules. Both should depend on abstractions.
>
> B. Abstractions should not depend upon details. **Details should depend upon abstractions.**

Abstractions can be both: interface definitions or abstract classes.

3.3.1 Duck Typing

Then there is *Duck Typing*. Let's start the explanation with the opposite of Duck Typing: Java.

If you pass something to any method in Java, you define a type hint for the value you are expecting. If the type hint is on `HttpClient`, the compiler verifies that you only use methods / properties defined by that interface. You cannot (should not) use the concrete implementation as something else. It does not matter which concrete instance is passed to the method.

When starting to develop with Java I thought this is horribly annoying. And I still think it is – more on that later.

Duck Typing on the other hand means, that you can use anything passed to your method as anything. You usually would check if a method exists and just call it right away. You do not enforce a base type or a formal interface definition.

In the example above this would mean, that anything which can act as a HTTP client would implement a method `request()` and you just go and use those classes as a HTTP client. The underlying problem here is, of course, that entirely different concepts also might define a `request()` method, which then might break a lot.

3.3.2 Prototyping

When developing prototypes (not talking about prototype based object orientation, even though it also gets interesting there) or Proof Of Concepts the concept of Duck

Typing is extremely important. And this is one of the reasons I usually prefer PHP over Java.

In PHP you can pass stuff around as you like and use the full power of duck typing. Want to see if a couple of components work together well and implement your business requirement? Want to play with a new library? Want to hack a feature seconds before a release date? Just do it.

BUT: If you are developing infrastructure components, Open Source libraries or any code other developers depend on: Please design and use your abstraction as if a small Java Compiler sits on your shoulder and pours hot coffee over you anytime you misuse an object.

It is the only way to make code extensible by others. A fact, even advocates of Duck Typing are aware of is that you must know your software really well, if you are using Duck Typing. If synonyms occur in method / property names, you can easily get code behaving in really strange ways. Also you cannot easily identify your ducks, since there is no clearly documented concept behind it. Wikipedia[8] says:

> In essence, the problem is that, if it walks like a duck and quacks like a duck, it could be a dragon doing a duck impersonation. One may not always want to let dragons into a pond, even if they can impersonate a duck.

3.3.3 Using Foreign Code

The *Open Closed Principle* (S.O.L.I.D.) wants us to change the behavior of our application without modifying code. This requires code to define abstractions which we can replace in the application configuration.

When using type hints on abstractions in your library you make the user of your code assume that she / he can just implement a new class based on this abstract concept and be done. If your code then calls additional methods it will just break. If it breaks immediately one can fix the code, but often enough it happens that the code will only break in obscure error conditions. And since a call to an undefined method even triggers a fatal error this will kill PHP immediately.

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

And often enough I want to replace such implementations / dependencies in foreign libraries. The most common thing is, that I want your library to use my Logger through an adapter, or anything similar. If you then call undocumented methods on that logger (in a special error condition)...

3.3.4 Package Visibility

There is one case where public methods are OK, even if they are not defined by a base concept: PHP misses package visibility. It occurs seldom "enough" (to me), but sometimes you want "internal" access to methods on a package level, even if this might not be part of the external API. Until we have package visibility in PHP, from my point of view, it is OK to define additional public methods. But please flag them clearly, to make sure everyone knows it is not part of the public API, like this:

```
class MyHttpClient extends HttpClient
{
    /**
     * @private
     */
    public function setCookie($key, $value)
    {
        // ...
    }
}
```

3.3.5 Conclusion

It is fine to employ duck typing. At the end of the day PHP is popular because the language does not impose any limits on the way you want to work with it.

But if you start with class based object orientation, please do it right. Mixing the concepts of class based object orientation and other stuff like Duck Typing messes with everybody's head. It *will* break code in obscure situations which might be really hard to cover by tests.

Abstractions are documentation. The best you can provide for developers. They are not easy to find, but you should really stay with them, once defined. Interfaces can, by the way, be used to extend them over time.

3.4 Abstract Classes vs. Interfaces

Kore Nordmann at 2. October, 2012[9]

Features of object oriented languages are often use from a purely technical perspective, without respect to their actual semantics. This is fine as long as it works for you, but might lead to problems in the long run. In this article I discuss the semantical differences between abstract classes and `interfaces`. I also outline why following the semantics of those language constructs can lead to better code.

Disclaimer: This is of course kind of artificial. If using the language constructs in other ways works for you, it's fine. It can make communication, extension of code and maintenance of code harder, though.

3.4.1 Definitions

First, we need to differ between `interface` and interface. When typeset in mono space I mean the language construct -- otherwise I talk about the public methods of any object / class. The second one often is also called "public interface", "class interface" or even "class signature".

3.4.2 Classes are Types

A class denotes a **type**.

We know that objects consist of internal state and methods to operate on this state. The interactions between objects are implemented by calling methods on other, related objects. Those methods might, or might not, modify the internal state of the called object.

Classes, as they define the blueprints for those objects, define the type of objects, thus we know how to use an object. When talking about types there are some natural implications, which apply especially when extending from another class, which means to create a sub-type. One is the Liskov Substitution Principle[10]:

> "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

[9] https://qafoo.com/blog/026_abstract_classes_vs_interfaces.html
[10] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

This basically translates to: "Do not mess up the interface". It is required, so you can use a sub-type (derived class) without any special knowledge, just by reading the documentation / interface of the parent class. Imagine a `Cache` class, from which you extend a `Cache\FileSystem`, `Cache\Memcached` and maybe a `Cache\Null`. You do not want to implement any kind of special handling in every place where an object of any of those derived classes is used in your application. All of them should just behave like a `Cache`. Keep this in mind.

If we can use any sub-type of a type just as the parent type defines, we talk about **Subtype Polymorphism**. Something you definitely want to achieve. Abstract classes are naturally used when you need such a parent type, which requires some specialization, like the `Cache` class mentioned above. I'll get back to why it makes no sense to extract an `interface` from this. And yes, you are allowed to define abstract classes, even if you don't intend to implement any code already and don't have any properties to define.

3.4.3 `interface`

An `interface` describes an aspect of the interface of a type.

PHP does not support multiple inheritance, but it allows you to implement any number of interfaces, right? The reason for this is, that an `interface` annotates usage aspects on the interface of a type. The Interface Segregation Principle[11] also claims:

> "Clients should not be forced to depend upon interfaces that they do not use."

This means that a type using another type should only depend on an interface, which is actually used. Depending on a god class, for example, is the exact opposite. You would not know which methods are actually called from looking at the injection signature. But this also implies that the interfaces should be designed small and task-specific.

To stay in the context of the example mentioned above, a proper interface would be `Cacheable` to annotate on certain classes that its instances can be cached. The

[11] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

natural counterpart of a `Cache`, if you want to handle objects. This is a minimal usage aspect of a class and other classes requiring this `interface` will most likely, and may only, use the methods required by this `interface`. Implementors of this interface would also not be forced to implement meaningless methods.

3.4.4 Telling Apart

Those definitions are nice, and fluffy -- but what happens if you define an `interface` instead of using an abstract class? (This is at least what I see most.)

- You invite others to create god classes
  Interfaces do not allow to already provide code or define properties. Their only "benefit" over abstract classes is that you can implement multiple of them in a class / type.
  So why would you define a `Cache` interface. Obviously you do not want that some class implements the `Cache` logic while also being a `Logger` and a `ORM` -- even all might access the same storage.
  While this, in theory, provides more "flexibility" for the future, every usage of this flexibility would hurt your software design -- a lot.
- You overload interfaces
  A common developer pattern is to define a type, and then extract one single `interface` from the class interface. Such interfaces do not define a single usage aspect any more. If someone wants to implement one aspect of this `interface` and two or three aspects from some other "`interfaces`" it would result in one *really big* class with dozens of empty methods. Or, even worse, dozens of implemented methods, which are never used. This would obviously be wrong.

3.4.5 But...

There is this one popular quote:

> “Code against interfaces, not implementations.”

"Interface" is not typeset in mono space. Look it up!

To phrase it better, like it is done in the Dependency Inversion Principle[12]: **"Depend on abstractions, not on concretions."**. "Interfaces" in the above quote may very well be abstract classes. Don't forget about those.

3.4.6 Examples & Hints

Classic examples for proper interfaces would be:

- `Cacheable`
- `Serializeable`

I have two mnemonics for people I discuss Object Oriented Design with:

- `interface` names should end with `able` or `ing`.
  This is obviously not always true. But ending on `able` is a strong indicator that such an `interface` just annotates one usage aspect.
- `interfaces` make sense to be implemented in entirely different types.
  An `interface` usually makes sense to be implemented in types, which otherwise have barely anything in common. A prime example again would be `Cacheable` or `Countable`, which even is a proper `interface` defined in the SPL.
  To come up with a "real-world" example: Let's consider an `interface Drinkable`. You could implement that one on cups, on the sea and maybe even on humans, if there are vampires around. Otherwise humans, seas and cups probably do not have that much in common.

Examples for proper abstract classes could be:

- `Log(ger)`
- `Cache`

Again I have a mnemonic to help you with the decision:

- An implemented abstract class can stand on its own.
  If you implement all remaining abstract methods of an abstract class, you get a proper member in your world. It does not require any additional methods to be usable by others. It probably will require some dependencies (like a storage), but most other objects will happily just call the methods provided by the abstract class.

[12] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Final hint:

An `interface` **must not** define a constructor. **Never**. The same is true for most abstract classes. By defining a constructor you predefine and limit what dependencies may be injected in implementations. It is very likely to change for different implementations. But this is actually a topic which deserves its very own blog post.

3.4.7 tl;dr

An `interface` annotates a usage aspect of a type / class. Do not just extract interfaces from classes. Use abstract classes, when defining a base type.

3.5 ContainerAware Considered Harmful

Tobias Schlitt at 7. October, 2013[13]

A while ago I tweeted[14]

> ContainerAware is the new Singleton.

While many people agreed by retweeting and faving. I feel the need to elaborate some more on this statement and safe the explaination for the future.

TL;DR: No class of your application (except for factories) should know about the Dependency Injection Container (DIC).

3.5.1 Background

The `ContainerAware` interface (actually `ContainerAwareInterface`, `ContainerAware` is a basic implementation of it) is part of the Symfony2[15] API[16], but a similar concept is known from many other frameworks and many applications rely on it. It defines only the one method `setContainer()`, which allows to inject the DIC into an object so that it can directly retrieve services from it.

It is most common to have controllers implement this interface. In fact, the Symfony2 base controllers[17] do so and there is a base class for shell commands[18] that does it.

3.5.2 Issues

Accessing the DIC in your classes can seriously harm maintainability and code re-usability in the long run.

[13] https://qafoo.com/blog/057_containeraware_considered_harmful.html
[14] https://twitter.com/tobySen/status/378780141826355200
[15] http://symfony.com
[16] https://qa.fo/book-ContainerAware
[17] https://qa.fo/book-Controller
[18] https://qa.fo/book-ContainerAwareCommand

Reduced Testability

Unit testing is the most common automatic test method in the PHP world. If you have the container injected into your objects, this becomes much harder. There are three strategies to approach testing a class that gets the container injected:

1. Mock the container and make it return the mocked service mocks,
2. Use the container in your test cases and make it return the mocks
3. Mock the subject of test to override the `get()` method that is commonly used to access services inside the class.

The first solution actually requires you to create a container mock that basically does the same thing as the real container. Except for mocking overhead you don't win much with this. So, if you are already in the situation that your classes have a dependency to the DI container, you're better of with version 2.

If you choose the second variant, you're formerly not writing a unit test, but an integration test instead. This is not a problem by default and integration tests are important, especially if you use an external framework for development. However, if you intend to write unit tests, you simply don't in this case, since another class (the DIC) is put under test.

The third variant is a simple no-go. You should never mock methods of your subject, because you cannot ensure that the mocked version of the code mimics what really happens. Your test cases lose a big amount of the safety that they should actually provide you with. Whenever you feel the need to mock a method of the test subject, that is a clear sign for the need to refactor (so-called *code smell*).

Hidden Dependencies

If you go for a real dependency injection approach (my favorite is constructor injection), you can easily see when a class does to much, just by looking at its dependencies. Think about a constructor like this:

```
<?php

class UserProfileController
{
    // ...

    public function __construct(
        UserRepository $userRepository,
        Search $search,
```

```
        MailGateway $mailGateway,
        Router $router,
        HttpClient $httpClient,
        PermissionService $permissions
    )
    {
        // ...
    }

    // ...
}
```

You can easily spot that there is something fishy. The class receives quite many dependencies and these do not fit very well together. Some are from the infrastructure layer, others seem to be business services and even others correlate with the MVC structure. If you see this constructor, it's eminent that there is need for refactoring. You will note that at the latest when you add another dependency to it.

Now compare that with the following code:

```
class UserProfileController extends ContainerAware
{
    // ...

    public function __construct()
    {
        // ...
    }

    // ...
}
```

There is no way to see what the actually dependencies of the class are. In order to see that, you need to go through the code and look for places where the Dependency Injection Container is used to retrieve objects. In real life, nobody will ever do that, especially if this is hidden behind `$this->get('serviceName')` calls.

Feature Sneak-In

This issue is a direct result of the previous one: If you can, you eventually will. And by this I mean access whatever service you might think you just need.

With access to the DIC a developer has the freedom to just grab any object and use it. This might be convenient to implement hacks, if you don't know where else

to put a certain functionality when being in a rush. But on the other side of the coin, there is nothing which forces you to clean up the mess.

A common result is, that more and more business logic is collected in the controllers, making them become huge transaction scripts, which leads to code duplication and hidden business rules.

3.5.3 Conclusion

Making your classes aware of the DIC is not an option. Even if you really feel you need to, just don't do it. To be on the safe side I even recommend to make your controllers services[19].

If you develop Symfony2 based applications and run into the issue of having to inject too many framework services[20] into your controllers, my co-worker Benjamin has a great post about that in his personal blog.

[19] https://qa.fo/book-service

[20] https://qa.fo/book-extending_symfony2_controller_utilities

3.6 Code Reuse By Inheritance

Kore Nordmann at 20. January, 2014[21]

3.6.1 Inheritance

To me, inheritance has two properties:

- Defining an *is-a* relationship
- Making it possible to share code between classes by extending from a common base class

The is-a relationship between classes is one thing I like to use inheritance for. So, to me, a concrete Cache - for example a `MemcacheCache` implementation - extending from an abstract `Cache` base class is a very sane thing to do. Implementing an interface means, in my opinion, adding an additional usage aspect / feature to a class, while the base class defines the *type* of a class. Mind: The base class, most likely, only defines abstract methods and does not provide any implementation in this case. I discussed this in more detail in a dedicated blog post[22], which is why I skip a detailed explanation now.

The other thing you can use inheritance for is to share code between multiple classes. A prime example for this is Active Record[23], where a concrete record class (for example a BlogPost) extends from a base class which adds the database access logic.

I personally think that the latter is one of the worst things you can do in object oriented design. Bear with me for a moment and let me try to explain why I think that.

3.6.2 Active Record

With Active Record, the problem is fairly obvious: The base class usually implements the storage logic, while the extending class is supposed to implement the business logic and maybe additional table query logic.

[21] https://qafoo.com/blog/063_code_reuse_by_inheritance.html
[22] https://qafoo.com/blog/026_abstract_classes_vs_interfaces
[23] https://en.wikipedia.org/wiki/Active_record

The business logic (or domain logic) is the most important part of your application – the stuff you earn your money with. You probably want to test this logic since you might be broke if it fails too hard.

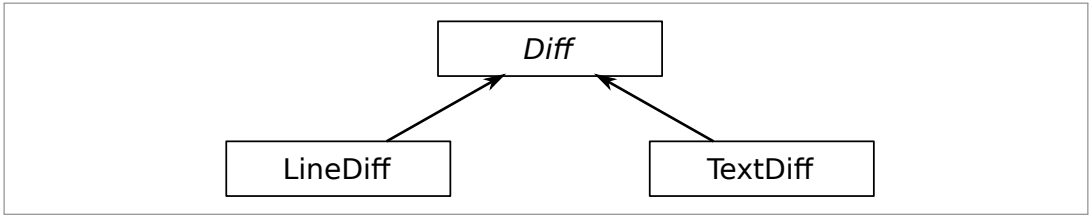
Unit-testing a class which uses logic from a base class, for example accessing the database, is a lot of work. The best way is usually to mock all methods (from the parent class) which access the database and then run the tests on the mocked subject while correctly simulating the return values from your database. This is damn tedious.

Not wanting to go into too much detail here, but testing an Active Record class as-is is often even worse since tests which hit your database are usually damn slow, harder to set up and harder to keep atomic. But most importantly, unit-tests should fail for only one reason, and also testing a database access layer will likely make it a lot harder to locate the exact reason for unit test failures.

3.6.3 A Smaller Example

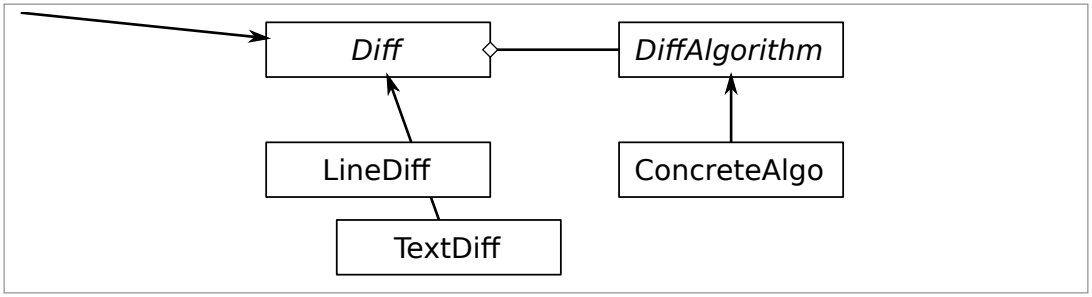
One of my preferred examples when describing this to others is a mistake in a component design I made not too many years ago. I wanted to write a little *diff* component, which makes it possible to evaluate the line-wise diffs in source code, but also paragraph and word-wise diffs for wiki articles and blog posts.

When writing the component, I started implementing the most common diff algorithm on top of "tokens", which should allow me to handle all the cases mentioned before. Once the algorithm worked, I derived from the class, implementing the different diff flavours:



That worked fine for some time, but once the diff component was integrated into PHPUnit, Sebastian Bergmann noticed problems with large texts and we realized that there are better diff algorithms for particular large texts. The `LineDiff` and

`TextDiff` classes, implementing their respective tokenizing rules, are still fine, but we wanted to replace the algorithm in the base class depending on the diff use case. This is, of course, not possible. A better class design would obviously be:



We can now replace the used diff algorithm, test it directly based on simple token streams, and every concrete `Diff` implementation (`LineDiff`, `TextDiff`, ...) will still work with all the diff algorithm implementations you can come up with. A downside, of course, is that creating the Diff object gets slightly more complex, but this is usually handled by a Dependency Injection Container, anyway.

3.6.4 The Helper Method

The points where Code-Reuse-By-Inheritance still keeps creeping into my code from time to time are small helper methods. For example for complex data structures it is so easy to just define a method in the data structure which calculates some kind of hash, to apply simple transformations or to validate some values. The next thing you notice is that you will move those methods up in the extension hierarchy or copy it around (maybe slightly modified).

Why not make it a (single) public method in some Hasher, Validator or Visitor class? You might be surprised how much easier stuff gets to test and that you might even be able to re-use that code in even more places. I guess the *fear of (many) classes* applies here again, which I consider void. But this is another blog post.

Of course it is valid to have such helper methods during prototyping, but if you start using such code in multiple places, start testing it or start to put it into production you should refactor it to follow the mentioned concerns.

In most cases, those helper methods are also a different concern which you start mixing into your class. Even the concern seems closely related, it probably does not hurt to move it somewhere else and make it explicit.

It may happen, though, that people start to over-engineer given these constraints. But every developer walks on a very fine line between missing abstraction and over-engineering all the time, anyway. A good base for a decision could be: Will it ever make sense to somehow re-use this piece of code or may it be possible that someone wants to replace this implementation in a similar use case or during testing?

3.6.5 Testing Private Methods

The urge to test private or protected methods is, in my opinion, a code smell which should directly lead to refactoring.

Once you separate your concerns and move code you share by inheritance into its own classes, every non-trivial code will be contained in public methods which are easy to test. And more importantly: easier to understand.

3.6.6 Depth Of Inheritance Tree (DIT)

Then there is the "Depth of Inheritance Tree" (DIT) metric with a common boundary value of 5, while the counting even stops at component borders. To me, the maximum value for this metric should be considered 2. Except for some struct classes[24] / value objects, there is, in my opinion, no reason for more then one level of extension of a class. If you use inheritance just for defining the _type_ of classes, you will never extend more than once. If you are tempted to do that, use aggregation instead and you are probably fine in 99% of all cases.

If you have valid use cases for an inheritance hierarchy greater than two, please share those with me.

[24] https://qafoo.com/blog/016_struct_classes_in_php

3.6.7 Summary

To me, by now, Code-Reuse-By-Inheritance is a clear code smell. Every time I am tempted to do this or find this in existing code, I will refactor out the code into a separate class as soon as possible.

3.7 Utilize Dynamic Dispatch

Tobias Schlitt at 16. October, 2014[25]

A while ago I replied to the tweet[26] by @ramsey[27]

> Traits are a nice way to provide common functionality to unrelated classes without using static methods on a global Util class.

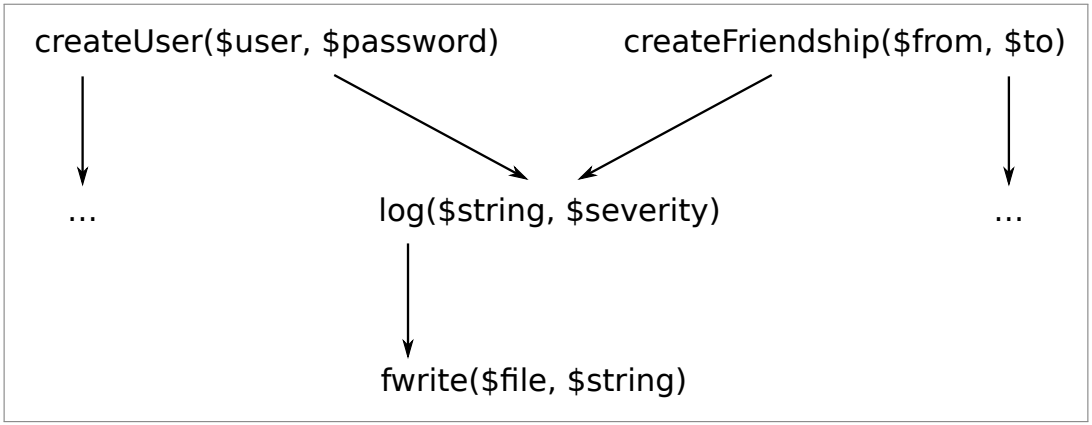
with[28]

> Which makes them exactly as evil as static access. Funktionality you dispatch to becomes irreplaceable destroying a fundament of OO: Dynamic dispatch.

I want to use this blog post to illustrate the concept of *dynamic dispatch* which I use a lot recently to motivate creating clean OO structures in my trainings. In my experience, this helps people to understand why we want to write code in this way. After that I will show why traits are bad in this direction.

Dynamic dispatch is actually an implementation detail of object oriented programming languages. To explain it I need to take a little leap back in time and illustrate the code flow in a classic procedural program:

[25] https://qafoo.com/blog/072_utilize_dynamic_dispatch.html
[26] https://twitter.com/ramsey/status/509028110465908737
[27] https://twitter.com/ramsey
[28] https://twitter.com/tobySen/status/509040311122022400

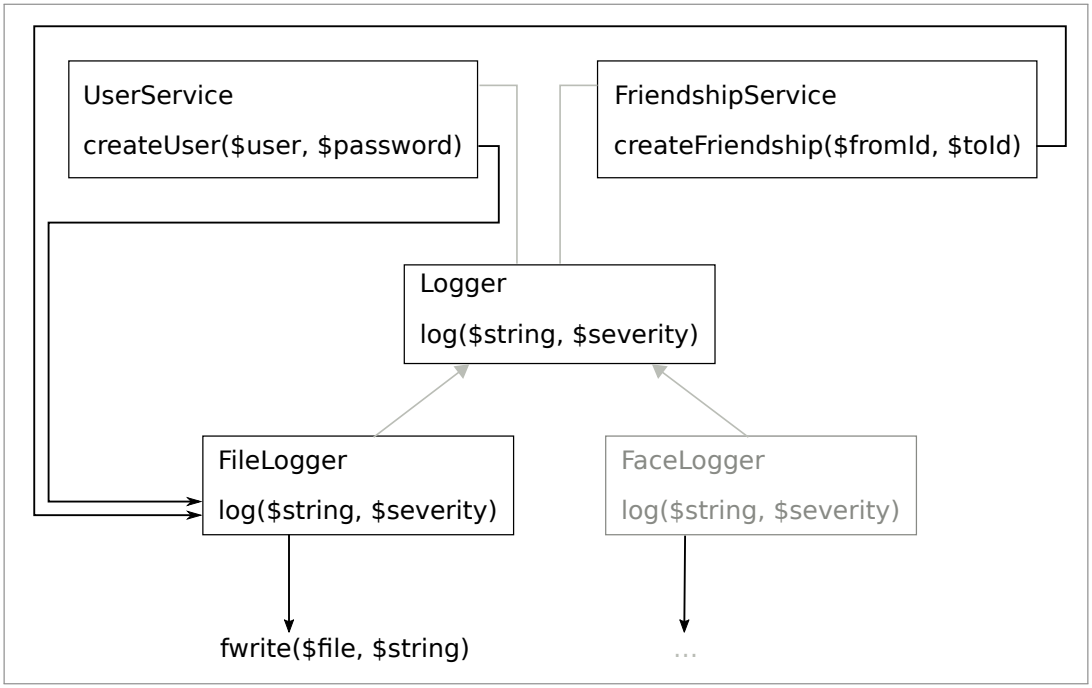


The graphic shows two procedures which call a shared `log()` procedure, the arrows visualize the execution flow of the program. As can be seen, a procedural program classically works top down. In order to solve a complex task, a procedure dispatches to other procedures where each fulfills a smaller fraction of the task. It is important to notice that the procedures to be executed are exactly known at compile time of the program (simplified). This is what is called a *static dispatch*.

