Principles of Package Design

Matthias Noback

Preparing your code for reuse

Principles of Package Design

Preparing your code for reuse

Matthias Noback

This book is for sale at http://leanpub.com/principles-of-package-design

This version was published on 2015-03-31

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Matthias Noback by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought ”Principles of PHP Package Design” by @matthiasnoback

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#

Also By Matthias Noback

A Year With Symfony

Un Año Con Symfony

To life just out
and long familiar

Contents

Introduction

The situation

This book assumes you are a developer writing object oriented code every day. As you design new classes, you constantly ask yourself (or your co-worker): does this method belong here? Is it okay for this class to know about …? Do I have to define an interface for this class? Is it important to the other class that I return an array? Should I return `null` or `false`?

While you are working on a class, there are many design decisions that you have to make. Luckily there are also many wise people who have written about class design, whose ideas can help you make those decisions. Think about the SOLID principles, the Law of Demeter, design patterns, concepts like encapsulation, data hiding, inheritance, composition. If you like to read about these subjects, you will know much about them already. But if you don't then you may still apply many of these principles just because you have some kind of programmer's intuition for them.

So you know your way around classes. The classes you produce make sense: they contain what you would expect them to contain, no more, no less. You can look at the code of a class and see what it is supposed to do. But now some of those well-designed classes start to cluster together. They form little aggregates of code that cover the same subject, do something in a similar way, or are coupled to the same third-party code. They belong together, so you put them in the same directory, or in the same namespace. In other words you are creating packages* of them.

And just like you are very conscious about the way you design your classes, you immediately start asking yourself some questions about the design of the packages you create: can I put this class in the same package, or does it deserve a new package? Is this package too big and should it be split? Is it okay if package A depends on package B?

Unfortunately, many developers don't have definitive answers for these questions. They all use different rules when it comes to package design. Those rules are the result of discussions on GitHub, or worse: they are based on gut feelings of the lead

developers of well-known open source projects. But guessing how to do package design is not necessary at all. We don't have to invent package design principles ourselves again and again. We can stand on the shoulders of giants here.

One of those giants is Robert C. Martin, a person I very much admire because of the clarity and strength with which he speaks about the principles that should govern your work as a software developer, both with regard to the quality of your code[1], as well as the quality of you as a human being[2] who creates software.

Robert wrote about package (or "component") design principles on several occasions. I first learned about these principles when reading the articles available for download at butunclebob.com[3]. Then I watched some of his videos about component design on cleancoders.com[4] in which he explained the same principles again, but in a more urgent manner.

These package design principles are actually not difficult to understand (in fact, they are much more straight-forward than the SOLID principles of class design). They provide an answer to the following questions:

- Which classes belong inside a package and which don't?
- Which packages are safe to depend on and which aren't?
- What can I do for my users to enhance the usability of a package?
- What can I do for myself to keep a package maintainable?

The discovery that there exist such clear answers to these questions has led me to undertake this effort: to learn more about these package design principles as explained by Robert Martin, then to explain and discuss them in a broader way.

I think that to learn about these package design principles will be of great use to you. If you have never created a package before you will know from the start which principles should govern your package design decisions. And if you have already created a few, or even many packages, you will be able to use the design principles to fix design issues in existing packages and make your next package even better.

[1] http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882

[2] http://www.amazon.com/The-Clean-Coder-Professional-Programmers/dp/0137081073

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

[4] https://cleancoders.com

Overview of the contents

The majority of this book covers package design principles. But first we must consider the contents of a package: classes and interfaces. The way you design them has great consequences for the characteristics of the package in which they will eventually reside. So before considering package design principles themselves, we first need to take a look at the principles that govern class design. These are the so-called *SOLID principles*. Each letter of this acronym stands for a different principle and we will briefly (re)visit them in the first part of this book.

The second part of the book (the main part) covers the six major principles of package design. The first three are about *cohesion*. While class cohesion is about which *methods* belong inside a *class*, package cohesion is about which *classes* belong inside a *package*. The *package cohesion principles* tell you which classes should be put together in a package, when to split packages, and if a combination of classes may be considered a "package" in the first place.

The second three package design principles are about *coupling*. Coupling is important at the class level, since most classes are pretty useless on their own. They need other classes with which to collaborate. Class design principles like the *Dependency inversion principle* help you write nicely decoupled classes. But when the dependencies of a class lie outside its own package, you need a way to determine if it's safe to couple the package to another package. The *package coupling principles* will help you choose the right dependencies. They will also help you recognize and prevent wrong directions in the dependency graph of your packages.

Notes about the code samples

Let me add some quick notes about the code samples in this book. Although this book tries to be agnostic about programming languages, the code samples are written in PHP. It's the programming language I know best, which is convenient for me. However, it should be no problem at all to understand it if you are more familiar with other programming languages.

The sample code should not be considered "useful in production". It just tries to bring across some technical points. You shouldn't copy/paste it into your projects (it wouldn't even execute well in most cases).

At many places I abbreviate the code samples using . . . , in particular when the exact statements are irrelevant for the example. I repeat as little of the code as possible in subsequent modified pieces of the same code, for brevity and clarity.

One last note: when the code samples contain interfaces, their name always has the suffix "Interface". I personally don't think this is best practice, but I did it nonetheless because this way it should be easier to remember which things are classes and which are interfaces.

Thank you

Before we dive into the subject matter of this book: just a quick word of thanks to some people in particular. First of all to Robert Martin, who has come up with a lot of what is inside this book. He encouraged me to further engage in the effort to complete this book. Then there were many people who provided me with some valuable feedback based on their proof-readings, like Brian Fenton, Kevin Archer, Luis Cordova, Mark Badolato, Matthijs van Rietschoten, Richard Perez, Rolf Babijn, Ross Tuck, Sebastian Stok, Thomas Shone and Peter Rehm. One of the proof- readers that I want to mention in particular is Peter Bowyer, who offered some detailed suggestions, about content as well as language. He also did the brilliant suggestion to turn this initially PHP-specific book into something that is interesting and useful for *every* software developer.

Even though this book is language-agnostic, I owe a lot to the particular community to which I belong: the PHP community. Thank you all for being awesome - providing everyone with great reading material, daily insights, friendly invitations and lots of good code.

Finally, thank you Lies, Lucas and Julia. Thank you for letting me step away from family business and write this book in solitude. And thank you for always embracing me when I stepped back in.

Class design

Introduction

SOLID principles

Developers like you and I need help with making our decisions: we have incredibly many choices to make, each day, all day. So if there are some principles we think are sound, we happily follow them. Principles are guidelines, or "things you should do". There is no real imperative there. You are not *required* to follow these principles, but in theory you *should*.

When it comes to creating classes, there are many guidelines you should follow, like: choose descriptive names, keep the number of variables low, use few control structures, etc. But these are actually quite general programming guidelines. They will keep your code readable, understandable and therefore maintainable. Also, they are quite specific, so your team may be very strict about them ("at most two levels of indentation inside each method", "at most three instance variables", etc.).

Next to these general programming guidelines there are also some deeper principles that can be applied to class design. These are powerful principles, but less specific in most cases, and it is therefore much harder to be strict about them. Each of these principles brings some room for discussion. Also, not all of them can or should be applied all the time (unlike the more general programming guidelines: when not applied, your code will most certainly start to get in your way pretty soon).

The principles I refer to are named "SOLID principles" coined by Robert Martin. In the following chapters I will give a brief summary of each of the principles. Even though the SOLID principles are concerned with the design of classes, a discussion of them belongs in this book, since the class design principles resonate with the package design principles we will discuss in the second part of this book.

Why follow the principles?

When you learn about the SOLID principles, you may ask yourself: why do I have to stay true to them? Take for example the *Open/closed principle*: "You should be able to extend the behavior of a class, without modifying it." Why, actually? Is it so bad to change the behavior of a class by opening its file in an editor and making some changes? Or take for instance the *dependency inversion principle*, which says: "Depend on abstractions, not on concretions." Why again? What's wrong with depending on concretions?

Of course, in the following chapters I will take great care in explaining to you why you should use these principles and what happens if you don't. But to make this clear before you take the dive: the SOLID principles for class design are there to prepare your code base for future changes. You want these changes to be local, not global, and small, not big.

Prepare for change

Think about this for a minute: why do you want to make as few and as little changes as possible to existing code? First of all there is the risk of one of those changes breaking the entire system. Then there is the amount of time you need to invest for each change in an existing class; to understand what it originally does, and where best to add or remove some lines of code. But there is also the extra burden in modifying the existing unit tests for the class. Besides, each change may be part of some review process, it may require a rebuild of the entire system, it may even require others to update their systems to reflect the changes.

This would almost lead us to the conclusion that changing existing code is something we don't want. However, to dismiss change in general would be way too much. Most real businesses change heavily over time, and so do their software requirements. So to keep functioning as a software developer, you need to embrace change* yourself too. And to make it easier for you to cope with the quickly changing requirements, you need to prepare your code for them. Luckily there are many ways to accomplish that, which can all be extracted from the following five SOLID class design principles.

The Single responsibility principle

The *Single responsibility principle* says that:

> A class should have one, and only one, reason to change.

There is a strange little jump here, from this principle being about "responsibilities" to the explanation being about "reasons to change". Well, this is not so strange when you think about it: each responsibility is also a reason to change.

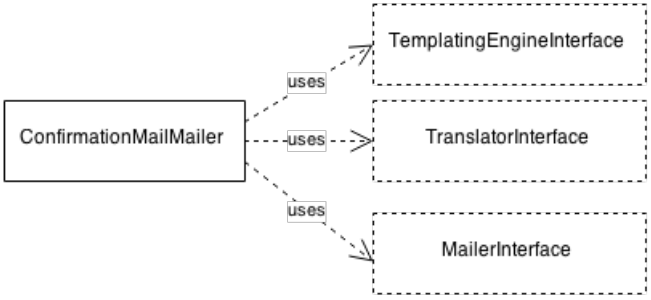
A class with too many responsibilities

Let's take a look at a concrete, probably recognizable example of a class that is used to send a confirmation mail to the email address of a new user. It has some dependencies, like a templating engine for rendering the body of the email message, a translator for translating the message's subject and a mailer for actually sending the message. These are all injected by their interface (which is good, see The dependency inversion principle):

```
class ConfirmationMailMailer
{
    private $templating;
    private $translator;
    private $mailer;

    public function __construct(
        TemplatingEngineInterface $templating,
        TranslatorInterface $translator,
```

```
        MailerInterface $mailer
    ) {
        $this->templating = $templating;
        $this->translator = $translator;
        $this->mailer = $mailer;
    }

    public function sendTo(User $user)
    {
        $message = $this->createMessageFor($user);

        $this->sendMessage($message);
    }

    private function createMessageFor(User $user)
    {
        $subject = $this
            ->translator
            ->translate('Confirm your mail address');

        $body = $this
            ->templating
            ->render('confirmationMail.html.tpl', [
                'confirmationCode' => $user->getConfirmationCode()
            ]);

        $message = new Message($subject, $body);

        $message->setTo($user->getEmailAddress());

        return $message;
    }

    private function sendMessage(MessageInterface $message)
    {
```

```
        $this->mailer->send($message);
    }
}
```



The initial situation

Responsibilities are reasons to change

When you would talk to someone about this class, you would say that it has two jobs, or two responsibilities: to *create* a confirmation mail, and to *send* it. These two responsibilities are also its two reasons to change. Whenever the requirements change regarding the creation of the message, or regarding the sending of the message, this class will have to be modified. This also means that when either of the responsibilities requires a change, the entire class needs to be opened and modified, while most of it may have nothing to do with the requested change itself.

Since changing existing code is something that needs to be prevented, or at least be confined (see the Introduction), and responsibilities are reasons to change, we should try to minimize the number of responsibilities of each class. This would at the same time minimize the chance that the class has to be opened for modification.

Because a class with no responsibilities is quite a useless class, the best we can do with regard to minimizing the number of responsibilities, is to reduce it to one. Hence the *single* responsibility principle.

Recognizing violations of the

Single responsibility principle

This is a list of symptoms of a class that may violate the *Single responsibility principle*:

- The class has many instance variables
- The class has many public methods
- Each method of the class uses other instance variables
- Specific tasks are delegated to private methods

These are all good reasons to extract so-called "collaborator classes" from the class, thereby delegating some of its responsibilities and making it adhere to the single responsibility principle.

Refactoring: using collaborator classes

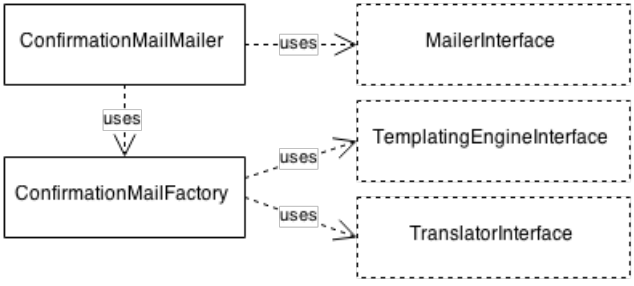
We now know that the `ConfirmationMailMailer` does too much and is therefore a liability. The way we can (and in this case should) refactor the class, is by extracting collaborator classes. Since this class is a "mailer", we let it keep the responsibility of *sending a message* to the user. But we extract the responsibility of *creating the message*. Creating objects traditionally calls for a "factory", hence we introduce the `ConfirmationMailFactory`:

```
class ConfirmationMailMailer
{
    private $confirmationMailFactory;
    private $mailer;

    public function __construct(
        ConfirmationMailFactory $confirmationMailFactory
        MailerInterface $mailer
    ) {
        $this->confirmationMailFactory = $confirmationMailFactory;
```

```
        $this->mailer = $mailer;
    }

    public function sendTo(User $user)
    {
        $message = $this->createMessageFor($user);

        $this->sendMessage($message);
    }

    private function createMessageFor(User $user)
    {
        return $this->confirmationMailFactory->createMessageFor($use\
r);
    }

    private function sendMessage(MessageInterface $message)
    {
        $this->mailer->send($message);
    }
}

class ConfirmationMailFactory
{
    private $templating;
    private $translator;

    public function __construct(
        TemplatingEngineInterface $templating,
        TranslatorInterface $translator
    ) {
        $this->templating = $templating;
        $this->translator = $translator;
    }
```

```
    public function createMessageFor(User $user);
    {
        /*
         * Create an instance of MessageInterface based on the
         * given User
         */
        $message = ...;

        return $message;
    }
}
```

Now the creation logic of the confirmation mail has been nicely put inside `ConfirmationMailFact`
It would be even better if an interface was defined for the factory class, but it's fine for now.



Introducing the `ConfirmationMailFactory`

Advantages of having a single responsibility

As a side-effect of the refactoring to single responsibilities, both of the classes are easier to test. You can now test both responsibilities separately. The correctness of the created message can be verified by testing the `createMessageFor()` method of `ConfirmationMailFactory`. And testing the `sendTo()` method of `ConfirmationMailMailer` is also quite easy now, because you can mock up the complete message creation process and just focus on sending the message.

In general you will notice that classes with single responsibilities are easier to test. Having a single responsibility will make a class smaller, so you have to write fewer tests to keep that class covered. This will be easier for your mind to grasp. Also these small classes will have fewer private methods with effects that need to be verified in a unit test.

Finally, smaller classes are also simpler to maintain: it is easier to grasp their purpose and all the implementation details are where they belong: in the classes responsible for them.

A small peek ahead: common closure

While the *Single responsibility principle* should be applied to classes, in a slightly different way it should also be applied to groups of classes (also known as *packages*). In the context of package design "having only one reason to change" becomes "being closed against the same kind of changes". The corresponding package principle is called the *Common closure principle*.

A somewhat exaggerated example of a package that doesn't follow this *Common closure principle*, would be a package which knows how to connect with a MySQL database *and* knows how to produce HTML pages. Such a package would have too many responsibilities and will be opened (i.e. modified) for all sorts of reasons. The solution for packages like this one is to split them into smaller packages, each with less responsibilities, and therefore less reasons to change.

There is another interesting similarity between the *Single responsibility principle* of class design and the *Common closure principle* of package design that I'd like to quickly mention here: following these principles in most cases reduces class (and package) coupling.

When a class has many responsibilities, it is likely to have many dependencies too. It probably gets many objects injected as constructor arguments to be able to fulfill its goal. For example the `ConfirmationMailMailer` needed a translator service, a templating engine and a mailer to create and send a confirmation mail. By depending on those objects it was directly coupled to them. When we applied the *Single responsibility principle* and moved the responsibility of creating the message itself to a new class `ConfirmationMailFactory`, we reduced the number of dependencies of `ConfirmationMailMailer` and thereby reduced its coupling.

The same goes for the *Common closure principle*. When a package has many dependencies, it is tightly coupled to each of them, which means that a change in one of the dependencies will likely require a change in the package too. Applying the *Common closure principle* to a package means reducing the reasons why a package would need to change. Removing dependencies, or deferring them to other packages is one way to accomplish this.

The Open/closed principle

The *Open/closed principle* says that:

> You should be able to extend a class's behavior, without modifying it.

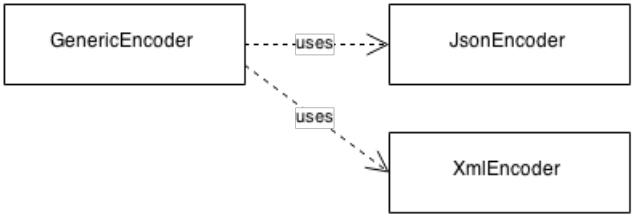
Again, a small linguistic jump has to be made from the name of the principle to its explanation: a unit of code can be considered "open" for extension when its behavior can be easily changed *without* modifying it. The fact that no actual modification is needed to change the behavior of a unit of code makes it "closed" for modification.

A class that is closed for extension

Take a look at the `GenericEncoder` class below. Notice the branching inside the `encodeToFormat()` method that is needed to choose the right encoder based on the value of the `$format` argument:

```
class GenericEncoder
{
    public function encodeToFormat($data, $format)
    {
        if ($format === 'json') {
            $encoder = new JsonEncoder();
        } elseif ($format === 'xml') {
            $encoder = new XmlEncoder();
        } else {
            throw new InvalidArgumentException('Unknown format');
        }

        $data = $this->prepareData($data, $format);
```

```
        return $encoder->encode($data);
    }
}
```



The initial situation

Let's say you want to use the `GenericEncoder` to encode data to the Yaml format, which is currently not supported. The obvious solution would be to create a `YamlEncoder` class for this purpose and then add an extra condition inside the existing `encodeToFormat()` method:

```
class GenericEncoder
{
    public function encodeToFormat($data, $format)
    {
        if (...) {
            ...
        } elseif (...) {
            ...
        } elseif ($format === 'yaml') {
            $encoder = new YamlEncoder();
        } else {
            ...
        }
        ...
    }
}
```

As you can imagine each time you want to add another format-specific encoder, the `GenericEncoder` class itself needs to be *modified*: you can not change its behavior without modifying its code. This is why the `GenericEncoder` class can not be considered *open for extension* and *closed for modification*.

Let's take a look at the `prepareData()` method of the same class. Just like the `encodeToFormat()` method it contains some more format-specific logic:

```
class GenericEncoder
{
    public function encodeToFormat($data, $format)
    {
        ...
        $data = $this->prepareData($data, $format);
        ...
    }

    private function prepareData($data, $format)
    {
        switch ($format) {
            case 'json':
                $data = $this->forceArray($data);
                $data = $this->fixKeys($data);
                // fall through
            case 'xml':
                $data = $this->fixAttributes($data);
                break;
            default:
                throw new InvalidArgumentException(
                    'Format not supported'
                );
        }

        return $data;
    }
}
```

The `prepareData()` method is another good example of code that is *closed for extension* since it is *impossible* to add support for another format without modifying the the code itself. Besides, these kind of switch statements are not good for maintainability. When you would have to modify this code, for instance when you introduce a new format, it is likely that you would either introduce some code duplication or simply make a mistake because you overlooked the "fall-through" case.

Refactoring: abstract factory

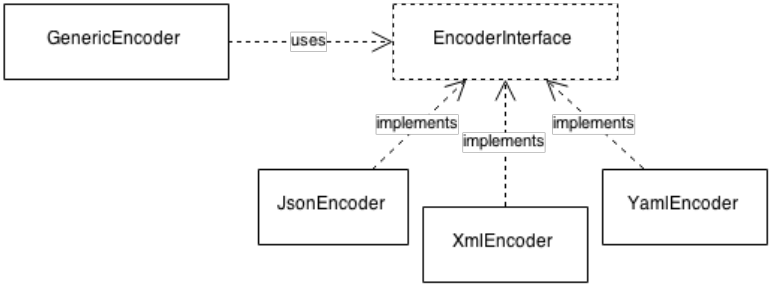
We'd like to fix this bad design, which requires us to constantly dive into the `GenericEncoder` class to modify format-specific behavior. We first need to delegate the responsibility of resolving the right encoder for the format to some other class. When you think of responsibilities as reasons to change (see the previous chapter), this makes perfect sense: the logic for finding the right format-specific encoder is something which is likely to change, so it would be good to transfer this responsibility to another class.

This new class might as well be an implementation of the "abstract factory" design pattern. The abstractness is represented by the fact that its `create()` method is bound to return an instance of a given interface. We don't care about its actual class; we only want to retrieve an object with an `encode($data)` method. So we need an interface for such format-specific encoders:

```
/**
 * Interface for format-specific encoders
 */
interface EncoderInterface
{
    /**
     * Encode the given data
     *
     * @param mixed $data
     * @return string
     */
    public function encode($data);
}
```

Now make sure every existing format-specific encoder implements this interface:

```
class JsonEncoder implements EncoderInterface
{
    ...
}

class XmlEncoder implements EncoderInterface
{
    ...
}

class YamlEncoder implements EncoderInterface
{
    ...
}
```



Introducing the `EncoderInterface`

Now we can move the creation logic of format-specific encoders to a class with just this responsibility. Let's call it the `EncoderFactory`:

```
class EncoderFactory
{
    public function createForFormat($format)
    {
        if ($format === 'json') {
            return new JsonEncoder();
        } elseif ($format === 'xml') {
            return new XmlEncoder();
        } elseif (...) {
            ...
        }

        throw new InvalidArgumentException('Unknown format');
    }
}
```

Then we have to make sure that the `GenericEncoder` does not create any format-specific encoders anymore. Instead, it should delegate this job to the `EncoderFactory`, which it receives as a constructor argument:

```
class GenericEncoder
{
    private $encoderFactory;

    public function __construct(
        EncoderFactory $encoderFactory
    ) {
        $this->encoderFactory = $encoderFactory;
    }

    public function encodeToFormat($data, $format)
    {
        $encoder = $this->encoderFactory->createForFormat($format);

        $data = $this->prepareData($data, $format);
```

```
        return $encoder->encode($data);
    }
}
```

By leaving the responsibility of creating the right encoder to the encoder factory the `GenericEncoder` now conforms to the *Single responsibility principle*.

Using the encoder factory for fetching the right encoder for a given format means that adding an extra format-specific encoder does not require us to modify the `GenericEncoder` class anymore. We need to modify the `EncoderFactory` class instead.

But when we take a look at the `EncoderFactory` there is still an ugly hard-coded list of supported formats and their corresponding encoders. Even worse, class names are still hard-coded. This means that now the `EncoderFactory` is *closed against extension*. It thereby violates the *Open/closed principle*.

Quick refactoring opportunity: dynamic class names?

It seems there is some low-hanging fruit here, ready to be taken. As you may have noticed there is a striking symmetry inside the `switch` statement: for the `json` format a `JsonEncoder` instance is being returned, for the `xml` format an `XmlEncoder`, etc. If your programming language supports dynamic class names like PHP does, this could be easily refactored into something that is not hard-coded anymore:

```
$class = ucfirst(strtolower($format)) . 'Encoder';
if (!class_exists($class)) {
    throw new InvalidArgumentException('Unknown format');
}
```

Yes, this is in fact equivalent code. It's shorter and it removes the need for a `switch` statement. It even introduces a bit more flexibility: in order to extend its behavior you don't need to modify the code anymore. In the case of the new encoder for the Yaml format, we only need to create a new class which follows the naming convention: `YamlEncoder`. And that's it. Q. However, using dynamic class names to make a class extensible like this introduces some new problems and doesn't fix some of the existing problems:

- Introducing a naming convention only offers some flexibility for you as the maintainer of the code. When someone else wants to add support for a new format they have to put a class in your namespace, which is possible, but not really user-friendly.
- A much bigger issue: creation logic is still being reduced to `new ...()`. If for instance an encoder class has some dependencies, there is no way to inject them (e.g. as constructor arguments). We will address this issue below.

Refactoring: making the abstract factory open for extension

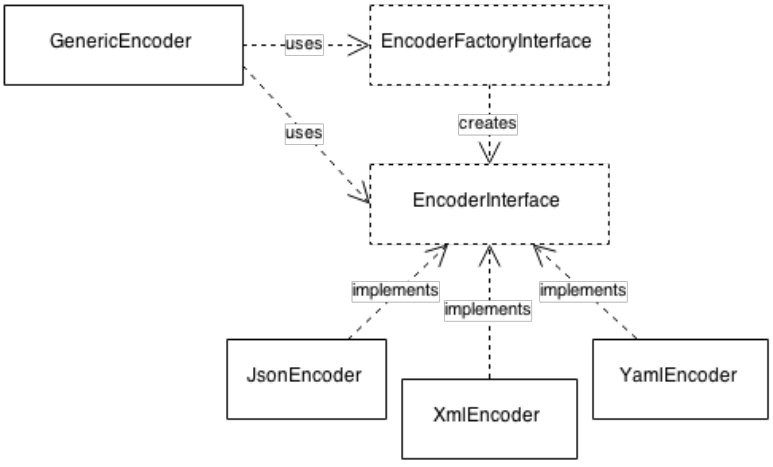
A first step we could take is to apply the *Dependency inversion principle* by defining an interface for encoder factories:

```
interface EncoderFactoryInterface
{
    /**
     * Create an encoder for the given format
     *
     * @param string $format
     * @return EncoderInterface
     */
    public function createForFormat($format);
}
```

The `EncoderFactory` we already have should implement this new interface and the constructor argument of the `GenericEncoder` should have the interface as a type-hint:

```
class EncoderFactory implements EncoderFactoryInterface
{
    ...
}

class GenericEncoder
{
    public function __construct(
        EncoderFactoryInterface $encoderFactory
    ) {
        ...
    }
    ...
}
```

By doing so we have added a first extension point: if any developer in the future would like to change the encoder creation logic, they would not be forced to modify an existing class. It would be sufficient to implement the `EncoderFactoryInterface` and provide their own implementation of an encoder factory as the constructor argument of the `GenericEncoder`.



Introducing the `EncoderFactoryInterface`

Although this is a first step in the direction of writing classes that are open for extension and closed for modification, re-implementing `EncoderFactoryInterface` just to add support for another format seems a bit inefficient. When a new format comes along, we want to keep using the same old `EncoderFactory`, but we want to support the new format without touching the code of the class itself. Also, if one of the encoders would need another object to fulfill its task, it's currently not possible to provide that object as a constructor argument, because the *creation logic* of each of the encoders is hard-coded in the `EncoderFactory` class.

In other words, it's impossible to extend or change the behavior of the `EncoderFactory` class without modifying it: the logic by which the encoder factory decides which encoder it should create and how it should do that for any given format can't be changed from the outside. But it's quite easy to move this logic out of the `EncoderFactory` class, thereby making the class open for extension.

There are several ways to make a factory like `EncoderFactory` open for extension. I've chosen to inject specialized factories into the `EncoderFactory`:

```
class EncoderFactory implements EncoderFactoryInterface
{
    private $factories = array();

    /**
     * Register a callable that returns an instance of
     * EncoderInterface for the given format.
     *
     * @param string $format
     * @param callable $factory
     */
    public function addEncoderFactory($format, callable $factory)
    {
        $this->factories[$format] = $factory;
    }

    public function createForFormat($format)
    {
        $factory = $this->factories[$format];

        // the factory is a callable
        $encoder = $factory();

        return $encoder;
    }
}
```

For each format it is possible to inject a callable[5]. The `createForFormat()` method takes that callable, calls it, and uses its return value as the actual encoder for the given format.

This fully dynamic and extensible implementation allows developers to add as many format-specific encoders they like. Injecting the format-specific encoders would look like this:

[5] http://www.php.net/manual/en/language.types.callable.php

```
$encoderFactory = new EncoderFactory();

$encoderFactory->addEncoderFactory(
    'xml',
    function () {
        return new XmlEncoder();
    }
);

$encoderFactory->addEncoderFactory(
    'json',
    function () {
        return new JsonEncoder();
    }
);

$genericEncoder = new GenericEncoder($encoderFactory);

$data = ...;

$jsonEncodedData = $genericEncoder->encode($data, 'json');
```

By introducing callable factories we have relieved the `EncoderFactory` from the responsibility of providing the right constructor arguments for each encoder. In other words, we pushed knowledge about *creation logic* outside of the `EncoderFactory`, which makes it at once adhere to both the *Single responsibility principle* and the *Open/closed principle*.

Refactoring: polymorphism

We have put some effort into implementing a nice abstract factory for encoders, but the `GenericEncoder` still has this ugly `switch` statement for preparing the data before it is encoded:

```
class GenericEncoder
{
    private function prepareData($data, $format)
    {
        switch ($format) {
            case 'json':
                $data = $this->forceArray($data);
                $data = $this->fixKeys($data);
                // fall through
            case 'xml':
                $data = $this->fixAttributes($data);
                break;
            default:
                throw new InvalidArgumentException(
                    'Format not supported'
                );
        }

        return $data;
    }
}
```

Where should we put this format-specific data preparation logic? Rephrased: whose *responsibility* would it be to prepare data before encoding it? Is it something the `GenericEncoder` should do? No, because preparing the data is format-specific, not generic. Is it the `EncoderFactory`? No, because it only knows about creating encoders. Is it one of the format-specific encoders? Yes! They know everything about encoding data for their own format.

So let's delegate the "prepare data" method to the specific encoders by adding a method `prepareData($data)` to the `EncoderInterface` and call it in the `encodeToFormat()` method of the `GenericEncoder`:

```php
interface EncoderInterface
{
    public function encode($data);

    /**
     * Do anything that is required to prepare the data for
     * encoding it.
     *
     * @param mixed $data
     * @return mixed
     */
    public function prepareData($data);
}

class GenericEncoder
{
    public function encodeToFormat($data, $format)
    {
        $encoder = $this->encoderFactory->createForFormat($format);

        /*
         * Preparing the data is now a responsibility of the
         * format-specific encoder
         */
        $data = $encoder->prepareData($data);

        return $encoder->encode($data);
    }
}
```

In the case of the `JsonEncoder` this would look like this:

```
class JsonEncoder implements EncoderInterface
{
    public function encodeToFormat($data, $format)
    {
        ...
    }

    public function prepareData($data)
    {
        $data = $this->forceArray($data);
        $data = $this->fixKeys($data);

        return $data;
    }
}
```

This is not a great solution, because it introduces something called "temporal coupling": before calling `encodeToFormat()` you always have to call `prepareData()`. If you don't, your data may be invalid and not ready to be encoded.

So instead, we should make preparing the data part of the actual encoding process inside the format-specific encoder. Each encoder should decide for itself if and how it needs to prepare the provided data before encoding it.

```
class JsonEncoder implements EncoderInterface
{
    public function encode($data)
    {
        $data = $this->prepareData($data);

        return json_encode($data);
    }

    private function prepareData($data)
    {
        ...
```

```
        return $data;
    }
}
```

In this scenario the `prepareData()` method is a private method. It is not part of the public interface of format-specific encoders, because it will only be used internally. The `GenericEncoder` is not supposed to call it anymore. We only have to remove it from the `EncoderInterface`, which now exposes a very clean API:

```
interface EncoderInterface
{
    public function encode($data);
}
```

Conclusion

The `GenericEncoder` we started with at the beginning of this chapter was quite specific. Everything was hard-coded, so it was impossible to change its behavior without modifying it. We first moved out the responsibility of creating the format-specific encoders to an encoder factory. Next we applied a bit of dependency inversion by introducing an interface for the encoder factory. Finally we made the encoder factory completely dynamic: we allowed new format-specific encoder factories to be injected from the outside, i.e. without modifying the code of the encoder factory itself.

Finally we made the `GenericEncoder` *actually* generic. When we want to add support for another format we don't need to modify its code anymore. We only need to inject another callable in the encoder factory. This makes both classes (`GenericEncoder` and `EncoderFactory`) *open for extension* and *closed for modification.*

Recognizing classes that violate the *Open/closed principle*

This is a list of characteristics of classes that may not be open for extension:

- The class has conditions to determine a strategy.
- Conditions using the same variables or constants are recurring inside the class or related classes.
- It contains hard-coded references to other classes or class names.
- Inside the class objects are being created using the `new` operator.

Packages need classes that are open for extension

Applying the *Open/closed principle* to every class in your project will greatly benefit the implementation of future requirements (or changed requirements) for that project. When the behavior of a class can be changed from the outside, without modifying its code, people will feel safe to do so. They won't need to be afraid that they will break something. They won't even need to modify existing unit tests for the class.

When it comes to packages, the *Open/closed principle* is important for another reason. A package will be used in many different projects and in many different circumstances. This means that the classes in a package should not be too specific and leave room for the details to be implemented in different ways. And when behavior *has* to be specific (at some point a package has to be opinionated about something), it should be possible to change that behavior without actually modifying the code. Especially since most of the time that code can not be changed by its users without cloning and maintaining the entire package themselves.

This is why the *Open/closed principle* is highly useful and should be applied widely and generously when you are designing classes that are bound to end up in a reusable package.

The Liskov substitution principle

The *Liskov substitution principle* can be stated as:

> Derived classes must be substitutable for their base classes.

The funny thing about this principle is that it has the name of a person in it: Liskov. This is because the principle was first stated (in quite different wordings) by Barbara Liskov. But otherwise, there are no surprises here; no big conceptual leaps. It seems only logical that derived classes, or "subclasses" as they are usually called, should be substitutable for their base, or "parent" classes. Of course there's more to it. This principle is not just a statement of the obvious.

Dissecting the principle, we recognize two conceptual parts. First it's about derived classes and base classes. Then it's about being substitutable.

The good thing is, we already know from experience what a *derived class* is: it's a class that extends some other class: the *base class*. Depending on the programming language you work with, a base class can be either a concrete class, an abstract class or an interface. If the base class is a *concrete class*, it has no "missing" (also known as *virtual*) methods. In this case a derived class, or subclass, overrides one or more of the methods that are already implemented in the parent class. On the other hand if the base class is an *abstract class*, there are one or more *pure virtual methods*, which have to be implemented by the derived class. Finally, if *all* of the methods of a base class are pure virtual methods (i.e. they only have a signature and no body), then generally the base class is called an *interface*.

```php
/**
 * A concrete class, all methods are implementated, but can be
 * overridden by derived classes
 */
class ConcreteClass
{
    public function implementedMethod()
    {
        ...
    }
}

/**
 * An abstract class: some methods need to be implemented by derived
 * classes
 */
abstract class AbstractClass
{
    abstract public function abstractMethod();

    public function implementedMethod()
    {

    }
}

/**
 * An interface: all methods need to be implemented by derived
 * classes
 */
interface AnInterface
{
    public function abstractMethod();
}
```

Now we know all about base classes and derived classes. But what does it mean for derived classes to be *substitutable*? There is plenty of room for discussion it seems. In general, being substitutable is about *behaving well* as a subclass or a class implementing an interface. "Behaving well" would then mean behaving "as expected" or "as agreed upon".

Bringing the two concepts together, the *Liskov substitution principle* says that if we create a class which extends another class or implements an interface, it has to behave as expected.

Words like "behaving as expected" are still pretty vague though. This is why pointing out violations of the *Liskov substitution principle* can be pretty hard. Amongst developers there may even be disagreement about whether or not something counts as a violation of the principle. Sometimes it's a matter of taste. And sometimes it depends on the programming language itself and the constructs it offers for object-oriented programming.

Nevertheless we can point out some general bad practices which can prevent classes from being good substitutes for their parent classes or from being good implementations of an interface. So even though the principle itself is stated in a positive way, what follows is a discussion of some recurring violations of the principle. This will give you an idea of what it means to behave *badly* as a substitute for a class or an interface. This will indirectly help you to form an idea about how to behave *well* as a derived class.

Violation: a derived class does not have an implementation for all methods

When a class does not have a proper implementation for all the methods of its parent class (or its interface for that matter), this results in a clear violation of the *Liskov substitution principle*. It is bad behavior of substitutes to not do everything they are supposed to do. Consider for instance this `FileInterface`:

```
interface FileInterface
{
    public function rename($name);

    public function changeOwner($user, $group);
}
```

It may seem obvious that a file always has a name and an owner and that both can be changed. But you may also imagine that for some files, changing the owner would not be possible at all. Take for instance files that are stored using a cloud storage provider like Dropbox. If we create a Dropbox implementation of the `FileInterface`, we have to prevent users from trying to change the owner of a file because that simply doesn't work for a Dropbox file:

```
class DropboxFile implements FileInterface
{
    public function rename($name)
    {
        ...
    }

    public function changeOwner($user, $group)
    {
        throw new BadMethodCallException(
            'Not implemented for Dropbox files'
        );
    }
}
```

Throwing exceptions like this should be considered bad behavior for substitutes: when someone calls the `changeOwner()` method of `DropboxFile`, their entire application might crash, without any warning.

Still, we don't want a user to call `FileInterface::changeOwner()` when the class they use is `DropboxFile`. So maybe we can ask the user to check that, using a simple conditional like this:

```
if (!($file instanceof DropboxFile)) {
    $file->changeOwner(...);
}
```

This will prevent the nasty exception inside `changeOwner()` from being thrown. Unfortunately this is not a viable solution. Most likely these lines will be repeated all over the user's codebase, quickly becoming a maintenance burden for them.