Imagine you now want to log all friendship related operations to the new *Facelog* server, because it generates incredibly useful insights for you. What are your options to implement this change?

You can a) change the `log()` procedure itself, i.e. patch it. But this is of course no real option here, because `createUser()` would also be affected by this change and most probably many other modules that use `log()`. Option b) is to touch the `createFriendship()` procedure (and any other friendship related modules) and change them to use another procedure instead of `log()`, e.g. `facelog()`. Following this approach is viable, but also means quite some work and potential for errors.

In the object oriented (OO) paradigm the situation is different:



The most obvious difference here is that there are two types of errors: Black arrows visualize the actual code flow again, while gray ones indicate object (actually class) dependencies. The `UserService` depends on `Logger`, `FileLogger` extends `Logger` and so on.

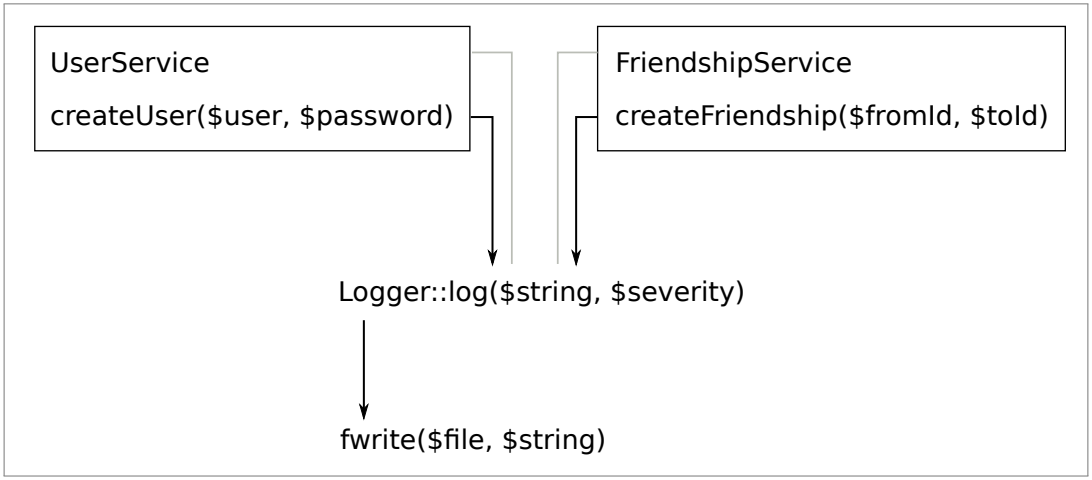
And this exactly is the crucial point: At compile time, an object oriented program typically only knows about the class dependencies. At this stage it is unclear, how the actual execution flow will be at run-time. It depends upon which particular object is given to the dependant, which might be influenced by configuration, user-input and so on. On a technical level, the programming environment decides out of a set of available method implementations which one is the correct one to use. This is what we call *dynamic dispatch*.

So how could you perform the required change in this environment? Of course you can just give a different object to `FriendshipService`, which provides the API defined by the `Logger` type but talks to Facelog instead of writing to a file. The

dynamic dispatch will take care. The nice thing: You neither need to touch the `FriendshipService` nor do you need to fear undesired side-effects on `UserService` and others by touching the `FileLogger`.

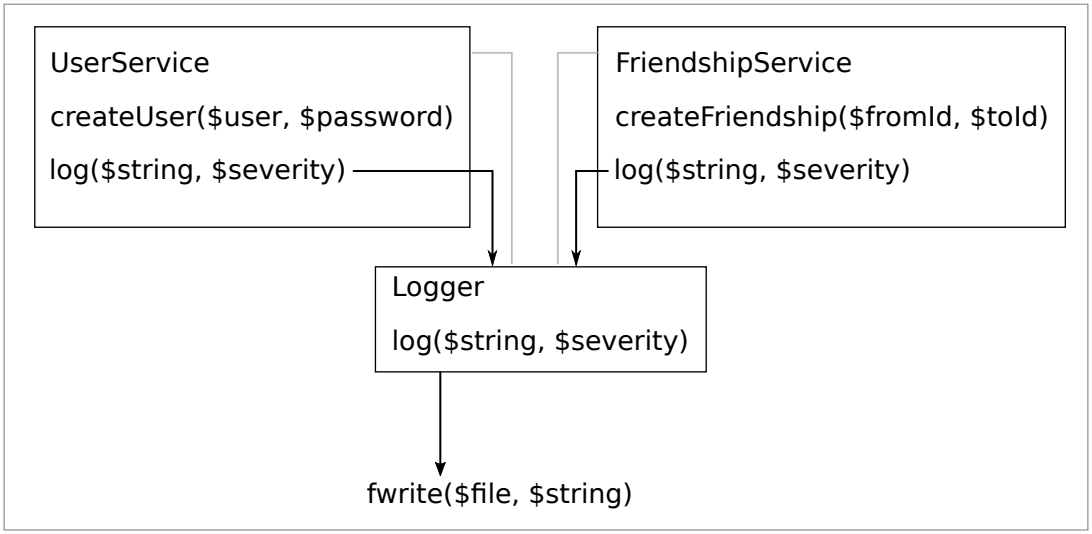
Now, that is dynamic dispatch: Your execution environment decides at run-time, which method is actually to be called, based on the actual object reference given for a type dependency. There is no need to recompile the program to achieve a change in this dispatching, a different user input or configuration can suffice.

Now take a look at a static method call as shown in the following graphic:



What do you realize? Right, a static method call is actually a procedural call. Indeed, the dispatch is static in this case: There is no way for you to replace this call fine grained, except for the two procedural approaches explained further up. Whenever you introduce a static method call, you throw away the powers that dynamic dispatch unveils to you.

Now look at the next graphic that shows a trait implementation for the `Logger`:



Using a trait copies a piece of code into your class, making it a part itself. How would you realize the desired change now? True, there is now way to utilize dynamic dispatch here, but only the procedural approaches work. *A trait is static dispatch, too.*

Disclaimers:

1. There is of course much more in the background of this very narrow view on object oriented mechanics, for example subtype polymorphism.
2. I'm aware that there are concepts in procedural languages to implement a dynamic dispatch, too, for example function pointers. However, for illustration purposes, it is helpful to take these out of scope here.

3.8 When to Abstract?

Kore Nordmann at 17. May, 2016[29]

One of the most difficult challenges in a developers life is finding the "right" abstraction, or at least the best one given the current circumstances. The core problem is that abstraction is a bet on the future development of the software and we know that future is volatile. The circumstances will change, so will the view on the best abstraction change.

But there is another dimension which influences this decision: What kind of software are you developing? When it comes to decisions about Object Oriented Design we separate between three different types of projects:

- Internal Project
  It is just used by you and your co-workers. You are working in a small team. When you are changing an API you can adapt all code using this API easily – especially if you are using a Monorepo[30] approach.
- Library
  There is a common entry point for your library, an interface which everybody is using. This could be everything from a popular Open Source library to a company internal library used in multiple teams or projects. Changing the API is already a no-go because it was develop to fulfill the use-case of the library. You cannot see who is using this API and what code will break. The only chance is a new major release for each change to this API.
- Adaptable Product
  There are generic software solutions out there which will be adapted by agencies and developers like Wordpress, Magento or similar products. Users of those products will use **any** API in the source code of those products to change the behaviour of the system. You cannot change any API without breaking someones application.

So how does this influence the decision about abstracting your code? In the first case (Internal Project) you can delay abstracting code pretty far while in the last case (Adaptable Product) you should get it right from the very beginning – since you shouldn't change anything afterwards.

[29] https://qafoo.com/blog/084_when_to_abstract.html
[30] https://qa.fo/book-monorepos

But, in an Adaptable Product and in Libraries your use case is usually well defined, thus you know what the interface looks and you can think about it, analyze it and come up with a sane abstraction. One could even say: You define the use case with your API, thus your abstraction is fine by definition. Your library and product will be used in other use cases as well, which you did not consider when writing the software, but it usually less of an issue to adapt to that later.

Authors of those libraries and software products are often well known and talk about their code in public. Also this is the code you, as a developer, will see when learning and trying to improve.

But especially in the area of Internal Projects the circumstances change often and fast. You will have no idea what the next requirement of the project stakeholders will be. So, let's be blatant: Do not abstract! Do not use interfaces or abstract classes.

OK, sometimes you still should use them. But at least wait until the requirements get obvious. I, by now, suggest a process like the following:

1. Implement the current requirement in a concrete class
2. Adapt the concrete class to the ever changing requirements
3. If a different implementation for the same use case is required create an abstraction, but not earlier

Example: If you are supposed to integrate with some newsletter provider just implement the code in a concrete class. When starting, I assure you, not all final requirements are provided. There will be some additional requirements even before the first release of this small part of your application. If you came up with an abstraction first you'll have to adapt the abstraction **and** the concrete class. Also both will be more complicated since you already thought about making this somehow generic. Usually with requirements which will **not** show up because the other stakeholders will think about different things then you do.

Only when the second and third newsletter provider shows up and you are supposed to connect them with the same classes create an abstraction. Since we are a lot later in the project you are now better aware of all requirements and have a better understanding for the actual use-cases and differences between the endpoints.

3.8.1 Summary

Trying to follow "best practices" from library development in other projects will lead to wrong abstractions and increased workload. Do not abstract before you are sure about the actual requirements and before there is a need to do so. Concrete classes without interfaces are fine in Internal Projects, while they are not in Libraries or Adaptable Products.

4. Testing

4.1 Finding the right Test-Mix

Benjamin Eberlei at 19. August, 2013[1]

The topic of Test Driven Development (TDD) and unit-testing usually creates heated discussions among developers. We at Qafoo are not an exception and the topic how to apply TDD in our daily work regularly leads to heated discussions. One reason for this is that both TDD and unit-testing combined are by some people seen dogmatically as the only way to do software development.

This blog post tries to summarize our current state of discussion on TDD, software testing and finding the right test-mix. It is written from my own perspective and all errors in description are mine, not my colleagues'. I also try to review the topic in context of the recent discussions of others.

Let's start with Uncle Bob, who is the primary voice of TDD as the only approach to software development. He recently wrote about this in his article "The startup trap[2]" spurring several discussions (HackerNews[3] , Storify[4], Greg Youngs Blog[5]). However, he also acknowledges in a blog post the day later[6], that he does not apply

[1] https://qafoo.com/blog/055_finding_the_right_test_mix.html
[2] https://qa.fo/book-TheStartUpTrap
[3] http://martinfowler.com/bliki/TestPyramid.html
[4] https://qa.fo/book-tdd-and-startups-uncle-bob-vs-nate-et-al
[5] https://qa.fo/book-tdd-and-startups-uncle-bob-vs-nate-et-al
[6] https://qa.fo/book-ThePragmaticsOfTDD

TDD dogmatically to every aspect of software development. Notably, he doesn't write tests for:

- state of objects, for example getters/setters and attributes themselves.
- GUIs, because they require lots of tinkering and small adjustments to configuration variables.
- Third-party libraries
- while experimenting with code (what I would call periods of "Spikes").

Most importantly, check how Uncle Bob never mentions the word unit-test in both blog posts. This is because TDD is about software design, not about testing your individual units (objects, functions, ...) or 100% code coverage. We think, TDD can be applied on any level of the testing pyramid[7]: the acceptance-, integration- and unit-test level. Kent Beck describes this in his book "Extreme Programming explained".

Why do I mention this? We have made the experience first hand that starting a new project with TDD and mostly unit-tests can actually slow down development considerably. This happens because during the development of new projects and features the requirements change frequently. This conflicts with tests locking the code to the current functionality through high test-coverage. Johann Peter Hartmann discussed this topic with Toby[8] in an interview on our blog some weeks ago.

Unit-tests are necessary to stabilize your code, but when you know the requirements are not stable yet, then having too many unit-tests can be a burden as well. It is very difficult to write unit-tests that are immune to change, especially if you don't know what changes will happen. Combined with a dynamic language like PHP that has poor automatic refactoring support and you will suddenly see yourself wasting lots of time adjusting unit-tests to changing requirements.

There are several learnings from this that we try to teach our customers in our trainings:

1. Don't focus on unit-tests exclusively. A good test-mix is required and uses acceptance-, integration- and unit-tests. A "good" ratio for software is something like 70% unit, 20% integration and 10% acceptance tests.

[7] http://martinfowler.com/bliki/TestPyramid.html
[8] https://qafoo.com/blog/051_testing_sweet_spot.html

2. During development-spikes, with frequent requirement changes and uncertainty, it can be better to drive the design using acceptance- and integration-tests instead of unit-tests. Unit-test only those parts of the application that are mission critical, highly reused parts of the infrastructure (high cohesion) or stable already.
3. Once the software stabilizes, you should refactor tests from the acceptance- and integration-levels towards the unit-test-level. This makes the tests run much faster, less prone to failure due to side-effects, and allows you to write much more tests for different input combinations and edge-cases. Failures in unit-tests are also much easier to analyze than failures in acceptance tests.
4. Apply risk management to different components of your software: Having a lot of unit-tests is only important for those parts of your software that have a high business value. Features that are less important don't need the same test-coverage as those features that generate the majority of the business value. A few acceptance-tests might already be enough for those less important components.
5. Learning how to write tests that don't break on every occasion. This is beyond the scope of this blog-post.

4.1.1 The Test-Mix Tradeoff

It is important to state that we don't advocate to stop testing. Instead, we are putting forth the notion of TDD decoupled from unit-testing and depending on business value (and risk) instead. This is a tradeoff by reducing the time for refactoring tests and increasing the time to run tests as well as the risk of failure due to uncovered code.

This approach to TDD is not a silver bullet, though: If your business doesn't allow for periods of stabilization or you wait too long before stabilizing, then you will end up with an inverted test-pyramid of many slow acceptance tests and just a few unit-tests or even worse, with no tests at all.

We found that our approach is closely related to the discussion on "Spike and Stabilize" in Liz Keogh's blog[9] (including the comments), with the difference that we

[9] https://qa.fo/book-beyond-test-driven-development

suggest using at least acceptance-tests during spikes. Her list of bullet points on applying TDD is a systematic checklist for the pragmatic choices Uncle Bob had in his blog-post.

4.1.2 Conclusion

TDD is about design and not about unit-testing and 100% coverage. Using acceptance- and integration-tests is a valid approach for TDD and serves well during periods of spikes and frequent requirement changes. This decision trades a slower test suite and less stability for more flexibility to adjust the code. Neglecting to introduce unit-tests during code stabilization however might lead your code base to rot in the long run. Only a decent unit-test-coverage provides a developer with security to change code on all levels of the application.

4.2 Mocking with Phake

Benjamin Eberlei at 13. March, 2013[10]

Update (14.3.2013): Introduced Test Double wording instead of using mock objects for everything to more cleanly describe the different concepts.

The use of Test Doubles is an important skill to learn when using Test Driven Development (TDD). Test Doubles allow you to replace dependencies of an object with lookalikes, much like crash test dummies are used during automobile safety tests so humans aren't harmed.

4.2.1 Test Doubles Explained

Creating a test double in PHP involves creating a new class that extends the public API of the original class with empty methods. You can safely use all methods of a test double and they will do nothing, rather than calling the original code of the original class. There are two ways to create these test doubles: You can write them yourself or use one of the many existing libraries.

I would strongly recommend to use any of the existing libraries that can simplify and automate this task for you. Technically they work using code-generation at run-time.

To allow interactions with test doubles there are three ways to configure them in any library:

- Add expectations of the arguments passed to a method (Verification)
- Add results that are returned from a method-call to the mock object (Stubbing)
- Delegate calls to the original code (Spying)

Test doubles using the first approach are called Mock Objects. Objects of the second type are called Stubs, of the third type Spies.

4.2.2 Benefits of Test Doubles

There are many reasons why test doubles are useful:

- Allow units (objects) to be tested in isolation of their dependencies. This is done by replacing the dependencies with test doubles.

[10] https://qafoo.com/blog/037_phake.html

- Allow verification of behavior between objects. In contrast to assertions that can only verify the state of objects in isolation. This makes them very useful with relation to Behavior Driven Development (BDD) inside your unit-tests.
- Test Doubles are useful to test-drive new interfaces based on required behavior without caring for the implementation at the moment.

That means testdoubles are invaluable to move from state-based object-oriented programming to a behavioral approach based on sending messages between objects.

4.2.3 Introduction to Phake

Using Test Doubles in PHPUnit tests means using the built-in MockObjects library[11] for quite some years. In the last years two contenders emerged that can be used as optional dependencies in PHPUnit:

- Mockery by Padraic Brady[12]
- Phake by Mike Lively[13]

Both can be installed using Composer and integrated into your projects very easily.

This blog post introduces Phake, because it works quite differently than both PHPUnit Mock Objects and Mockery:

1. In the well known four phase test[14] with Setup, Exercise, Verify, Teardown both PHPUnit mocks and Mockery require expectations to be part of the "setup" phase.
   This is unfortunate, because mock expectations are much more related to the "Verify" phase instead. This happens in PHPUnit and Mockery, because they don't explicitly differentiate between methods that are mocked (verification) or stubbed (returning results). Phake introduces a differentiation by requiring different APIs to be used for configuration.
2. Instead of using strings for method names and builder methods for arguments, Phake let's you prototype the method-call in actual PHP code. This simplifies the mock object configuration considerably and requires much less typing.

[11] https://github.com/sebastianbergmann/phpunit-mock-objects
[12] https://github.com/padraic/mockery
[13] https://github.com/mlively/Phake
[14] http://xunitpatterns.com/Four%20Phase%20Test.html

Let's see an example containing both mock and stub objects in one test for loading the weather data for a given location.

```
<?php

class LoaderTest extends \PHPUnit_Framework_TestCase
{
    public function testGetWeather()
    {
        // 1. setup
        $logger = \Phake::mock('Qafoo\\Weather\\Logger');
        $functional = \Phake::mock('Qafoo\\Weather\\Service');

        // Stubbing with Phake::when()
        \Phake::when($functional)->getWeatherForLocation()->thenReturn(
            new Struct\Weather( 'Fair', 23, 0, 'NW' )
        );

        $loader = new Loader($functional, $logger);

        // 2. exercise
        $locatedWeather = $loader->getWeatherForLocation(
            new Struct\Location( 'Berlin', 'Germany' )
        );

        // 3. verify
        $this->assertInstanceOf(
            'Qafoo\\Weather\\Struct\\LocatedWeather',
            $locatedWeather
        );

        // Verification with Phake::verify()
        \Phake::verify($logger)->log('Fetched weather for Berlin Germany.');
    }
}
```

Using the `Phake::when()` call and passing a mock object you can prototype what a method call should return for your code to show a desired behavior. See how we can just call `->getWeatherForLocation()` as a prototype for how the stub behaves instead of PHPUnits `->method('getWeatherForLocation')` or Mockerys `->should Recieve('getWeatherForLocation')`.

Using `thenReturn` specifies the return value of this method call and completes the description of how the stub works. If you want to return different values on

consecutive calls, just chain multiple `thenReturn` calls. You can use `thenThrow` to throw exceptions.

Verification is done with `Phake::verify()` after the tested code was actually exercised. We again prototype what method calls and which arguments we want to verify, in this case `->log('Fetched weather for Berlin Germany');` on the logger mock.

This is a very simple stubbing and verification example with Phake. Comparable to PHPUnit and Mockery, we could add more complex expectations:

- Using argument matchers to verify the structure of arguments used in method calls.
- Checking multiple invocations such as exactly n-times, at least N-times or others.
- Verify no interaction with a mock happened at all.
- Verify no further interaction happened than the ones already verified.

4.2.4 Conclusion

Phake is a great mocking library and can be easily integrated into PHPUnit. Its new approach to prototype mocks and stubs and the separation between stubbing and verification phases is very refreshing and easy to use.

If you want to go into more detail and learn about Phake, you should check out the extensive documentation[15].

[15] http://phake.digitalsandwich.com/docs/html/

4.3 Testing Effects of Commands With Phake::capture()

Benjamin Eberlei at 8. March, 2016[16]

Today I want to share a simple trick for the excellent Mocking library Phake[17] (I wrote about it before (See: Mocking with Phake)) when testing state on APIs that don't return values.

Testing becomes harder when you are using command / query separation in your code and service operations don't return values anymore. If your command method creates objects and passes them down the stack, then you usually want to make assertions on the nature of the changes.

Take the following example that creates a new `Order` object:

```
<?php

class CheckoutHandler
{
    private $orderRepository;
    private $productRepository;

    public function __construct($orderRepository, $productRepository)
    {
        $this->orderRepository = $orderRepository;
        $this->productRepository = $productRepository;
    }

    public function checkout(Checkout $command)
    {
        $order = new Order();
        $order->setAddress($command->address);

        foreach ($command->productIds as $id => $amount) {
            $product = $this->productRepository->find($id);
            $order->addItem($product, $amount);
        }

        $this->orderRepository->save($order);
    }
}
```

[16] https://qafoo.com/blog/078_phake_capture.html
[17] http://phake.readthedocs.org/en/2.1/

A "usual" PHPUnit test for this class can only make a single assertion that the `OrderRepository` is called with an `Order` object. But we might want know if a product was correctly assigned.

With `Phake::capture($value)` we can assign the argument passed to `OrderRepository#save($order)` to a variable that is available inside the Unit-Test, ready to run assertions on.

```
<?php

class CheckoutHandlerTest extends \PHPUnit_Framework_TestCase
{
    public function testCheckout()
    {
        $orderRepository = \Phake::mock(OrderRepository::class);
        $productRepository = \Phake::mock(ProductRepository::class);

        $product = new Product();
        \Phake::when($productRepository)->find(42)->thenReturn($product);

        $handler = new CheckoutHandler($orderRepository, $productRepository);
        $handler->checkout(new Checkout([
            'productIds' => [42 => 1],
            'address' => new Address(),
        ]));

        \Phake::verify($orderRepository)->save(\Phake::capture($order));

        $this->assertEquals(1, count($order->getProducts()));
        $this->assertSame($product, $order->getProducts()[0]);
    }
}
```

See after the `\Phake::capture($order)` call, the `$order` variable contains the argument that was passed to the OrderRepository from your code.

This argueably reaches into the tested class quite a bit, but when you use Command / Query separation and London-Style TDD the only way to observe behaviour and state is mocking. I still think Phake is the best mocking library for PHP and the capture method is another good argument for it.

4.4 Using Mink in PHPUnit

Benjamin Eberlei at 5. April, 2016[18]

Another day for a short PHPUnit trick. If you want to use PHPunit to control a browser for functional or acceptence tests, then you can easily do this using the Mink library. Mink is well known from the Behat community to facilitate Behaviour-Driven Development (BDD), but it is a standalone library that can be used with PHPUnit just as easily.

This is more flexible than using dedicated browser abstractions such as Selenium directly from PHPunit, because you can switch between different implementations or even run tests with multiple implementations using the same code base.

To start install Mink into your PHP project using Composer:

```
$ composer require behat/mink behat/mink-goutte-driver --dev
```

This will install Mink and the Guzzle/Goutte based Driver to crawl your site using a very simple cURL based browser abstraction.

Lets start using it for a simple PHPUnit test that verifies Wikipedia Search:

```
<?php

class WikipediaTest extends \PHPUnit_Framework_TestCase
{
    public function testSearch()
    {
        $baseUrl = 'https://en.wikipedia.org/wiki/Main_Page';

        $driver = new \Behat\Mink\Driver\GoutteDriver();
        $session = new \Behat\Mink\Session($driver);
        $session->start();
        $session->visit($baseUrl);

        $page = $session->getPage();
        $page->fillField('search', 'PHP');
        $page->pressButton('searchButton');

        $content = $session->getPage()->getContent();

        $this->assertContains('PHP: Hypertext Preprocessor', $content);
        $this->assertContains('Rasmus Lerdorf', $content);
    }
}
```

[18] https://qafoo.com/blog/081_phpunit_mink_functional_tests.html

Setting up the Driver and Session over and over again can become quite complicated, lets introduce a reusable trait:

```
<?php

trait MinkSetup
{
    private $minkBaseUrl;

    private $minkSession;

    /**
     * @before
     */
    public function setupMinkSession()
    {
        $this->minkBaseUrl = isset($_SERVER['MINK_BASE_URL'])
            ? $_SERVER['MINK_BASE_URL']
            : 'http://localhost:8000';

        $driver = new \Behat\Mink\Driver\GoutteDriver();
        $this->minkSession= new \Behat\Mink\Session($driver);
        $this->minkSession->start();
    }

    public function getCurrentPage()
    {
        return $this->minkSession->getPage();
    }

    public function getCurrentPageContent()
    {
        return $this->getCurrentPage()->getContent();
    }

    public function visit($url)
    {
        $this->minkSession->visit($this->minkBaseUrl . $url);
    }
}
```

The `@before` annotation is relatively new, it makes sure that the annotated method is called during each test cases setup phase, whenever we use the `MinkSetup` trait in a test class.

This allows us to write the actual test in a much simpler way:

```
<?php
```

```
class WikipediaTest extends \PHPUnit_Framework_TestCase
{
    use MinkSetup;

    public function testSearch()
    {
        $this->visit('/');

        $page = $this->getCurrentPage();
        $page->fillField('search', 'PHP');
        $page->pressButton('searchButton');

        $content = $this->getCurrentPageContent();

        $this->assertContains('PHP: Hypertext Preprocessor', $content);
        $this->assertContains('Rasmus Lerdorf', $content);
    }
}
```

If you followed the `MinkSetup` implementation, you saw the `MINK_BASE_URL` environemnt variable. We can configure this from the `phpunit.xml` configuration:

```
<?xml version="1.0" ?>
<phpunit bootstrap="vendor/autoload.php">
    <php>
        <env name="MINK_BASE_URL">http://en.wikipedia.org/wiki</env>
    </php>
</phpunit>
```

You can improve this by adding more helper methods onto the `MinkSetup` trait, for example by closely following the possibilities that Mink provides inside of Behat (See MinkContext[19]).

[19] https://qa.fo/book-MinkContext

4.5 Introduction To Page Objects

Manuel Pichler at 6. September, 2016[20]

A while ago we wrote about writing acceptance tests (end-to-end tests) with Mink and PHPUnit (See: Using Mink in PHPUnit). While this is a great set of tools for various applications such tests tend be susceptible to changes in the frontend. And the way they break is often hard to debug, too. Today I will introduce you to Page Objects[21] which can solve some of these problems.