Instead of throwing an exception we might just be secretive about the fact that we can't change the owner of a `DropboxFile`. Implementing this simple solution might be very tempting:

```
class DropboxFile implements FileInterface
{
    ...

    public function changeOwner($user, $group)
    {
        // shhh... this is not supported, but who needs to know?
    }
}
```

Unfortunately, we can't do this: changing the owner of a file is a *significant operation*. It's about security after all. Some other parts of the system may count on `DropboxFile::changeOwner()` to *really* change the owner of the file, for instance to make it unavailable for a previous owner. If for some reason this is not possible for a given type of file, it should be clear by its contract. In other words: its interface should not offer methods that make it *seem* as if this is possible.

The best solution would be to split the interface (see also *The interface segregation principle*).

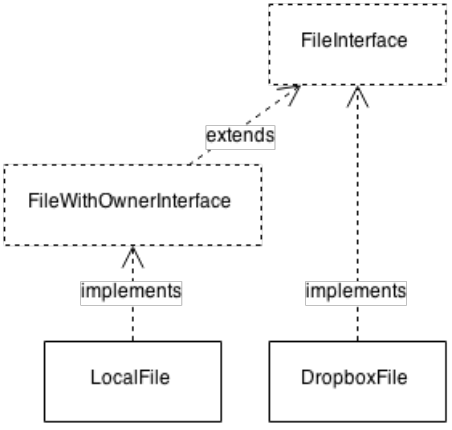
```
interface FileInterface
{
    public function rename($name);
}

interface FileWithOwnerInterface extends FileInterface
{
    public function changeOwner($user, $group);
}
```

Together these interfaces form a hierarchy of file types. There is the generic file type defined by `FileInterface` (which only offers a method for it to be renamed). Then there is a subtype of files of which the owner can be changed. When we have defined these interfaces the `DropboxFile` class would implement only the generic `FileInterface`:

```
class DropboxFile implements FileInterface
{
    public function rename($name)
    {
        ...
    }
}
```

And any other file type which supports a change of ownership, like `LocalFile`, implements `FileWithOwnerInterface`:

```
class LocalFile implements FileWithOwnerInterface
{
    public function rename($name)
    {
        ...
    }

    public function changeOwner($user, $group)
    {
        ...
    }
}
```



The new hierarchy of file classes

Finally we made the code adhere to the *Liskov substitution principle* principle again. Each of the derived classes (`DropboxFile`, `LocalFile`) now behave well as substitutes for their base class (`FileInterface`, `FileWithOwnerInterface`): all of the methods of the base classes are properly implemented in the derived classes.

Violation: different substitutes return things of different types

This violation applies in particular to programming languages that are not strictly typed, like PHP. These languages allow for a lot of uncertainty with regard to the type of, for instance, return values.

If a programming language has no way to pin down the type of the return value of a method, a common solution is to just mention it inside the docblock of the method, like this:

```
interface RouterInterface
{
    /**
     * @return Route[]
     */
    public function getRoutes();

    ...
}
```

The `getRoutes()` method is supposed to return something iterable (hence the `[]`) containing `Route` objects. But different router implementations may return different types of iterable things, like arrays or (in PHP) an object that implements `Traversable`[6]. For instance, the `SimpleRouter` returns a simple array of `Route` objects:

[6] http://php.net/traversable

```
class SimpleRouter implements RouterInterface
{
    public function getRoutes()
    {
        $routes = [];

        // add Route objects to $routes
        $routes[] = ...;

        return $routes;
    }
}
```

But the `AdvancedRouter` returns a much more advanced `RouteCollection` object, which implements `Traversable` (actually, it implements `Iterator`[7], which itself implements `Traversable`):

```
class AdvancedRouter implements RouterInterface
{
    public function getRoutes()
    {
        $routeCollection = new RouteCollection();

        ...

        return $routeCollection;
    }
}

class RouteCollection implements Iterator
{
    ...
}
```

[7] http://php.net/iterator

Now `AdvancedRouter` and `SimpleRouter` look like good substitutes for `RouterInterface`, but in reality they are not. Even though both classes implement the `getRoutes()` method, they both return a value of a different type.

This violation of the *Liskov substitution principle* may go unnoticed for a while, when people only iterate over the return value of `getRoutes()` using a simple `foreach` loop:

```
// $router implements RouterInterface, so $routes is iterable
$routes = $router->getRoutes();

foreach ($routes as $route) {
    // $route is a Route object
}
```

This is bound to work in all situations, because `foreach` loops over the values in an array as well as over the values provided by an iterator. But since many things iterable (or at least all arrays) are also countable, one day someone may try to do this:

```
if (count($routes) > 10) {
    ...
}
```

Using the `SimpleRouter` this will work, but using the `AdvancedRouter` this won't work, since the `RouteCollection` does not implement `Countable`[8]. So it becomes clear that there is a problem with the relation between parent classes and their derived classes.

Contrary to the previous violation that we discussed, the problem is not that `SimpleRouter` and `AdvancedRouter` are bad substitutes for `RouterInterface`. The real problem is the ambiguously defined return type of its `getRoutes()` method: `Route[]`.

The solution to the problem is to define the type of the return value more strictly and to not allow for accidental deviations from the expected type. So interfaces and

[8] http://php.net/countable

abstract classes should always document their return value in a strict way, using specific types[9]:

```
/**
 * @return array<Route>
 */
```

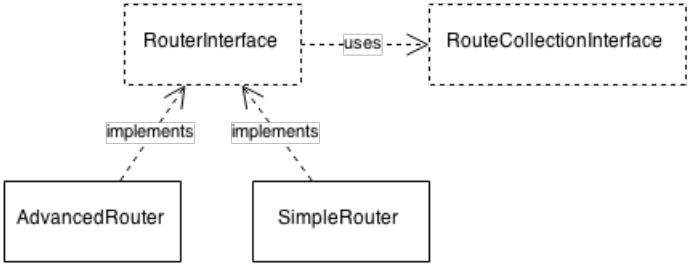
However, when we use the `array` type-hint, this still leaves room for some questions. Arrays are pretty vague data structures. The implementer of this interface might wonder what type of keys they should use: integers or strings. And what are the expected values for those?

Because we still have this ambiguity, preferably we would introduce a new interface to make sure that there can be no doubt:

```
interface RouterInterface
{
    /**
     * @return RouteCollectionInterface
     */
    public function getRoutes();
}

interface RouteCollectionInterface extends Iterator, Countable
{
}
```

With the introduction of this new interface for collections of routes, it will be much easier for the derived classes (i.e. `SimpleRouter` and `AdvancedRouter`) to behave well as substitutes for their base class (i.e. `RouterInterface`): now it's clear what their `getRoutes()` method is supposed to return.

[9] http://phpdoc.org/docs/latest/references/phpdoc/types.html



The new dependency diagram

More specific return types *are* allowed

The *Liskov substitution principle* does not allow for wrong or unspecific return types. Still, derived classes are allowed to return values which are *a subtype* of the type prescribed by the base class.

Consider the return type `RouteCollectionInterface`: any value which is an object that implements this interface will suffice as a proper return value:

```
class AnyRouteCollectionClass
    implements RouteCollectionInterface
{
    ...
}
```

Any route collection class is a derived class of `RouteCollectionInterface`, hence allowed as a return value. But the same goes for any class that *extends* such a route collection class, because the extending class is also supposed to be a well-behaving substitute for `CollectionInterface`.

Violation: a derived class is less permissive with regard to method arguments

As we saw earlier: to be a good substitute means to implement all the required methods and make them return the right things, according to the contract of the base class. When it comes to method arguments, a substitute needs to be *equally or more permissive* than that contract defines.

What does it mean for a method to be "more or less permissive" about its method arguments? Well, let's take a look at a so-called "mass mailer". Its interface says it should have a single method: `sendMail()`:

```
interface MassMailerInterface
{
    public function sendMail(
        TransportInterface $ransport,
        MessageInterface $message,
        RecipientsInterface $recipients
    );
}
```

Derived classes of this interface (i.e. base class) should use the provided mail transport to send a message to all recipients at once. The `TransportInterface` hides the messy details of how the message should be physically sent to the recipients. For instance, there may be implementations of `TransportInterface` that use `sendmail`, SMTP or PHP's built-in `mail()` function to deliver mails.

Below you will find a partial implementation of the `MassMailerInterface` which uses SMTP to send an email to lots of recipients at once. The first thing it does is verify that the user has provided the right type of argument for `$transport` (after all, this class only works with SMTP, so it needs an SMTP transport):

```
class SmtpMassMailer implements MassMailerInterface
{
    public function sendMail(
        TransportInterface $transport,
        MessageInterface $message,
        RecipientsInterface $recipients
    ) {
        if (!($transport instanceof SmtpTransport)) {
            throw new InvalidArgumentException(
                'SmtpMassMailer only works with SMTP'
            );
        }

        ...
    }
}
```

By restricting the set of allowed arguments like this, the `SmtpMassMailer` violates the *Liskov substitution principle*. As a substitute of the base class `MassMailerInterface` it's supposed to work with *any* mail transport, as long as it's an object of type `TransportInterface`. Instead, `SmtpMassMailer` is *less permissive* with regard to method arguments than the base class. This is bad substitute behavior.

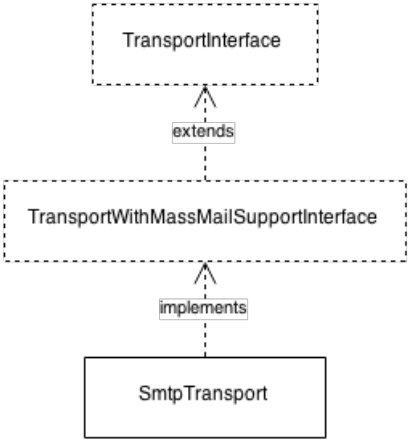
The only way to fix this is to make sure that the contract of the base class better reflects the needs of derived classes. Apparently `TransportInterface` as a type-hint for `$transport` is not sufficiently specific because it turns out that not every kind of mail transport is suitable for mass mailing.

Whenever we reason about class design like this, we need to keep an eye on phrases like:

> not every ... is a ...
>
> not every ... can be used as a ...

They usually indicate that there is something wrong with the type hierarchy of our classes. In this particular situation, our base classes/interfaces need to reflect that

there are different kinds of mail transports. Redefining our class hierarchy, we might define a generic `TransportInterface` and one specialized `TransportWithMassMailSupportInterface` which extends `TransportInterface`. `SmtpTransport` should then implement `TransportWithMassMailSupportInterface` and the other transports merely implement `TransportInterface`.



The new class diagram

```
class SmtpTransport implements TransportWithMassMailSupportInterface
{
    ...
}
```

Finally we can change the expected type of the `$transport` argument to `TransportWithMassMailSupportInterface` to prevent the wrong type of transport from being provided to the `sendMail()` method:

```
interface MassMailerInterface
{
    public function sendMail(
        TransportWithMassMailSupportInterface $transport,
        MessageInterface $message,
        RecipientsInterface $recipients
    );
}
```

Then we can modify `SmtpMassMailer` and remove the extra check on the type of the provided `$transport` argument:

```
class SmtpMassMailer implements MassMailerInterface
{
    public function sendMail(
        TransportWithMassMailSupportInterface $transport,
        MessageInterface $message,
        RecipientsInterface $recipients
    ) {
        /*
         * No need to validate $transport anymore, it supports
         * mass mailing
         */

        ...
    }
}
```

Finally, `SmtpMassMailer` adheres to the *Liskov substitution principle*. It behaves well as a substitute because it does not put more restrictions on the input arguments than its base class (`MassMailerInterface`) does.

Depending on your particular situation, it may not be justifiable to introduce this extra layer of abstraction. Maybe you are trying to redefine things in an abstract way which are really *just concrete things*. Or maybe you are trying to find similarities between things that can't be found because they don't exist. For example, it might

be impossible to use any other transport for mass mailing than the SMTP transport. This means there can't be any other mass mailer than an SMTP mass mailer. Then we could just as well define everything as a concrete class:

```
class SmtpMassMailer
{
    public function sendMail(
        SmtpTransport $transport,
        MessageInterface $message,
        RecipientsInterface $recipients
    ) {
        ...
    }
}
```

Since `SmtpMassMailer` is not derived from a base class, it doesn't violate the *Liskov substitution principle* anymore.

Violation: secretly programming a more specific type

Base classes like interfaces are used to expose an explicit public API. For instance, the public API of the `HttpKernelInterface` below consists of just one method which is by definition public:

```
interface HttpKernelInterface
{
    /**
     * @return Response
     */
    public function handle(Request $request);
}
```

Sometimes derived classes have additional public methods. These methods constitute its *implicit* public API:

```
class HttpKernel implements HttpKernelInterface
{
    public function handle(Request $request)
    {
        ...
    }

    public function getEnvironment()
    {
        ...
    }
}
```

The `getEnvironment()` method is not defined in the `HttpKernelInterface`. So whenever you want to use this method, you have to explicitly depend on the `HttpKernel` class, instead of the interface, like `CachedHttpKernel` does. It wraps an `HttpKernel` instance and adds some additional HTTP caching functionality to it:

```
class CachedHttpKernel implements HttpKernelInterface
{
    public function __construct(HttpKernel $kernel)
    {
        if ($kernel->getEnvironment() === 'dev') {
            ...
        }
    }

    public function handle(Request $request)
    {
        ...
    }
}
```

As the creator of the `CachedHttpKernel` we might want to make it a bit more generic by allowing users to wrap *any* instance of `HttpKernelInterface`. This requires just a simple modification to the constructor of the class:

```
class CachedHttpKernel implements HttpKernelInterface
{
    public function __construct(HttpKernelInterface $kernel)
    {
        if ($kernel->getEnvironment() === 'dev') {
            ...
        }
    }

    ...
}
```

You may have spotted the problem already: we still use the `getEnvironment()` method which was legitimate when the `$kernel` was guaranteed to be an instance of `HttpKernel`, yet we can't be sure anymore: now we only know that `$kernel` is an instance of `HttpKernelInterface`.

It might still be an instance of `HttpKernel` and since this is a PHP example, the code would run perfectly well in that case. The validity of the code is only determined at runtime, so even though the types don't match, we might still be able to call the `getEnvironment()` method on the `$kernel`, if it exists.

So the `CachedKernel` pretends to be part of a nice hierarchy of substitutable classes, while in fact it isn't. It breaks the tradition of implementing and requiring just the `handle()` method of `KernelInterface` and thereby it violates the *Liskov substitution principle*.

The solution to this problem is to be careful about respecting the contracts of the base class. For example, we could expand the interface to contain the required `getEnvironment()` method:

```
interface KernelInterface
{
    public function handle(Request $request);

    public function getEnvironment();
}
```

Or we could split the interface, just like we did on several previous occasions:

```
interface HttpKernelInterface
{
    public function handle(Request $request);
}

interface HttpKernelWithEnvironmentInterface
    extends HttpKernelInterface
{
    public function getEnvironment();
}
```

Then you can require more specific types of objects:

```
class CachedHttpKernel implements HttpKernelInterface
{
    public function __construct(
        HttpKernelWithEnvironmentInterface $kernel
    ) {
        ...
    }
}
```

As a last resort, you may verify that the actual argument implements the desired interface before calling its `getEnvironment()` method:

```
class CachedHttpKernel implements HttpKernelInterface
{
    public function __construct(HttpKernelInterface $kernel)
    {
        if ($kernel instanceof HttpKernelWithEnvironmentInterface) {
            $environment = $kernel->getEnvironment();
            ...
        }
    }
}
```

Besides being an ad-hoc solution, it feels a lot like there is a missed opportunity for polymorphism here.

Conclusion

The *Liskov substitution principle* demands from derived classes that they are good substitutes. We discussed some examples of being a bad substitute. Based on these negative examples of bad substitutes, we can form an idea of what "being a good substitute" would mean. A good substitute:

- Provides an implementation for all the methods of the base class.
- Returns the type of things the base class prescribes.
- Doesn't put extra constraints on arguments for methods.
- Doesn't make use of non-strict typing to break the contract that was provided by the base class.

The Interface segregation principle

The fourth SOLID principle is the *Interface segregation principle*. It gives us the following instruction:

> Make fine-grained interfaces that are client specific.

"Fine-grained interfaces" stands for interfaces with a small amount of methods. "Client specific" means that interfaces should define methods that make sense from the point of view of the client that *uses* the interface.

In order to reach an understanding of this principle we will, just like in the previous chapter, discuss some common violations of it. Each violation is followed by a change in the code that would fix the problem.

Violation: leaky abstractions

Let's reconsider an example from the chapter about the *Liskov substitution principle*, the `FileInterface`:

```
interface FileInterface
{
    public function rename($name);

    public function changeOwner($user, $group);
}
```

This interface serves as an abstract representation of a file, on any filesystem or machine imaginable: local or remote, physical or in-memory, etc. The `FileInterface`

tries to define some common ground between different kinds of underlying *real* files. Each file can at least be renamed and its owner can be changed, no matter what filesystem is used.

According to the *Liskov substitution principle* each class that implements `FileInterface` should be a good substitute for that interface. You should be able to use each of them in the same way. Therefore if a class implements `FileInterface` it should implement all of the methods defined in that interface to accomplish the desired behavior on the respective filesystem.

Take for example the `DropboxFile` class which contains the implementation details for storing files in a Dropbox[10] folder (Dropbox is a cloud storage service):

```
class DropboxFile implements FileInterface
{
    public function rename($name)
    {
        /*
         * Make a call to the Dropbox API to change the name
         * of this file
         */

        ...
    }

    public function changeOwner($user, $group)
    {
        // irrelevant for Dropbox files, so no implementation
    }
}
```

This Dropbox implementation of `FileInterface` obviously violates the *Liskov substitution principle* (as we already discussed). It's not a proper subclass of `FileInterface` since not every method is implemented. When called, the `changeOwner()` method does not do what we expect from it.

[10] https://www.dropbox.com/

Dropbox as a storage platform is not to blame for this. You can store files in a Dropbox folder, you can rename them, but you just can't change their owner. So instead we have to conclude that something is wrong with the `FileInterface`: we initially thought that a change of ownership would be possible for all kinds of files, stored by any means possible. But this appears to be a false belief. This means that the `FileInterface` is an improper generalization of the "file" concept. Such an improper generalization is usually called a "leaky abstraction".

Leaky abstractions is something I first learned about when reading Joel Spolsky's article on leaky abstractions[11]. In it he mentions the *Law of leaky abstractions*:

> All non-trivial abstractions, to some degree, are leaky.

As programmers we are looking for *abstractions* all day. We want to treat a specific thing as a more general thing. When we do this consistently, we can later fearlessly replace any specific thing with some other specific thing. The system will not fall apart because every part of it depends only on general, abstract things (see also the next chapter about the *Dependency inversion principle*.

The problem with most (all?) abstractions, as the Law of Leaky Abstractions says, is that they are *leaky*, which means that it will never be possible to abstract away every underlying specificness. In the example of the `FileInterface`, it became clear that it is not truly a general property of any type of file that its owner can be changed. Thus the `FileInterface` should apparently be called a "leaky abstraction".

Refactoring: create separate interfaces and use multiple inheritance

To solve this problem there are several solutions. The obvious one would be to enhance the `FileInterface`. We could add a method that can be used to check if the "change ownership" behavior is supported by a particular type of file:

[11] http://www.joelonsoftware.com/articles/LeakyAbstractions.html

```
interface FileInterface
{
    /**
     * @return boolean Whether or not the owner can be changed
     */
    public function canChangeOwner();

    public function changeOwner($user, $group);
}
```

Calling `canChangeOwner()` on a specific implementation of `FileInterface` that does not support a change of ownership should return `false`. And when a user would nevertheless call `changeOwner()`, an exception should be thrown.

This solution would require all subclasses of `FileInterface` to implement the same method, just for the sake of some classes that don't support a change of ownership. This is not desirable. Besides, every *client* of this interface needs to call `canChangeOwner()` first, before it may call `changeOwner()`. This is called "temporal coupling" and is a design smell. It will result in a lot of code duplication (and a big maintenance burden) and hard-to-spot bugs.

A better option is to split the `FileInterface` into more fine-grained interfaces:

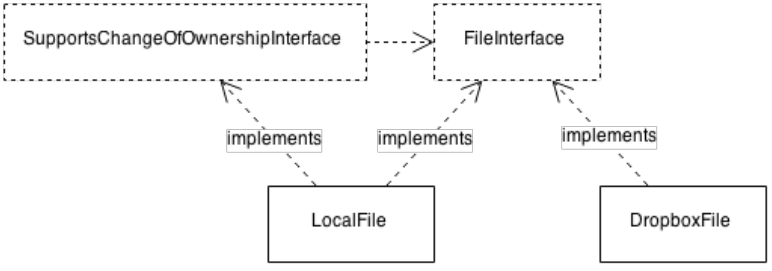
```
interface FileInterface
{
    public function rename($name);
}

interface SupportsChangeOfOwnershipInterface
{
    public function changeOwner($user, $group);
}
```

The regular `LocalFile` class implements both interfaces:

```
class LocalFile
    implements FileInterface, SupportsChangeOfOwnershipInterface
{
    public function rename($name)
    {
        ...
    }

    public function changeOwner($user, $group)
    {
        ...
    }
}
```

While the `DropboxFile` class implements just one of them:

```
class DropboxFile implements FileInterface
{
    public function rename($name)
    {
        ...
    }
}
```



File class hierarchy

Splitting the original `FileInterface` (or "segregating" it, as the name of the *Interface segregation principle* calls it) has resulted in two finer grained interfaces.

What did we do?

By segregating the original interface we have made more fine-grained interfaces. We were somewhat forced to do this because we learned something about our "files" domain. We discovered that you can't change the owner of every imaginable type of file.

The resulting *fine-grained* interfaces are now also more *client-specific*, as we want them to be. Consider the `FileImporter` class:

```
class FileImporter
{
    public function import(FileInterface $file)
    {
        ...

        $file->rename(...);
    }
}
```

The `FileImporter` class is a *client* of `FileInterface`. It doesn't need a way to change the owner of the file, it just needs to be able to rename that file. Hence, it accepts any object which is an instance of the more general `FileInterface`.

Now consider the `FilePermissionManager` class:

```
class FilePermissionManager
{
    public function transferOwnership(
        SupportsChangeOfOwnershipInterface $file,
        $newOwner,
        $newGroup
    ) {
        ...

        $file->changeOwner($newOwner, $newGroup);
    }
}
```

The `FilePermissionManager` class is another client. It needs just a way to change the ownership of the file. Hence, it accepts objects which implement `SupportsChangeOfOwnershipInterface` since that interface has just the right method to fulfill this task.

Violation: multiple use cases

Sometimes the interface of a class (i.e. its public API) contains too many methods not because of a leaky abstraction but because it serves multiple use cases. Some clients of the object will call a different set of methods than other clients of the same object.

Almost every existing *service container* implementation serves as a great example of a class that has different clients, since many service containers are used both as a dependency injection (or inversion of control) container *and* as a service locator.

A service container is an object you use to retrieve other objects (i.e. *services*):

```
interface ServiceContainerInterface
{
    public function get($name);

    ...
}

// $serviceContainer is an instance of ServiceContainerInterface
$mailer = $serviceContainer->get('mailer');
```

The `mailer` service will return a fully initialized object that can be used as a mailer. This allows for lazy loading of services. Because the service container locates services for you, a service container is also called a "service locator" (read more about why using a service locator is not a good idea in most cases in this article by Paul Jones[12]).

Before you can retrieve a service from a service container, some other part of the system should configure it correctly. The container should be instructed how to initialize services like the `mailer` service. This is the aspect of a service container that makes it a *dependency injection* container:

[12] http://paul-m-jones.com/archives/4792

```
interface ServiceContainerInterface
{
    public function get($name);

    public function set($name, callable $factory);
}

// $serviceContainer is an instance of ServiceContainerInterface

// configure the mailer service, which requires a transport service
$serviceContainer->set(
    'mailer',
    function () use ($serviceContainer) {
        return new Mailer(
            $serviceContainer->get('mailer.transport')
        );
    }
);

// configure the mailer transport service
$serviceContainer->set(
    'mailer.transport',
    function () use ($serviceContainer) {
        return new MailerSmtpTransport();
    }
);
```

Other parts of the application don't need to worry anymore about how they should instantiate and initialize the `mailer` service - the creation logic is all handled by something else (this is why it is called *inversion of control*).

What's interesting is that the use case of configuring the service container (i.e using it as an inversion of control container) is entirely different from the use case of fetching services from the service container (i.e. using it as a service locator). Still, both use cases are implemented in any service container class since both `get()` and `set()` methods are defined in the `ServiceContainerInterface`.

This means that any client which depends on the `ServiceContainerInterface` can both *fetch* previously defined services and *define* new services. In reality most clients of the `ServiceContainerInterface` only perform one of these tasks. A client either configures the service container (for example when the application is bootstrapped) *or* fetches a service from it (when the application is up and running).

When an interface tries to serve several types of clients at once, like the `ServiceContainerInterface` does, it violates the *Interface segregation principle*. Such an interface is not fine-grained enough to be client-specific.

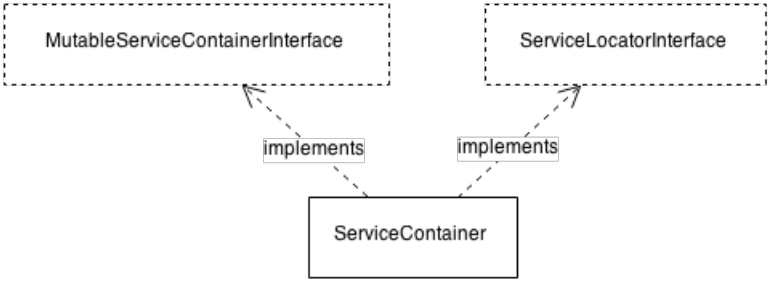
Refactoring: separate interfaces and multiple inheritance

One type of client is the part of the application which bootstraps the service container by configuring the available services. Such a client would only need the part of the `ServiceContainerInterface` that makes it mutable, i.e. its `set()` method. Another type of client is for instance a controller which fetches a service to process a request. This type of client only needs `get()`, *not* `set()`. The difference between clients should be reflected in the interfaces that are available, for instance by splitting the interface into a `MutableServiceContainerInterface` and a `ServiceLocatorInterface`.

```
interface MutableServiceContainerInterface
{
    public function set($name, callable $factory);
}

interface ServiceLocatorInterface
{
    public function get($name);
}
```

Now each client can require its own appropriate type of service container. In practice there would be a single `ServiceContainer` class, which serves both types of clients at the same time by implementing both `MutableServiceContainerInterface` *and* `ServiceLocatorInterface`:

```
class ServiceContainer implements
    MutableServiceContainerInterface
    ServiceLocatorInterface
{
    public function set($name, callable $factory)
    {
        ...
    }

    public function get($name)
    {
        ...
    }
}
```



`ServiceContainer` class hierarchy

None of the clients need to be bothered by this, since none of them will depend on the `ServiceContainer` class, only on one of the interfaces:

```
class Kernel
{
    public function initializeServiceContainer(
        MutableServiceContainerInterface $serviceContainer
    ) {
        $serviceContainer->set(...);
    }
}

class SomeController
{
    private $serviceLocator;

    public function __construct(
        ServiceLocatorInterface $serviceLocator
    ) {
        $this->serviceLocator = $serviceLocator;
    }

    public function indexAction()
    {
        $mailer = $this->serviceLocator->get('mailer');

        ...
    }
}
```

Having the `ServiceContainer` class implement both interfaces is not strictly necessary. It will just make it easier for you to maintain the code. The point to take from this is: it doesn't matter if a *class* does not strictly follow the *Interface segregation principle*. That's no problem, as long as all parts of the application depend only on one small, client-specific part of the public API of that class. To enable clients to do this, make sure that you always offer an interface. Then split that interface whenever you notice that different clients tend to use a different subset of its methods.

Container Interoperability

An interesting project to mention here is the Container Interoperability project[13] which tries to define common interfaces for service containers within the PHP ecosystem. They follow the same reasoning as we did here: the two separate use cases of configuring the container and fetching entries from it should be reflected by separate interfaces.

Violation: no interface, just a class

Say you are working on a package called `FabulousORM`, which is supposed to contain a better object-relational mapper than any of the existing ones. You define an `EntityManager` class which can be used to persist entities (objects) in a relational database. It uses a unit of work[14] to calculate the actual changes that need to be made to the database. The `EntityManager` class has some public methods (`persist()` and `flush()`) and one private method which internally makes the `UnitOfWork` object available to other methods:

```
class EntityManager
{
    public function persist($entity)
    {
        ...
    }

    public function flush()
    {
        ...
    }

    private function getUnitOfWork()
    {
```

[13] https://github.com/container-interop/container-interop

[14] http://martinfowler.com/eaaCatalog/unitOfWork.html

```
        ...
    }
}
```

People who use your package in their project can depend on the `EntityManager` in their own classes, like the `UserRepository` does:

```
class UserRepository
{
    public function __construct(EntityManager $entityManager)
    {
        ...
    }
}
```

Unfortunately we can't use an interface as the type-hint for `$entityManager`. The `EntityManager` class doesn't implement an interface. So the best we can do is to use the class as a type-hint.

Even though there is no *explicit* interface for the `EntityManager` class, it still has an *implicit* interface. Each method of the class comes with a certain scope (`public`, `protected` or `private`). When a client like `UserRepository` depends on the `EntityManager` class, it actually depends on all the public methods of `EntityManager`: `persist()` and `flush()`. None of the methods with a different scope (i.e. `protected` or `private`) can be called by a client. So the public methods combined form the *implicit interface* of `EntityManager`.

Implicit changes in the implicit interface

One day you decide to add a `Query` class to your ORM package. It can be used to query the database and retrieve entities from it. This `Query` class needs the `UnitOfWork` object that is used internally by `EntityManager`. So you decide to turn its private `getUnitOfWork()` method into a public method. That way, the `Query` class may depend on the `EntityManager` class and use its `getUnitOfWork()`:

```
class EntityManager
{
    ...

    /**
     * This method needs to be public because it's used by the
     * Query class
     */
    public function getUnitOfWork()
    {
        ...
    }
}

class Query
{
    public function __construct(EntityManager $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    public function someMethod()
    {
        $this->entityManager->getUnitOfWork()->...
    }
}
```

This new public method, `getUnitOfWork()` will automatically become part of the implicit interface* of `EntityManager`. From this moment on all clients of `EntityManager` implicitly depend on this method too, even though they may only need the `persist()` and `flush()` methods.

This is a dangerous situation. Maybe some clients start using the publicly available method `getUnitOfWork()` too. They may do some pretty dangerous things with the unit of work, which you would normally never authorize.

Adding methods to the implicit interface of a class is also bound to cause backwards compatibility problems. Say that one day you refactor the `Query` class and remove its dependency on the `EntityManager` class. Since none of your classes need the public `getUnitOfWork()` method anymore, you then decide to make that method private again. Suddenly all the clients which use the previously public `getUnitOfWork()` method will break.

Refactoring: add header and role interfaces

You can solve this problem by defining an interface for each use case that the `EntityManager` class provides. For example you may define the primary use case of "persisting entities" as the `PersistsEntitiesInterface`:

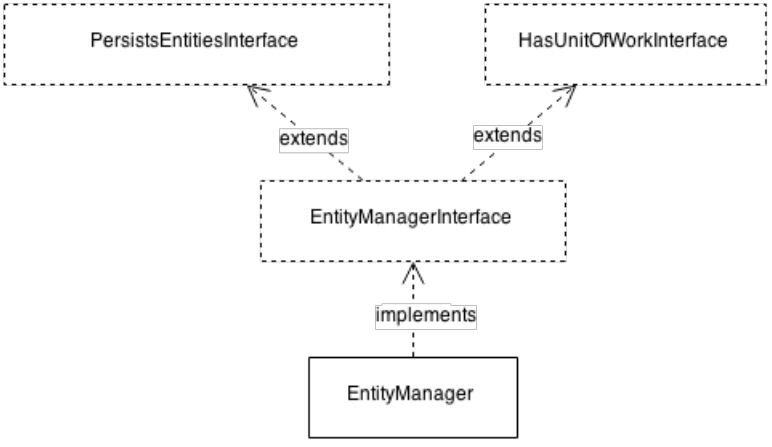
```
interface PersistsEntitiesInterface
{
    public function persist($entity);

    public function flush();
}
```

And you can introduce a second interface, `HasUnitOfWorkInterface`, to define a second use case:

```
interface HasUnitOfWorkInterface
{
    public function getUnitOfWork();
}
```

Then you can add a main interface which combines the two interfaces, and a class which implements the main interface:

```
interface EntityManagerInterface extends
    PersistsEntitiesInterface,
    HasUnitOfWorkInterface
{
}

class EntityManager implements EntityManagerInterface
{
    ...
}
```



`EntityManager` class hierarchy

Several of the interfaces we have just defined describe the *roles* that a class can play: the `PersistsEntitiesInterface` and the `HasUnitOfWorkInterface`. Then there is one interface which combines these roles and together constitutes a thing we know as an *entity manager*, which "can persist entities" and "has a unit of work": the `EntityManagerInterface`.

Martin Fowler calls these different types of interfaces role interfaces and header interfaces[15] respectively. You can determine role interfaces for a class by looking at the different clients which make use of the class. Then you group the methods which are used together in separate interfaces, like we did for the `EntityManager` class.

[15] http://martinfowler.com/bliki/RoleInterface.html

Header interfaces are usually the easiest to define, since:

> all you have to do is duplicate the public methods [of the class], no thought needed.

Often defining only a header interface is not enough, just like with the `EntityManager`: clients won't need any other public methods than `persist()` and `flush()`. If the header interface would contain some other public methods, like `getUnitOfWork()`, they would be superfluous. As Robert Martin puts it:

> Clients should not be forced to depend on methods they do not use.

So you always have to split an interface according to which methods will be used by different types of clients. This will have several advantages:

- An interface is a promise concerning the public API of your classes. Offering small interfaces will help you keep your promises, since the chance that they need to be changed will be much smaller.
- You are free to change the implementation of your class by removing or renaming public methods, as long as they are not defined in the interface.

As a class designer it is important to provide interfaces for your classes (header as well as role interfaces). This is because the last of the SOLID principles - the *Dependency inversion principle* - says that you should depend on abstractions, not on concretions. Classes are concrete, interfaces are abstract. So when you only offer classes and no interfaces, other people will not even be able to follow the *Dependency inversion principle*, since you force them to depend on concretions.

The Dependency inversion principle

The last of the SOLID principles of class design focuses on class dependencies. It tells you what kinds of things a class should depend upon:

> Depend on abstractions, not on concretions.

The name of this principle contains the word "inversion", from which we may infer that without following this principle we usually depend on concretions, not on abstractions. The principles tells us to invert that direction: we should always depend on abstractions.

Example of dependency inversion: the FizzBuzz generator

There is a well-known programming assignment which serves as a nice example of dependency inversion. It is called "FizzBuzz" and is often used as a little test to see if a candidate for a programming job could manage to implement a set of requirements, usually on the spot. The requirements are these:

- Generate a list of integers, from `1` to `n`.
- Numbers that are divisible by `3` should be replaced with "Fizz".
- Numbers that are divisible by `5` should be replaced with "Buzz".
- Numbers that are both divisible by `3` *and* by `5` should be replaced with "FizzBuzz".

A straightforward implementation might look like this:

```
class FizzBuzz
{
    public static function generateList($limit)
    {
        $list = [];

        for ($number = 1; $number <= $limit; $number++) {
            $list[] = self::generateElement($number);
        }

        return $list;
    }

    private static function generateElement($number)
    {
        if ($number % 3 === 0 && $number % 5 === 0) {
            return 'FizzBuzz';
        }

        if ($number % 3 === 0) {
            return 'Fizz';
        }

        if ($number % 5 === 0) {
            return 'Buzz';
        }

        return $number;
    }
}
```

Only static methods are used since we don't make use of object properties. We could have used plain old functions as well.

Given the assignment this is a very *specific* implementation of the requirements. Reading through the code, we are able to find every aspect of the requirements in

one specific line: the requirements regarding the divisibility of the numbers and the requirement that the list of numbers starts at 1 and ends at the given number, etc.

When the candidate has produced some code like this, the interviewer adds another requirement:

> It should be possible to add another rule, without modifying the `FizzBuzz` class.

Making the `FizzBuzz` class open for extension

Currently the `FizzBuzz` class is not open for extension nor closed for modification. If numbers divisible by 7 should one day be replaced by "Whizz", it is impossible to implement this change without actually modifying the code of the `FizzBuzz` class.

Pondering about the design of `FizzBuzz` and how we can make it more flexible, we note that the `generateElement()` method contains a lot of details. Within the same class the `generateList()` method is rather generic. It just generates a list of incrementing numbers, starting with 1 (which is somewhat specific), and ending with a given number. So `FizzBuzz` has two responsibilities: it generates lists of numbers, and it replaces certain numbers with something else, based on the FizzBuzz rules.

These FizzBuzz rules are liable to change. And the requirement is that when the rules change, we should not need to modify the `FizzBuzz` class itself. Let's apply some things that we learned in the chapter about the *Open closed principle*. For example, we can externalize the rules and use them like this:

```
class FizzBuzz
{
    public static function generateList($limit)
    {
        ...
    }

    private static function generateElement($number)
    {
        $fizzBuzzRule = new FizzBuzzRule();
```

```
        if ($fizzBuzzRule->matches($number)) {
            return $fizzBuzzRule->getReplacement();
        }

        $fizzRule = new FizzRule();
        if ($fizzRule->matches($number)) {
            return $fizzRule->getReplacement();
        }

        $buzzRule = new BuzzRule();
        if ($buzzRule->matches($number)) {
            return $buzzRule->getReplacement();
        }

        return $number;
    }
}
```

The details about the rules can be found in the specific rule classes. For example the `FizzRule` class would look like this:

```
class FizzRule
{
    public function matches($number)
    {
        return $number % 3 === 0;
    }

    public function getReplacement()
    {
        return 'Fizz';
    }
}
```

This is one step in the right direction. Even though the exact numbers (3, 5, 3 and 5) have been moved to the specific rule classes, the code in `generateElement()`

remains very specific. Adding a new rule would still require a modification of the `generateElement()` method, so we have not made the class open for extension yet.