The basic idea behind a Page Object is that you get an object oriented representation of your website. The Page Objects maps the HTML (or JSON) to an object oriented structure you can interact with and assert on. This is more initial work then than writing tests with PHPUnit and Mink directly, but it can be worth the effort. I will introduce you to Page Objects by writing some simple tests for Tideways – our application performance monitoring platform[22].

4.5.1 Groundwork

We will again use the awesome Mink[23] to simulate browsers and make it easy to interact with a website. Thus we are actually re-using the `FeatureTest` base class from the Using Mink in PHPUnit (See: Using Mink in PHPUnit) blog post. We have set up a repository[24] where can take a full look at the working example and maybe even try it out yourself.

You'll need some tools to set this up – in the mentioned repository it is sufficient to execute `composer install`. Setting it up in a new projects you'd execute something like:

```
composer require --dev phpunit/phpunit behat/mink behat/mink-goutte-driver
```

The FeatureTest base class[25] handles the basic mink interaction and has already been discussed in the last blog post so that we can skip it here.

[20] https://qafoo.com/blog/089_introduction_to_page_objects.html
[21] http://martinfowler.com/bliki/PageObject.html
[22] https://tideways.io
[23] http://mink.behat.org/en/latest/
[24] https://github.com/QafooLabs/PageObjects
[25] https://qa.fo/book-FeatureTest

4.5.2 A First Test

As mentioned we want to test Tideways[26] and Tideways requires you to login. Thus we start with a simple login test:

```
class LoginTest extends FeatureTest
{
    public function testLogInWithWrongPassword()
    {
        $page = (new Page\Login($this->session))->visit(Page\Login::PATH);

        $page->setUser(getenv('USER'));
        $page->setPassword('wrongPassword');
        $newPage = $page->login();

        $this->assertInstanceOf(Page\Login::class, $newPage);
    }

    // ...
}
```

This test already uses a page object by instantiating the class `Page\Login`. And by using this one it makes the test very easy to read. You instantiate the page, `visit()` it and then interact with it in an obvious way. We set username and password, and then call `login()`. Since we set a wrong password we expect to stay on the login page.

This already is the nice thing with page objects. The test are readable and this is something we want to optimize for, right?

One the other hand the logic must be implemented in the Page Object. By implementing it in a Page Object it is re-usable in other tests as you will see later. So let's take a look at this simple Page Object:

```
use Qafoo\Page;

class Login extends Page
{
    const PATH = '/login';

    public function setUser($user)
    {
        $this->find('input[name="_username"]')->setValue($user);
    }
```

[26] https://tideways.io

```
    public function setPassword($password)
    {
        $this->find('input[name="_password"]')->setValue($password);
    }

    public function login()
    {
        $this->find('input[name="_submit"]')->press();

        return $this->createFromDocument();
    }
}
```

Since we use Mink and implement some logic in the `Qafoo\Page` base class this still does not look that complex. What you should note is the fact that the method `setUser()` (and alike) hide the interaction with the DOM tree. If the name of those form fields change you'll have to change it in one single location. The methods `find()` and `visitPath()` can be found in the Page base class[27] and just abstract Mink a little bit and provide better error messages if something fails.

The `login()` method will execute a HTTP request to some page. If the login failed we will be redirected back to the login page (like in the test above), otherwise we expect to be redirected to the dashboard:

```
public function testSuccessfulLogIn()
{
    $page = (new Page\Login($this->session))->visit(Page\Login::PATH);

    $page->setUser(getenv('USER'));
    $page->setPassword(getenv('PASSWORD'));
    $newPage = $page->login();

    $this->assertInstanceOf(Page\Dashboard::class, $newPage);
}
```

We expect the user name and password to be set as environment variables since there are no public logins for Tideways. If you want to run the tests yourself, just create an account and provide them like mentioned in the README[28].

There is one "magic" method left in the page object shown before – the method `createFromDocument()`. The method maps the path of the last request back to

[27] https://qa.fo/book-Page

[28] https://github.com/QafooLabs/PageObjects

Page Object. Something like the router in about every framework would do, but we map to a Page Object instead of a controller. This method will get more complex for complex routes but it helps us to make assertions on the resulting page.

4.5.3 Refactoring The Frontend

We recently migrated the dashboard from being plain HTML rendered using Twig templates on the server into a React.js component. What happens to our page objects in this case? Let's take a look at our dashboard tests first:

```
class DashboardTest extends FeatureTest
{
    use Helper\User;

    // ...

    public function testHasDemoOrganization()
    {
        $this->logIn();

        $page = (new Page\Dashboard($this->session))->visit(Page\Dashboard::PATH);

        $organizations = $page->getOrganizations();
        $this->assertArrayHasKey('demo', $organizations);
        return $organizations['demo'];
    }

    // ...
}
```

This test again makes assertions on a page object – now `Page\Dashboard` which can be instantiated after logging in successfully. The test itself does not reveal in any way if we are asserting on HTML or some JSON data. It is simple and asserts that we find the `demo` organization in the current users account (which you might need to enable[29]).

So let's take a look at the dashboard Page Object, where the magic happens:

```
class Dashboard extends Page
{
    const PATH = '/dashboard';
```

[29] https://app.tideways.io/settings/profile

```
    public function getOrganizations()
    {
        $dataElement = $this->find('[data-dashboard]');
        $dataUrl = $dataElement->getAttribute('data-dashboard');

        $data = json_decode($this->visitPath($dataUrl)->getContent());
        \PHPUnit_FrameWork_Assert::assertNotNull($data, "Failed to parse JSON
            response");

        $organizations = array();
        foreach ($data->organizations as $organization) {
            $organizations[$organization->name] = new Dashboard\Organization(
                $organization, $data->applications);
        }

        return $organizations;
    }
}
```

We are currently migrating (step by step) from jQuery modules to React.js components and are still using data attributes to trigger loading React.js components in the UI. Instead of asserting on the HTML, what we would have done when still rendering the dashboard on the server side, we check for such a data attribute and load the data from the server. For each organization found on the page we then return another object which represents a partial (organization) on the page.

Using this object oriented abstraction of the page allows us to transparently switch between plain HTML rendering and React components while the test will look just like before. The only thing changed is the page object, but this one can still power many tests which can make the effort worth it. On top of those `Organization` partials we can then execute additional assertions:

```
/**
 * @depends testHasDemoOrganization
 */
public function testMonthlyRequestLimitReached(Page\Dashboard\Organization
    $organization)
{
    $this->assertFalse($organization->getMonthlyRequestLimitReached());
}

/**
 * @depends testHasDemoOrganization
 */
public function testHasDemoApplications(Page\Dashboard\Organization $organization)
```

```
{
    $this->assertCount(3, $organization->getApplications());
}

// ...
```

4.5.4 Problems With Page Objects

As you probably can guess providing a full abstraction for your frontend will take some time to write. Those page objects can also get slightly more complex, so that you might even feel like testing them at some point.

Since end-to-end tests also will always be slow'ish (compared to unit tests) we advise to only write those tests for critical parts of your application. The tests will execute full HTTP requests which take time – and nobody runs a test suite which takes multiple hours to execute.

Also remember that, like with any integration test, you probably need some means to setup and reset the environment the tests run on. In this simple example we run the test on the live system and assume that the user has the demo organization enabled. In a real-world scenario you'd boot up your application (provision a vagrant box or docker container), reset the database, mock some services, prime the database and run the tests against such a reproducible environment. This takes more effort to implement, again.

While the tests are immune to UI changes this way (as long as the same data is still available) they are not immune to workflow changes. If your team adds another step to a checkout process, for example, the corresponding Page Object tests will still fail and you'll have to adapt them.

4.5.5 Conclusion

Page Objects can be a good approach to write mid-term stable end-to-end tests even for complex applications. By investing more time in your tests you can get very readable tests which are easy to adapt to most UI changes.

4.6 Database Tests With PHPUnit

Tobias Schlitt at 4. October, 2016[30]

Most of us do not use PHPUnit solely for Unit Tests but we also write Integration or Acceptance Tests with PHPUnit. One very common question then is how to interact with the database correctly in those tests. Let me show you the different options and their trade offs...

There are multiple aspects of database tests where our decision has impact on test atomicity and test runtime. All decisions boil down to: More test atomicity leads to longer test runs, and we can buy test speed by omitting test atomicity. One might be tempted to immediately favour test atomicity but fast tests are a feature which is often underappreciated. If your tests are slow (more then 30 seconds or even taking minutes) developers will not run those tests before each commit any more – they feel it just takes them too much time. Your tests will still be run on your Continuous Integration server but even when your tests fail it is annoying to debug them, because you have to wait so long until the failing test is reached.

4.6.1 Removing Data versus Schema Reset

You have a schema of your database laying around somewhere, right? It should even be somewhere in your source code repository to be able to initialize a full build of your application. We prefer using something like DBDeploy[31] to maintain and document changes to our schema but the solution does not really matter. With a schema you can drop the entire database to clean it up before a test and re-apply the schema. This works well with MySQL or SQLite but takes a lot of time with systems like Oracle.

The other idea is to reset the tables your test modified. Since this does not do any schema changes but just removes data it is usually faster. The most common implementation is to remove data manually in a `tearDown()` method in all tables you know the test modified. The problem now is that your application will change. This means that the tables data is written into will also change. At some point you'll forget removing some data in some table and this will cause a side effect on another

[30] https://qafoo.com/blog/090_database_tests_with_phpunit.html
[31] http://dbdeploy.com/

test – a horribly hard to debug side effect. And this only works for new data, it will not revert changed data.

Some people are adding something like a `changed` column to each table which automatically recieves the time of the last change to a row. If you are doing this you can execute something like `DELETEFROM $table WHEREchanged >= $test StartTime` on each table. But then again, this only works for data added in a test and would remove all rows which were changed in a test.

Summary

Resetting the full schema is the cleanest approach, but also takes the most time. Resetting a selected number of tables is faster, but also more cumbersome and error-prone. If you have some kind of change management you might be able to use that. Especially for large and complex schemas (where you would prefer a cleaner approach) resetting the full schema often takes far too much time.

4.6.2 Point of Data Reset

Now we know different ways to reset the data in our database, but *when* should we do the reset. There are basically three options:

- Before each test
- Before each test class
- Once before the whole test run

Before Each Test

The `setUp()` method in `PHPUnit_Framework_TestCase` is the right point to run initialisations before each test and we can initialize the database here if our test wants to access the database and clean it up later in the `tearDown()` method. This way we preserve the atomicity of our tests. But since a single cleanup can take up to some seconds this is only feasible for simple and small schemas.

Before Each Test Class

PHPUnit offers the two (static) methods `setUpBeforeClass()` and `tearDownAfterClass()` which are run once before any test in the test case class is executed and respectively after all tests in the test case class have been executed.

These methods allow us to reset or setup the database once for entire test case. If you do something like this you must remember that the tests in this test case depend on each other. The first test might create some row and the second test then accesses this row and uses it for something. I usually add `@depends` annotations on the tests to make this extra obvious. Also the second test will not work without the first one anyways. You can see an example for such a test case in our TimePlanner demo project[32].

The initialisation of the database happens in the mentioned methods in a base class for our integration tests[33]. The methods look slightly more complicated because the tests are run against three different databases (MySQL, SQLite and CouchDB) and all are resetting the entire schema using Doctrine.

With this method you cannot easily reorder your tests in a test case anymore, but you can still run your tests cases in any order, since they are still supposed to be independent. If you want to debug a single test it still means that you have to run all tests which it depends on before it – which can be annoying.

Before the whole test run

PHPUnit allows you to specify a bootstrap file, either on the CLI or in the `phpunit.xml`. Nowadays this is mostly used to just specify the `vendor/autoload.php` from composer, but you can do more in this file. For example you can initialize your database schema. You can also do this in some build step which is always executed before each test run or somewhere similar.

This allows you to initialize the database once before all tests. This approach is supposed to be the fastest because it minimizes database interaction and thus IO. But there are some drawbacks:

- If you don't specify the order of your test cases manually the order they are picked up by PHPUnit might changed depending on your OS or file system. This can lead to very weird and hard to debug test failures.
- State is leaked between all your tests across your complete test suite. This again can lead to situations which are very hard to debug.

[32] https://qa.fo/book-PublicHolidayGatewayTest
[33] https://qa.fo/book-IntegrationTest

- There will be situations where you can debug one test only when running the entire test suite because it depends on the state of some random test you might not even be aware of.

Summary

Except for very small and simple projects it is usually best to initialize the database before each test class. This seems to be the be best compromise between test stability and speed for most projects – your mileage might vary.

4.6.3 Mocking the Database Away

Another entirely different option which comes to mind is mocking the entire interaction with your database. But such complex mocks are often error prone and a lot of effort to implement and maintain. If your database interaction is very simple it might work for you. In any project with non trivial queries (aggregations, joins, ...) you probably do not want to walk this path.

In a MySQL project you can think about using a SQLite in-memory database, but at some point you'll discover differences between these two database management systems. And this could mean production bugs which will not be discovered during testing.

4.6.4 Conclusion

In general you should try to reduce access to global state (like a database) as much as possible in your tests. But at some point you'll want to test the interaction with your database – in this case you must decide for a way which works best for your project. In most projects the best way for us seems to reset the full schema before each test case which interacts with the database.

4.7 Database Fixture Setup in PHPUnit

Tobias Schlitt at 15. November, 2016[34]

We already discussed when and how to reset a database between tests (See: Database Tests With PHPUnit). But on top of that you often need some basic or more complex data to run the tests against. We discuss different variants and their respective trade-offs in this post...

4.7.1 Dump & Insert Live Data

The simplest approach - we still see this in the wild - for schema and test data management is to dump and replay live data for testing purposes. The benefit here is that you only have to adapt data in one place when the schema changes and it is really easy to set up. But there are a couple of drawbacks to this approach:

- **Large live data sets**
  The live data usually is "large" – at least a couple of hundred MB (while database are usually not considered large before reaching TB). It takes a lot of time to reset a database even with such a dump. You do not want to spend such an amount of time before each test, test suite or even before each test run.
- **Boundary value test cases**
  There is often a need for special test cases, like strange characters or other type of boundary values in your tests. Those data sets might not exist in the live data so you end up creating this data in your tests on top of the data dump anyways.
- **Data privacy**
  Depending on your use case and business your developers should not have access to all live data. There might be sensitive information which should be locked down on your servers. In this case a live SQL dump is no option. Especially in Germany we might be required by law to lock certain data away from certain people.
- **Changing data on the live system**

[34] https://qafoo.com/blog/091_database_fixture_setup_in_phpunit.html

It is obvious that data on the live system changes over time. This can make it hard to author tests that are stable and reproducible. In addition, tests might become less meaningful if you need to craft them in a way that they can cope with changing live data.

Modified Live Data Set

To avoid the problems mentioned above a next step usually is a modified SQL file, which is smaller and does contain sensible test data, like boundary values or stable data without any sensitive information for reproducible tests.

The problem which arises now is that you have to adapt two files when you change the data structure. And the schema and properties of the data *will* divert over time – no matter how careful you are. In the end this approach is always hard to maintain, so what can be done?

First we suggest you implement some kind of sensible schema management like DBDeploy[35], or even Doctrine Migrations if this works for your use-case. In this post we want to focus on the data / fixture management, though.

4.7.2 Base Data

Most applications require a set of base data, like default or admin users, some groups and permissions or something similar. This will be same in all installations (production, staging, development, testing, ...). Depending on your schema management solution you will be able to insert this data during schema initialization. With DBDeploy you can just add some `INSERT` statements with Doctrine you could use Doctrine data fixtures[36] (or even the Symfony bundle[37]).

4.7.3 Test Data

The more important thing is the test data. Inserting sensible test data is tightly coupled to your test database resetting strategy (See: Database Tests With PHPUnit) which we discussed earlier.

[35] http://dbdeploy.com/
[36] https://github.com/doctrine/data-fixtures
[37] https://qa.fo/book-index

Before Each Test

When you reset your database before each test you also want to insert the test data right inside the test. This is very obvious, makes tests easy to read and understand. In theory this is clearly the best solution. But as mentioned in the referenced blog post this will cost a lot of time to execute and likely be to slow for any non-trivial application.

Before Each Test Class

When you reset the database before each test case you can create the data throughout the test case. An simple common CRUD example could be the following tests:

1. Create a new data set
2. Load data set
3. Fail loading a non-existent data set
4. Load listing (with one item)
5. Delete data set
6. Fail deleting non-existent data set

Those tests depend on each other which should be indicated by using `@depends` annotations like in this example from our demo project[38].

This approach is still very clean and from reading the test case you can understand the full context. Another developer will still be able to understand what is going on. This is a lot harder when the data is inserted in another place, since you'll always have to look in multiple places to get the full image. And new developers might not yet know all the places to look at. Tests which provide all the context you need to know in one file are very helpful for everybody.

This strategy will get more complex if you have dependent data – like tests for user permissions, which also require users and groups to exist. You could either use custom helpers or tools like Alice[39] for this.

Before the whole test run

If you decided to only reset the database once before all tests are run you usually need a more complete data fixture. You will want to fill all tables with some basic

[38] https://qa.fo/book-PublicHolidayGatewayTest
[39] https://github.com/nelmio/alice

data you can work with. Especially in such a case tools like Alice[40] are very useful. By relying on Faker[41] you even get sensible looking data without too much custom work.

4.7.4 Conclusion

The way you initialise your database fixtures depends on your test schema management. We suggest to reset schemas at the begin of each test case and creating the data right in the test case. This proved to be a good compromise between speed, test (case) atomicity and test readability. Tools like Alice[42] and Faker[43] can ease the process of creating data a lot.

[40] https://github.com/nelmio/alice
[41] https://github.com/fzaninotto/Faker
[42] https://github.com/nelmio/alice
[43] https://github.com/fzaninotto/Faker

4.8 Using Traits With PHPUnit

Kore Nordmann at 29. November, 2016[44]

As we already wrote that "Code Reuse By Inheritance" (See: Code Reuse By Inheritance) has lots of problems and we consider it a code smell. You should always aim to use Dependency Injection, most likely Constructor Injection. But with test cases in PHPUnit we cannot do this because we have no control about how and when our test cases are created. There are a similar problem in other frameworks, like we discussed in "Object Lifecycle Control" (See: Object lifecycle control). We also blogged about traits as a Code Smell (See: Utilize Dynamic Dispatch), but let me show and explain why they might be fine to use in your test cases.

So in PHPUnit it is common to reuse code by inheritance. Even by now we often create some base class providing common functionality. This works fine and is OK if this really defines an "is a"-relationship. Let's continue with an example from our Page Object test repository[45]. The patterns described here are usually not required for Unit Tests (which you should always also write) but mostly for integration or functional tests.

4.8.1 An Example

The example repository[46] implements functional tests for a website using the Page Object pattern (See: Introduction To Page Objects). This means that the tests access a website and assert on its contents, try to fill forms, submit them and click links. Such tests are a useful part of your test mix (See: Finding the right Test-Mix), but should never be your only tests.

Let's see the options for code reuse we employ in this little test project – and the reasoning behind it. We start with a simple test case:

```
class LoginTest extends FeatureTest
{
    public function testLogInWithWrongPassword()
    {
        $page = (new Page\Login($this->session))->visit(Page\Login::PATH);
```

[44] https://qafoo.com/blog/092_using_traits_with_phpunit.html
[45] https://github.com/QafooLabs/PageObjects
[46] https://github.com/QafooLabs/PageObjects

```
        $page->setUser(getenv('USER'));
        $page->setPassword('wrongPassword');
        $newPage = $page->login();

        $this->assertInstanceOf(Page\Login::class, $newPage);
    }

    // ...
}
```

This test case extends from a base class `FeatureTest` to re-use its functionality. The base class uses PHPUnits `setUp()` method to setup and start the mink test driver which will act as a browser to access the website. And it provides a default `tearDown()` method to reset the browser state again.

In this case the inheritance defines a clear "is a" relationship with the `FeatureTest` and it overwrites default methods in the PHPUnit stack which should be overwritten in any feature test. The `FeatureTest` again extends an `IntegrationTest` which is empty in this example but normally would provide access to the application stack to reset a database, access random services or something similar. We just do not need anything like this in this little test project. Since functional tests can be considered a superset of integration tests this is fine again and provides common functionality which belongs to all tests of this type.

4.8.2 Traits

Let's take a look at a slightly more complex test case now:

```
class DashboardTest extends FeatureTest
{
    use Helper\User;

    // ...

    public function testHasDemoOrganization()
    {
        $this->logIn();

        $page = (new Page\Dashboard($this->session))->visit(Page\Dashboard::PATH);

        $organizations = $page->getOrganizations();
        $this->assertArrayHasKey('demo', $organizations);
        return $organizations['demo'];
```

```
    }

    // ...
}
```

In this case we using the `Helper\User` trait to include some functionality – it provides the `logIn()` method which is used in the test `testHasDemoOrganization()`. Not every feature test might need this aspect and in "normal" software you would provide such helpers through constructor injection. But since we do not have any control on the test case creation we include the code using a trait.

The trait enables code reuse – we can use it any test case which requires login. The trait extracts this concern and we do not clutter every test case requiring login with this kind of code.

4.8.3 Whats The Difference?

The trait helps us in this example, the code looks clean, so you might want to ask: **Why would traits ever be considered a code smell?**

One of the most important reasons is that in a test case there probably won't be a reason to change a dependency without adapting the code (Open Closed Principle). In other words: There is no reason for dynamic dispatch.

A trait establishes a dependency to another class which is defined by the name of the trait (instead of an instance of some class which could be a subtype). There is no easy way to change the actually used implementation from the outside. If you include a `LoggerTrait` there is no way to change the used `LoggerTrait`, during tests or when the requirements change, without changing code. Traits establish a `static` dependency which is hard to mock and hard to replace during runtime or configuration.

But we will never mock our test cases, right? And if the use cases change we will change the test cases. This can happen a lot as compared to unit tests.

Especially (Open Source) libraries and extensible software commonly has the requirement that people should be able to change the behaviour without changing the code. Most likely because they do not have direct access to the code or it would have side effects to other usages of the code. But nobody uses your test cases in such a way, thus you are "allowed" to sin in here – at least a little bit.

And there are no other options. This, generally, can be another reason to use traits. Traits are often an option when refactoring legacy software to temporarily use common code before we can migrate to sensible dependency injection. Being a code smell they even help knowing about places which are still not done.

4.8.4 Summary

In tests traits can be a great tool to reuse common code, while we still consider traits a code smell in almost every other case.

4.9 Testing the Untestable

Manuel Pichler at 2. May, 2017[47]

A long time ago I wrote a blog post about Testing file uploads with PHP[48] where I have used a CGI PHP binary and the *PHP Testing Framework* (short PHPT), which is still used to test PHP itself and PHP extensions.

Since the whole topic appears to be still up-to-date, I would like to show a different approach how to test a fileupload in PHP in this post. This time we will use PHP's namespaces instead of a special PHP version to test code that utilizes internal functions like `is_uploaded_file()` or `move_uploaded_file()`. So let's start with some *code under test* example source:

```
namespace Qafoo\Blog;

class UploadExample
{
    protected $target;

    public function __construct(string $target)
    {
        $this->target = rtrim($target, '/') . '/';
    }

    public function handle(string $name): void
    {
        if (false === is_uploaded_file($_FILES[$name]['tmp_name'])) {
            throw new FileNotFoundException();
        }

        $moved = move_uploaded_file(
            $_FILES[$name]['tmp_name'],
            $this->target . $_FILES[$name]['name']
        );
        if (false === $moved) {
            throw new FileNotMovedException();
        }
    }
}
```

Even if we can mockout the magic `$_FILES` super global variable that we use here::

```
public function handle(array $files, string $name): void
```

[47] https://qafoo.com/blog/102_testing_the_untestable.html
[48] https://qa.fo/book-013_testing_file_uploads_with_php

```
{
    if (false === is_uploaded_file($files[$name]['tmp_name'])) {
        throw new FileNotFoundException();
    }

    $moved = move_uploaded_file(
        $files[$name]['tmp_name'],
        $this->target . $files[$name]['name']
    );
    if (false === $moved) {
        throw new FileNotMovedException();
    }
}
```

we use the internal functions `is_uploaded_file()` and `move_uploaded_file()`, which work one some internal request data structure that we cannot access nor modify. Despite this internal handling we still can at least test the negative path:

```
namespace Qafoo\Blog;

use PHPUnit\Framework\TestCase;

class UploadExampleTest extends TestCase
{
    /**
     * @expectedException \Qafoo\Blog\FileNotFoundException
     */
    public function testHandleThrowsFileNotFound(): void
    {
        $files = [
            'file_invalid'   => [
                'name'     => 'foo.txt',
                'tmp_name' => '/tmp/php42up23',
                'type'     => 'text/plain',
                'size'     => 42,
                'error'    => 0
            ]
        ];

        $upload = new UploadExample(sys_get_temp_dir());
        $upload->handle($files, 'file_invalid');
    }
}
```

But it's impossible to write tests for the happy path of the `handle()` method.

4.9.1 So What Can We Do?

We can use a small trick that utilizes namespaces and PHP's lookup behavior for functions to inject/mock our own implementations of the two functions during the tests.