Removing the specificness from the `FizzBuzz` class

We can remove this specificness from the `FizzBuzz` class by introducing an interface for the rule classes and allowing multiple rules to be injected to a `FizzBuzz` object:

```
interface RuleInterface
{
    public function matches($number);

    public function getReplacement();
}

class FizzBuzz
{
    private $rules = [];

    public function addRule(RuleInterface $rule)
    {
        $this->rules[] = $rule;
    }

    public function generateList($limit)
    {
        ...
    }

    private function generateElement($number)
    {
        foreach ($this->rules as $rule) {
            if ($rule->matches($number)) {
                return $rule->getReplacement();
            }
```

```
        }

        return $number;
    }
}
```

Now we need to make sure that every specific rule class implements the `RuleInterface` and then the `FizzBuzz` class can be used to generate lists of numbers with varying rules:

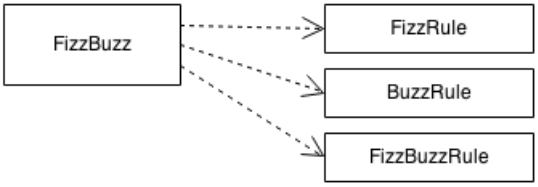
```
class FizzRule implements RuleInterface
{
    ...
}

$fizzBuzz = new FizzBuzz();
$fizzBuzz->addRule(new FizzBuzzRule());
$fizzBuzz->addRule(new FizzRule());
$fizzBuzz->addRule(new BuzzRule());
...

$list = $fizzBuzz->generateList(100);
```

Now we have a highly general piece of code, the `FizzBuzz` class, which "generates a list of numbers and replaces certain numbers based on a flexible set of rules".

There is no mention of "FizzBuzz" in that description and there is no mention of "Fizz" nor "Buzz" in the code of the `FizzBuzz` class. Actually, the `FizzBuzz` class should be renamed so that it better communicates its responsibility. Of course, naming things is one of the hardest parts of our job and `NumberListGenerator` isn't a particularly expressive name, but it would better describe its purpose than `FizzBuzz`.

Looking at the initial implementation of the `FizzBuzz` class it now becomes clear that the class already had an abstract task: it just generated a list of numbers. Only the rules were highly detailed (being divisible by 3, being divisible by 5, etc.). To use the words from the *Dependency inversion principle*, an abstraction depended on

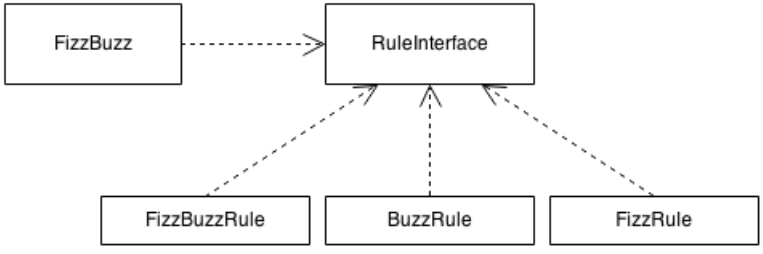
details. According to the principle the direction of this dependency is wrong and should have been inverted. It caused the `FizzBuzz` class to be closed for extension as it was impossible to add another rule, without modifying it.



`FizzBuzz` having concrete dependencies

By introducing the `RuleInterface` and adding specific rule classes that implement `RuleInterface` we fixed the dependency direction. The highly abstract `FizzBuzz` class now started to depend on other abstract things, called "rules". At the same time, the detailed things - the specific rule classes - depended on an abstract thing, the `RuleInterface`. And this is exactly the way things should be according to the *Dependency inversion principle*:

> Abstractions should not depend upon details. Details should depend upon abstractions.



`FizzBuzz` having abstract dependencies

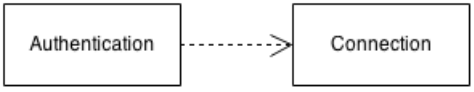
Now that we've seen the *Dependency inversion principle* in action we can take a look at some situations where it is clearly being violated. As in the previous chapters, each violation is followed by a suggested refactoring, which fixes the problems.

Violation: a high-level class depends upon a low-level class

The first violation arises from *mixing different levels of abstraction.* Consider the following `Authentication` class:

```
use Doctrine\DBAL\Connection;

class Authentication
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function checkCredentials($username, $password)
    {
        $user = $this->connection->fetchAssoc(
            'SELECT * FROM users WHERE username = ?',
            [$username]
        );

        if ($user === null) {
            throw new InvalidCredentialsException('User not found');
        }

        // validate password
        ...
    }
}
```

The `Authentication` class needs a database connection (in this case represented by a `Connection` object from the Doctrine DBAL library[16], which is based on PDO[17]). It uses the connection to retrieve the user data from the database.



The `Authentication` class depends on `Connection`

There are many problems with this approach. They can be articulated by answering the following two questions about this class:

> *Is it important for an authentication mechanism to know where exactly user data comes from?*
>
> Definitely not. The only thing the `Authentication` class really needs is user data, as an array or preferably an object representing a user. The *origin* of that data is irrelevant.

> *Is it possible to fetch user data from some other place than a database?*
>
> Currently it's impossible. The `Authentication` class requires a `Connection` object, which is a *database* connection. You can not use it to retrieve users from, for instance, a text file or from some external web service.

In conclusion, both the *Single responsibility principle* and the *Open/closed principle* have been violated in this class. The underlying reason is that the *Dependency inversion principle* has been violated too: the `Authentication` class itself is a *high-level abstraction*. Nevertheless it depends on a very *low-level concretion*: the database connection. This particular dependency makes it impossible for the `Authentication` class to fetch user data from any other place than the database.

In reality, all that the `Authentication` class would need is something which provides the user data - let's call that thing a "user provider". The `Authentication` class doesn't need to know anything about the actual process of fetching the user data (whether

[16] http://docs.doctrine-project.org/projects/doctrine-dbal/
[17] http://php.net/pdo

it originates from a database, a text file, an LDAP server, etc.). It only needs the user data.

It is a good thing for the `Authentication` class not to care about the origin of the user data itself. At once, the class becomes highly reusable. All the implementation details about fetching user data would be left out of that class. This will make it easy for users of the class to implement their own "user providers".

Refactoring: abstractions and concretions both depend on abstractions

Refactoring the high-level `Authentication` class to make it adhere to the *Dependency inversion principle* means first removing the dependency on the low-level `Connection` class. Then we add a higher-level dependency on something which provides the user data, the `UserProvider` class:

```
class Authentication
{
    private $userProvider;

    public function __construct(UserProvider $userProvider)
    {
        $this->userProvider = $userProvider;
    }

    public function checkCredentials($username, $password)
    {
        $user = $this->userProvider->findUser($username);

        if ($user === null) {
            throw new InvalidCredentialsException('User not found');
        }

        // validate password
        ...
```

```
    }
}

class UserProvider
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function findUser($username)
    {
        return $this->connection->fetchAssoc(
            'SELECT * FROM users WHERE username = ?',
            [$username]
        );
    }
}
```

The `Authentication` class has nothing to do with a database anymore. Instead, the `UserProvider` class does everything that is needed to fetch a user from the database.

`Authentication` depends on `UserProvider`

It's still not easy to switch between different user provider implementations. The `Authentication` class depends on the concrete `UserProvider` class. If anybody wants to fetch their user data from a text file, they'd have to extend this class and override its `findUser()` method:

```
class TextFileUserProvider extends UserProvider
{
    public function findUser($username)
    {
        ...
    }
}
```

They would thereby inherit any behavior that was implemented in the `UserProvider` class itself and that is not a desirable situation. The solution is to provide an interface for every class that wants to be a user provider:

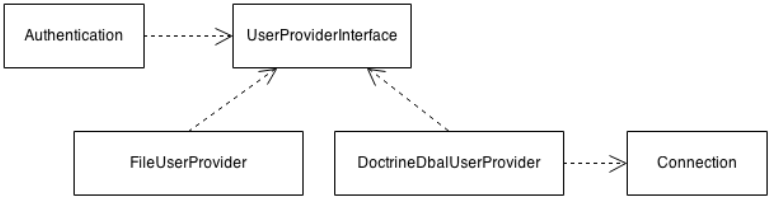
```
interface UserProviderInterface
{
    public function findUser($username);
}
```

Then every class implementing the `UserProviderInterface` can and should also have a more meaningful name, like `DoctrineDbalUserProvider`:

```
class DoctrineDbalUserProvider implements UserProviderInterface
{
    ...
}

class TextFileUserProvider implements UserProviderInterface
{
    ...
}
```

And of course we have to change the type-hint in the constructor of the `Authentication` class:

```
class Authentication
{
    private $userProvider;

    public function __construct(UserProviderInterface $userProvider)
    {
        $this->userProvider = $userProvider;
    }

    ...
}
```



`Authentication` depends on `UserProviderInterface`

As you can see in the dependency diagram, the high-level class `Authentication` does not depend on low-level, concrete classes like `Connection` anymore. Instead, it depends on another high-level, abstract thing: `UserProviderInterface`. Both are conceptually on more or less the same level. Lower-level operations like reading from a file, and fetching data from a database are performed by lower-level classes - the concrete user providers. Those user provider classes also depend on a high-level abstraction because they implement `UserProviderInterface`. This completely conforms to the *Dependency inversion principle* which states that:

> High level modules should not depend upon low level modules. Both should depend upon abstractions.

A nice side-effect of the changes we made is that the maintainability of the code has greatly improved. When a bug is found in one of the queries used for fetching user data from the database, there is no reason to modify the `Authentication` class anymore. The necessary changes will only occur inside the specific user provider,

in this case the `DoctrineDbalUserProvider`. This means that this refactoring has greatly reduced the chance that you will accidentally break the authentication mechanism itself.

Simply depending on an interface is not enough

The step from `UserProvider` to `UserProviderInterface` was an important one because it helps users of the `Authentication` class to easily switch between user provider implementations. But just adding an interface to a class is not always sufficient to fix all problems related to dependencies.

Consider this alternative version of the `UserProviderInterface`:

```
interface UserProviderInterface
{
    public function findUser($username);

    public function getTableName();
}
```

This is not at all a helpful interface. It is an immediate violation of the Liskov substitution principle. Not all classes which implement this interface will be able to be good substitutes: some of them won't be able to return a "table name". But more importantly: the `UserProviderInterface` mixes different levels of abstraction, and combines something high-level like "finding a user" with something low-level like "the name of a database table".

So even though we would introduce this inadequate interface to make the `Authentication` class depend on abstractions instead of concretions, this goal would not be reached. In fact the `Authentication` class still depends on something concrete and low-level, namely a user provider that is table-based.

Violation: a class depends upon a class from another package

In this section we'll discuss a common violation of the *Dependency inversion principle* which is especially relevant to package developers. Say a class needs some kind of a way to fire application-wide events. The usual solution for this is to use an event dispatcher (sometimes called "event manager"). The problem is: there are many event dispatchers available, and they all have a slightly different API. For instance, the Symfony `EventDispatcherInterface`[18] looks like this:

```
interface EventDispatcherInterface
{
    public function dispatch($eventName, Event $event = null);

    public function addListener(
        $eventName,
        $listener,
        $priority = 0
    );

    ...
}
```

Note that events are supposed to have a name, which is a string (e.g. "new_user"), and when firing (or "dispatching") the event you can provide an event object carrying additional contextual data. The event object will be enriched and used as the first argument when the event listener (which can be any PHP callable) is notified of the occurrence of an event:

[18] https://github.com/symfony/EventDispatcher/blob/master/EventDispatcherInterface.php

```
use Symfony\Component\EventDispatcher\Event;

class NewUserEvent extends Event
{
    private $user;

    public function __construct(User $user)
    {
        $this->user = $user;
    }

    public function getUser()
    {
        return $this->user;
    }
}

class EventListener
{
    public function onNewUser(NewUserEvent $event)
    {
        ...
    }
}
```

An event dispatcher from another framework, Laravel[19] looks like this:

[19] http://laravel.com/api/source-class-Illuminate.Events.Dispatcher.html

```
class Dispatcher
{
    public function listen($event, $listener, $priority = 0)
    {
        ...
    }
    public function fire($event, array $payload = [])
    {
        ...
    }

    ...
}
```

Note that it doesn't implement an interface. And instead of an event object, the contextual data for events (the "payload") consists of an array, which will be used as function arguments when a listener is notified of an event:

```
class EventListener
{
    public function onNewUser(User $user)
    {
        ...
    }
}

$dispatcher = new Dispatcher();
$dispatcher->listen('new_user', [new EventListener(), 'onNewUser']);

$user = new User();
$dispatcher->fire('new_user', [$user]);
```

It appears that you can do more or less the same things with both event dispatchers, i.e. fire events and listen to them. But the way you do it is quite different.

Now the package you are working on contains a `UserManager` class. Using this class you can register new users. Afterwards you want to dispatch an application-wide event so other parts of the application can respond to the fact that a new user now exists (for instance, maybe new users should receive a welcome email).

```
use Illuminate\Events\Dispatcher;

class UserManager
{
    public function register(User $user)
    {
        // persist the user data
        ...

        // fire an event: "new_user"
    }
}
```

Let's assume you want to use the package containing the `UserManager` class in a Laravel application. Laravel already provides an instance of the `Dispatcher` class as the `dispatcher` service in its Inversion of Control (IoC) container. This means you can inject it as a constructor argument of the `UserManager` class:

```
use Illuminate\Events\Dispatcher;

class UserManager
{
    private $dispatcher;

    public function __construct(Dispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function create(User $user)
```

```
    {
        ...

        $this->dispatcher->fire('new_user', ['user' => $user]);
    }
}
```

A couple of weeks later you start working on a project built with the Symfony framework. You want to reuse the `UserManager` class and install the package containing it inside this new project. A Symfony application also has an event dispatcher readily available as the `event_dispatcher` service in its service container. But this event dispatcher is an instance of `EventDispatcherInterface`. It is *impossible* to use the Symfony event dispatcher as a constructor argument for the `UserManager` class because the type of the argument wouldn't match the type of the injected service. You have effectively *prevented reuse* of the `UserManager` class.

If you still want to use the `UserManager` class in a Symfony project you would need to add an extra dependency on the `illuminate/events` package to make the Laravel `Dispatcher` class available in your project. You would have to configure a service for it, next to the already existing Symfony event dispatcher. You would end up having two global event dispatchers. Then you would still need to bridge the gap between the two types of dispatchers, since events fired on the Laravel `Dispatcher` would not be fired automatically on the Symfony event dispatcher too. In fact, they even have incompatible types (event objects versus arrays).

Solution: add an abstraction and remove the dependency using composition

As we discussed earlier depending on a concrete class can be problematic all by itself because it makes it hard for users to switch between implementations of that dependency. Therefore we should introduce our own interface which decouples this class from any concrete event dispatcher implementations:

```
interface DispatcherInterface
{
    public function dispatch($eventName, array $context = []);
}
```

This abstract event dispatcher is not framework-specific, it just offers one method that can be used to dispatch events. Now we can change the `UserManager` class to only accept an event dispatcher which is an instance of our very own `DispatcherInterface`:

```
class UserManager
{
    private $dispatcher;

    public function __construct(DispatcherInterface $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
}
```

The `UserManager` is now fully decoupled from the framework. It uses its own event dispatcher, which is quite general and contains the least amount of details possible.

Of course our `DispatcherInterface` is not a working event dispatcher itself. We need to bridge the gap between that interface and the concrete event dispatchers from Laravel and Symfony. We can do this using the *Adapter* pattern. Using object composition we make the Laravel `Dispatcher` class compatible with the `DispatcherInterface`:

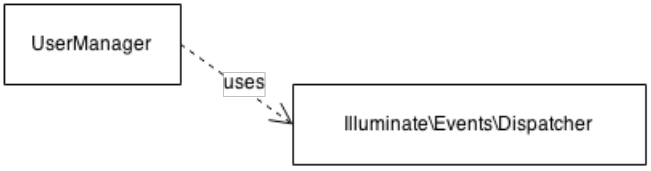
```
use Illuminate\Events\Dispatcher;

class LaravelDispatcher implements DispatcherInterface
{
    private $dispatcher;

    public function __construct(Dispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function dispatch($eventName, array $context = [])
    {
        $this->dispatcher->fire($eventName, array_values($context));
    }
}
```

By introducing the `DispatcherInterface`, we have cleared the way for users of other frameworks to implement their own adapter classes. These adapter classes only have to conform to the public API defined by the `DispatcherInterface`. Under the hood they can make use of their own specific type of event dispatcher. For example, the adapter for the Symfony event dispatcher would look like this:

```
use Symfony\Component\EventDispatcher\EventDispatcherInterface;
use Symfony\Component\EventDispatcher\GenericEvent;

class SymfonyDispatcher implements DispatcherInterface
{
    private $dispatcher;

    public function __construct(EventDispatcherInterface $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
```

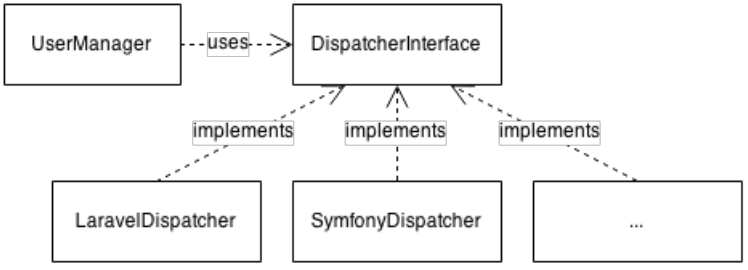
```
    public function dispatch($eventName, array $context = [])
    {
        $this->dispatcher->dispatch(
            $eventName,
            new GenericEvent(null, $context)
        );
    }
}
```

Before we introduced the `DispatcherInterface`, the `UserManager` depended on something *concrete* - the Laravel-specific implementation of an event dispatcher:



The `UserManager` has a dependency on the concrete Laravel `Dispatcher`

After we added the `DispatcherInterface` the `UserManager` class now depends on something *abstract*. In other words, we inverted the dependency direction, which is exactly what the *Dependency inversion principle* tells us to do:



The `UserManager` has a dependency on an abstract dispatcher, upon which several adapters depend

Conclusion

As we have seen in this chapter, following the *Dependency inversion principle* is helpful when others start using your classes. They want your classes to be abstract,

only depending on other abstract things, and leaving the details to a couple of small classes with specific responsibilities.

Applying the *Dependency inversion principle* in your code will make it easy for users to exchange specific parts of your code, with other parts that are tailored to their specific situation. At the same time *your* code remains general, abstract and therefore highly reusable.

Peeking ahead: abstract dependencies

Although the *Dependency inversion principle* is a class design principle, it is all about the relation between classes. This relation often transcends the boundaries of a package. Therefore the *Dependency inversion principle* resonates strongly at a package level.

Classes should depend on abstractions, not on concretions. Parallel to this, packages should depend in the direction of abstractness, away from concretions. Since packages contain both interfaces and classes, each package should only depend on other packages that have a higher or equal ratio of interfaces versus concrete classes. The package design principle that tells us to do so, is the last one, called the *Stable abstractions principle*.

Package design

Principles of cohesion

Code consists of *statements*, grouped into *functions*, grouped into *classes*, grouped into *packages*, combined into *systems*. There are several insights about this chain of concepts that I would like to discuss here, before we dive into the actual *principles of package design*.

Becoming a programmer

First of all, it occurred to me that in my programming career I learned about these different concepts in the exact same order in which I just mentioned them. The first thing I learned about PHP as a young website builder was to insert PHP statements into a regular HTML file. By doing so it was possible to turn a static HTML page into a dynamic one. You could conditionally show some things on the page, dynamically build a navigation tree, do some form processing and even fetch something from a database.

```
<?php
$name = htmlentities($_GET['name'], ENT_QUOTES);
$day = date('l');
?>
<html>
    <head>
        <title>My first homepage</title>
    </head>
    <body>
        <h1>Welcome, <?=$name?></h1>
        <p>Today it's <?=$day?></p>
    </body>
</html>
```

When the pages I created became more complex I intuitively felt the need to organize things in a better way, to make my work easier and to support my future self when a client would change their mind again. At first I resorted to so-called *include files*. The behavior of those include files could be changed using global variables:

```
<?php
// display_day.php

global $day;

?><p>Today it's <?=$day?></p>
```

```
<?php
// index.php

global $day;

$day = date('l');
include('display_day.php');
```

Looking back at this kind of crazy code from my oldest projects, I realize that I was actually using those include files as some kind of *functions*. Not really in a true functional manner though, because these "functions" had some nasty side effects (i.e. sending output directly to the client).

Using include files as functions worked fine for some time, until customer requirements got more complicated and I had to build an authentication mechanism for users, using a login form. I copied some code from the Internet which contained some *actual functions*. Of course I just pasted the code in my project (and it worked). But then I started to unravel it, diving into this "new" concept of a function.

When I got it, things changed dramatically: I started using functions all over the place. I put them in a file called `functions.php`. In each PHP file I included this file containing all the functions and then I could execute those functions from anywhere:

```
function wrap($text, $maxLineLength) {
    ...
}

function fetch_user_data($id) {
    ...
}

function array_merge_deep($array1, $array2) {
    ...
}

function copy_shopping_cart() {
    ...
}

function rename_file($source, $destination) {
    ...
}

...
```

Even though many of those functions still echoed things directly to the client (instead of buffering the output), I felt that my applications were already becoming pretty advanced.

At some point (I don't remember when exactly) I was browsing through the php.net[20] website and stumbled upon the page about classes[21]. I recognized this "class" thing as a way of grouping functions that were related to each other. So I created a big `Page` class which became the core of the first CMS I ever built. I have added the source of the `Page` class as an appendix to this book for your enjoyment, but let me show you some of the more interesting parts of the code here:

[20] http://php.net

[21] http://php.net/manual/en/language.oop5.basic.php

```
class Page
{
  public $uri = null;
  public $page = array();
  public $site_title =  '';
  public $breadcrumbs = array();
  public $js = array();
  public $css = array();
  public $auto_include_dir = '';
  /* @public $smart Smarty */
  public $smarty = null;
  public $default_template = '';
  public $template = '';
  public $cms_login = null;
  public $user_login = null;
  public $is_user = false;
  public $is_admin = false;
  public $languages = array();
  public $default_language = null;
  public $language = null;
  public $menu_items = array();

  protected $_extra_request_parameters = array();

  public function __construct($uri)
  {
    $this->connect_db();
    header('Content-Type: '.HEADER_CONTENT_TYPE);
    $this->smarty = new Smarty;

    if (isset($_GET['clear_cache']))
    {
      $this->smarty->clear_cache();
    }
```

```
if (DEBUGGING)
{
  $this->smarty->caching = false;
  if (trusted_ip())
  {
    $this->smarty->debugging = true;
  }
}

if (trusted_ip())
{
  ini_set('display_errors', '1');
  error_reporting(
      E_ERROR | E_PARSE | E_WARNING | E_USER_ERROR
      | E_USER_NOTICE | E_USER_WARNING
        );
}
else
{
  $this->smarty->debugging = false;
  ini_set('display_errors', '0');
  error_reporting(0);
}

    ...

if (!table_exists('content'))
{
  require(ROOT.'/includes/install.php');
  install();
}

$this->add_title_part(SITE_TITLE);

$this->cms_login = new LoginClass('admins', 'cms_login');
```

```
    $this->user_login = new LoginClass('users', 'user_login');

    if ($this->cms_login->isLoggedIn())
    {
      $this->is_admin = true;
    }

    if ($this->user_login->isLoggedIn())
    {
      $this->is_user = true;
    }

        ...

    $this->open_page();
  }

  public function connect_db()
  {
    $this->db_connection = @mysql_connect(
        MYSQL_HOST,
        MYSQL_USER,
        MYSQL_PASSWORD
        );

    if ($this->db_connection)
    {
      $this->db = @mysql_select_db(MYSQL_DB);
      if (!$this->db)
      {
        ?><p class="warning">Geen database!</p><?
        exit;
      }
    }
    else
```

```
    {
      ?><p class="warning">Geen verbinding!</p><?
      exit;
    }
  }
}
```

In summary:

- The `Page` class has about 20 or 30 instance variables.
- It depends on about 20 constants that are defined outside that class (more specifically: in the `config.php` file).
- It refers to so-called super-globals, like `$_GET`.
- It globally modifies the settings of the PHP process by configuring the PHP error mode in the constructor.
- Error messages (in Dutch) are being echoed directly to the user, after which the program simply terminates.
- It connects to a MySQL database and verifies that all the required tables are there. If not, it runs the install script.
- It sends response headers.

Although I would never write such code today, when I look at the `Page` class now, I don't feel ashamed about what I did back then. It's clear to me that I wasn't struggling to get things *working* (because everything just worked, I always did a good job in that respect). Rather, I was struggling to get things *organized*.

The hardest part

It took me a couple of years before I learned how to make my classes *moderately good*. And still every day there's something new to learn about class design, some old habit to drop, some new principle to apply. From this I draw the conclusion that organizing code into classes is a *difficult thing*. Of course it's fairly easy to learn all you can about the keywords that a programming language provides for defining

classes (`class`, `extends`, `implements`, `abstract`, `final`, etc.). Learning how to use them well, that's much harder.

Let's get back to this chain of concepts: statements, grouped into functions, grouped into classes, grouped into packages, combined in a system. Let me ask: what is the "beef" of a program? Well, it's the *statements*. Statements actually make things happen. If we were to transform all class methods of a program to regular functions and then inline those functions we'd get one long page of statements and, when executed, the program would still do the same thing.

From this we can draw the conclusion that code does not *need* to be organized if you only need it to *just work*. For the computer, what counts is statements. Still, we make great efforts to *modularize* our statements. We put them in class methods, and we group the classes together in packages. And judging by the order in which you learn things as a developer, writing classes and creating packages is much more difficult than writing just statements.

Cohesion

Over the years you often struggle to organize your code in the best way possible. While doing so, you gradually develop a strong sense of "belonging together". It helps you decide whether or not two pieces of code belong together. This intuition keeps evolving forever: when you are writing statements and put them into functions, when you write classes for those functions, and when you combine those classes into packages.

This intuition you have as a programmer, this sense of "belonging together", is actually about something called *cohesion*. Cohesion is a *degree of relatedness*. Some things are highly cohesive, some are less cohesive, depending on how much they are related to each other.

Early in life you learn how to decide whether or not things are cohesive. In school you get these little exercises: "One of the following words does not belong in the list, which is it? Duck, frog, fish, camel." When you have the list "duck, frog, fish", you have a highly cohesive list of words. The words stand for things that are highly related to each other (because they are all names of animals that live or survive in water). When you add "camel" to the list, the list definitely becomes less cohesive.

This resembles your job as a programmer: you need to find out which things you can add without making the whole less cohesive, and which things you can remove to make the whole more cohesive.

In the context of package design cohesion is mainly about which classes belong together in a package. There are many different ways in which you can arrange and combine classes and all of them produce a different kind of cohesion. For example, you can group all classes that serve as a controller, or all classes that are entities. The result is something called *logical cohesion*. But when you group the "blog post" controller and the "blog post" entity, the result is *communicational cohesion*: all classes in such a package operate on the same data (blog post records from the database).

There are several other types of cohesion[22] but the most important type of cohesion, the one you should strive for, is *functional cohesion*. Functional cohesion is achieved when all things in a "module" (e.g. a package) together can be used to perform a single well-defined task.

Class design principles benefit cohesion

Earlier I said that the programmer's sense of belonging together was based on intuition, shaped by experience. Of course there's also a rational side of that sense: if you know about class design principles then that will help you write highly cohesive code. For example when you apply the SOLID principles to your classes, they will automatically become more cohesive. You will end up with classes having fine-grained, client-specific interfaces which makes it unlikely that those interfaces contain methods that don't belong there. So applying the *Interface segregation principle* will give you highly cohesive classes. You will also have classes with just one reason for change, which means they are not "all over the place". So applying the *Single responsibility principle* also has the beneficial effect of making classes more cohesive.

Package design principles, part I: Cohesion

Once we know how to create highly cohesive classes, we can take the next step. After statements, functions and classes we arrive at: packages. Packages are groups

[22] http://en.wikipedia.org/wiki/Cohesion_%28computer_science%29

of classes, and just like everything that is a group of things, a package has cohesion (a certain degree of relatedness) too. If classes were grouped arbitrarily, the package containing them would have *coincidental cohesion*. Of course, the trick is to group classes in such a way that the package has a high level of functional cohesion: all the classes in the package should serve to perform that single well-defined task.

There are three package design principles that support you in creating highly cohesive packages. In the following chapters we will discuss each of them extensively. They are called respectively the *Release/reuse equivalence principle*, the *Common reuse principle* and the *Common closure principle*.

The Release/reuse equivalence principle

The first of the actual package design principles discussed in this book is the *Release/reuse equivalence principle*. This principle says:

> The granule of reuse is the granule of release.

This principles has two sides. First of all, you should release as much code as you (or others) can reasonably reuse. It makes no sense to invest all the time and energy needed to properly release code if nobody is going to use it in another project anyway. This may require you to do some kind of research to establish the viability of your package once you would privately or publicly release it. Maybe the package only *seems to be reusable*, but in the end it turns out to be useful in your specific use case only.

The other side of the principle is: you can only reuse the amount of code that you can *actually release*. By applying all the principles of class design, you may have created perfectly general, reusable code. But if you never release that code, than it's not reusable after all. So before you start making all your code reusable, try to answer this question first: are you going to be able to release that code, and manage future releases too?

Being aware of the effort that is required for releasing a package will help you decide on the number and the size of the packages that you are going to create. For example, releasing hundreds of tiny packages is something you can't possibly do. Each package requires a certain amount of time and energy from its maintainer. Think about tracking and fixing issues, adding version tags to new releases, keeping the documentation up-to-date, etc. On the other hand, releasing one very big package is equally impossible. It will undergo so many changes related to different parts of the package that it will be a very volatile package, a constantly moving target. This is not helpful at all for its users.

As I explained in the introduction, cohesion is always about "belonging together". And so the cohesion principles of package design offer strategies to decide if classes should be grouped in a package. The *Release/reuse equivalence principle* helps you decide *if* you would be able to release such a package at all. It makes you aware of the fact that a released package requires the careful nurturing of its maintainer.

The remaining sections of this chapter will give you an overview of the kind of things you need to take care of when you start releasing packages. While the previous part of the book was about the way in which you can prepare your *classes* for reuse, this chapter is about how you can prepare your *package* of classes to be reused, i.e. to be released. It is mostly not about code, but about all kinds of *meta* things.

Because this is a theoretical book, the following descriptions will be somewhere between theoretical and practical. They are not specific to any programming language or tool.

Keep your package under version control

The first thing you need to do is set up a version control system for your package. You need to be able to keep track of changes by you or any of the contributors, and people need to be able to pull in the latest version of the package. So even though mailing around code snippets would technically be a kind of version control (the sent date of the message could be used as the version number of the package), you should always use a *real* version control system (like Git[23]).

If you have an idea for a package (which would primarily be a coherent set of classes), the first thing you do is set up a version control repository for it. This will enable you to revert to previous situations if one particular change endangered the whole project. If you work in a team, using version control also helps you prevent conflicting changes. It also enables you to work on and test a new or experimental feature in a separate *branch*, without jeopardizing the stability of the master branch of the package.

The version control repository should be treated as a full description of the *history* of the project. You and your team are going to use the version control repository as a way to figure out when or why a bug was introduced. Make sure to only commit

[23] http://git-scm.com/

changes to the repository that are cohesive (i.e. belong together) and add descriptive and elaborate comments when you commit something.

In order to make your package available, you have to make sure it is hosted somewhere. Depending on your needs this can be something public or private, hosted or self-hosted.

Add a package definition file

Most programming languages have a standardized way of defining packages. And often this is just a simple file which provides some or all of the following properties of the package:

- Name of the package
- Maintainers, possibly some contributors
- URL and type of the version control repository
- Required dependencies, like other packages, specific language versions, etc.

Read as much as you can about your different options. A package containing a rich definition file that utilizes all the options in the right way is likely to be a well-behaving package in the package ecosystem of your programming language.

Once you have created a correct package definition file, you probably have to register the package to some sort of a central package repository or registry. Each programming language has its own remote package repositories, with different manuals and requirements.

Use semantic versioning

When you release a package you have to answer the following questions and make your intentions clear:

- Do you introduce changes in the API of your code with great care?

- Will you try to make sure that those changes don't break the way in which users interact with your package?
- In which situations would you allow yourself to heavily change your API?

In other words: how are you going to take care of *backward compatibility*? Package maintainers generally follow a versioning strategy called "semantic versioning". The outline of this strategy is this:

- You can add new things to your package, but make sure to release a new *minor* version each time you do so (e.g. `x.1.x` ⇒ `x.2.x`). When you want to deprecate things, just keep them around for a while.
- Remove deprecated parts or introduce backwards incompatible changes when you release a new *major* version (e.g. `1.x.x` ⇒ `2.x.x`).
- Constantly fix bugs and release them as *patch* versions (e.g. `x.x.1` ⇒ `x.x.2`).

Semantic version numbers are expected to always consist of three incremental numbers, like `0.1.1`, `2.0.10` or `3.1.0`. Each of the three parts of a version number conveys a particular meaning, which is why this kind of versioning is called *semantic*, which translates to "having meaning" (as opposed to being just random numbers).

The first part of a package's version number, the number before the first dot, is called the *major version*. The first major version is usually `0`. This version should be considered unfinished, experimental, heavily changing without too much care for backward compatibility. Starting from major version `1`, the public API is supposed to be stabilized and the package has a certain trustworthiness from that moment on. Each next increment of the major version number marks the moment that part of the code breaks backward compatibility. It is the moment when method signatures change, deprecated classes or interfaces are being removed. Sometimes even a complete rework of the same functionality is being released as a new *major* version.

The second part of the version number is the *minor version*. It also starts counting from `0`, though this has no special significance, except "being the first". Minor versions can be incremented when new functionality has been added to the package or when parts of the existing public API have been marked as deprecated. The promise of a new minor version is: nothing will change for its users, existing ways

in which they use the package will not be broken. A minor version only adds new ways of using the package.

The last part of the version number is the *patch version*. Starting with version `0` it is incremented for each patch that is released for the package. This can be either a bug fix, or some refactored private code, i.e. code that is not accessible by just using the public API of the package. Since refactoring means "changing the structure of code, without changing its behavior", refactoring private package code will not have any negative side-effect on existing users.

Immediately after the version number (consisting of the major, minor and patch version, separated by dots), there may be a textual indication of the state of the package: `alpha`, `beta`, `rc` (release candidate), optionally followed by another dot and another incremental number.

The number combined with the optional meta identifier of the package's version can be used to compare version numbers:

```
1.9.10
2.0.0
2.1.0
2.1.1-alpha
2.1.1-beta
2.1.1-rc.1
2.1.1-rc.2
2.1.1
```

Comparison is done in the natural way, so `2.1.1` is a lower version than `2.10.1`. There is no limit to each part of the version number, so you can just keep incrementing it.

Design for backward compatibility

When you use semantic versioning for your packages, providing backward compatibility means that you strive to provide the exact same functionality in minor version `x + 1` as in the previous minor version `x`. In other words: if some user's code relies on a feature provided by version `1.1.0`, you promise that this same feature will be

available in `1.2.0`. Using it will have exactly the same effects in both versions. Not only would it have the same behavioral effects, the feature can also still be invoked in the same way.

Of course, you may have fixed some bugs between two minor versions, and you may have *added* some features. But none of these things should pose any problems for users who upgrade their dependency on your package to the next minor version. All their tests should still pass, and everything should still work as it did before upgrading the dependency.

As you can imagine, and you probably know this already from your daily work as a developer, providing *true* backward compatibility can be really hard. You want to make some progress, but your promise for backward compatibility can hold you back. Still, if you want your package to be used by other developers, you need to give them both new features *and* continuity.

There is a time when you don't *have* to provide the continuity, which is when your package's major version is still `0.x.x`. During this period your package will be considered unstable anyway and you can move everything around. This may enrage some early adopters, but since they are aware of the fact that the package is still unstable, they can't complain really.

Working forever on `0.*` versions of a package would seem to alleviate you from the pain of keeping backward compatibility. However, an *unstable* package will likely not be used in any serious project that itself intends to be *stable*. In such a project people might depend on your unstable package, but will get really mad when they upgrade such a package and nothing works anymore. They would have to add extra integration tests, to test the boundaries between their and your code, so they will notice any compatibility problems early on. They will be scared to upgrade your package and therefore also miss all relevant bug or security vulnerability fixes.

In conclusion: you should make up your mind about the design of your code and as soon as you have tested your package in one or two of your projects, release it as version `1.0.0`. If you then really hate the design of your code, your strategy could be to start working on version `2.0.0` and announce that you will stop developing features for version `1.0` soon. Using version control branches you would still be able to provide fixes for the previous major version if you want.

Rules of thumb

Even though you could get away with only releasing major versions, the more likely scenario is that you will release minor versions too. So designing for backward compatibility should be part of your strategy from the moment you release the first major version of your package. In the following sections I will discuss some things you should or should not do in order to provide backward compatibility.

These are just examples and rules of thumb. There are many more ways in which you can prevent a backward compatibility break and still they can accidentally happen. Software is already complex by nature, but there are also ways in which people use your code that you don't officially support, or don't know of yet. This means you will never be fully covered. But you can at least maximize the potential damage.