Let's have a look at how PHP resolves functions within namespaced source code. In the following example both calls will invoke the same internal function `is_uploaded_file()`, ...

```
namespace Qafoo\Blog {
    var_dump(is_uploaded_file('test'));
    var_dump(\is_uploaded_file('test'));
}
```

... while in this example the first call will call our own implementation of `is_uploaded_file()` and the second call still invokes the internal function:

```
namespace Qafoo\Blog {
    function is_uploaded_file($name) {
        return ('awesome' === $name);
    }

    var_dump(is_uploaded_file('test'));
    var_dump(\is_uploaded_file('test'));
}
```

This happens because PHP first makes a function lookup in the local namespace for all function calls that don't have a leading \ and only if no local declaration exists it makes a lookup in the global namespace. For us that means we have now found an approach to mock out the internal functions in our test case, because we can overwrite the two upload functions in the namespace:

```
namespace Qafoo\Blog;

function is_uploaded_file($tmpName): bool
{
    return in_array($tmpName, ['/tmp/php42up23', '/tmp/php23up17']);
}

function move_uploaded_file($tmpName, $to): bool
{
    return in_array($tmpName, ['/tmp/php42up23']);
}
```

And our final test case that tests all execution paths will look like:

```
namespace Qafoo\Blog;

require __DIR__ . '/UploadExample.php';

use PHPUnit\Framework\TestCase;

class UploadExampleTest extends TestCase
{
    private $files = [
        'valid'    => [
            'name'     => 'foo.txt',
            'tmp_name' => '/tmp/php42up23',
        ],
        'invalid'    => [
            'name'     => 'bar.txt',
            'tmp_name' => '/tmp/php42up17',
        ],
        'move_fail' => [
            'name'     => 'baz.txt',
            'tmp_name' => '/tmp/php23up17',
        ],
    ];

    /**
     * @expectedException \Qafoo\Blog\FileNotFoundException
     */
    public function testHandleThrowsFileNotFound(): void
    {
        $upload = new UploadExample(sys_get_temp_dir());
        $upload->handle($this->files, 'invalid');
    }

    /**
     * @expectedException \Qafoo\Blog\FileNotMovedException
     */
    public function testHandleThrowsFileNotMoved(): void
    {
        $upload = new UploadExample(sys_get_temp_dir());
        $upload->handle($this->files, 'move_fail');
    }

    /**
     *
     */
    public function testHappyPath(): void
    {
        $upload = new UploadExample(sys_get_temp_dir());
```

```
        $upload->handle($this->files, 'valid');

        $this->addToAssertionCount(1);
    }
}
```

That's it, now you know how to write fast and reliable test for code that handles file uploads.

But wait, why have I titled this post with "Testing The Untestable"? Because this provides you much much more than just testing file uploads: It gives you a new and powerful testing toolbox. Imagine you are using ext/filter or you are using any of the file functions to access an external service. All this can be mocked out with this technique, like here:

```
namespace Acme\Services;

class ExternalDataProvider
{
    private $apiUrl = 'http://api.example.com/v/2.1/';

    public function getItems(): array
    {
        // ...
        $data = file_get_contents($this->apiUrl);
        // ...
    }
}

namespace Acme\Services;

use PHPUnit\Framework\TestCase

class ExternalDataProviderTest extends TestCase
{
    public function testGetItems(): void
    {
        // ...
    }
}

function file_get_contents($path) {
    if (preg_match('~^https?://~', $path) {
        // Load some fixture here
    }
    // Call the original here
```

```
    return \file_get_contents($path);
}
```

This isn't something new and was already possible in 2010 when I wrote the original post, but I hope this gives you a powerful tool.

4.10 Outside-In Testing and the Adapter and Facade Patterns

Benjamin Eberlei at 5. July, 2016[49]

We at Qafoo are big fans of outside-in testing as described in the book "Growing Object-Oriented Software, Guided by Tests[50]" (Steve Freeman and Nat Pryce). As part of our workshops on Test-Driven Development we explain to our customers[51] how testing from the outside-in can help find the right test-mix.

The technique puts a focus on test-driven-development, but instead of the traditional approach starts at the acceptance test level. The first test for a feature is an acceptance test and only then the feature is implemented from the outside classes first (UI and controllers), towards the inner classes (model, infrastructure). Mocks are used to describe roles of collaborators when starting to write tests for the outside classes. When a class is tested, the roles described by interfaces (See: Abstract Classes vs. Interfaces) are implemented and the testing cycle starts again with mocks for the collaborators.

Outside-In testing leads to interfaces that are written from what is useful for the client object using them, in contrast to objects that are composed of collaborators that already exist. Because at some point we have to interact with objects that exist already, we will need three techniques to link those newly created interfaces/roles to existing code in our project:

1. The adapter pattern, mostly for third-party code
2. The facade pattern, mostly to structure your own code into layers
3. Continuous refactoring of all the interfaces and implementations

This blog post will focus on the facade adapter pattern as one of the most important ingredient to testable code.

Even if very well tested, APIs of third party libraries or frameworks are usually not suited for projects using any form of automated testing, either because using them directly requires setting up external resources (I/O) or mocking them is complex, maybe even impossible.

warning

[49] https://qafoo.com/blog/087_outside_in_testing_adapter_pattern.html
[50] http://www.growing-object-oriented-software.com/
[51] https://qafoo.com/blog/051_testing_sweet_spot.html

> Side Fact: This even affects libraries that put a focus on testing, such as Doctrine2, Symfony2 or Zend Framework. The reason is that libraries often provide static APIs, fluent APIs, facades with too many public methods or complex object interactions with law-of-demeter violations.

Take a common use-case, importing data from a remote source into your own database. In a Symfony2 project with Doctrine and Guzzle as supporting libraries, the feature could easily be implemented as a console command, using only the third-party code and some glue code of our own:

```
<?php
namespace Acme\ProductBundle\Command;

class ImportProductCommand extends ContainerAwareCommand
{
    protected function configure() { /* omitted */ }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $client = $this->getContainer()->get('guzzle.http_client');
        $entityManager = $this->getContainer()->get('doctrine.orm.default_entity_
            manager');

        $request = $client->get('http://remote.source/products.xml');
        $response = $request->send();

        $products = $this->parseResponse($response);

        foreach ($products as $product) {
            $entityManager->persist($product);
        }

        $entityManager->flush();
    }

    protected function parseResponse($response) { /** omitted */ }
}
```

Looks simple, but in the real world, you can safely assume there is quite some complexity in `parseResponse` and possibly even more logic inside the loop over all the `$products`.

In this scenario, the code is completely untestable, just by combining the APIs of Guzzle, Doctrine and Symfony.

Lets take a different approach, starting with the highest level description of the feature as a test written in Gherkin for Behat:

```
Scenario: Import Products
    Given a remote service with products:
        | name | description       | price
        | A    | Nice and shiny    | 100
        | B    | Rusty, but cheap  | 10
    When I import the products
    Then I should see product "A" in my product listing
    Then I should see product "B" in my product listing
```

This will be our acceptance (or end-to-end) test that will use as many external systems and I/O as possible. We might need to mock data from the remote product service, but when possible we should use the real data. This test will require the UI to be done as well as the necessary database and remote system APIs and therefore will fail until we have implemented all the code necessary. So lets focus on the code directly with our first unit-test and lets imagine how we want the code to be used:

```
<?php
namespace Catalog;

class ImportProductTest extends \PHPUnit_Framework_TestCase
{
    public function testImportTwoProducts()
    {
        $remoteCatalog = \Phake::mock('Catalog\RemoteCatalog');
        $productGateway = \Phake::mock('Catalog\ProductGateway');

        $productA = $this->createSampleProduct();
        $productB = $this->createSampleProduct();

        \Phake::when($remoteCatalog)->fetch()->thenReturn(array($productA,
            $productB));

        $importer = new ProductImporter($productGateway);
        $importer->import($remoteCatalog);

        \Phake::verify($productGateway)->store($productA);
        \Phake::verify($productGateway)->store($productB);
    }
}
```

When we fetch 2 products from the `$remoteCatalog` then those products should be passed to the `$productGateway`, our storage system. To describe the use-case in this simple way, we have abstracted `RemoteCatalog` and `ProductGateway`, which will act as facades used in our `ProductImporter`. The implementation is very simple:

```
<?php

namespace Catalog;

class ProductImporter
{
    /**
     * @var ProductGateway
     */
    private $productGateway;

    public function import(RemoteCatalog $remoteCatalog)
    {
        $products = $remoteCatalog->fetch();

        foreach ($products as $product) {
            $this->productGateway->store($product);
        }
    }
}
```

We haven't seen the Guzzle and Doctrine code here again, instead we have written code that is entirely written in our business domain and uses concepts from this domain.

Lets move to the implementation of the `RemoteCatalog` starting with a test:

```
<?php

namespace Catalog\RemoteCatalog;

class HttpCatalogTest extends \PHPUnit_Framework_TestCase
{
    public function testGetAndParseData()
    {
        $client = \Phake::mock('Catalog\Adapter\HttpClient');
        $parser = \Phake::mock('Catalog\RemoteCatalog\Parser');
        $url = 'http://remote.local';

        $productA = $this->createSampleProduct();
```

```
        \Phake::when($client)->get($url)->thenReturn('<xml>');
        \Phake::when($parser)->parse('<xml>')->thenReturn(array($productA));

        $catalog = new HttpCatalog($url, $client, $parser);
        $data = $catalog->fetch();

        $this->assertSame(array($productA), $data);
    }
}
```

This test again perfectly clear explains how fetching a catalog with HTTP should work on a technical level. Notice how we choose a very simple API for the HTTP client, one that is fetching and $url and retrieving the body as string. We don't need more.

```
<?php
namespace Catalog\RemoteCatalog;

use Catalog\RemoteCatalog;

class HttpCatalog implements RemoteCatalog
{
    private $url;
    private $client;
    private $parser;

    public function fetch()
    {
        $body = $this->client->fetch($this->url);
        return $this->parser->parse($body);
    }
}
```

The HttpClient is a real adapter for us now, we want this very simple API and we are going to use Guzzle to implement it:

```
<?php
namespace Catalog\Adapter\Guzzle;

use Guzzle\Http\Client;

class GuzzleHttpClient implements HttpClient
{
    private $client;

    public function __construct(Client $client)
```

```
    {
        $this->client = $client;
    }

    public function fetch($url)
    {
        $request = $this->client->get($url);
        $response = $request->send();

        return $response->getBody(true);
    }
}
```

Implementing the Guzzle client we have reached a "leaf" of our object graph. It is important to see how only the Guzzle adapter actually uses Guzzle code. A complete solution would also require to handle the Guzzle Exceptions, but that is only a trivial task to introduce as well.

You can continue with this example and implement the `Parser` and the `ProductGateway` objects. The technique stays the same: Think in terms of what you want the API to look from the outside and invent collaborators that help you think about the problem. Then implement them until you get to the "leafs" of the object graph. Only the leafs should actually contain code to third party software.

Starting with new requirements of the system we will probably be able to reuse some of the code: The HttpClient interface is very useful in a good number of use-cases. The HttpCatalog can be used with specific Parser implementations to import many different product catalogs that have an HTTP interface. And if your customer uses some other protocols like FTP, BitTorrent or anything else then we are good as well. The `ProductGateway` will probably be used in various places to give us access to finding and storing products.

4.10.1 Conclusion

Specification and testing of the requirements from the outside requires you to think about what you need first and not on what you have and how you might combine this. This helps to design simple and reusable objects.

But outside-in testing has additional benefits: When you don't test everything in your system with unit-tests (which is not uncommon) for any of the various reasons (time, prototyping, not core domain, ..) then you still have at least one acceptance

test that verifies the complete feature and maybe some unit-tests for the tricky implementation details.

Overall we think making yourself familiar with outside-in testing is beneficial to designing testable and maintainable applications.

4.11 Behavior Driven Development

Tobias Schlitt at 8. March, 2013[52]

While unit, integration and system tests - especially combined with the methodology of Test Driven Development (TDD) - are great ways to push the technical correctness of an application forward, they miss out one important aspect: the customer. None of these methods verify that developers actually implement what the customer desires. Behavior Driven Development[53] (BDD) can help to bridge this gap.

The methodology of BDD is actually derived from TDD. Instead of writing a test for a single code unit upfront, an acceptance test for a certain behavior is formulated. More important, the test case is formulated in the ubiquitous language of the project, allowing non-technical persons to read it as the specification of the desired feature.

In the following I want to outline the methodology of BDD from a developer's perspective.

4.11.1 Example

So much for theory. In practice I prefer to explain things on the basis of examples. So here is an acceptance test:

```
Feature: The existing CFPs should be listed
    As a visitor I want to get a listing of all existing CFPs.

    Scenario: If there are no CFPs, I see an empty list.
        Given there are no CFPs
        When I view "CFP Listing"
        Then I see 0 CFPs listed
```

The snippet above shows an acceptance test formulated in a language that is based upon the Gherkin framework for Domain Specific Languages (DSLs). The first lines are narrative, documenting the feature to be tested. They do not fulfil any technical purpose, think of it as if it was a class level doc block.

[52] https://qafoo.com/blog/036_behavior_driven_development.html
[53] https://en.wikipedia.org/wiki/Behavior-driven_development

What follows is the definition of a specific scenario within the feature. A scenario is basically a user story, as you probably recognize it from your agile development mode. Again, the introductory sentence is documentation and not processed.

The block following afterwards is the actual test case. These three sentences are on the one hand human readable. On the other hand, they are structured so simple and generic that they can be processed by a computer with very little effort. The first word on every line is a keyword from the Gherkin language, while `Given` indicates a precondition, the `When` part indicates the test stimulus and `Then` formulates an expectation.

The wonderful thing about this specification is that your customer can actually read and verify it. You can write down what you, as a developer, understood how your customer wants the system to behave. You can send the specification to her/him and ask her/him if this is exactly what she/he desires. If she/he agrees, you can start implementing right away and verify your progress against the specification, as it is executable through a BDD test tool.

Possibly, if you have a technically skilled customer and train him quite a bit, he will probably even be able to adjust scenarios or even write some on his own.

4.11.2 Behat

The Gherkin example from above is almost useless without a proper tool to execute it. Of course, there is no tool that can do it right away, because no software can understand the project specific sentences. However, frameworks exist that offer you a basis. For the PHP world, the toolkit of choice is called Behat[54].

Behat provides you with the basic infrastructure for BDD and enables you to easily work with custom sentences. Specifically, you create a so-called `FeatureContext` for your test cases that contain a method for every sentence in your projects ubiquitous language. For example:

```
class ListingFeatureContext extends BehatContext
{
    // ...

    /**
     * @Given /^there are no CFPs$/
```

[54] http://behat.org/

```
     */
    public function thereAreNoCfps()
    {
        $this->cleanupDatabase();
    }
}
```

The extract from the `ListingFeatureContext` above shows the method that reacts to the sentence `Given there are no CFPs`. Using an annotation, Behat connects the method to the sentence. Whenever it discovers that sentence in a test scenario, it will execute the method.

4.11.3 Rationale

Of course, the examples shown above are only very rudimentary, missing e.g. variables and other advanced features. However, they should have explained what BDD is all about: Communication. Especially in teams which follow the Domain Driven Design (DDD) approach, a ubiquitous language for the project domain is already practiced. Toolkits such as Behat provide you with the environment to express expectations in this language and make these executable.

The two essential ideas behind this are a) to ease communication with the client and b) to bridge the gap between (important!) tests for technical correctness and business expectations.

4.11.4 Conclusion

BDD is an interesting approach that can work especially well for projects that build on extensive business logic and such that follow Domain Driven Design. Besides that, Behat can become a tool of choice for acceptance tests also in respect to its integration with Mink[55] and Symfony2[56].

What are your experiences with BDD and with Behat?

[55] http://extensions.behat.org/mink/
[56] http://extensions.behat.org/symfony2/

4.12 Code Coverage with Behat

Manuel Pichler at 3. April, 2013[57]

There is generally no point in having code coverage for Behat test cases because of their nature: The purpose of an acceptance test is to assert a certain behavior of an application, not to technically test a piece of code. Therefore, there is no point in checking for uncovered code pieces in order to write a Behat test for it.

That said, there is still a scenario where you want to peek at code coverage of Behat tests: When creating them as wide-coverage tests before starting to refactor legacy code. Behat in combination with Mink provides you with a great tool for such tests.

Before you can start with refactoring legacy code you need tests to ensure that you don't break working functionality. Web acceptance tests on basis of Behat and Mink are a great tool to realize these. But how can you detect if the code you are about to refactor is touched by at least one test? Code coverage can be of assistance there.

4.12.1 Preparation

Since Behat does not ship with code coverage (for very good reason), you need some hand work to get that done, but not much. In order to get started, you need to install the PHP_CodeCoverage library[58] and phpcov[59], most probably via PEAR using:

```
$ pear config-set auto_discover 1
$ pear install -a pear.phpunit.de/phpcov
```

4.12.2 Collecting Code Coverage

Since the Behat tests stimulate your application through external calls, it is not possible to generate code coverage right from the test code. Instead, you need to trigger the code coverage collection from your application code:

[57] https://qafoo.com/blog/040_code_coverage_with_behat.html
[58] https://github.com/sebastianbergmann/php-code-coverage
[59] https://github.com/sebastianbergmann/phpcov

```
<?php

// ... forbid production access here ...

$calculateCoverage = file_exists("/tmp/generate-behat-coverage");

if ($calculateCoverage) {
    require 'PHP/CodeCoverage/Autoload.php';

    $filter = new PHP_CodeCoverage_Filter();
    $filter->addDirectoryToBlacklist(__DIR__ . "/../vendor");
    $filter->addDirectoryToWhitelist(__DIR__ . "/../src");

    $coverage = new PHP_CodeCoverage(null, $filter);
    $coverage->start('Behat Test');
}

// ... run your application here ...

if ($calculateCoverage) {
    $coverage->stop();

    $writer = new PHP_CodeCoverage_Report_PHP;
    $writer->process($coverage, __DIR__ . "/../log/behat-coverage/" . microtime(
        true) . ".cov");
}
```

At first the code detects if code coverage information should be gathered by checking if the file `/tmp/generate-behat-coverage` exists. You can touch and delete that one manually or from your test setup.

The next code block loads and initializes the code coverage collection, creates a filter for 3rd party code and starts the code coverage collection. After that, the comment indicates to run your application, which might e.g. be a Symfony2 kernel handling call.

The final lines write the code coverage information into a file for further processing. It will create a dedicated file for each request, where these files then need to be merged later.

4.12.3 Running Tests

With the shown code in place, you can trigger a Behat test run with code coverage using the following Ant code, for example:

```
<target name="behat-coverage" depends="clean, initialize">
    <delete dir="${commons:logsdir}/behat-coverage" />
    <mkdir dir="${commons:logsdir}/behat-coverage" />

    <touch file="/tmp/generate-behat-coverage" />
    <antcall target="behat" />
    <delete file="/tmp/generate-behat-coverage" />

    <exec executable="phpcov" failonerror="false" dir="${basedir}">
        <arg value="--merge" />
        <arg value="--html" />
        <arg value="${commons:logsdir}/../coverage/behat" />
        <arg value="${commons:logsdir}/behat-coverage" />
    </exec>
</target>
```

The Ant target first cleans up code coverage from previous runs. It then touches the file that indicates to the application to run code coverage, executes Behat and removes the trigger file again. Then the `phpcov` utility is executed to merge the results of all requests into a single coverage report and generate HTML from it.

4.12.4 Conclusion

Code coverage is completely out of scope for acceptance tests. However, if you abuse Behat to create wide-coverage tests before refactoring, it might be of help to you to see what is still missing before you start hacking.

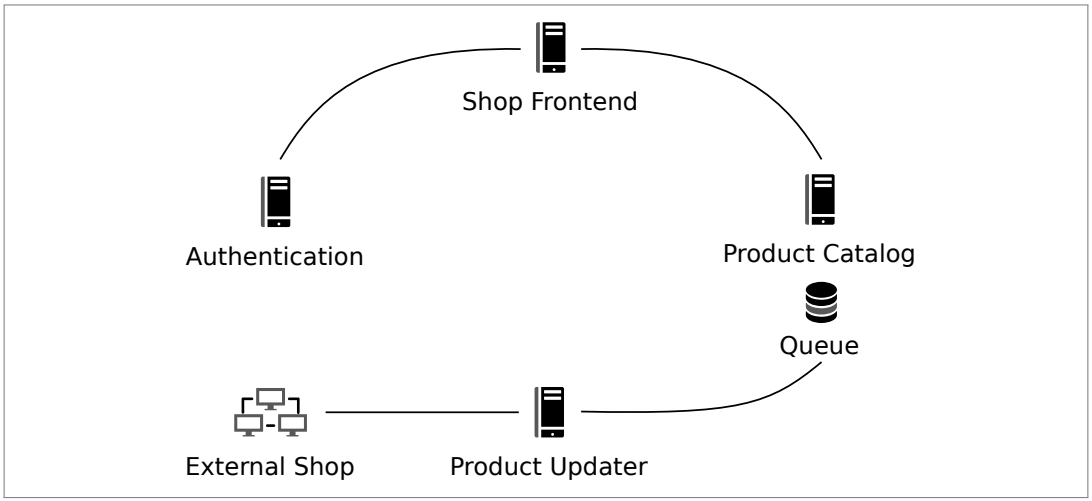
4.13 Testing Micro Services

Tobias Schlitt at 16. September, 2014[60]

I recently had a short exchange with Ole Michaelis on Twitter about how to end-to-end test micro services[61]. Since I didn't have time to make my whole case, Ole suggested that I blog about this, which I'm happily doing now.

The idea behind micro service architecture was discussed by Martin Fowler[62] in a nice to read blog post, so I won't jump on that topic in detail here.

At Qafoo we run a flavor of what would today be called a micro service architecture for Bepado[63]. I sketched a simplified fraction of that below. While the frontend is roughly a classical web application, the other shown components are modelled as independently deployable, slim web services. The *Product Updater* service is responsible for fetching product updates from remote shops. It validates the incoming data and queues the product for indexing. Indexing is then performed in a different service, the *Product Catalog*.



[60] https://qafoo.com/blog/071_testing_micro_services.html
[61] https://twitter.com/CodeStars/status/501336599419187200
[62] http://martinfowler.com/articles/microservices.html
[63] http://bepado.com

While this shows only a small part of the full architecture, typical problems with system-testing such environments become evident. Sure, you can unit test classes inside each micro service and apply classical system tests to each service in isolation. But, how would you perform end-to-end system tests for the full application stack?

Trying to apply the classical process of "arrange, act, assert" did not work because of two reasons: a) there is a good portion of asynchronity (queuing) between services which prevents from reasoning about the point in time when a certain state of the system can be asserted. b) setting up the fixture for the full system to trigger a certain business behavior in isolation would be quite complex task.

A fix for a) could be to mock out the queues and have synchronous implementations for the test setup. Still, the accumulated service communication overhead (HTTP) makes timing hard. Again, you could mock out the service layer and replace it with something that avoids API roundtrips for testing. But this would mean that you need to maintain two more implementations of critical infrastructure for testing purposes only and it would make your system test rather unreliable, because you moved a large bit away from the production setup.

A similar argument applies to topic b): Maintaining multiple huge fixtures for different test scenarios through a stack of independent components is a big effort. Micro fixtures for each service easily become inconsistent.

For these reasons, we moved away from the classical testing approach for our end-to-end tests and instead focused on metrics.

The system already collects metrics from each service for production monitoring. These metrics include statistical information like "number of updates received per shop", various failover scenarios e.g. "an update received from a shop was invalid, but could be corrected to maintain a consistent system state", errors, thrown exceptions and more. For system testing we actually added some more metrics which turned out to be viable in production, too.

In order to perform a system test, we install all micro services into a single VM. This allows the developer to easily watch logs and metrics in a central place. A very simple base fixture is deployed that includes some test-users. In addition to that, we maintain very few small components that emulate external systems with entry points to our application. These "emulation daemons" feed the system with ran-

domized but production-like data and actions. One example is the *shop emulator*, which provides product updates (valid and invalid ones) for multiple shops.

As a developer I can now test if the whole application stack still works in normal parameters by monitoring this system and asserting the plausibility of metrics. This helps us to ensure that larger changes, and especially refactorings across service borders, do not destroy the core business processes of the system.

While this is of course not a completely automated test methodology, it serves us quite well in combination with unit and acceptance tests on the service level: The amount of time spent for executing the tests is acceptable, while the effort for maintaining the test infrastructure is low.

What are your experiences with testing micro service architectures? Do you have some interesting approaches to share? Please feel invited to leave a comment!

P.S. I still wonder if it would be possible to raise the degree of automation further, for example by applying statistical outlier detection to the metrics in comparison to earlier runs. But as long as the current process works fine for us, I'd most probably not invest much time into research in this direction.

4.14 Five Tips to Improve Your Unit Testing

Tobias Schlitt at 13. June, 2017[64]

After you got the hang of unit testing there is still so much space for improvement. In this post I want to share five tips with advanced testers I have seen to influence testing in the right direction.

Do you have additional tips and tricks to improve testing? Please leave a comment!

4.14.1 1. Be Pragmatic About a "Unit"

"A unit is a class" or even "a unit is a single method" are two dogmata people use to explain unit testing. This for good reasons: Following a straight line helps to grasp the concept when getting started, recognizing the code smells discovered through testing issues on that basis is very important and aiming for coherent classes and pure methods/functions is always a good idea.

But as you have practiced unit testing on basis of these dogmas you will notice that in some cases pragmatism beats the dogma. That is perfectly valid. There can be many reasons to test a bunch of classes together instead of focussing on just a single one:

- A class uses multiple rather trivial other classes where it is pure overhead of mocking these (e.g. DTOs with very few logic).
- You are in the phase of refactoring and don't know if the result turns out exactly to be what you want.
- A class dispatches logic to some other classes to make the code re-usable but the logic is only complete in combination.
- The individual classes are rather simple, but them playing together results in an algorithm that's worth testing.
- ...

But beware: Never use an external system (e.g. database, hard disk, web-service) in a unit test.

[64] https://qafoo.com/blog/105_five_tips_improve_unit_testing.html

4.14.2 2. Test Where the Logic is

I'm not a fan of CodeCoverage. But when you just get started with unit testing it is a great tool to reflect your tests and, even more important, the parts of your code which are **not** touched by tests. But following Code Coverage after this initial learning phase often drives people into testing trivials like getters/setters, constructors and so on.

Please do not test these trivials explicitely. It's a nice practice for your first few unit tests, sure. And cases might where you feel the need to test a setter (e.g. if it contains rather complex validation). In all other cases: Trust yourself that you will use the trivial code in other test cases, for examle such from tip 1 or 5.

Instead of focussing on trivials, look where there really *is* logic. In technical terms: Where are the loops, conditions, private methods and so on? Focus on these places. Accept the challenges they offer and write tests for stuff that matters.

4.14.3 3. Continuously Refactor Test Code

As your system grows your test code will (hopefully) grow, too. Some people insist that changing tests is a no-go because there no guarantee to not break a test. I disagree: As within your production code you will find better arrangement for your code and will get a deeper understanding on how to realize certain requirements over time. You will produce duplication and (hopefully!) become aware of it. You will implement hacks to achieve results fast and clean them up when you find better solutions over time.

It is important to reflect over your test code in a very similar way than you do it with the productive code. However, there are two important hints to follow:

a) **Never** change production and test code at the same time. When you refactor your test code, the production code is the reference to asserts tests are still working as they should. And of course your tests are the assertion while working on your production code.

b) Keep test code **simple**. Simple does not mean dirty, it means easy to read and understand. Your goal should neither be to reduce the ammount of test code to a bare minimum nor to find the highest degree of automation. Tip number 4 will go a little bit more into detail here.

4.14.4 4. Build Your Own Set of Utilities

PHPUnit (and other unit test frameworks) ship with a large set of generic tools and utilities. Ranging from assertions over data generators to mock frameworks - you have a large selection inside of frameworks and extensions to download. While these can give you a good technical basis for your tests, none of them provides the golden hammer to suite all your needs.

You will start writing utility methods for setting up certain fixtures (e.g. a `User`, a `Product`) and implement custom assertions (e.g. `assertUniqueProductSet()`) based on your data structures and code patterns. That is a good thing! Identify the patterns that evolve in your test suite and reduce duplication by extracting utilities.

Tool methods can typically reside in common base classes (due to PHPUnits inheritance scheme) or traits (See: Using Traits With PHPUnit).

4.14.5 5. Always Write Tests for Bugs

You can discuss a lot about worthwhile tests and opinions on this topic vary widely. But there is one category that always makes sense: Regression tests for bugs. Whenever you encounter a bug in your software, write a test that fails due to this bug and fix the bug afterwards.

If someone discovers a bug in your software that means that the code is actually in use, so it's important for your users and therefore deserves a test. The pure existence of the bug shows that there is room for another test. And, you will need to reproduce the bug anyway. Doing this by hand is the same effort that doing it through a test case. So the latter version will safe you time and gain you a test where there would have been only a bug fix before.

5. Refactoring

5.1 Loving Legacy Code

Tobias Schlitt at 4. April, 2017[1]

Many developers want to "rewrite the whole application" and "get rid of all that sh*t". Most of them are pretty blank when I tell them that I really like working on such code bases, even if I just jumped into the code. I recently talked about that to the other Qafoo members and all of them agreed to my views. Therefore I want to explain our love of legacy code in this post.

The first thing to realize about existing code bases - no matter how bad you feel their quality is - is that *it provides business value*. The code is in production and people use it and gain value from that. For us as developers that means: The business rules that the appplication is meant to reflect **are already written down in code**. There is no need to discuss the purpose, how things are meant to work and which goals should be achieved. We already have a machine readable version of the business vision and it works. One of the biggest issues in software development has already been taken from our shoulders: Understanding the business and implementing a solution accordingly.

When implementing a new program on the green field you have a high amount of uncertainty about the eventual goals. Extracting and documenting requirements is a tough part for all stakeholders and many iterations need to happen before

[1] https://qafoo.com/blog/100_loving_legacy_code.html

we even reach the stage where software is usable and production ready. Which applications that have grown throughout years these steps have already been done. As developers we can read the eventual decisions how the software should work: It is already there, written down in a language we can naturally understand: Code!

Even better we also know what are common change and extension points and use the technical patterns which are optimal for the change we saw in a couple of years. Developers are often wrong when anticipating change, but in this case we can analyze the past development and have data to base on when anticipating change.

Now we "only" need to move around the code so that its functionality does not change but that we can better maintain it and get rid of technical dept. Of course this is not a picnic, too. But instead of tackling on an additional layer of complexity we can focus on refactoring. If we created a good refactoring basis already, we can even perform that task on the go beside implementing new features and fixing bugs.

I could even go that far to tell you that implementing software on the green field is rather boring for me. Of course there are always smaller parts which are interesting and challenging but most tasks are just boring. It's different with unknown legacy code bases. You need to be creative to apply at least a minimal amount of testing before you can start to refactor. You need to follow the minds of multiple developers of which some even might not be in sight anymore. You need to understand what patterns are realized in a software, what would improve the code quality and find ways to reach that goal without disturbing every day work. This is the interesting stuff. :)

> In legacy software you can let technical patterns emerge for a proven domain.
>
> -- Tobias Schlitt

You can read more about refactoring approaches which can help you in our posts on Extract Methods (See: Basic Refactoring Techniques: Extract Method) and Extract Services (See: How to Perform Extract Service Refactoring When You Don't Have Tests).

5.2 Refactoring with the Advanced Boy Scout Rule

Tobias Schlitt at 30. May, 2017[2]

When we join teams to coach them with refactoring their legacy code base, many of them are overwhelmed by the sheer mass of code. That typically results in the request for "some refactoring sprints" or even "a complete rewrite". Both is obviously not a solution from the business perspective - feature development and bug fixing needs to go on and the refactoring should not eat up the larges portion of time. But where and how should the team start and how should? What we call the "Advanced Boy Scout Rule" has helped many teams to come over this staleness and reach fast results while continuing to deliver business value.

The Boy Scout Rule says

> Always leave the camping ground cleaner than you found it.

Or translated to software development

> Always leave the code cleaner than you found it.

This mantra is hopefully part of your team philosophy latest since every member read "Clean Code". (If not: Book a training with us, now![3])

Building on this idea we apply the "Advanced Boyscout Rule" as follows:

1. When you feel pain while working on a specific code piece, stash your changes and try to resolve the pain through refactoring right now.
2. If you managed to fix it, commit and resume your original work.
3. If you did not manage to resolve the issue within `$x` (maybe 15-20) minutes:
   - Revert the refactoring attempt
   - Add a `@refactor` annotation describing shortly what your issue is
   - If there already is a `@refactor` annotation, append a `!` to it
4. After 1-2 sprints, grep your code for `@refactor` and sort the output by the number of `!` descending.
5. Pick the highest priority issue(s), define a solution strategy and add regular tickets to execute the refactoring in the upcoming sprint.

[2] https://qafoo.com/blog/104_refactoring_advanced_boysout_rule.html
[3] https://qafoo.com/services/workshops/refactoring

This procedure yields you several benefits, like:

- Code becomes better and better in small steps every day
- You reach the most hurting pains first, improving every day development fast
- Immediate visible business value from your development is not lowered significantly
- Instead, you even gain business value by stabilizing the parts of your software first which are touched most

Of course this is only a draft for the concrete implementation in your team. You should change this accordingto your needs. For example instead of leaving comments in the code, some teams prefer to add a post it with the class name to a wall in the office and add dots on their back if there is already one. For other teams it makes sense to focus on specific aspects first like "extract SQL statements into gateways/repositories" or "migrate from arrays to data transfer objects".

I'd like to thank Michael Marlberg[4] who made me aware of this method[5] at first in a project we worked on together.

[4] https://twitter.com/mmahlberg

[5] http://aim42.github.io/#Introduce-Boy-Scout-Rule

5.3 Extended Definition Of Done

Kore Nordmann at 21. February, 2017[6]

When software projects grow it is considered helpful if the software follows an established structure so that every developer finds their way easily. If the used structures and patterns change per developer or even every couple of months or years it will get really hard to maintain the software. There are multiple reasons for this:

- Collective Code Ownership
  Every developer in a team should have the feeling that the code is "their" code. Otherwise they will defer responsibility for features or bug fixes to somebody else. This is not helpful especially when somebody leaves the company or even just is sick or on vacation.
- Common principles ease understanding
  If the code follows common structures it is a lot easier to get to the business / domain logic and find issues there. Otherwise you first need to understand the structures of the code before you can even start thinking about the business / domain logic.
- Focus on what matters
  In the end our code is there to solve certain business / domain problems and not be creative about the patterns used. If you, as a developer, know immediately which patterns you are supposed to use you can focus much more easily on the business / domain concepts. Otherwise many developers will often try to come up with fancy abstraction layers distracting from the actual business / domain logic.

This may sound boring for developers but in our experience with many different teams developers welcome the possibility to focus on the business / domain part. They welcome not having to discuss the "correct" patterns again and again.

To get the acceptance for this **it is crucial** to do a workshop together with the whole development team and agree on the patterns to use with everybody involved with the code. We moderate workshops like this regularly and always find a sensible set of patterns to agree on. Guided by our expertise the already existing patterns

[6] https://qafoo.com/blog/097_extended_definition_of_done.html

and the patterns the developers themself think are the right ones to use will be discovered and agreed on.

In such workshops we mutually agree on set of definitions and define them as a guide for code reviews which are then part of the "Definition Of Done" for the team. This also means that the team gets clear guidelines for Code Reviews and the reviewer knows what to look for. With several teams we even assist during Code Reviews for some time by reviewing the pull requests ourselves according to the guidelines we agreed on. This helps to get a deeper understanding of the structure and patterns and resolves remaining issues.

A common Definition Of Done we agree on could look like the following points. Remember that this might vary a lot depending on the domain, the team and the already existing patterns:

- Always exceptions for error handling
  - Never return `null` or `false` in case of an error
- Use data objects
  - **Never** use arrays as data structures
  - Data objects **must not** aggregate "active" dependencies (gateways, services)
  - Only logic modelling eternal truth
- Services
  - Max 4 dependencies, which are all injected using Constructor Injection
  - No dependencies on externals – each external class should be wrapped behind a facade
  - **Must be "fully" tested**
- Use Gateways / Repositories to load and save data
  - Return and / or receive data objects
  - Services depend on Gateways (interfaces)
- No logic in Controller, besides
  - Catch domain exceptions
  - Simple authorization ("is logged in")
  - Convert incoming data into object and outgoing data from object

There are usually more rules then this simple set which are then more specific to the given domain. But most of these rules are simple and fast to review (not simple

enough to write automatic checks, though) and following such a defined set of rules already greatly simplifies and unifies the code.

5.3.1 Conclusion

While working with many different teams we understood that common rules for code structure and patterns on top of what PSR-2 defines are helpful for developers and speed up the development. Strong rules simplify code reviews and strengthen the sense of Collective Code Ownership. We suggest to build up a rule set for common problems in your domain and use them in your daily work. Get the whole team together and agree on a rule set with everyone.

5.4 How to Refactor Without Breaking Things

Tobias Schlitt at 1. June, 2016[7]

Refactoring means to change the structure of your code without changing its behavior. It is an essential part of everyday programming and should become knee-jerk for your whole development team. Refactoring is very helpful to cleanup feature spikes, revise earlier decisions and keep a maintainable codebase in the long run. In a perfect project world - with extensive automated tests of various types - this is just a matter of getting used to. But there are only very few such projects. So getting into proper refactoring is much harder. This article will show you important tips to master this challenge with your team.

From our experience in various (legacy) projects successful refactoring depends on the following points:

1. Tests
2. Baby steps

5.4.1 Tests

Tests help you to ensure that the behavior of your application does not break while restructuring the code. But in many cases you will want to apply techniques of refactoring just to make your code more testable and to come to a stage where writing unit tests gets cheap. There the dog seems to chase its own tail.

We found out that high-level functional tests can deal as a good basis to get started with refactoring. Even very old legacy applications can usually be tested through the browser using Mink with PHPUnit (See: Using Mink in PHPUnit) (or Mink with Behat). Which of both solutions you choose depends on your project team: If you are familiar with PHPUnit and don't plan of involving non-technical people in testing later, PHPUnit + Mink is a solid choice for you.

Before you start writing tests you need to setup at least a rudimentary automation system that can reset your development installation (most likely the database). The goal must not be to get a fully fledged infrastructure automation (See: Why you need infrastructure and deployment automation) (which is of course still desirable) but to get a predictable, reproducible starting state for your tests. Maybe you just

[7] https://qafoo.com/blog/085_how_to_refactor_without_breaking.html

hack up a shell script or use PHPUnits `setupBeforeClass()` to apply a big bunch of SQL.

Then you start writing tests through the front-end for the parts of your code that you want to touch first, e.g. a really bad controller action (which code to refactor first will be part of another article).

Make sure to concentrate on the essentials: Keep the current behavior working. Don't care too much about good test code. You can throw these tests away after you finished your refactoring or just keep a few of them and clean them up later. As usual this is a matter of trade-off (See: Developers Life is a Trade-Off). What you want to achieve here is an automated version of the click-and-play tests you'd do manually in the browser to verify things still work.

Code coverage can be of good help here to see if you have already enough tests to be safe. We have a blog post on using code coverage with Behat (See: Code Coverage with Behat) for exactly this purpose. The same technique can be applied to running PHPUnit with Mink. But beware: the goal is not `$someHighPercent` code coverage! The goal is to give you a good feeling for working with the underlying code. Once you have reached that state, stop writing tests and focus on the actual refactoring again.

5.4.2 Baby Steps

When you start with restructuring your code, do yourself a favor and don't be too ambitious. The smaller your steps are, the easier it gets. Ideally, you will only apply a single refactoring step (e.g. *extract method* or even *rename variable*) at once, then run your tests and commit.

We know that this is hard to get through in the first place, especially when you did not do much refactoring before. But reminding yourself over and over again to go very small steps into a better direction is really helpful for multiple reasons:

1. It reduces the risk of breaking something. The human brain can only cope with a limited amount of complexity. The larger the change is, the more things you need to keep in mind. This raises the chance of messing things up and waist time.
2. If you messed up the current step (for example by changing behavior or realizing that your change did not lead to a good result) large changes make

it harder for you to just reset and restart. You will think about the time you already invested and will probably go on trying to fix the state. However, this typically makes it worse. Reset to HEAD and restart the refactoring step should be the way to go instead.

3. While you might have a big picture in mind where your refactoring should lead, this might not be the best goal. Maybe there are better solutions you did not think about in the first place. Doing baby steps will keep the door open for correcting your path at any time.
4. Chances that you will get through a large refactoring without being disturbed are low. There is always an emergency fix to be applied, a very important meeting to be joined, good coffee to be drunken or just Facebook that will require you to stop. Getting back into your working stack later will be hard and committing a non-working state should be a no-go. With baby steps, you can just cancel the current step or finish it within seconds and leave safely.

Long story short: Do yourself the favor and get used to baby steps. This will sometimes even result in more ugly intermediate steps. Get over it, things will eventually be better!

As a side note: People often ask us, if committing each and every baby step won't lead to polluting your version history. If you feel that way, rather go for squashing your commits later than doing larger steps with each commit.

What are your tips for successful refactoring? Leave us a comment!

The current issue of the German PHP Magazin[8] also has a slightly more extensive article on this topic by us.

[8] https://qa.fo/book-php-magazin-4-16-244272

5.5 Getting Rid of `static`

Kore Nordmann at 10. January, 2017[9]

When people start (unit-)testing their code one of the worst problems to tackle are static calls. How can we refactor static calls out of an existing application without breaking the code and while producing new features? How can we get rid of this big test impediment?

5.5.1 The Problem

Illustrating the problem with example code is only partially possible – the sheer amount of static calls found in real-world software is way too large. The problem is worse by at least a magnitude from everything you'll see in this blog post. But here goes an example anyways:

```
class UserService {
    /* ... */

    public static function getUser($userId) {
        $cacheKey = 'user-' . $userId;
        if (Cache::has($cacheKey)) {
            return Cache::get($cacheKey);
        }

        $userData = DB::query('SELECT * FROM user WHERE id = :id', ['id' => $userI
            d]);
        $user = new User($userData);

        Cache::set($cacheKey, $user);
        return $user;
    }

    /* ... */
}
```

Using the cache statically is only one common thing to do. Usually it is also the logger, the database and about any service. Probably it is even the `UserService` itself with calls like `UserService::getUser(42)` spread all over your code.

Today it is an established best-practice that we should test our code. People working with code like the one shown above know this and want to apply automated

[9] https://qafoo.com/blog/094_getting_rid_of_static.html

testing. It is also clear that the most desired testing method are unit tests. When writing new methods in the `UserService`: How can we test those? Or how can we test the existing methods in the `UserService`?

The core problem when testing the `UserService` is that we cannot mock the `Cache` instance for testing. The original `Cache` class will always be called, thus we will always test the `Cache` class together with the `UserService`. This is, by definition, not an Unit Test any more. Depending on the used cache implementation this might also require a far more complex setup to run the tests. If the cache implementation directly caches into Redis for example, you need a working Redis server just to run the `UserService` tests.

But getting away from static calls isn't a thing you should do in a single refactoring step. In fact, migration must be smooth and longer running to align with business perspectives. To achieve this, we show you multiple steps in such a migration.

5.5.2 Step 1: Replaceable Singletons

The workaround for the primary testing problem is obvious and employed by many developers – you can even find complete testing frameworks build on this approach, like the one from Laravel[10]: Make the implementation behind the static calls replaceable. Laravel calls this "Facade", while the Facade Pattern[11] actually is something different.

The idea is that the `Cache` class gets a setter for its actually used cache implementation, like:

```
class Cache {
    private static $implementation;

    public static function setImplementation(CacheImplementation $cache) {
        self::$implementation = $cache;
    }

    /* ... */
}
```

In your test case you can now use something like this:

[10] https://laravel.com/docs/5.3/mocking#mail-fakes
[11] https://en.wikipedia.org/wiki/Facade_pattern

```
class UserServiceTest extends \PHPUnit_Framework_TestCase {
    public function testLoadUser() {
        // Could happen in setUp()
        $originalCache = Cache::getImplementation();

        /* Set up mock */

        Cache::setImplementation($cacheMock);

        /* Actual test setup */
        /* Stimulus */
        /* Assertion */

        // Reset should happen in tearDown()
        Cache::setImplementation($originalCache);
    }
}
```

Why is this still problematic?

1. Global side effects
   The worst thing here is the global side effect of this change. Since `$implementation` has to be a `static` variable in the `Cache` class it is also changed for any other code, like future tests. To get atomicity of your tests you must also reset the cache implementation after the test again.
2. More complex test setup
   If you would use dependency injection the test code would only consist of the commented out code and would not require the setting and re-setting of the cache implementation. This might look trivial in this code but there are usually more classes used.
   Also tests should focus on readable code even more then any other code since they are often the starting point to understanding code for other developers. Anything messing unnecessarily with the test code can be considered bad.
3. API differences due to indirection
   The APIs of the `Cache` class and the `CacheImplementation` class might be different which means that you have to mock the internal usage inside of the `Cache` class and not the usage inside of the `UserService` class. You must at least ensure that those APIs are the same. This means that your brain must

keep more context which leaves less focus on the actual implementation and test.

This is a valid workaround introducing additional complexity because of global state and unnecessary indirection. The static calls allow you to skip dependency injection by introducing additional complexity while understanding the code and while testing the code. A trade-off I am not willing to accept as a migration step but not in the long run.

5.5.3 Step 2: Service Locator

Looking at the actual target of our refactorings, the `UserService` we desire looks like this:

```
class UserService {
    private $cache;
    private $database;

    public funtion __construct(CacheImplementation $cache, Database $database) {
        $this->cache = $cache;
        $this->database = $database;
    }

    /* ... */

    public function getUser($userId) {
        $cacheKey = 'user-' . $userId;
        if ($this->cache->has($cacheKey)) {
            return $this->cache->get($cacheKey);
        }

        $userData = $this->database->query('SELECT * FROM user WHERE id = :id', ['
            id' => $userId]);
        $user = new User($userData);

        $this->cache->set($cacheKey, $user);
        return $user;
    }

    /* ... */
}
```

What changed? We migrated all static dependencies to dependency injection via Constructor Injection and removed the `static` keyword for the `getUser()` method (and all others).

This code is testable. We can inject mocks for the `CacheImplementation` and `Database` directly and will not have any global side effects affecting out test atomicity. There is no environment related test setup required any more. The method itself could be cleaner and easier to test, but this is not the point right now. One important remark: **You can also call every static method dynamically.** Thus we can just pass an instance of `Cache` or `CacheImplementation` and `Database` and there should not be any problems.

The problem is that the API of our `UserService` changed. If we had code earlier calling `UserService::getUser(42)` it will not work any more. A simple step to get this code working is introducing a static Service Locator as a workaround during refactoring:

```
class ServiceLocator {
    private static $userService = null;

    /* Same for cache(), database(), ... */

    public static function userService() {
        if (!self::$userService) {
            self::$userService = new UserService(self::cache(), self::database());
        }

        return self::$userService;
    }
}
```

The we need to migrate all calling code to call `ServiceLocator::userService()->getUser(42)` instead of `UserService::getUser(42)`. This refactoring is easy enough and can usually be done by Search & Replace: `s/UserService::/ServiceLocator::userService()->/g`

The code using the `UserService` is still not clean, but we cleaned up the `UserService` itself to use dependency injection and be testable. We are not done yet, but this already is a big step forward.

Until now all steps are quick to accomplish, but the next one will take time.

5.5.4 Step 3: Dependency Injection

Now we have the `UserService` in the desired state. We already changed the code using it to indirect static calls through the `ServiceLocator`. But we should eventually get rid of all the static access, including the static access to the `ServiceLocator` itself.

Problems with using a Service Locator are:

1. Still global side effects (when used statically)
   If you test a class which itself accesses the Service Locator you have to replace all requested services with mock objects in the Service Locator. Since the Service Locator implementation uses static state it will still affect all future tests (without proper resetting logic). This breaks test atomicity again.
2. ..., thus also: Complex test setup
3. Hidden dependencies
   If a class has access to the Service Locator you can only understand its dependencies from reading its entire code. In a class using Constructor Injection we know all its (required) dependencies by just looking at the constructor signature.
   A class which has access to the Service Locator can request any class anywhere in its code from it. How do we know what to mock in a test case or what setup to create if we want to use the class somewhere else? You'll have to read the entire code.
   This is also true for Service Locators with dynamic access which are passed around. They can be another intermediate refactoring step but usually provides nothing of additional value.

Consequence: **Pass the `UserService` instance to every class which needs access to it.** Or phrased differently: Use Dependency Injection for all your code – or at least all code which is supposed to be tested or re-used at some point.

This means that, in production, you'll have just one instance of the `UserService` which is passed to any class which needs to access the users. This seems like a lot of work and it really is. But most of the work can actually be taken over by even the simplest Dependency Injection Container:

```
class DependencyInjectionContainer {
    private $userService = null;
```

```
    /* Same for cache(), database(), ... */

    public function userService($dic) {
        if (!$this->userService) {
            $this->userService = new UserService($this->cache(), $this->database()
                );
        }

        return $this->userService;
    }

    public function userController($dic) {
        return new UserController($this->userService());
    }
}
```

OK, this looks really similar to the `ServiceLocator` class shown before – only dropping all `static`. This is right. The difference between a Service Locator and a Dependency Injection Container is not the implementation but the usage:

The `DependencyInjectionContainer` is **only** used inside your `index.php` and not passed to any class. As an additional migration step you *may* use it inside your Service Locator until we migrated away from it entirely.

If you have a simple routing definition like with Silex[12] you should use the Dependency Injection Container right there – and nowhere else:

```
$app = new Silex\Application();
$dic = new DependencyInjectionContainer();

$app->get('/user/{id}', function ($id) use ($app, $dic) {
    return $dic->userController()->getAction($id);
});

$app->run();
```

The Dependency Injection Container will now resolve all the required dependencies **only when the route is called**. You can even replace the code above by using simple Dependency Injection Containers like Pimple[13].

This is the last step to have only testable classes all using Dependency Injection. The last step is a lot of work because many classes must be adapted. But you achieved actually two goals by doing this:

[12] http://silex.sensiolabs.org/

[13] http://pimple.sensiolabs.org/

1. Make everything testable
2. Extract application configuration
   Which cache is used by the user service should not be the concern of the user service itself but is nothing but application configuration. Now the `DependencyInjectionContainer` contains all your application configuration (it may access parameter files or similar) and you configure the used cache implementation there – and it will be "magically" used everywhere.

5.5.5 Conclusion

Migrating away from static calls can be quite some work. This blog post showed you a migration strategy with functional software in every step. Consider `static` a code smell because of all the reasons mentioned in this post and migrate away from it eventually. Take your time.

5.6 Refactoring Should not Only be a Ticket

Tobias Schlitt at 24. January, 2017[14]

A while ago I tweeted

> #Refactoring should never only be a dedicated task on your board. It should be an essential part of every other task you work on.
>
> -- https://twitter.com/tobySen/status/783610875047505920

In this blog post I would like to elaborate a bit further on what I mean and why I think this is important.

When we do quality workshops and trainings on-site at our customers we see various approaches to refactoring which typically fail, for example:

1. A general ticket "Refactoring" is added to every sprint
2. Dedicated refactoring sprints are requested

The problem here is that refactoring is not seen as an essential part of the daily work, but instead as a dedicated task that requires additional time on top of daily work.

Compare your work as a programmer to the job of any type of craftsman: does that craftsman charge additional time for cleaning up the construction site? Of course not. Either you clean up your working place after finishing a task or you need to do it before starting the next. Both ways are possible, but just skipping to clean your workplace until you get dedicated time is not an option.

This is exactly the way how you should approach refactoring: When starting a new task you need to analyze the existing code anyway. If you stumble over some dirt, clean it up as you go. When you finished your task reflect what you just did. Maybe a method grew too large? Maybe you could avoid duplication? Maybe you chose a bad name? Fix it – now!

If your team accepts refactoring as an essential part of every work they perform, you will experience how fast your code base will improve at exactly the places you work on a lot.

Of course you will still discover bigger challenges while trying to clean up the construction site. There will be steps which turn out to be too large to be done on

[14] https://qafoo.com/blog/095_refactoring_should_not_be_a_ticket.html

the go. These are exactly the parts which should be made dedicated tickets. But beware to just name a ticket "Refactoring". Be specific instead and explain exactly what needs to be achieved to put the team in a better position to clean up on the go.

5.7 Extracting Data Objects

Tobias Schlitt at 7. February, 2017[15]

Extracting data objects from your code will make it easier to read and write, easier to test and more forward compatible. This post shows you the two most common cases where introducing a data object makes sense and how to do it.

5.7.1 Too Many Parameters

Every project has them, the method signatures where you just add another parameter. Query methods are a very typical example:

```
public function findProducts($phrase, $categories = array(), $minPrice = 0,
    $maxPrice = null, $productTypeFilters = array(), $limit = 10, $offset = 0)
{
    // ...
}
```

There are several issues with such method signatures: It is really hard to remember which parameter is at which position, additional information will require you to add even more parameters and introducing more mandatory data will even force you to change the parameter order which will most probably be a large amount of work.

Inspecting the parameters closely you can find a common pattern for most of them: 5 of 7 parameters are criteria for product search. This already reveals the name for the data object to choose:

```
class ProductCriteria
{
    public $phrase;

    public $categories = array();

    public $minPrice = 0;

    public $maxPrice;

    public $productTypeFilters = array();

    public function __construct($phrase)
    {
        $this->phrase = $phrase;
```

[15] https://qafoo.com/blog/096_refactoring_extract_data_objects.html

```
    }
}
```

Using this data object strips down the method signature to three parameters:

```
public function findProducts(ProductCriteria $criteria, $limit = 10, $offset = 0)
{
    // ...
}
```

It is much more readable now and it is much easier to introduce additional criteria. You can even change the structure used inside the criteria fields with some effort and without affecting the using code pieces.

It might make sense to use a base class for your data objects, as described in an earlier post (See: Struct classes in PHP), since PHP does not have native support for data objects and it can provide you with additional convenience.

5.7.2 Associative Arrays

Arrays in PHP are a powerful data type. Whenever there is data to be structured it is easy to just create a (potentially deeply nested) mixture of struct and list out of thin air. That makes them a really good tool for prototyping, for example:

```
public function getDiscounts(array $checkout)
{
    // ....
}
```

But once the prototyping phase is over they will soon become a real pain: There is no defined way to document array structures so the IDE will not be able to tell you which fields exist, what their purpose is and what type the fields expect. The only way to know is reading the code that creates and the code that uses the array structure. Due to the lack of auto-completion on field names there is a high risk for typos. And because it is so easy to add new fields people will eventually add whatever they need at a single place making your array more and more god like.

It is therefore a good idea to replace any associative array structure with a data object once the structure has stabilized a bit. For example:

```
class Checkout
{
    /**
```

```
     * @var CheckoutItem[]
     */
    public $items;

    /**
     * @var Address[]
     */
    public $shippingAddress;

    // ...
}
```

With this approach you actually solidify the structure you prototyped as an array and create sensible documentation and auto-completion support for it. In addition you raise the barrier for adding arbitrary new fields by adding one more thinking step.

5.7.3 Smooth Migration

In most cases a migration towards using a data object cannot be accomplished within some minutes. This only works if the method for which you are attempting to change the signature is used infrequently. If that is the case: lucky you, go ahead and perform the changes. Otherwise you should perform a smooth migration over time. You can most probably apply the following steps mechanically.

Create a New Method

Because you cannot simple change the original method signature you need a new method right beside the original one. For example:

```
/**
 * @deprecated Use calculateDiscounts() instead!
 */
public function getDiscounts(array $checkout)
{
    // ....
}

public function calculateDiscounts(Checkout $checkout)
{
    // ....
}
```

The `@deprecated` annotation added to the original method is quite handy, because IDEs can display warnings to developers still using the old method.

Of course, having these two methods lurking around right beside each other is not nice. But remember that this is only a temporary state until you finished the refactoring entirely.

Dispatch Old Method To New

After adding the new method you probably have code duplication. To remove that, remove the body of the old method and call the new one instead, migrating the incoming array to the new data object:

```
/**
 * @deprecated Use calculateDiscounts() instead!
 */
public function getDiscounts(array $checkout)
{
    return $this->calculateDiscounts(Checkout::fromLegacyArray($checkout));
}

public function calculateDiscounts(Checkout $checkout)
{
    // ....
}
```

To have the conversion from the original array to the new object in a single place I added a factory method (one of the few cases where `static` is OK) to the `Checkout` class.

Change Use Case

Now it's time to change the use-case you are working on - the place which motivated you to actually start the refactoring:

```
// ... calling code ...
$discounts = $whereverTheMethodIs->calculateDiscounts(
    Checkout::fromLegacyArray($checkoutArray)
);
```

Congrats, you finished the first step into eliminating the deprecated method from your project. :)

Iterate

You should make it a rule in your project to perform this refactoring step whenever you encounter a use of the old method. After some weeks, search your code for

the (hopefully few) remaining method calls and change them. Once you reached that state you can safely remove the deprecated method.

While you are a big step further now the end is still not reached. Look through your code and find all usages of the `Checkout::fromLegacyArray()` methods. These are the places where the original array structure is still used. You can now start replacing these cases in a similar way as explained here.

5.8 Basic Refactoring Techniques: Extract Method

Benjamin Eberlei at 7. March, 2017[16]

Refactoring is the process of restructuring code without changing its behaviour and the technique "Extract Method" is one of the most important building blocks of refactoring.

With extract method you move a fragment of code from an existing method into a new method with a name that explains what it is doing. Therefore this technique can be used to reduce complexity and improve readability of code.

In this post I want to explain the mechanics of extract method using an example so that you have a checklist of steps when performing this refactoring. Extract method is a technique that you can use even without tests, because the potential risks of breaking are manageable when you follow the steps.

I have performed these steps countless times myself and the more often you perform them the less likely will you break the code.

Knowing all the manual steps that are necessary for extract method is a great benefit even if you are using PHPStorm's powerful automated Extract Method functionality in the end. Just understanding each step helps you selecting the best code blocks for refactoring, something that PHPStorm cannot do for you.

The example is a method from a controller that directly uses a library called Solarium to access a Solr database, including some relatively complex low level filtering code:

```
public function searchAction(Request $request)
{
    if ($request->has('query') || $request->has('type')) {
        $solarium = new \Solarium_Client();
        $select = $solarium->createSelect();

        if ($request->has('type')) { // filter by type
            $filterQueryTerm = sprintf(
                'type:%s',
                $select->getHelper()->escapeTerm($request->get('type'))
            );
            $filterQuery = $select->createFilterQuery('type')->setQuery($filterQ
                ueryTerm);
            $select->addFilterQuery($filterQuery);
```

[16] https://qafoo.com/blog/098_extract_method.html

```
        }

        // more filtering logic here

        $result = $solarium->select($query);

        return ['result' => $result];
    }

    return [];
}
```

As a rule of thumb, code in a method should work on the same level of abstraction (high- vs low-level code) to hide unnecessary details from the programmer when reading code. Mixing high level controller with low level data access does not hold up to that rule.

5.8.1 Step 1: Identify code fragment to extract

We want to extract all the Solarium related code into a new method on the controller to hide the details of how searching with Solarium works on the low level.

The primary goal is find all consecutive lines that belong together semantically. Which lines should be part of the new method and which should stay? This first step is not always easy, practice is everything.

Everything from line 4 (instantiating Solarium) to line 15 (calling select) belongs to this concern. We start using the solarium object and its helpers in line 4 and never use them anymore after line 15.

Don't think about this too long though, keep in mind that refactorings can be easily reverted and redone.

5.8.2 Step 2: Create empty method and copy code

If we have a candidate block of code to extract, we create a new empty method without arguments and give it a name that describes what the block is doing:

```
private function search()
{
}
```

The next step is to copy over lines 4-15 into the new method:

```
private function search()
{
    $solarium = new \Solarium_Client();
    $select = $solarium->createSelect();

    if ($request->has('type')) { // filter by type
        $filterQueryTerm = sprintf(
            'type:%s',
            $select->getHelper()->escapeTerm($request->get('type'))
        );
        $filterQuery = $select->createFilterQuery('type')->setQuery($filterQueryT
            erm);
        $select->addFilterQuery($filterQuery);
    }

    // more filtering logic here

    $result = $solarium->select($query);
}
```

This method will not work yet, but little steps are the key to avoid breaking the code. The next steps in the refactoring will make this new method usable.

5.8.3 Step 3: Identify undeclared variables that must be arguments

All variables that have been declared above line 4 in our original method are missing from the new method now and the solution is to pass them as arguments.

How to find all these variables? If you are using an IDE the previous code block should now be littered with references to using undeclared variables. If you are using Vim or another editor you must find these occurrences yourself.

In our example code, the only variable that is used inside the new method and was declared before line 4 is `$request`, so we pass it as argument:

```
private function search(Request $request)
{
    // ...
}
```

5.8.4 Step 4: Identify variables that are still used in old method

The next step is to check which variables declared inside our new method `search` are still used after the last extracted line 15.

Your IDE can help you with this. Simply comment out the lines you extracted then it will warn you about using undeclared variables used after the extracted lines. If you use an editor you must again find this out yourself by studying the code.

In our example this applies to `$result` which is again used in line 17. All variables of this kind must be returned from the new method and assigned to a variable with the same name to require as little changes as possible:

```
private function search(Request $request)
{
    // ...

    return $result;
}
```

What if there are more then one variable being declared inside and used outside the method? There are several solutions that each has their own set of downsides:

1. Return an array of the variables (emulation of multiple return values). You can use `list()` to assign them to non-array variables in the old method.
2. Only return scalar values and pass objects as arguments and modify them
3. Pass scalar variable into new method by reference and modify it

The first method is the mechanically simplest and should be preferred, because there is less risk of breakage with this approach. Ignore the nagging desire to introduce an object or a complex array to make this code less ugly. You can do that if you want after the refactoring is done and the code works.

5.8.5 Step 5: Call new method from original method

We have commented out the original code in the previous step to find return values, so we must now call the new method instead.

Pass all the arguments you identified in step 4 and 5 and declare all return values with variables with the same they will be used with later:

```
public function searchAction(Request $request)
{
    if ($request->has('query') || $request->has('type')) {
        $result = $this->search($request);
        // commented out original code here

        return ['result' => $result];
    }
```

```
    // ...
}
```

Now I can execute this code again (either manually or with existing integration tests). The original code is just commented out so that when problems occur I can read it next to the new code and easily compare for mistakes. Delete this code if you are sure the extract method has worked.

Congratulations, you have applied the heuristics to perform extract method as safely as possible even if you don't have tests. Still there are some risks with every code block you extract that you should know to check for.

5.8.6 Risky Extract Method Checklist

There is some risk with extract method, even if you performed the mechanics perfectly it can still alter the behaviour of your original code. You should think about the side effects of your new method before executing it the first time. With experience you learn to spot potential problems before even selecting a code fragment to extract.

- Arrays are not passed by reference, but many methods subtly change them in a way that has an effect on the parent method. Example `next()` or `sort()`.
- Side effects to instance variables or in the global state can sometimes have different outcomes when extracted into a method. Make sure to check this more carefully when your extracted method is called in a loop.
- Variables that are declared before and used after the extracted method require special care as you must pass them as argument (step4) and returning them (step5) and are sometimes better passed by reference instead.

5.8.7 Fin

Extract method is especially powerful and reduces the complexity if the new method contains one or many variables that are declared inside the new method and are not returned, because they are not needed afterwards. As a programmer this reduces the mental capacity needed for understanding the original method massively.

From my experience it takes a lot of training to select the right lines to extract and extract method is a technique I still practice actively and improve on.

Extract Method is a fundamental building block for more advanced refactorings such as Extract Service and refactoring towards different design patterns. These are topics we will cover in future blog posts about refactoring.

5.9 How to Perform Extract Service Refactoring When You Don't Have Tests

Benjamin Eberlei at 21. March, 2017[17]

When you are refactoring in a legacy codebase, the goal is often to reduce complexity or separate concerns from classes, methods and functions that do too much work themselves. Primary candidates for refactoring are often controller classes or use-case oriented service classes (such as a `UserService`).

Extracting new service classes is one popular refactoring to separate concerns, but without tests it is dangerous because there are many ways to break your original code.

This post presents a list of steps and checklists to perform extract service when you don't have tests or only minimal test coverage. It is not 100% safe but it provides small baby-steps that can be applied and immediately verified.

The primary risk of failure is the temptation to do too many steps at the same time, delaying the re-execution and verification that the code still works for many minutes or even hours. Read more about this in How to Refactor Without Breaking Things (See: How to Refactor Without Breaking Things).

This post builds upon the previous post on Extract Method (See: Basic Refactoring Techniques: Extract Method), where we already moved a dedicated concern with low-level code to query a Solr database into a new method.

```
public function searchAction(Request $request)
{
    if ($request->has('query') || $request->has('type')) {
        $result = $this->search($request);

        return ['result' => $result];
    }

    return [];
}

private function search(Request $request)
{
    $select = $this->solarium->createSelect();

    if ($request->has('type')) { // filter by type
        $filterQueryTerm = sprintf(
```

[17] https://qafoo.com/blog/099_extract_service_class.html

```
            'type:%s',
            $select->getHelper()->escapeTerm($request->get('type'))
        );
        $filterQuery = $select->createFilterQuery('type')->setQuery($filterQueryT
            erm);
        $select->addFilterQuery($filterQuery);
    }

    // more filtering logic here

    $result = $solarium->select($query);

    return $reesult;
}
```

Our goal is to extract all the Solr code into a new `SolrSearchService` class. Before you start you should have already used Extract Method to create one or several methods that you want to move to a new class.

5.9.1 Step 1: Create Class and Copy Method

Similar to the extract method refactoring we start by copying code 1:1 without changing it for now.

```
class SolrSearchService
{
    private function search(Request $request)
    {
        $select = $this->solarium->createSelect();

        if ($request->has('type')) { // filter by type
            $filterQueryTerm = sprintf(
                'type:%s',
                $select->getHelper()->escapeTerm($request->get('type'))
            );
            $filterQuery = $select->createFilterQuery('type')->setQuery($filterQ
                ueryTerm);
            $select->addFilterQuery($filterQuery);
        }

        // more filtering logic here

        $result = $solarium->select($query);

        return $reesult;
    }
```

```
}
```

5.9.2 Step 2: Fix Visibility, Namespace, Use and Autoloading

You can modify the `private` visibility to `public` as the first step, because we need to call this method from the original class it must be public.

You must also copy over all the `use` statements that are in the original class, because otherwise the new class could import a class from the wrong location.

```
use Symfony\Component\HttpFoundation\Request;

class SolrSearchService
{
    public function search(Request $request)
    {
        // ...
    }
}
```

I usually copy *all* use statements if I extract a larger block of code, because then I don't miss a reference in the moved codeblock. If the block is smaller I only copy the ones that are actually used in the block. An IDE helps here because it can highlight the superfluous use statements and recommend missing ones.

As a last point in this section, verify that this new class is autoloadable, so that you can easily use it from tests and the original class.

5.9.3 Step 3: Check for Instance Variable Usage

As a next step you must look at the extracted code and find all references to instance variables, because our new class doesn't have those instance variables itself, they are still on the original class.

There are several cases of instance variable uses that need different handling:

1. You find an instance variable that is **only** used in the extracted method itself. Then you can copy (don't delete it just yet) the variable to the new service. You could perform a refactoring to convert the instance variable to a local variable if the state is not needed across multiple calls to the method.

2. The instance variable is another object that is injected into the originals class constructor or with setter injection. Copy the instance variable over to the new object and introduce a constructor that sets this dependency.
3. The instance variable is used for state that is only read in the new class. Convert the instance variable into a local variable on the new class and pass it as a new argument to the extracted method.
4. The instance variable is used for state that is read, changed or both in the new class. Convert the instance variable into a local variable on the new class and pass it to a setter on the new service before you call your news method, and retrieve it back with a getter after you called the service.

Method 4 works for all use cases, always use it when you are unsure, even if it requires the most code and does not look very clean.

It requires a bit of experience to quickly categorize instance variables into these groups and what the best course of action is. Don't be discouraged if the first attempts lead to nothing or weird APIs, remember that refactoring is a constant process and intermediate steps may actually make the code worse.

In our example we have one instance variable `$this->solarium` which is a service used in the parent class as well, point 2 in our list. We add the following code to `SolrSearchService`:

```
class SolrSearchService
{
    private $solarium;

    public function __construct(\Solarium_Client $solarium)
    {
        $this->solarium = $solarium;
    }
}
```

This class should now be usable independently of the original class. You should now write some tests for it, in this case integration tests, because we rely on third party APIs and a database.

Make sure to use dependency injection for the new class even if your dependencies are available as singletons or from a static registry. This way you at least this class is testable and you can make use of mock objects to write those tests.

5.9.4 Step 4: Use New Class Inline

The original class still uses the old extracted method. Comment the body of the method out and instantiate the new class inline, pass all the dependencies into the constructor or as arguments to the new method:

```
private function search(Request $request)
{
    // commented out old code
    $solrSearchService = new SolrSearchService($this->solarium);
    $result = $solrSearchService->search($request);

    return $result;
}
```

This method looks more complex if you have instance variables mentioned as item 3, 4 or 5 in the previous section.

The original code is now runnable. It should work out of the box if you have followed all the steps correctly. Remove the commented out old code and remove instance variables that were entirely moved to the new class (type 1).

You should test the code manually (or with functional tests (See: Using Mink in PHPUnit)) and commit your current changes now.

5.9.5 Step 5: Inline Method

The `search` method on the original class is not needed anymore, you can inline the method into the location where its called from:

```
public function searchAction(Request $request)
{
    if ($request->has('query') || $request->has('type')) {
        $solrSearchService = new SolrSearchService($this->solarium);
        $result = $solrSearchService->search($request);

        return ['result' => $result];
    }

    return [];
}
```

5.9.6 Step 6: Move Instantiation into Constructor or Setter

This step is optional, but instantiating the `SolrSearchService` in your runtime code is not a good practice. You can create a new instance variable `solrSearchService` and move the new to where the `solarium` instance variable is already assigned:

```
private $solarium;
private $solrSearchService;

public function __construct(\Solarium_Client $solarium)
{
    $this->solarium = $solarium;
    $this->solrSearchService = new SolrSearchService($solarium);
}

public function searchAction(Request $request)
{
    if ($request->has('query') || $request->has('type')) {
        $result = $this->solrSearchService->search($request);

        return ['result' => $result];
    }

    return [];
}
```

This step makes sense if you continue extracting additional methods from the original class to the new `SolrSearchService`, because they can now reuse the same instance.

5.9.7 Step 7: Cleanup Dependency Injection

This step is optional and is only possible if you are already injecting dependencies using a factory or containers. After extracting **all** methods that work on the `solarium` instance variable to the new service we don't need the instance variable on the original class anymore.

At this point we can switch the injected dependency to be the SolrSearchService directly:

```
private $solrSearchService;

public function __construct(SolrSearchService $solrSearchService)
{
    $this->solrSearchService = $solrSearchService;
```

```
}
```

5.9.8 Fin

Compared to the extract method refactoring, extracting a service requires more steps and each of them is more risky. On top of that IDEs usually don't provide this refactoring as an automatic procedure, so you have to do it manually. But even though the refactoring is risky, you should learn and master it, because it is very effective at splitting up code that started out simple and got more complex over time.

5.10 How You Can Successfully Ship New Code in a Legacy Codebase

Benjamin Eberlei at 19. April, 2017[18]

The greek philosopher Heraclitus already knew that "change is the only constant" and as software developers we know this to be true for much of software development and business requirements.

Usually the problems software needs to solve get more complex over time. As the software itself needs to model this increased complexity it is often necessary to replace entire subsystems with more efficient or flexible solutions. Instead of starting from scratch whenever this happens (often!), a better solution is to refactor the existing code and therefore reducing the risk of losing existing business rules and knowledge.

A good strategy for this kind of change is called "Branch By Abstraction". While the term is certainly clumsy and overloaded, the idea itself is genious.

Instead of introducing a long running branch in your version control system (VCS) where you spend days and months of refactoring, you instead introduce an abstraction in your code-base and implement the branching part by selecting different implementations of this abstraction at runtime.

Explaining Branch By Abstraction without an example is not a good idea, so lets take a look at three different examples to get an understanding how to make use of branch by abstraction:

5.10.1 Example 1: Replacing the Backend in a CMS

Up to eZPublish version 4, the popular CMS had a very relational-database centric API and model where all the abstractions made it quite clear that the underlying database is relational. In addition the content repository was leaky about another relational abstraction, the fact that one attribute is stored in one row because of the use of a highly sophisticated Entity-Attribute-Value model.

Those assumptions in the API and the inability to implement different data models made it very hard to scale eZPublish 4 beyond a certain number of content objects. With the rise of Solr and ElasticSearch customers wanted to use the powerful search engines, but the API limitations made this nearly impossible.

[18] https://qafoo.com/blog/101_branch_by_abstraction.html

With eZPublish 5 a new API was introduced with a fully object-oriented interface and a new content model abstraction.

The idea was to allow developers to work towards switching the storage model step by step. You could replace usages of the old API with the new API which would already allow to benefit of NoSQL search based implementations, with a fallthrough to the legacy database schema. After a full switch to the new API it would also be possible to replace the legacy database schema entirely.

Critical in using branch by abstraction in such a scenario is finding the right API that is not a leaky abstraction for the abstracted storage engine anymore.

```
<?php

$query = new Query([
    'filter' => new Criterion\LogicalAnd([
        new Criterion\ContentTypeIdentifier("article");
        new Criterion\Field("status", Operator::EQ, "xyz"),
        new Criterion\FullText("Hello World");
        new Criterion\Visibility(Criterion\Visibility::VISIBLE),
    ])
]);

$searchResult = $searchService->findContent($query);
```

You can see how this API exposes query functionality generically, but makes sure to add semantic meaning such as the FullText criterion. This allowed eZ Systems to implement the API both with the old legacy database schema as a first implementation, but also experiment with Solr and other search technologies to achieve much better performance and scalability.

5.10.2 Example 2: Rewriting a submodule without changing public API

Very early in my career I was responsible for maintaining a fairly large newsletter system that had pretty bad APIs and was a large mess of spaghetti code.

One center piece of this system was the parser that took free form HTML, some configuration flags and text input and generated an HTML email that included tracking data and links, inline CSS styles, images uploaded to a CDN and mangled the HTML such that Outlook and the likes rendered it correctly.

The code for this large system was hidden in a single god-function inside a class:

```
class Mailing
{
    public function generateMail()
    {
        // 2000-3000 lines of PHP parsing goodness!

        $this->save(array('generated_mail' => $parsedHtmlBody));
    }
}
```

The code was hard to test and was a frequent source of small and nasty bugs. When we onboarded a second customer onto this system and he needed different configuration inputs everything fell apart.

We built a new mail parser from scratch with a nice new API. It was entirely stateless and accepted a large object of inputs and produced a large object of outputs that could then be put into the existing database as required. The code was feature flagged to our new customer first:

```
class Mailing
{
    public function generateMail()
    {
        if (Config::getTenant() === 'new_customer') {
            $parser = new MailParser(/* some dependencies */);
            $parsedMail = $parser->parse(
                new MailDraft(array(/* tons of inputs */))
            );

            $this->save(array('generated_mail' => $parsedMail->body));

            return;
        }

        // old 2000-3000 lines of PHP goodness!
    }
}
```

The combination of highly unit tested code and some weeks of production experience with a subset of customers finally gave us confidence to get rid of the old code entirely and we ended up with a shiny nice API within this legacy code base, but entirely decoupled from it.

5.10.3 Example 3: Github reimplements Merge button

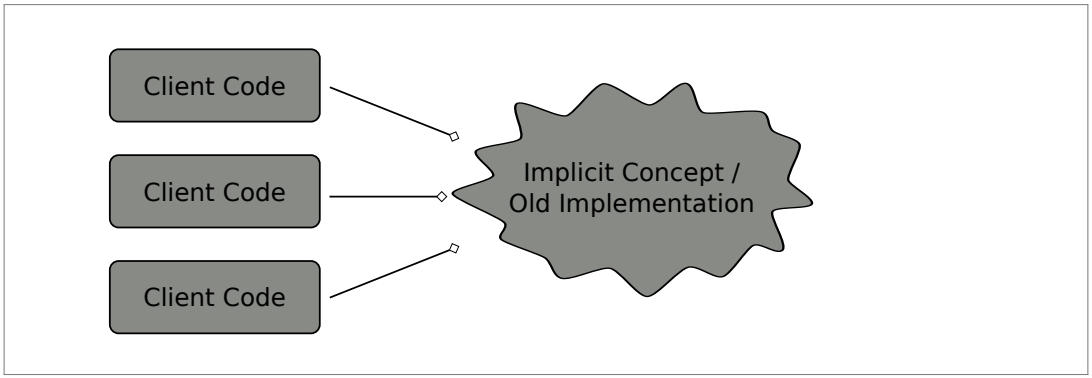
In December 2015, Github blogged[19] about their use of branch by abstraction to replace how the merge button works across the whole site. This critical feature of Github needed extensive testing so they used a library called Scientist to devise an experiment.

They refactored the old code into a dedicated method and then wrote new code with the same method signature. The scientist library then allows them to run both new and old code after each other. The library checks if the response of both methods is the same and logs an error if its not the case.

This strategy is a very powerful use-case of branch by abstraction.

5.10.4 The Process

The process of implementing Branch By Abstraction usually follows a certain set of steps while each step already provides you with a sensible outcome. The basic we have to work with looks something like this:



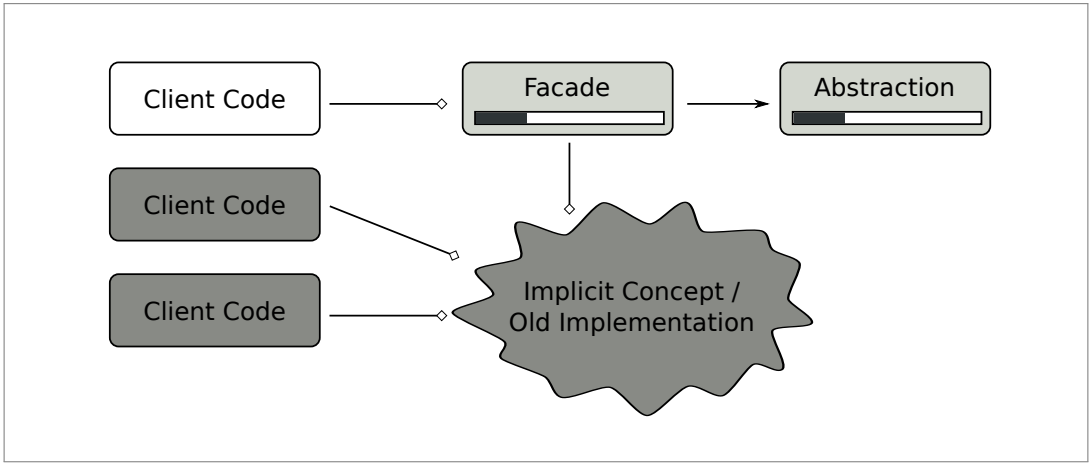
We have several "Clients" (classes, functions) which use and probably implement an implicitly defined concept like in one of the examples above. This might be inline code (Example 2) or already code in other classes which is called using APIs which do not describe the actual domain well (Example 1). Usually it is a combination of both.

[19] https://githubengineering.com/move-fast/

Step 1: Refactoring The First Client

We start by picking one of those "Clients" (not the most complex one) and see how this Client uses the implicitly defined concept. We create a new API for this. This is very similar to Extract Methods (See: Basic Refactoring Techniques: Extract Method) or Extract Classes (See: How to Perform Extract Service Refactoring When You Don't Have Tests) refactorings. While moving the old code we also create an abstraction (interface, set of interfaces).

Do not over-analyze in this step. Just extract the Facade which is required by this one Client and do not try to already cover all other Clients concepts. We will get there. It is best to just *move* the old code behind a Facade (implementing the abstraction) to make sure the system behaves like before and we do not break something. Functional Tests (See: Using Mink in PHPUnit) are really useful when doing such refactorings (See: How to Refactor Without Breaking Things).

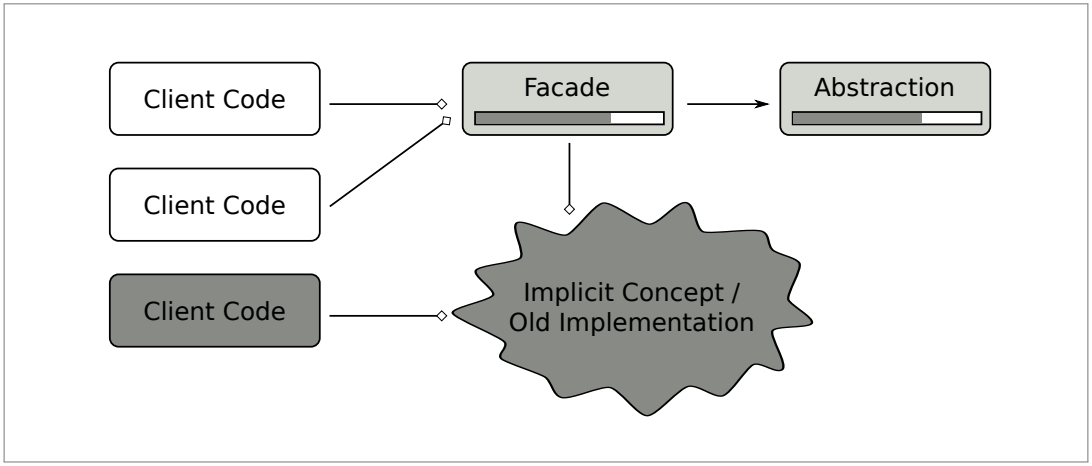


Already after this step the code in the first Client will be much more readable. You already did something beneficial to the software.

Step 2: Iterate Across Clients

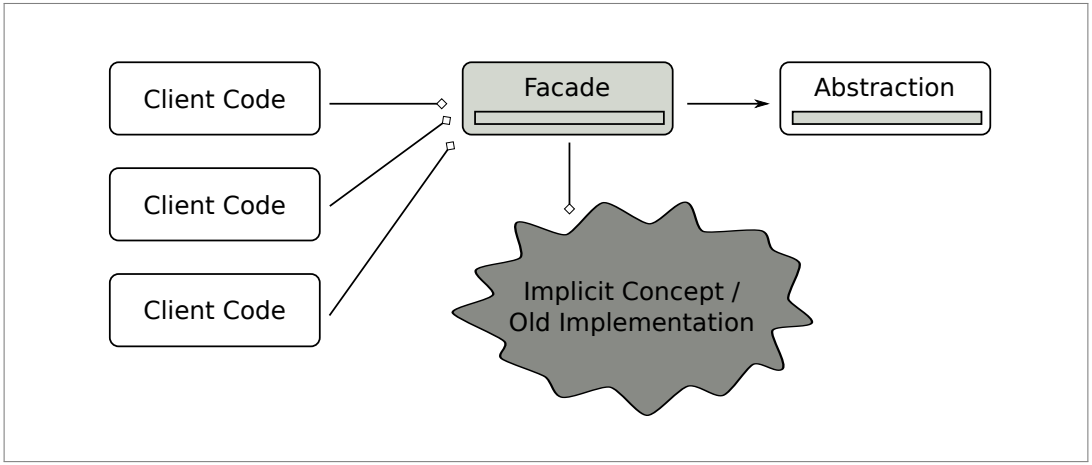
We continue to do this with every client. You will have to adapt the Facade and the abstraction during this process. There will always be edge cases in in some clients

you will not have thought of initially. But this is also the beauty behind this approach. We slowly discover and understand the implicit concept / our domain.



Step 3: Finish The Clients

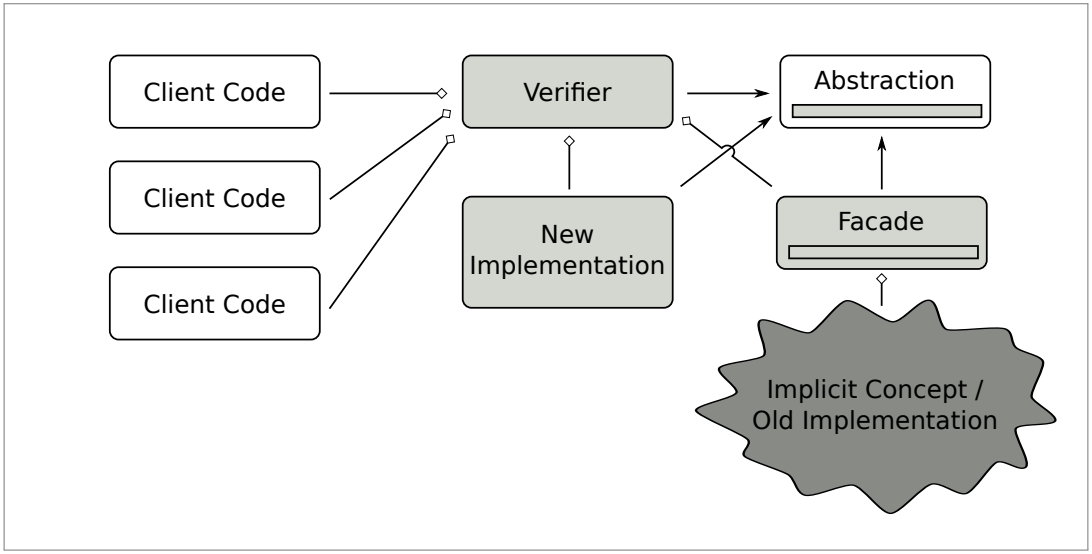
We do this until all clients are refactored. The clients are done now and so is our abstraction. We now understood the concept and have modelled it with a new set of interfaces. This is great!



But there still is this Facade which contains all the old ugly code. If the code is working well enough and we do not have to change it often it is valid to just keep it there and ignore it. It will not be maintainable, but if there are no requirements to change, adapt or extend it – then this is fine. Feel free to just stop here.

Step 4: Starting a new Implementation

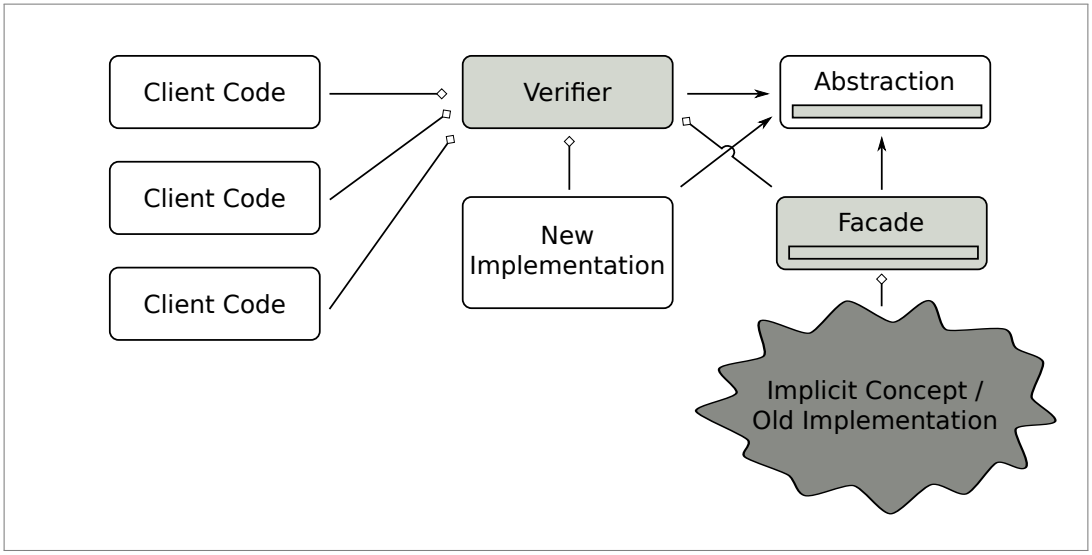
If we decided that we need a new implementation because the old one is just unmaintainable we can start with this now. We have an abstraction which we "just" need to implement. Please do this test-driven. With tests we can be pretty sure that the new implementation will succeed and fulfil all the requirements of our abstraction. Since the Facade, the Verifier and the new implementation all fulfill the same interface we should be able to replace the implementation used by the Clients using our Dependency Injection Container or with some Factory.



But we only really know that the new implementation is a success if we are testing it in production. And with the given abstraction we can implement a *Verifier* which ensures exactly this.

The Verifier knows the legacy Facade and the new code – it also implements the abstraction. The Verifier now always calls the Facade wrapping the old code

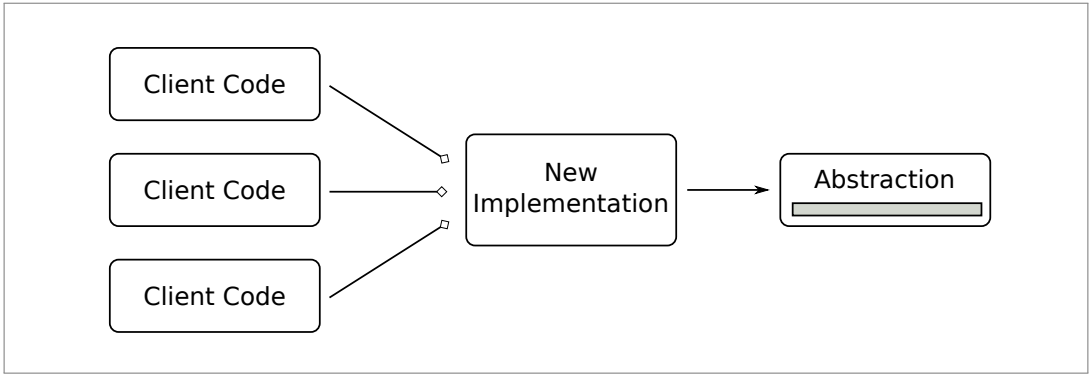
and the new implementation and compares their output. It can optionally even compare additional metrics like speed and memory usage. By using this in production for some time and logging all differences we can be pretty sure that the new implementation will be ready for production.



In this step you might also find bugs in the old code, just like Github did in the example linked before.

Step 5: Delete the Old Code

Now comes the most beautiful step: Deleting the old code.



Once we are sure everything works there is no need to keep the Verifier nor the Facade wrapping the old code. You can throw everything away and now only maintain and adapt the new clean code.

5.10.5 Conclusion

Branch by abstraction is not an easy skill to master and the implementation is always extremely specific to the use-case. But the benefits of avoiding long-running branches and complex merges, being able to test old vs new code and gradually rolling out the new functionality to only a subset of users are so large, that you should definately learn about it. To be honest, for me it is one reason why refactoring in a large legacy code can be so much fun!

5.11 Extracting Value Objects

Benjamin Eberlei at 16. May, 2017[20]

Software systems usually get more complex over time. In the beginning a variable starts out to represent something very simple with very few rules and constraints that can are enforced in a single location of the code.

Take this code example where the user selects a start and an end date to query a list of events:

```
class EventController
{
    public function listAction(Request $request)
    {
        $start = new \DateTime($request->query->get('start', '-60 minute'));
        $end = new \DateTime($request->query->get('end', 'now'));

        if ($start > $end) {
            $tmp = $end;
            $end = $start;
            $start = $tmp;
        }

        return [
            'events' => $this->eventRepository->findBetween($start, $end),
        ];
    }

    public function listTodayAction()
    {
        $start = new DateTime('today 00:00:00');
        $end = new DateTime('today 23:59:59');

        return [
            'events' => $this->eventRepository->findBetween($start, $end),
        ];
    }
}
```

This simple switch of start and end date when they are inverted is common and the simplicity of the code often means it is copied rather than abstracted into a method.

But how do we extract a method for this code? We could add a method `switchStartEnd()` on the `EventController`, but look how ugly that looks like:

[20] https://qafoo.com/blog/103_extracting_value_objects.html

```
private function switchStartEnd($start, $end)
{
    if ($start > $end) {
        $tmp = $end;
        $end = $start;
        $start = $tmp;
    }

    return array($start, $end);
}

public function listAction(Request $request)
{
    $start = new \DateTime($request->query->get('start', '-60 minute'));
    $end = new \DateTime($request->query->get('end', 'now'));

    list ($start, $end) = $this->switchStartEnd($start, $end);

    return [
        'events' => $this->eventRepository->findBetween($start, $end),
    ];
}
```

Plus, the biggest downside of this refactoring is the fact that you cannot use `switchStartEnd` in other places that perform date range handling.

The problem here is a code smell that is widespread in every codebase I have ever seen and is called "Primitive Obsession". It means that as developers we often rely on the most basic types of our programming language, instead of increasing the abstraction and introducing new types. In object oriented programming a type is equivalent to a new class.

Object oriented systems often have tons of classes that work on fullfilling a use-case, but they are not really types like string, integer or DateTime are.

In our example we are missing a DateRange class, and introducing it will immediately simplify our code and allow us to heavily unit-test business logic related to date ranges.

```
class DateRange
{
    public function __construct(DateTime $start, DateTime $end)
    {
        if ($start > $end) {
            $tmp = $end;
            $end = $start;
```

```
            $start = $tmp;
        }

        $this->start = $start;
        $this->end = $end;
    }

    public function getStart()
    {
        return $this->start;
    }

    public function getEnd()
    {
        return $this->end;
    }
}
```

Writing a unit-test for this is simple. Writing a test for the same code embedded into the Controller may be way too much work for the benefit.

We don't have to stop here though, we also have code constructing the DateRange in our controller that we can extract into the new value object:

```
class DateRange
{
    public static function today()
    {
        $start = new DateTime('today 00:00:00');
        $end = new DateTime('today 23:59:59');

        return new self($start, $end);
    }

    public static function fromStrings($start, $end)
    {
        return new self(new DateTime($start), new DateTime($end));
    }
}
```

Again, these methods on the `DateRange` can be easily tested. If we use the DateRange everywhere in our code we could easily add more code into the `fromStrings` method that does proper error handling when the strings are not valid dates for example.

Meanwhile the controller code is refactored into something very boring, all the logic is hidden in small testable classes:

```
public function listAction(Request $request)
{
    $range = DateRange::fromStrings(
        $request->query->get('start', '-60 minute'),
        $request->query->get('end', 'now')
    );

    return [
        'events' => $this->eventRepository->findBetween($range),
    ];
}

public function listTodayAction()
{
    return [
        'events' => $this->eventRepository->findBetween(DateRange::today()),
    ];
}
```

Introducing value objects is extremely helpful in structuring data and making small business rules reusable and abstracted across a large code base. The best candidates for this kind of refactoring in web applications are classes related to date (Week, DateRange, DateIterator, ...), Money, Email, IPAddress, URLs, slugged Strings, integers used as bitmasks and many others.

As soon as you detect business rules in your code that operate on primitive strings, integers or PHPs Date objects (they are not too powerful) you should think about extracting a value. If you wan't to avoid creating tons of object you can wait for 3-5 different rules on the same kind of primitive type or the same rule spread in 3-5 locations.

5.12 Refactoring Singleton Usage to get Testable Code

Benjamin Eberlei at 27. June, 2017[21]

So your code base is littered with singletons and using them? Don't worry, you can start refactoring them out of your code base class by class and introduce increased testability at every step. This strategy is very simple to implement and the propability of breaking your code is very low, especially when you are becoming more experienced with this technique.

Take the following example code of a SearchService that acceses a singleton to perform its work:

```
class SearchService
{
    public function searchAction($queryString, $type)
    {
        /** @var $solarium \Solarium_Client */
        $solarium = Solarium::getInstance();
        $select = $solarium->createSelect();

        // More and complex filtering logic to test

        $result = $solarium->query($select);

        return $result;
    }
}
```

To make this code testable without the singleton, we can use the lazy initialization pattern. The first step is to extract the method for the line that is fetching the singleton:

```
public function searchAction($queryString, $type)
{
    /** @var $solarium \Solarium_Client */
    $solarium = $this->getSolarium();
    // ...
}

protected function getSolarium()
{
    return Solarium::getInstance();
}
```

[21] https://qafoo.com/blog/107_refactoring_singletons_testability.html

You now have two options for testability. The most obvious is to create a test class that extends the original SearchService and overwrites the protected `getSolarium` to return a mock. But it is not very flexible and additional classes necessary for testing are not a good practice to follow.

Instead introduce a new instance variable and fetch the singleton only if this is null, making use of the so called lazy initialization pattern:

```
private $solarium;

private function getSolarium()
{
    if ($this->solarium === null) {
        $this->solarium = Solarium::getInstance();
    }
    return $this->solarium;
}

public function setSolarium(\Solarium_Client $solarium)
{
    $this->solarium = $solarium;
}
```

Since you would want to use constructor injection for all mandatory dependencies you could also introduce an optional constructor argument, like:

```
private $solarium;

public function __construct(\Solarium_Client $solarium = null)
{
    $this->solarium = $solarium ?: Solarium::getInstance();
}
```

Now this code is already testable using mocks:

```
class SearchServiceTest extends PHPUnit_Framework_TestCase
{
    public function testSearchFilter()
    {
        $solariumMock = \Phake::getMock(SolariumClient::class);
        $service = new SearchService($solariumMock);

        \Phake::when($solarium)->createSelect()->thenReturn(new \Solarium_Query_S
            elect($solarium));

        $service->search('Foobar', 'some_type');
```

```
        \Phake::verify($solarium)->query(\Phake::capture($select));

        // Perform assertions on $select
    }
}
```

If you perform this refactoring often you can entirely remove singletons from parts of your code base and move towards a more testable dependency injection.

6. Architecture

6.1 Why Architecture is Important

Kore Nordmann at 22. March, 2016[1]

We experience that the system architectures of our customers grow more and more complex. This is either because of scaling requirements or because developers like to try out new technologies like implementing Microservices in other languages then PHP (Node.js, Go, ...) or different storage technologies (MongoDB, CouchDB, Redis, ...). Depending on the application it is also common to introduce dedicated search services (Elasticsearch, Solr, ...), queueing systems (Beanstalk, RabbitMQ, ZeroMQ, ...) or cache systems (Redis, Memcache, ...).

Often there are very valid reasons to do this but there is also an important problem: You are creating a distributed system and they are really hard to get right & operate. Every system spread across multiple nodes in a network is a distributed system. A system consisting of a MySQL server and a PHP application server is already distributed, but this is a well known problem for most teams. Architecture decisions start to get critical once the data is distributed across multiple systems. Why is this the case?

One of the things which are hardest to repair in existing systems are inconsistencies of your data. Repairing this often even requires manual checks and sanitization which, depending on the amount of data, can take up really large amounts of time.

[1] https://qafoo.com/blog/079_why_architecture_matters.html

There are even studies [2] pointing out the costs of bugs in your architecture. If they are discovered late, when the system is already in production, then fixing these bugs can amplify the costs hundredfold. This is why we suggest to investigate and analyze your architecture before distributing your data and be careful doing so.

What are the main points you should check when designing a system architecture for a new project, during scaling an existing project or when introducing new storage technologies (search, storage, cache, ...)? There are a couple of questions we can ask ourselves:

- How can the consistency of data be ensured across multiple systems?
- How do we verify that the chosen systems fulfil their requirements?
- What are the technical and operational risks of newly introduced systems?
- How will the system handle latencies and failures of nodes?
- Is the overall application resilient against single node failures or how can this be accomplished?

On top of that those decisions should be documented and valued by certain criteria. There are even frameworks for documenting system architecture decisions and risks, which you might want to follow like ATAM[4]. Important assessment points are:

- Consistency and security of data
- Performance (latency, transaction throughput)
- Modifiability (Applicability to new products, future change costs)
- Availability (Hardware failure, software bugs)

6.1.1 Summary

When introducing new systems you should be careful especially when you plan to distribute your data across multiple nodes. Technology and architecture decisions should not be made because some topic is hot right now (like Microservices) but you should assess that the chosen system architecture actually fulfills your requirements and will be beneficial. Since there will be no *perfect* architecture for your use case one should always document the respective benefits, drawbacks and reasoning why some kind of architecture was implemented.

[2] "Code Complete (2nd ed.)"[3] by Steve McConnell (ISBN 0735619670)

[4] https://en.wikipedia.org/wiki/Special:BookSources/0735619670

6.2 Scaling Constraints of Languages

Kore Nordmann at 2. August, 2016[5]

Micro-Services or any set of small services are common again right now. While it can make a lot of sense to use a dedicated service for a well defined problem those services are sometimes used just to play with a different server software. While it is pretty obvious for most that selecting the right database is important the same is true for selecting the right language (virtual machine) for the job.

There are different types of services or server applications where different types of virtual machines (executing the opcodes / bytecode of the compiled source code) make more or less sense. What are the criteria we should base such a decision on and which language should we choose when?

When coming from a PHP background there are a couple of technology stacks you come across regularly:

- PHP (Connected to your webserver through FPM or mod_php with Apache)
- Node.js (JavaScript)
- Go (different server paradigms possible, most is also true for stuff like Elixir/OTP or Erlang/OTP)
- Java (doing the heavy crunching)

6.2.1 Why PHP?

Why did we start using PHP or how did it get so popular after all? One reason is **LCoDC$SS** as described in Roy T. Fieldings dissertation[6]. The abbreviations which describes the network architectural properties of HTTP / REST stands for:

- Layered (L)
  We are using a protocol which allows layering. You can put a reverse caching proxy or a load balancer in front of your application servers.
- Code on Demand (CoD)
  We are delivering HTML, CSS (& JavaScript) which are interpreted by browsers. The rendering does not happen on the server.

[5] https://qafoo.com/blog/088_scaling_constraints_of_languages.html

[6] Architectural Styles and the Design of Network-based Software Architectures[7] by Roy T. Fielding (2000)

- Client-Server (CS)
  The Browser (Client) interacts with a Server.
- Cached ($)
  The result of certain HTTP verbs (`GET`, `HEAD`) can be cached. This speeds up the web especially for static resources (images, ...).
- **Stateless (S)**
  The client always transmits the full state which is required to generate the response by the server. This might include session cookies. If your application is implemented correctly any of your frontend servers (serving the same application) can answer any request – no matter if the same client was connected to a different server in an earlier request.

PHP was built for this. With it's so called shared-nothing architecture it handles stateless HTTP requests perfectly. The PHP engine throws away any state (variables, file pointers, database connections, ...) after returning the response. It is almost impossible to build shared state between multiple requests (it is, of course, but don't do this).

This way PHP does not only use every CPU core on a server but usually it is even trivial to add additional servers and just scale your frontend this way. Only works if you respect the statelessness of HTTP requests, though.

6.2.2 So, Why Not PHP?

We just learned that PHP seems perfect for the web – does it serve all use cases perfectly? No.

Besides HTTP browsers also speak other protocols like WebSocket. With WebSockets you do not have the HTTP request response cycle but bi-directional permanent connections. PHP is not built for this.

Let's start with a use case where WebSockets are commonly used: A chat. You want that a message written by some user is immediately passed on to every other user in the same chat. This requires state on the server (who is in the chat) as well as passing messages between multiple connections. The incoming message should be passed on to all connected users.

You can implement this with HTTP (long polling), PHP and some software managing the shared state (database, in-memory storage) but it will be complicated and doesn't make much sense. This is where technologies like Node.js come into play.

Node.js

With Node.js it is really simple to develop your own server software. Using HTTP, Websockets or anything else is pretty straight forward and you probably already know the language from some frontend development. Using asynchronous IO Node.js allows to do some work while waiting for the network or file system. This allows to answer many requests per second while sharing some state – like for a chat.

On the other hand does Node.js not really support multiple cores in a sensible way. You can fork multiple processes or start multiple Node.js servers but you are loosing the benefit of shared state immediately. And managing this state will get hard and cumbersome again.

What I like to use Node.js for is quickly developing simple servers. For services which require state and can easily be handled by a single CPU core, Node.js might even be ready to serve production ready services.

Go

Developed by Google Go as a language is not as easy to pick up as Node.js when you come from a pure web development background. This is even worse for languages like Elixir or Erlang.

On the other it is fairly easy to write servers with those languages and they even utilize multiple CPU cores and still maintain shared state. If you need to scale a WebSocket-based application you might want to take a look at those.

When a single server is not large enough any more you'll have to come up with intelligent ideas to distribute your service - but this is far beyond "micro" then.

Java

Java is commonly used for complex backend services or to integrate with existing (legacy) stacks in companies. The Java VM actually allows to build both fast applications embracing the shared nothing nature of HTTP, but also servers with a shared state. Since state is so easy to share, Java applications are often not scal-

able beyond a single server. The Java VM on the other hand scales really well on a single server.

6.2.3 Summary

Choosing the language (virtual machine) is not just a matter of taste. Because of different paradigms the languages / virtual machines are build with the decision has architectural consequences. There are also other factors like developer experience, license costs or similar when it comes to choosing the right technology for your team & application.

6.3 How To Synchronize a Database With ElasticSearch?

Kore Nordmann at 14. June, 2016[8]

Since search engines like Apache Solr and ElasticSearch are easy to use and setup more and more applications are using them to index their content and make it searchable by the user. After all the underlying Lucene index engine provides far more powerful features then a plain MySQL full text search or similar solutions. With Apache Solr and ElasticSearch you can enhance the performance and the functionality of your website.

What we often stumble across, though, is the naiive approach of synchronizing both data storages. It boils down to:

- Store data in database
- Store data in search index

The problem with this approach is, as in any distributed system, that both the first and the second write **will** fail at some point. When the first write to the database fails usually nothing is indexed. When the second write to the search index fails you have content which exists but the user won't be able to find it in the search index.

Depending on your domain this might not be a problem. But if a price update of a product in an online shop fails you might actually loose money or at least customers will be angry about the inconsistencies on your site. There will be similar problems in other domains.

6.3.1 Transactions

The second idea of most developers now have is to build some kind of transaction support across both storages. We know transactions well from our Computer Science courses and relational database management systems, thus it is an intuitive decision. With a transaction wrapped around you'll get something like this:

```
try {
    $database->store($document);
    $searchIndex->index($document);
} catch (IndexException $e) {
    $database->remove($document);
    // Inform user that write failed
}
```

[8] https://qafoo.com/blog/086_how_to_synchronize_a_database_with_elastic_search.html

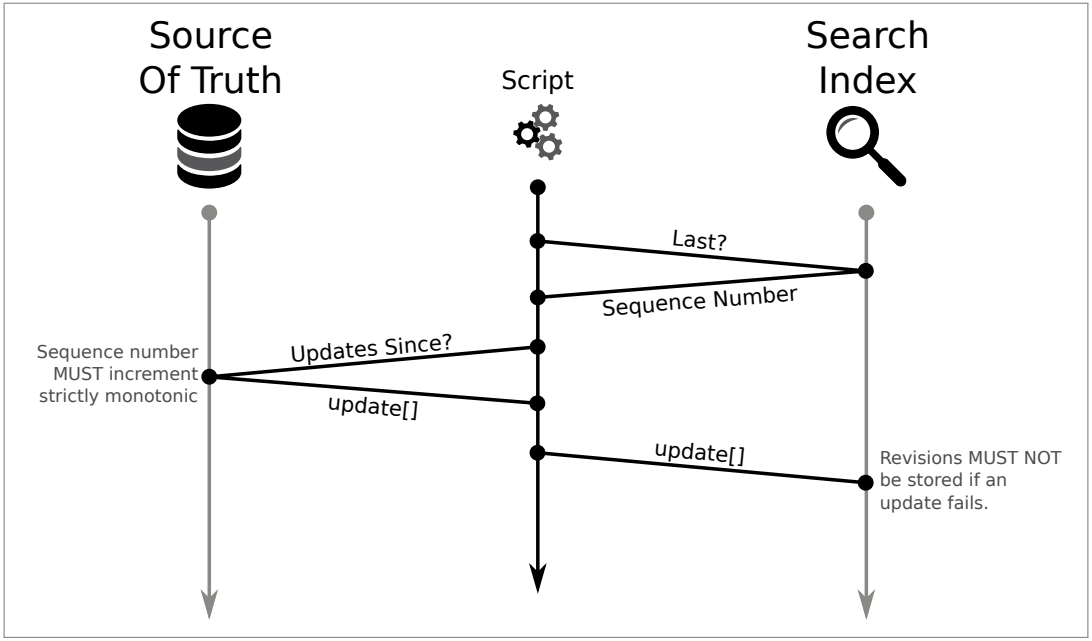
This works, but it puts the burden of resubmitting the document and handling the failure in your application on the user. Resubmitting a complex document might or might not work in a sane way for your users. If the user edited a long and complex text and all edits are lost because the search index failed to update – your user might not be that happy.

6.3.2 Changes Feed

There is a better way to implement this, often used in systems where you have one primary storage and any number of secondary storages. In this scenario your primary storage defines the state of your system – it is commonly called "Source Of Truth". In this example your database knows the definitive truth since everything is first stored there. Any other system, like your search index or caches should be updated based on the state in the database.

What we want to do is passing every change in the database on to the search index. A changes feed can do this. Some databases like CouchDB[9] offer this out of the box for every document, but how can this done with a relational database management system like MySQL? And how can the data be passed on safely to the search index? The process behind this can be illustrated by this simplified sequence diagram:

[9] http://guide.couchdb.org/draft/notifications.html



The idea behind this is that we start asking the target system (search index) what document it already knows. For this we need some kind of sequence number or revision – which **MUST** be sortable & strictly monotonic[10] to make sure that we do not loose any updates. With the last sequence number returned from the target system we can now load all or just a batch of changes from our database. These changes are then pushed to the search index. This sounds simple, right? But there are two questions remaining:

- How to I get MySQL to produce correct sequence numbers?
- How can I store the last sequence number in the search index?

6.3.3 Generating Correct Sequence Numbers

This is the hardest part with the described architectural pattern and a little bit harder to get right then one would assume in the first place. Every sequence number **MUST** be bigger then the last sequence number otherwise changes can be lost.

[10] https://en.wikipedia.org/wiki/Monotonic_function

The easiest way to implement this is an additional table with the following schema:

```
CREATE TABLE changes (
    sequence_number INT NOT NULL AUTO_INCREMENT,
    document_id INT NOT NULL,
    PRIMARY KEY (sequence_number)
);
```

The `document_id` would reference a data set in another table – if you want you can even define a foreign key relation with the data indexed in the search index. Defining the `sequence_number` number as `AUTO_INCREMENT` ensures that MySQL takes care of incrementing the sequence number.

With every change to a document we now also append a row to the `changes` table. Do this inside one transaction. Afterwards we can just query the changes like:

```
SELECT
    sequence_number, document.*
FROM changes
JOIN -- ... --
WHERE sequence_number > :since
ORDER BY sequence_number ASC
LIMIT 0, 100;
```

This would request the next 100 changes and join them with the actual data.

This table will grow fast you say? You're right.

But this can be optimized. The only thing we must keep in this table is the latest sequence number of each `document_id` so that we can run a clean import in the correct order. If there are dependencies between your documents this can get a little more complex but can still be solved.

Unnecessary at first, but at some point you might also have to handle the case where the `sequence_number` overflows.

6.3.4 Storing The Sequence Number

The sequence number must not be increased in the search index if no document was stored. Otherwise we would loose the document since the next request for new documents to the database will use the already increased sequence number.

Since systems like ElasticSearch do not support transaction we should store the sequence number associated with the update right in the document. Using a MAX query[11] and an index on the sequence number field we can still fetch the last sequence number from the search index.

Another option would be to store the last sequence number in a special document or somewhere else like the file system. If ElasticSearch now loses some or all documents we will not be aware of it and some documents will again be missing from the search index. Solutions like ElasticSearch tend to only persist their in memory data every few seconds and if the node crashes in the mean time they **will** loose some data. The described pattern ensures the index is brought up to date once the node recovers. Even with a complete data loss the architectural pattern described here will automatically ensure the entire index is rebuilt correctly.

6.3.5 Conclusion

The architectural pattern discussed in this blog post is nothing we invented. It is actually used by replication mechanisms of databases or other storage systems. We just showed you an adapted version for the concrete use case of synchronizing contents between a database and ElasticSearch.

The shown pattern is stable and resilient against failures. If ElasticSearch is not available while storing the data it will catch up later. Even longer down times or complete node failures will be automatically resolved. The mechanism is also called *eventual consistency* because of this.

The problem with this approach is that you'll need an extra script or cron job to synchronize both systems. The search index will also always lag slightly behind the database, depending on how often the script is run.

We generally advise to use this pattern to keep multiple system in sync when there is one source of truth. The pattern does not work when data is written to multiple different nodes.

[11] https://qa.fo/book-search-aggregations-metrics-max-aggregation

6.4 Common Bottlenecks in Performance Tests

Kore Nordmann at 19. April, 2016[12]

Most developers by now internalized that we should not invest time in optimizations before we know what happens exactly. Or as Donald Knuth wrote:

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. [13]

This is true for optimizations in your PHP code but also for optimizations regarding your infrastructure. We should measure before we try to optimize and waste time. When it comes to the assumed performance problems in your system architecture most people guess the root cause will be the database. This might be true but in most projects we put under load it proved to be false.

So, how can we figure out where the problems are located in our stack?

You need to test the performance, as always. For optimizing your PHP scripts you'd use XDebug and KCacheGrind or something like Tideways[15]. You'd trigger a single request and see what is slow in there and optimize that part.

6.4.1 System Test Setup

It is slightly more complicated to test your full stack. In the optimal case you simulate the real user behaviour. It is definitely not sufficient to just run `ab` or `siege` against your front page. For an online shop typical user tasks could be:

- Browsing the catalogue (random browser)
- Product searches
- User sign up & login
- Order checkout

We usually discuss the common tasks on a website with the owner of the website. Then we discuss the numbers for each of those task groups which we should

[12] https://qafoo.com/blog/082_common_bottlenecks_in_performance_tests.html

[13] Computing Surveys (PDF)[14], Vol 6, No 4, December 1974

[15] https://jmeter.apache.org/

simulate to mimic a certain scenario. With this information a jMeter[16] test can be authored simulating the real behaviour of your users. After a test run you can compare the access logs of the test with common access logs from your application to verify you simulated the right thing – if those logs are available.

6.4.2 Stack Analysis

Once you wrote sufficiently generic tests you will be able to simulate larger numbers of users accessing your website – some may call it DDOS'ing[17] your own page. When running these tests you can now watch all metrics for your system closely and you'll be able to identify the performance problems in your stack.

There are couple of tools which might help you here, but the list is far from extensive and depends a lot on the concrete application:

- vmstat watches CPU usage, free memory and other system metrics
- iftop shows if network performance is an issue
- Tideways[18] or XHProf for live analysis of your application servers

On top of that you should definitely watch the error logs on all involved systems.

6.4.3 It is Not Always The Database

The root cause for performance problems on websites we put under test weren't rooted in the database for most of our test runs:

- Varnish & Edge Side Includes (ESI)
  We tested a large online shop which used ESI so extensively that the application servers running PHP were actually the problem. The used framework had a high bootstrap time so that this was the most urgent performance impediment. The database wasn't even sweating.
- Network File System (NFS) locking issues
  Once you put a high load on a server sub systems will behave differently. NFS, for example, tries to implement some locking for a distributed file system. When multiple servers are accessing the same set of files it can stall your

[16] https://qa.fo/book-Denial-of-service_attack
[17] https://tideways.io
[18] https://qafoo.com/services/workshops/performance.html

application entirely. Something you will almost never hit during development but in load tests or later in production.

There are even configuration issues which only occur under load and degrade your performance more then any slow query will do.

- Broken server configurations
  In one case a web hoster who claimed to be specialized on cluster setups provided us with a setup where we ran the tests on. The cluster allowed a lot more FPM children to spawn then database connections. Once put under load the MySQL server rejected most of the incoming connections which meant the application failed hard.
- Opcode cache failures
  Wrong Opcode cache (APC, eAccelerator, ...) configuration can even degrade PHP performance. But this is also something you will not notice without putting the system under load. So you'll only notice this when many customers try to access your website or during a load test.

6.4.4 Summary

If you want to ensure the application performs even under load you should simulate this load before going live. The problems will usually not be where you thought they would be. Test, measure and analyze. The time invested in a load test will be a far more efficient investment then random optimizations based on guesswork.

Kore will talk about "Packt Mein Shop das?" ("Will My Shop Manage This?") at Oxid Commons[19] in Freiburg, Germany on 02.06.2016 about load testing.

[19] https://qa.fo/book-p261-knuth

6.5 Embedding REST Entities

Tobias Schlitt at 13. June, 2013[20]

During my talk at IPC Spring[21] I showed an approach for embedding entities in REST responses. This methodology might be useful if the standard use-case for your API is to request multiple related entities together. The downside of such an approach is the raised complexity of caching and especially purging logic. In this blog post I will further elaborate on the approach of resource embedding.

6.5.1 Entities

As an example I chose the resources *Product* and *Category* from the e-commerce domain. A product entity could be encoded through the media type `application/vnd.com.example.product+xml` as shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<product
  xmlns="urn:com.example.product"
  xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self"
    type="application/vnd.com.example.product+xml"
    href="http://example.com/products/23" />
  <name>Glow Stone Driveway</name>
  <description>Awesome ...</description>
  <atom:link rel="collection"
    type="application/vnd.com.example.category+xml"
    href="http://example.com/categories/geek_toys" />
  <!-- More links ... -->
</product>
```

A product has a `name` and a `description`. Besides that, the representation provides Atom[22] links as hyper media controls, using relations as recommended by the IANA[23].

Corresponding, the following example shows an encoded category entity:

```
<?xml version="1.0" encoding="UTF-8"?>
<category
```

[20] https://qafoo.com/blog/048_embedding_rest_entities.html
[21] https://qafoo.com/blog/047_pragmatic_rest_bdd_ipc.html
[22] https://tools.ietf.org/html/rfc4287
[23] https://qa.fo/book-link-relations

```
  xmlns="urn:com.example.category"
  xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self"
    type="application/vnd.com.example.category+xml"
    href="http://example.com/categories/geek_toys" />
  <name>Geek Toys</name>
  <products>
    <atom:link rel="item"
      type="application/vnd.com.example.product+xml"
      href="http://example.com/products/23" />
    <!-- ... -->
  </products>
  <!-- Links: overview, paging, sorting, ... -->
</category>
```

Note that here is a list of product items linked from a category, which could for example be paged, sorted, filtered and so on using URL parameters.

6.5.2 The Use-Case

Imagine the major use-case for the API is to retrieve the top 20 products from a category. To retrieve these together with product details, a client needs 21 `GET` requests on the first go. For each subsequent execution of the use case, the 21 requests must be repeated, although the response might be that the entities did not change.

Alternatively, the server can set expiry times for the entities, so that clients do not need to re-check that frequently. But such time spans are hard to calculate or even guess and can easily lead to stale caches.

So, depending on the requirements for data accuracy and clients, this might easily lead to a huge number of requests.

6.5.3 Resource Embedding with HAL

A promising idea to solve the named issue is to deliver resources in an embedded way while making explicit that the delivered data partially belongs to a different resource that can be found at a different URI. Searching the web reveals that there is already an approach for this out there, which is called Hypertext Application Language (HAL)[24].

[24] http://stateless.co/hal_specification.html

Embedding products into a category using a HAL encoding is shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<resource href="/categories/geek_toys">
  <name>Geek Toys</name>
  <products>
    <resource rel="product" href="/products/23">
      <name>Glow Stone Driveway</name>
      <description>Awesome ...</description>
      <link rel="category" href="/categories/geek_toys" />
    </resource>
    <!-- ... -->
  </products>
</resource>
```

HAL encodes any resource using the <resource> tag and has its own <link> tag. As can be seen, the products which are part of a category are embedded into its representation, but the categories that are assigned to a product are linked.

While this approach appears charming at first glance, it has some flaws:

First of all, HAL is meant to encode all entities in a standardized manner of a <resource> tag. That conflicts with one of the fundamental ideas of hyper media: to encode an entity in a semantically meaningful way.

Second, the HAL specification does not provide its own namespace, making it ugly to re-use among other XML languages. Finally, the specification introduces its own <link> element instead of re-using existing standards like Atom[25] or XLink[26].

6.5.4 Better Resource Embedding

Since the idea of HAL is essentially quite nice, I made up my mind to make the approach more standards-compliant and to mitigate the named issues. The basic idea is to re-use the Atom <link> elements, as in the original drafts, and embed resources as their children:

```
<?xml version="1.0" encoding="UTF-8"?>
<category xmlns="..." xmlns:atom="..."
  xmlns:p="urn:com.example.product"><!-- ... -->
  <products>
```

[25] https://tools.ietf.org/html/rfc4287

[26] http://www.w3.org/TR/xlink/

```
    <atom:link rel="item" type="..." href="...">
      <p:product><!-- ... -->
        <p:name>Glow Stone Driveway</p:name>
        <p:description>Awesome ...</p:description>
        <atom:link rel="collection"
          type="application/vnd.com.example.category+xml"
          href="http://example.com/categories/geek_toys" />
        </p:product>
    </atom:link>
    <!-- ... -->
  </products>
</category>
```

This example left out some unimportant details, it is basically derived from the original example in this post. However, in addition to the category namespace, the namespace for encoding product entities is also imported using the shortcut `p`. Inside of the `<atom:link>` for a product, the entity representation itself is embedded using this namespace.

Naturally, the mentioned drawbacks of HAL are mitigated: Clear semantics are kept, each media type ships with a decent XML namespace and a standard `<link >` element is used (Atom). Luckily, the Atom specs allow any external element to occur inside of `<atom:link>` so embedding the product entity is perfectly valid.

Furthermore, if clearly documented, that would allow the REST provider to transparently switch on and off embedding of specific resources. A client then must not rely on embedded resources and must be capable of fetching them through their link references. However, if available, it can use the embedded information to cache an embedded resource transparently.

6.5.5 Bottom Line

Of course, resource embedding is nothing I would recommend to you in general. It can be considered a *hack* for special use-cases. One of its drawbacks is to weaken the caching capabilities RESTful web APIs provide: The category resource must be re-fetched every time one of the embedded products changes. This of course also affects your application's caching logic, because you need to purge caches accordingly.

So, if you think about making use of embedded resources, make sure you analyzed the use-cases of your API carefully and keep the hack as local as possible.

7. Workflow

7.1 Coding in Katas

Tobias Schlitt at 18. February, 2013[1]

In almost any kind of sports you hone your skills by repeating a small piece of practice over and over again. Pretty much the same works for learning to play a musical instrument. The idea of Code Katas applies this simple but effective method of exercise to the world of programming.

A programmer's everyday life mostly consists of a single goal: getting things done. New features are to be shipped, bugs to be found and fixed. While you can certainly learn many new things thereby, this scenario lacks two essential factors for effective learning and self-development: **Time to try and possibility to fail.** Everyday work is typically not the situation where taking a risk and failing by intention is desired.

So, if you want to grow beyond yourself, you need to create an environment for your practice where failure is allowed. Starting a spare time project or even an open source project can help there. But finding a sufficiently clear and defined task to practice your skills is hard. That's where the idea of Code Katas jumps in:

> Kata (or literally: "form") is a Japanese word describing detailed choreographed patterns of movements practiced either solo or in pairs.

[1] https://qafoo.com/blog/034_coding_in_katas.html

```
-- https://en.wikipedia.org/wiki/Kata
```

A Code Kata is basically a well-defined, small task to be solved by programming. The time to solve a Kata should be reasonable to allow you to get started rapidly and to see first results quickly. However, you usually set yourself a time limit, say 1 hour, and stop precisely when the limit is reached. The actual result of this work is only a secondary concern, same applies to finishing the task. What counts is the actual practice. Pushed even further: You might actually want to throw away the code later.

The important part is to reflect on what you did: How did you approach the problem? Where did your approach lead you? What were the problems? What went well? Ideally, ask another developer for feedback or, if possible, directly work in pairs to learn from each other.

And when you are done: Start over from scratch. Maybe directly after finishing the first iteration or a week later. Do the same Kata again. Try a different approach or try the same and see where it leads you after your reflection. Reflect again and set yourself a goal for the next iteration.

With these characteristics, a Kata gives you a good basis to try out something new, to let off steam and to practice your skills as a programmer. Maybe you could try a purely test driven approach now. Maybe you will think about a complete OO model upfront then. And maybe you could apply Object Calisthenics in another iteration.

One of the most famous Code Katas is the Bowling Game Kata[2] by Robert C. Martin. The task of this Kata is to implement bowling with its simple rules. Not a tough challenge, but something with a connection to reality, clearly structured and well-defined. Uncle Bob used this Kata to visualize the methodology of Test Driven Development.

At Qafoo we tend to use Katas more and more frequently in our trainings for practice. Of course, Katas can neither replace practices derived from the real world nor working on own code. But they offer our training attendees the possibility to perform quick practices in-between theoretical lectures and to reflect on different approaches to certain problems in practical trying.

[2] https://qa.fo/book-ArticleS_UncleBob

There is a Kata Catalogue[3] where you can find quite some Katas. Tell us about your experience with Coding Katas!

[3] http://codingdojo.org/cgi-bin/wiki.pl?KataCatalogue

7.2 Why you need infrastructure and deployment automation

Benjamin Eberlei at 28. February, 2014[4]

The amount of time wasted on setup, synchronization and deployment of applications is often ignored by teams and managers. Manual server management and application deployment are a huge waste of time and ultimately money. In addition, manually performing these tasks is often prone to error and a big risk for uninterrupted uptime of production.

Software quality does not stop with tests and good CodeSniffer and PHP Mess Detector scores. The deployment and setup is equally important to the quality of an application. From our experience, this is a field where many teams could still achieve huge productivity gains.

As a yardstick for the optimal solution, we can take a look at what the Joel test[5] has to say about deployment and infrastructure and add some of our own additional requirements (some from the OpsReportCard[6]):

- Can you make a build in one step?
- Do you make daily builds?
- Do you use configuration management tools to automate infrastructure?
- Is the development setup documented and automated?
- Can you rollout and rollback deployments in one step?
- Can applications be deployed to a new server setup without changes to the code?

I want to discuss these bullets in short paragraphs and mention why solving these problems allows you to save time and money, if done properly.

7.2.1 Can you make a build in one step?

One of the Joel test requirements is building an application in one step. In the PHP world that means that - by executing one command or pushing a button in the CI system - a process is triggered that creates a running version of your project.

What does a complete build include?

[4] https://qafoo.com/blog/065_infrastructure_automation.html
[5] http://www.joelonsoftware.com/articles/fog0000000043.html
[6] http://www.opsreportcard.com/

- Code deployment to the appropriate machines
- Configuration
- Database setup or migrations
- Service location
- Cache Warming

Tools to automate code deplyoment include Bashscript + rsync[7] + symlinks, Capistrano[8] (with Capifony plugin for Symfony), Ansible[9] or Ant[10]. It is perfectly fine to automate with bash scripts, some automation is always better than none. However, the amount of boilerplate with Bash is often quite big compared to the other solutions that explicitly handle most of the deployment issues already. For example, we regularly implement a combination of Ansible and Ant deployment for our customers.

For database migrations there are tools such as DBDeploy[11], Liquibase[12] or FlywayDB[13]. These can normally be integrated and automated during deployment via Capistrano, Ansible or Ant. This requires the database schema to be always backwards compatible, something to carefully watch out for.

A compelling argument for introducing a reliable one-step build is that it can be safely triggered by any member of the team and therefore reduces the possibility of errors drastically.

7.2.2 Do you make daily builds?

If you don't have a one-step deployment, it is very unlikely that you are able to make daily builds in a cost-effective way. Without daily builds you lose one benefit of PHP, the rapid pace of development.

You lose money because you might be slower with new features than your comptetitor or because you cannot remove costly bugs from your software fast enough.

[7] http://rsync.samba.org/
[8] http://capistranorb.com/
[9] http://www.ansible.com/home
[10] http://ant.apache.org/
[11] http://dbdeploy.com/
[12] http://www.liquibase.org/
[13] http://flywaydb.org

7.2.3 Do you use configuration management tools to automate infrastructure?

When running many different projects, all with their own unique server setup, you can easily see how this will cost you time and money to maintain all these servers. Automating server maintenance helps a lot.

CFengine[14] was the first tool that automated the setup of servers and it was released in 1993. Competitors such as Puppet, Chef and Ansible are much younger, but still old enough to be considered stable and frankly there is no excuse not to use them anymore.

Configuration management tools all solve the problem of consistenly automating the setup of new and old servers. The goal is to set up all servers in a similar, consistent way by the use of automation instead of manual SSH'ing, package and configuration management.

Tasks you want to automate with configuration management:

- Deployment of SSH authorized keys
- Security updates of packages
- Management of log files (logrotate, fetching, aggregating)
- PHP, Apache/Nginx, MySQL, NoSQL, ... configuration
- Updates of commonly used software such as Wordpress, TYPO3, Drupal, Shopware, Oxid and their third party extensions.

Recently we recommend using Ansible[15] to our customers, because it is so much easier to understand than other tools.

Puppet[16] is much more complex to understand initially from our own experience and also has a much higher learning curve.

7.2.4 Is the development setup documented and automated?

There are two obvious reasons why not automating the setup of development machine is dangerous:

- Bugs and problems of the category "it works on my machine" are much more likely to happen, especially when development and production machines significantly differ (PHP 5.2 vs 5.3 was such a difference).

[14] http://cfengine.com/
[15] http://www.ansible.com/home
[16] http://puppetlabs.com/

- You cannot easily introduce new developers to new projects because it always requires a huge amount of time. This contributes to knowledge silos and significantly increases the bus factor of projects.

At a time where php applications use much more services than just the LAMP stack, it is vital that every developer has the same production-like environment to develop on. This can be achieved with virtual machines and tools like Vagrant[17]. Combined with configuration management tools like Puppet[18], Chef[19] or Ansible[20], the tools for easy setup of development machines exist and are stable.

A more detailed post from us on Vagrant will be forthcoming in the next weeks.

7.2.5 Can you rollout and rollback deployments in one step?

Nothing **always** works and for deployments of applications this is even more true. Not accounting for failure is carless. Failure during deployments is not a black swan event, it's real and probably in the >1% likelihood.

You can reduce the cost of failure by being able to rollback to an old, working version.

You need a deployment system where several versions of your code are available on the production machines and where you can switch between them with a single atomic operation. This also requires asset compilation and database schema migrations that are backwards and forwards compatible.

7.2.6 Can applications be deployed to a new server setup without changes to the code?

One common mistake in application development is hardcoding configuration variables. Especially in the early development of greenfield projects, too much configuration got hardcoded into the code. The side effect of this is that you will probably lose a lot of time once the first real production setup should be deployed. Hardcoded configuration variables also often complicate the installation of the applica-

[17] http://www.vagrantup.com/
[18] http://puppetlabs.com/
[19] http://www.getchef.com/chef/
[20] http://www.ansible.com/home

tion on different servers. For continous integration, staging machines or for multiple tenant setups, however, this is extremly important.

Not properly taking care of application configuration will make deployment much harder.

7.2.7 Conclusion

Not automating deployment and infrastructure is expensive and can cause frequent human errors. This article highlighted several deployment problems and discussed tools and procedures to fix them. In the next years, the question of infrastructure automation will become even more important, as automation has positive effects on time to market and allows scaling development teams up. As usual, companies that don't automate will have a hard time competing with companies that do.

If you need help automating your infrastructure and deployment, don't hesitate to contact us[21]. We can help you with the details and problems that commonly occur and we will make sure you get started very quickly.

[21] https://qafoo.com/contact.html

Impressum

Erstellt und vertrieben wird dieses Buch durch:

Qafoo Gesellschaft mit beschränkter Haftung
Büngelerstr. 64
46539 Dinslaken

E-Mail: contact@qafoo.com
Telefon: +49 (0)209 40501252
Registergericht: Amtsgericht Duisburg
Registernummer: HRB 27078

Umsatzsteuer-Identifikationsnummer gem. §27a UStG: DE272316452
Inhaltlich Verantwortlicher: Kore Nordmann (Anschrift, s.o.)

Vertretungsberechtigte Geschäftsführer:

- Kore Nordmann
- Manuel Pichler
- Tobias Schlitt