Don't throw anything away Whenever you add something to your package, make sure it still exists in the next version. This applies to things like:

- Classes
- Methods
- Functions
- Parameters
- Constants

A class *exists* if it can be auto-loaded, so classes don't necessarily need to be in the same file, just make sure the class loader is always able to find them. This means you may move a class to another package and add that package as a dependency.

When you rename something, add a proxy Renaming classes is possible, but make sure that the old class can still be instantiated:

```
/**
 * @deprecated Use NewClass instead
 */
class DeprecatedClass extends NewClass
{
    // will inherit all methods from NewClass
}

class NewClass
{
    ...
}
```

Renaming a method is possible, but make sure you forward the call to the new method.

```
class SomeClass
{
    /**
     * @deprecated use newMethod() instead
     */
    public function deprecatedMethod()
    {
        return $this->newMethod();
    }

    public function newMethod()
    {
        ...
    }
}
```

Or if you have moved the functionality to another class, make sure it still works when someone uses the old method:

```php
class SomeClass
{
    /**
     * @deprecated Use Something::doComplicated() instead
     */
    public function doSomethingComplicated()
    {
        $something = new Something();

        return $something->doComplicated();
    }
}
```

Add `@deprecated` annotations

Whenever you deprecate an element of your code, be it a class, a method, a function or a property, you should not remove it immediately, but keep it around until you release the next major version. In the meantime, make sure it has the `@deprecated` annotation. Don't forget to add a little explanation and tell users what they should do instead, or how they can modify their own code to make it ready for the next major version in which the deprecated things will be removed.

Renaming parameters of a method is not problematic (in PHP at least), as long as their order and type doesn't change. Renaming parameters of a method defined in an interface is also not problematic. Classes that implement an interface may always use different names, as long as the parameter types correspond.

```
// interface defined inside the package
interface SomeInterface
{
    public function doSomething(ObjectManager $objectManager);
}

// class created by a user of the package
class SomeClass implements SomeInterface
{
    public function doSomething(ObjectManager $entityManager)
    {
        ...
    }
}
```

Only add parameters at the end and with a default value When you need to add a parameter to a method, make sure you add it at the end of the existing list of parameters. Also make sure that the new parameter has a sensible default value.

```
// current version
class StorageHandler
{
    public function persist($object)
    {
        $this->entityManager->persist($object);

        /*
         * The current implementation always flushes
         * the entity manager
         */
        $this->entityManager->flush();
    }
}

// next version
```

```
class StorageHandler
{
    public function persist($object, $andFlush = true)
    {
        $this->entityManager->persist($object);

        // the new implementation only flushes if requested
        if ($andFlush) {
            $this->entityManager->flush();
        }
    }
}
```

The extra parameter `$andFlush` has been introduced with a default value `true` to make sure that the new method behaves exactly the same as the old method, which already flushed the entity manager by default.

Methods should not have side effects Don't tempt users of your code to rely on a particular side effect of calling a method. When you later change the code, the side effect is likely to disappear, which breaks the user's code.

```
// previous version
class Stream
{
    public function open($file)
    {
        /*
         * The previous implementation creates a directory
         * if necessary
         */
        $this->createDirectoryIfNotExists($file);

        $this->handle = fopen($file, 'w');
    }
```

```
    private function createDirectoryIfNotExists($file)
    {
        ...
    }
}

// next version
class Stream
{
    public function open($file)
    {
        /*
         * The new implementation does not create a directory
         * automatically
         */
        $this->handle = fopen($file, 'w');
    }
}
```

In the previous version `Stream::open()` implicitly created a directory when a file was opened. This behavior was not desirable, so the next version of `Stream::open()` leaves it to the user to make sure the directory exists. They can use `Filesystem::isDirectory()` and `Filesystem::createDirectory()` for this:

```
class Filesystem
{
    public function createDirectory($directory)
    {
        ...
    }

    public function isDirectory($directory)
    {
        ...
    }
}
```

Of course this change causes a backward compatibility break. But this could have been prevented in the first place: make sure every method has no side effects and does one thing (and one thing only), which is clearly defined and is unlikely to change in the future.

Dependency versions should be permissive If your package has some dependencies itself, make sure you don't put too many restrictions on their version numbers. For instance when you write the code for a new package you may prefer to work with the latest version of one of its dependencies, let's say version `2.4.3`.

Since the maintainer of that package might have taken proper care of backward compatibility, it's likely that your package works well with version `2.3`, and maybe even with `2.2` or `2.1`.

By requiring version `2.4.3` or higher of the dependency you have effectively excluded all users who have lower versions of that same dependency installed in their project, even though your package would work fine with these older versions.

There are two solutions: force your users to upgrade to a new version of that dependency *or* make your own requirements less restrictive. Since the first option may break things in their project (a pain of which you don't want to be the cause), it is almost always best to choose the second option: make your code compatible with older versions and loosen your own requirements.

This is also true the other way around: if a new stable version of a package becomes available. As a package manager you are expected to make your package work with that new version too. You should check if it already does by installing the new version of the dependency and running the tests of your package. If necessary, make some changes to your code until all the tests pass. Of course, you need to make sure that the package continues to work with the previous version of the dependency. You might set up some continuous integration process to do this automatically for you.

Use objects instead of primitive values In order to provide backward compatibility, it's a good idea to use objects where you would normally use arrays or scalar values. Consider the following incremental changes to the `HttpClientInterface`:

```php
// version 1.0.0
interface HttpClientInterface
{
    public function connect($host);
}

// version 1.1.0
interface HttpClientInterface
{
    public function connect(
        $host,
        $port = 80
    );
}

// version 1.2.0
interface HttpClientInterface
{
    public function connect(
        $host,
        $port = 80,
        $ssl = false
    );
}

// version 1.3.0
interface HttpClientInterface
{
    public function connect(
        $host,
        $port = 80,
        $ssl = false,
        $verifyPeer = true
    );
}
```

Instead of adding more and more parameters (with default values because of the previously explained rule about adding parameters to existing methods), it would be much easier if we had some kind of value object from the beginning:

```
interface HttpClientInterface
{
    public function connect(
        ConnectionConfigurationInterface $configuration
    );
}

class ConnectionConfigurationInterface
{
    public function getHost();
    public function setHost($host);
    public function getPort();
    public function setPort($ort);
    public function shouldUseSsl();
    public function useSsl($useSsl);
    public function shouldVerifyPeer();
    public function verifyPeer($verifyPeer);
}
```

This would help us provide backward compatibility, while allowing us to add extra configuration options to the connection configuration. Users of this package would not be in trouble because of an extra method on the `ConnectionConfigurationInterface`, as long as it would return a sensible default value if they have provided no option for it.

Use private object properties and methods for information hiding The `connect()` method of `HttpClientInterface` expects an object of type `ConnectionConfigurationInterface`. The great thing about using an object instead of a primitive value is that you can use information hiding. A value object like `ConnectionConfiguration` normalizes its values, and makes sure it always behaves consistently:

```
class ConnectionConfiguration
{
    private $host = 'localhost';
    private $port = 80;

    public function setHost($host)
    {
        if (strpos($host, ':') !== false) {
            list($host, $port) = explode($host);
            $this->setPort($port);
        }

        $this->host = $host;
    }

    public function setPort($port)
    {
        $this->port = (integer) $port;
    }

    public function getHost()
    {
        return $this->host;
    }

    public function getPort()
    {
        return $this->port;
    }
}
```

An object of type `ConnectionConfiguration` always behaves consistently, at any moment in time, because it has sensible default values. On top of that it accepts different types of input for the host name (either a plain host name or a host name with a port), to ensure compatibility with previous versions of `ConnectionConfiguration`.

Users of older or newer versions of this class are not aware of this difference, since it has been *encapsulated* by the `setHost()` method.

Use object factories It is likely that between different package versions a class would have different dependencies.

```
// previous version
class Validator
{
    public function __construct()
    {
        ...
    }
}

// new version
class Validator
{
    public function __construct(MetadataFactory $metadataFactory)
    {
        ...
    }
}
```

Users of the previous version simply do this to create a `Validator` object:

```
$validator = new Validator();
```

But when users upgrade the validator package to the new version, this will obviously raise an error because of the missing constructor argument. From then on they should first create a `MetadataFactory`:

```
$metadataFactory = new MetadataFactory();
$validator = new Validator($metadataFactory);
```

To prevent such backward compatibility breaks it would be better if we had provided a factory for `Validator` objects from the start. This way, users only need to create a factory first (which should require no constructor arguments), and from then on they could use the factory to create new validators, without having to worry about other constructor arguments down the line:

```
class ValidatorFactory
{
    public function createValidator()
    {
        $metadataFactory = new MetadataFactory();

        return new Validator($metadataFactory);
    }
}
```

If in a future version the `Validator` class would need any other constructor parameter, the factory will add it behind the scenes and the user does not need to know about it.

And so on By now you will get the idea. There are many ways in which you can make your code backward compatible and still allow for future changes. For more on this subject I would like to point you to an interesting article by Garrett Rooney: Preserving Backward Compatibility[24]. He describes many interesting ways in which developers of the Subversion project have tried to maintain backward compatibility, while enabling *forward compatibility*. Another interesting document is "Our Backward Compatibility Promise"[25] delivered by the Symfony framework team. It may become a good guide for you too.

[24] http://www.onlamp.com/pub/a/onlamp/2005/02/17/backwardscompatibility.html

[25] http://symfony.com/doc/current/contributing/code/bc.html

Add meta files

The meta files that are absolutely necessary are a quick start guide in the form of a "README" file, some legal stuff in the form of a license file, and a change log.

`README` and documentation

The `README` file should be in the root directory of the package. It contains everything a user needs to get started. The `README` file may be the only official documentation for a package. Of not, it should contain a link to some other source of documentation inside the package (for instance in its `meta/doc` directory) or a dedicated website. Whichever strategy you choose, the `README` file is *mandatory* since it's the starting point for people to learn more about your package.

A `README` file is a text file. The lines should be wrapped at an appropriate width (e.g. 80 columns) in order to make it readable in the terminal. It's also a good idea to apply some styling and structuring to it. Its fairly conventional to write the file in Markdown[26], which gives you some basic markup options. You can write some words in italics or bold, add code blocks, and section headers. If you use Markdown in your `README` file, rename the file to `README.md`.

The `README` file should at least contain the following sections:

Installation and configuration

This can be as simple as mentioning the command by which you can install the package in a project, for example:

```
composer require matthiasnoback/some-package
```

Then tell the users everything else they need to do in order to use the package. Maybe they need to set up or configure some things, clear a cache, add some tables to a database, etc.

[26] http://daringfireball.net/projects/markdown/syntax

Usage

You need to show users how they can use the code in your package. This requires a quick explanation of some use cases and *code samples* for those situations.

Extension points

If the package is designed to be extended, if there are plugins for it, or bundles/modules that make it easy to integrate the library in a framework-based project, make sure you mention those *extension points*.

Limitations (optional)

You should mention use cases for which the package currently offers no solution. You should also mention known problems (bugs or other limitations) and maybe some features that you intend to implement some time.

License

Another file that is mandatory is the `LICENSE` file. Even though you have probably already provided the *name* of the license that applies to your package in the package definition file, you should still add the *full* license to your package. It should be in a file called `LICENSE` in the root of the package. In case you choose the MIT license, this is what the file contains:

```
Copyright (c) <year(s)> <name(s)>

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom
the Software is furnished to do so, subject to the following
conditions:
```

```
The above copyright notice and this permission notice shall
be included in all copies or substantial portions of the
Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS
OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

If you wonder why it is important to have a license file in your package: it depends on the country you live in, but some companies need your explicit permission to use your code in the way they intend to. They want to prevent legally uncomfortable situations caused by accidental copyright infringement. Equally important: it relieves you of any damage your code may cause when used by somebody else.

Change log

Each release of your package probably consists of multiple commits, solving at least one issue. Maybe you have added some features, maybe some bug fixes. Possibly also some really important security fixes. Users should be able to quickly see what has changed between two versions and whether or not they need to upgrade their dependencies. The place where people will look for such a list of changes is the `CHANGELOG` file.

Each new version (major, minor or patch) gets its own section in the change log in which you describe the changes that were made since the previous version. Briefly describe new features that were added, things that were deprecated (but not removed), problems that were fixed. Possibly point to issues in the issue tracker. An example of one of my own `CHANGELOG.md` files:

```
# Changelog

## v0.5.0

- Automatically resolve a definition's class before comparing it to
the expected class.

## v0.4.0

- Added `ContainerBuilderHasSyntheticServiceConstraint` and
corresponding assertion (as suggested by @WouterJ).

...
```

is no standard format, though the one proposed on keepachangelog.com[27] is both simple and complete.

Upgrade notes

Each section of the change log may contain some upgrade notes which tell users what they need to do when they upgrade to a newer version. For instance, if some classes were deprecated, it's a good idea to mention in the change log in which version you actually removed them.

In some cases these upgrade notes start taking too much space, which will muddle the view on the actual change log. Then it's time to move the upgrade notes to specific `UPGRADE-x` files. For example `UPGRADE-3.md` will contain instructions for upgrading the dependency on this package from version `2.x.x` to `3.x.x`. Remember that in between major versions no actions from the user should be required, because minor and patch versions only introduce backward compatible changes.

[27] http://keepachangelog.com/

Quality control

We have already discussed many characteristics of a package that would make it qualify as a good package (or a "good product"). Most of these characteristics were related to the infrastructure of the package: a package should display some good manners when it comes to version control, the package definition file, dependencies and their versions, and backward compatibility. Several meta files need to be in place, for the package to be usable, like documentation and a license file, etc.

You may have noticed that until now we haven't given much attention to the actual code in your package. We will of course discuss the required characteristics of classes in a package at great length in the next chapters. But in the last sections of this chapter I will first point out some aspects of package infrastructure that will help you create packages with high-quality code.

Quality from the user's point of view

A package makes some implicit promises about the code it contains. It basically says:

> You can add me to your project. *My* code will fulfill *your needs*. You won't have to write this code yourself. And that will make you very happy.

When I stumble upon a package that may "fulfill my needs", the first thing I do is read the `README` file (and possibly any other documentation that is available). When the description of the package resembles my own ideas about the code that I was going to write if this package would not exist, the next thing I do is dive into the code. I quickly scan the directory structure, the class names, then the code inside those classes, which I will then critically evaluate.

In the first place I look for the use cases that the package supports. The package maintainer has probably created this package to support one of their particular use cases. Most likely my own use case is (slightly if not vastly) different from theirs. So one particular characteristic I'm looking for is extensibility: is it possible to change the behavior of some of the classes in a package without actually modifying the code itself? Some good signs of extensibility are the use of interfaces (see also The

Dependency inversion principle and dependency injection (see also The *Open/closed principle*).

Furthermore, the package's code probably contains bugs, which need to be fixed. While wading through the code, I try to estimate the amount of work needed to fix any problem with the code - does the package contain classes with too many responsibilities? Would it be possible to swap out faulty implementations by simply implementing an interface defined in the package, or would I be forced to copy long pages of code to replicate its behavior?

Finally, I take a look at the automated tests that are available inside this package. Are there *enough* tests? Do they consist of clean code themselves? Do they make sense or are they just there for test coverage? What if there are no tests at all (which is the case for *many* packages out there)? How can I trust this code to work in my own project? How could I ever have the courage to put this code on a production server and let it be executed by real users?

The reason for my cautiousness when adding a dependency to my project is that once it is installed and I start using the code it contains, I become responsible[28] for it. Although many package maintainers are quite serious about delivering support for their packages, not every one of them will always fix any problem that is reported, or add any feature that is missing, even if you are so nice to create a pull request for it. Chances are you will be on your own when the package does not meet your expectations.

So you need to be able to fix bugs, and add features to the package, without modifying the code inside the package (since you are not actually able to do so). You need to be comfortable with that.

What the package maintainer needs to do

As a package maintainer you need to write code following established design principles, like the SOLID principles explained in the previous part of this book. But there are some other (much simpler) guidelines you should follow to produce good, or "clean" code.

[28] https://igor.io/2013/09/24/dependency-responsibility.html

Static analysis

To verify that code quality has a certain level and doesn't degrade over time, you can leverage automated static analysis tools. These tools can inspect the code and bring out a verdict based on a set of rules that in most cases can be fully configured to reflect your own quality standards.

Add tests

Of course it's important that your code looks good. But it's even more important that it runs well. And how would you be able to verify that? By adding an "appropriately sized" suite of tests to your package.

There are vastly different opinions about what this means exactly: how many tests should you write? Do you write the tests first[29], and later the code? Should you add integration tests, or functional tests? What is the amount of code coverage your package needs?

The crucial question you should ask yourself is this: *do I care about the future of my code*? Tests are meant to allow for safe refactoring later on. If you just write the code and use it in one project then you may not feel the need to write tests for this code. On the contrary you'd definitely need quite a large test suite if that code is going to be used *by anyone else* in *any other project*. In that case you want to keep fixing bugs or add new features to the package. If you have no tests, making those changes becomes difficult and dangerous. How can you trust the package to work as expected after you've made the changes?

So tests support refactoring. They will greatly help you prevent regressions in future commits. But tests also serve as the specification of your code. They describe the expected behavior when a user would do something with the code in some specific situation. This is why tests could in theory serve as documentation for the code.

This is not really true of course, because tests only tell little parts of the story, but never the whole story. They have no introduction, no epilogue, they don't fill in any (conceptual) knowledge gaps. Nevertheless, tests as a specification of the code and

[29] http://blog.8thlight.com/uncle-bob/2013/09/23/Test-first.html

a description of its behavior are important because they let users of the code know which method calls they can make, what kind of arguments they should provide, and which preconditions are required before they can do so.

If a package has no tests or too little tests, this is what it communicates to users:

> I don't care about the future of this code, I'm not sure that, when I change something, everything will keep working. In fact, I give you no hope that this code is reliable at all. I use it today, I don't care about tomorrow. YOLO :)

Set up continuous integration

All tests need to be run often. Of course you run tests all the time while developing. But every time you create another branch (for a new version), patch branches with some bug fix, or accept pull requests, you would have to run the tests again. Otherwise you will not know for sure that everything works as expected. Doing all of this manually would be too much work. And this is where continuous integration comes in handy.

Continuous integration means that every change to a project's repository will trigger its build process. If anything goes wrong, the project team will receive a notification and they can (immediately) fix the problem.

For software products that will be shipped, a build process may include the creation of an executable, or a ZIP file. For most projects the build process is mainly interesting because all the tests will be run. Some other artifacts that may be produced by the build process are code coverage and code quality metrics.

Conclusion

Most of the things that we discussed in this chapter were of a very practical nature. The underlying reason for this was: you need to get the infrastructure of your package right, before you can make it reusable in the first place. For you, your teammates,

or external developers from all over the world to be able to use your package in their projects, it needs to be *a really good product*. You (or the package maintainer succeeding you) need to be able to release the package once and to support future releases by means of a good infrastructure. Users should be able to understand what the package is all about, how they can use it, and what they can expect from you with regard to future versions.

Code being released as one package is what constitutes the first aspect of the *cohesion* of a package. If the *release* process of a package is unmanageable or not managed at all, it can not be properly *reused*. This chapter gave you an overview of what it means for a package to be released in a manageable way, from the first release, to any future release and from patch versions to major versions. There are many details to this process which we didn't cover here, but those are often specific for the programming language that you use.

One last remark before we continue to discuss the second *cohesion* principle, the *Common reuse principle*: it is possible that I have scared you, writing about all these things that you need to do to create "good" packages. Maybe you are tempted to put this book down and to let go of your dream to one day publish a package that is used by many, many people. But don't give up! Of course, creating your first package might give you some trouble. It will take some time, you may feel a bit insecure about the steps you take, you may forget some things, you may make some mistakes. The good thing is: you will learn quickly, develop some kind of habit, and in my experience other people are not shy to give you useful feedback on your packages, the code and meta files in it, the way you add version numbers for each change, etc.

After you finished reading this book, just go ahead, look for some practical suggestions to do the things that were described in a more abstract way in this chapter, and become part of the lively, code-sharing community of developers.

The Common reuse principle

In the previous chapter we discussed *Release/reuse equivalence principle*. It is the first principle of package cohesion: it tells an important part of the story about which classes belong together in a package, namely those that you can properly release and maintain as a package. You need to take care of delivering a package that is a true product.

If you follow every advice given in the previous chapter, you will have a well-behaving package. It has great usability and it's easily available, so it will be quickly adopted by other developers. But even when a package behaves well *as a package*, it may at the same time not be very *useful*.

When you group classes into packages there are two extremes that need to be avoided. You may have a very nice collection of very useful classes, implementing several interesting features. If you release all the classes as one package, you force your users to pull the entire package into their project, even if they use just a very small part of it. This is quite a maintenance burden for them.

However, if you put every single class in a separate package, you will have to release a lot of packages. This increases your own maintenance burden. At the same time users have a hard time to manage their own list of dependencies and keep track of all the new versions of those tiny packages.

In this chapter we discuss the second *package cohesion* principle, which is called the *Common reuse principle*. It helps you decide which classes should be put together in a package and what's more important: which classes should be moved to another package. When we are selecting classes or interfaces for reuse, the *Common reuse principle* tells us that:

> Classes that are used together are packaged together.

So when you design a package you should put classes in it that are going to be used *together*. This may seem a bit obvious; you are not going to put completely unrelated

classes that are never used together in one package. Things become somewhat less obvious when we consider the other side of this principle: you should not put classes in a package that are *not* used together. This includes classes that are likely *not* to be used together (which leaves the user with irrelevant code imported into their project).

In this chapter we first discuss some packages that obviously violate this rule: they contain all sorts of classes that are not used together. Either because those classes implement isolated features, or because they have different dependencies. Sometimes the package maintainer puts those classes in the same package because they have some conceptual similarity. Some may think it enhances the usability of the package for them or its users.

At the end of this chapter we try to formulate the principle in a more positive way and we discuss some guiding questions which can be used to make your packages conform to the *Common reuse principle*.

Signs that the principle is being violated

There are many signs, or "smells", by which you can recognize a package that violates the *Common reuse principle*. I will discuss some of these signs, and I will use some real-world packages as examples. A quick word before we continue: in no way I want to dispute the greatness of these packages: they are well-established packages, created by expert developers, and used by many people all over the world. However, I do not agree with some of the package design choices that were made. So you should take my comments below not as angry criticism, but as a gesture towards what I think is the ideal of package design.

Feature strata

The most important characteristic of packages that violate the *Common reuse principle* is what I call "strata of features". I really like the term *strata*, and this is the perfect time to use it: a *stratum* is (dictionary.com[30]):

> A layer of material, naturally or artificially formed, often one of a number of parallel layers one upon another.

[30] http://dictionary.reference.com/browse/strata

I'd like to define "feature strata" as features existing together in the same package, but not dependent on each other. This means that you would be able to use feature A without feature B, but adding feature B is possible without disturbing feature A. It also means that afterwards disabling feature B is no problem, and won't cause feature A to break. Feature A and B don't touch, they work in *parallel*.

In the context of packages feature strata often manifest themselves as classes that belong together because they implement some specific feature. But then after one feature has been implemented, the maintainer of the package kept adding new features, consisting of conglomerates of classes, to the same package. In most cases this happens because the features are *conceptually* related, but not *materially*.

Obvious stratification

Sometimes you can recognize a stratified package by the fact that it literally contains a different namespace for each *feature stratum*. There are many examples of this, but let's take a look at the symfony/security[31] package, which contains the Symfony Security component[32]. As you can see by looking at its directory/namespace tree, it has four major namespaces, `Acl`, `Core`, `Csrf` and `Http`, each of them containing many classes:

```
.
├── Acl
│   ├── Dbal
│   ├── Domain
│   ├── Exception
│   ├── Model
│   ├── Permission
├── Core
│   ├── Authentication
│   ├── Authorization
│   ├── Encoder
│   ├── Event
│   ├── Exception
```

[31] https://packagist.org/packages/symfony/security
[32] https://github.com/symfony/Security

```
|   ├── Role
|   ├── User
|   ├── Util
|   └── Validator
├── Csrf
└── Http
```

Classes from these major namespaces don't have to be used together at the same time: classes in `Acl` only depend on `Core`, classes in `Http` depend on classes in `Core` and optionally on classes in `Csrf`. Not the other way around. This means that if someone were to install this package to use the `Csrf` classes, they would only use a small subset of this package. So if we think about what the *Common reuse principle* was all about ("Classes that are used together are packaged together") then this is a clear violation of this principle.

As you can see, some of the major namespaces are split up into minor namespaces. And as it turns out, many of the classes from the minor namespaces can be used separately from classes in the other namespaces. This again indicates that the *Common reuse principle* has been violated, and that the package should be split. Fortunately the package maintainers already decided to split this package into several other packages, so now at least the major namespaces have their own package definition file and can be installed separately.

Obfuscated stratification

In many other cases a package has feature strata which aren't so easy to spot. The classes that are grouped around a certain functionality are not separated by their namespace, but by some other principle of division, for instance by the *type* of the class. Take a look at the `nelmio/security-bundle`[33] package which contains the NelmioSecurityBundle[34] which can be used in Symfony projects to add some specific security measures that are not provided by the framework itself. Because of the nature of the Symfony HttpKernel[35] everything related to security can be implemented as an event listener, which hooks into the process of converting a

[33] https://packagist.org/packages/nelmio/security-bundle
[34] https://github.com/nelmio/NelmioSecurityBundle
[35] https://github.com/symfony/HttpKernel

request to a response. Some security-related event listeners will prevent the kernel from further handling the current request (for instance based on the protocol used), and some listeners modify the final response (e.g. encrypt cookies or session data).

By looking at the directory tree of this package you can easily recognize the fact that indeed most of the features are introduced as event listeners, some of which may use utility classes like `Encrypter` and `Signer`:

```
.
├── ContentSecurityPolicy
│   └── ContentSecurityPolicyParser.php
├── Encrypter.php
├── EventListener
│   ├── ClickjackingListener.php
│   ├── ContentSecurityPolicyListener.php
│   ├── EncryptedCookieListener.php
│   ├── ExternalRedirectListener.php
│   ├── FlexibleSslListener.php
│   ├── ForcedSslListener.php
│   └── SignedCookieListener.php
├── Session
│   └── CookieSessionHandler.php
└── Signer.php
```

As you could have guessed by the names of the listeners, each listener has a particular functionality and each of the listeners can be used separately. This guess can be confirmed by looking at the available configuration options for this package:

```
nelmio_security:
    # signs/verifies all cookies
    signed_cookie:
        names: ['*']

    # encrypt all cookies
    encrypted_cookie:
        names: ['*']

    # prevents framing of the entire site
    clickjacking:
        paths:
            '^/.*': DENY

    # prevents redirections outside the website's domain
    external_redirects:
        abort: true
        log: true

    # prevents inline scripts, unsafe eval, external
    # scripts/images/styles/frames, etc
    csp:
        default: [ self ]

    ...
```

It becomes clear that all of these listeners represent a different functionality which can be configured on its own and any of the listeners can be disabled, while the other one will still keep working. This forces us to conclude that when you use one class from this package, you will not always use all the other (nor even most of the other) classes inside this package. And thus, the package violates the *Common reuse principle* in a very clear way. Somebody who wants to use only one of the features provided by this bundle (and they can!), still has to install the entire bundle.

It becomes slightly more interesting when we look at the remaining configuration options, where it seems that some parts of this package are even mutually exclusive:

HTTPS handling can not be “forced” and “flexible” at the same time:

```
nelmio_security:
    ...

    # forced HTTPS handling, don't combine with flexible mode
    # and make sure you have SSL working on your site before
    # enabling this
#    forced_ssl:
#        hsts_max_age: 2592000 # 30 days
#        hsts_subdomains: true

    # flexible HTTPS handling
#    flexible_ssl:
#        cookie_name: auth
#        unsecured_logout: false
```

This means that when you use one particular class in this package, i.e. the `ForcedSslListener` you will definitely not use *all* other classes of this package, in fact you will *with certainty* not use `FlexibleSslListener`. Of course, this definitely asks for a package split, so users would not have to be concerned with this exclusiveness.

Classes that can only be used when ... is installed

It should be noted that the previous examples were about packages that provide *separate* feature strata, but each of those features have the *same dependencies* (in this case other Symfony components or the entire Symfony framework). The following examples will be about feature strata within packages that have different dependencies themselves.

Let’s take a look at the `monolog/monolog`[36] package which contains the Monolog[37] logger. The primary class of this package is the `Logger` class (obviously). However, the real handling of log messages is done by instances of `HandlerInterface`. By

[36] https://packagist.org/packages/monolog/monolog
[37] https://github.com/Seldaek/monolog

combining different handlers, activation strategies, formatters and processors, every part of logging messages can be configured. The package tries to offer support for anything you can write log messages to, like a file, a logging server, a database, etc. This results in the following list of files (it's quite big and still it's shorter than the real list):

```
.
├── Formatter
│   ├── ChromePHPFormatter.php
│   ├── FormatterInterface.php
│   ├── GelfMessageFormatter.php
│   ├── JsonFormatter.php
│   ├── LogstashFormatter.php
│   └── WildfireFormatter.php
├── Handler
│   ├── AmqpHandler.php
│   ├── BufferHandler.php
│   ├── ChromePHPHandler.php
│   ├── CouchDBHandler.php
│   ├── CubeHandler.php
│   ├── DoctrineCouchDBHandler.php
│   ├── ErrorLogHandler.php
│   ├── FingersCrossedHandler.php
│   ├── FirePHPHandler.php
│   ├── GelfHandler.php
│   ├── HandlerInterface.php
│   ├── HipChatHandler.php
│   ├── MailHandler.php
│   ├── MongoDBHandler.php
│   ├── NativeMailerHandler.php
│   ├── NewRelicHandler.php
│   ├── NullHandler.php
│   ├── PushoverHandler.php
│   ├── RavenHandler.php
│   ├── RedisHandler.php
│   ├── RotatingFileHandler.php
```

```
|   ├── SocketHandler.php
|   ├── StreamHandler.php
|   ├── SwiftMailerHandler.php
|   ├── SyslogHandler.php
|   ├── TestHandler.php
|   └── ZendMonitorHandler.php
├── Logger.php
└── Processor
    ├── MemoryProcessor.php
    ├── ProcessIdProcessor.php
    └── PsrLogMessageProcessor.php
```

A developer can install this package and start instantiating handler classes for any storage facility that is *already available in their development environment*. As a user you don't need to think about which package you should install, it's always the main package. This would seem to have a high usability factor: isn't this easy? Whatever your situation is, just install the `monolog/monolog` package and you can use it right-away (this is known as the "batteries included" approach).

As we will see, this design choice complicates things a lot, for the user as well as the package maintainer. The thing is: this package isn't entirely honest about its dependencies. It contains code for all kinds of things, but all this code needs many different extra things to be able to function correctly. For instance the `MongoDBHandler` needs the `mongo` PHP extension to be installed. Now when I install this package, Composer does not verify that the extension is installed, because it is listed as an optional dependency (`ext-mongo`, under the `suggest` key) in the package definition file:

```
{
  "name": "monolog/monolog",
  ...
  "require": {
    "php": ">=5.3.0",
    "psr/log": "~1.0"
  },
  ...
  "suggest": {
    "mlehner/gelf-php": "Send log messages to GrayLog2",
    "raven/raven": "Send log messages to Sentry",
    "doctrine/couchdb": "Send log messages to CouchDB",
    "ext-mongo": "Send log messages to MongoDB",
    "aws/aws-sdk-php": "Send log messages to AWS services"
    ...
  },
  ...
}
```

So if I don't have the `mongo` extension installed in my development environment, yet I install the `monolog/monolog` package, and I want to use its `MongoDBHandler` for storing my log messages in a Mongo database, I have to *manually* add `ext-mongo` as a dependency to my own project:

```
{
    "require": {
        "ext-mongo": ...
    }
}
```

The first blocking issue for me is that I don't know which version of the `mongo` PHP extension I need to install to be able to use the `MongoDBHandler`. There is no way to find out, other than to just try installing the latest stable version and *hope* that it is supported by `MongoDBHandler`. I will only know this for sure if the `MongoDBHandler` comes with unit tests that fully specify its behavior. Then I could run the tests and

see if they pass with the specific version of `ext-mongo` that I just installed. This is my first objection to the design choice of making the `MongoDBHandler` part of the core Monolog package.

The second objection to this approach is that it forces me to add the `mongo` extension to the list of dependencies *of my own project*. This is very wrong, since it is not a dependency of *my* project but of the `monolog/monolog` package: it is a dependency of a *class inside that package*. My own project might not contain any code related to MongoDB at all, yet I have to require the `mongo` extension in my project because I want to use the `MongoDBHandler` class.

So the `monolog/monolog` package is not able to handle its dependencies well, and I have to do this myself. But this is quite contrary to the reason that I use a dependency or package manager and install each of *my* dependencies as vendor packages. I want packages to *fully take care of their own dependencies*: it is not my responsibility to know the dependencies of each class inside a package and to guess which ones I need to manually install to be able to use them. Furthermore, I should not be the one who needs to find out which version of a dependency is compatible with the code in the package. This is the task of the package maintainer.

But why didn't the maintainer of `monolog/monolog` add the `ext-mongo` dependency to the list of required packages then? Well, imagine a developer who only wants to use the `StreamHandler`, which just adds log messages to a file on an actual filesystem. They don't need a Mongo database nor the `mongo` extension to be available. If `ext-mongo` would be a required dependency, installing the `monolog/monolog` package would force them to also install the `mongo` extension, even though the code that really needs the extension will never be executed in their environment. This is why `ext-mongo` is just a *suggested dependency*.

But this reasoning applies to each of the specific handlers which the `monolog/monolog` package supplies. And nearly all handlers will be used exclusively by any particular user, which means that the package violates the *Common reuse principle* an equal number of times. For any handler in the package, the other handlers and their optional dependencies will probably never be used at the same time.

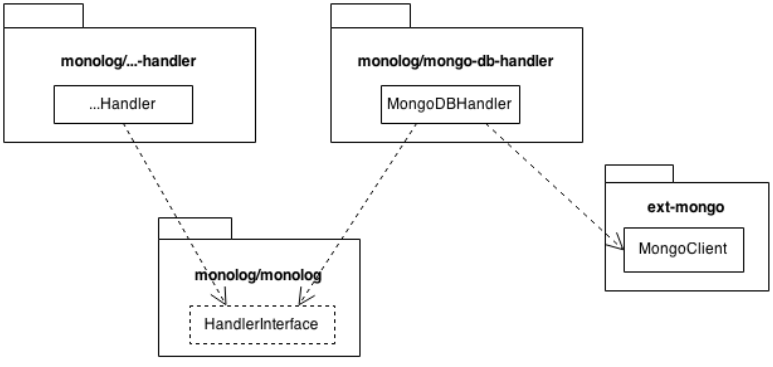
Suggested refactoring

The solution to this problem is easy: split the `monolog/monolog` package. Each handler should have its own package, with its own *true* dependencies. For example, the

`MongoDBHandler` would be in the `monolog/mongo-db-handler` package:

```
{
    "name": "monolog/mongo-db-handler",
    "require": {
        "php": ">=5.3.2",
        "monolog/monolog": "~1.6"
        "ext-mongo": ">=1.2.12,<1.6-dev"
    }
}
```

This way, each specific handler package can be explicit about its dependencies and the supported versions of those dependencies, and there are no optional dependencies anymore (think about this a little longer: how can a dependency be "optional" really?). If I choose to install the `monolog/mongo-db-handler`, I can rest assured that every dependency will be checked for me and that after installing the package, every line of code inside it is executable in my development environment.



`monolog` packages with explicit dependencies.

Previously the package definition file of `monolog/monolog` contained some useful suggestions as to what other extensions and packages the user could install to unlock some of its features. Now that the `ext-mongo` dependency was moved to the `monolog/mongo-db-handler` package, how does a user know they can use a Mongo database to store log messages? Well, this `monolog/mongo-db-handler` package itself could be listed under the suggested dependencies for the `monolog/monolog` package:

```
{
  "name": "monolog/monolog",
  ...
  "suggest": {
    "monolog/mongo-db-handler": "Send log messages to MongoDB",
    "monolog/gelf-handler": "Send log messages to GrayLog2",
    "monolog/raven-handler": "Send log messages to Sentry",
    ...
  },
  ...
}
```

A package should be "linkable"

Let's take one last look at the `MongoDBHandler` as it is currently still a part of the `monolog/monolog` package. We already concluded that after installing the package, you would not be able to use this class without also installing the `mongo` PHP extension first. If we don't do this, and we try to use this specific handler, we would get all kinds of errors, in particular errors related to classes that were not found. However, the code inside the `MongoDBHandler` is strictly correct, it just doesn't work in the context of this project.

We need to make a conceptual division here when it comes to correctness, a division that is not often made by PHP developers. In many programming languages problems with the code can occur at *compile* time or at *link* time. Compiling code means checking its syntax, building up an abstract syntax tree and converting the higher level code to lower level code. The result of the compile step are object files, which need to be linked together, in order to be executable. During the link process, references to classes and functions will be verified. If a function or class is not defined in any of the object files, the linker produces an error.

One of the characteristics of PHP is that it has no link process. It compiles code, yes, but if a class or a function does not exist, it will only be noticed at runtime, and even then in most cases very late.

I strongly believe that even though PHP allows us to be very flexible in this regard, we must teach ourselves to think more in a "compiled language" way. We have to

ask ourselves: would this code compile? Definitely, otherwise it would just be malformed code. And regarding every explicit, non-optional dependency of my package: would this code "link"? The answer would be "No" if the `MongoDBHandler` stays inside the `monolog/monolog` package, which has the `mongo` extension as a suggested dependency. If we move the `MongoDBHandler` to its own `monolog/mongo-db-handler` package, which has an explicit dependency on `ext-mongo`, the answer would be "Yes", as it should be.

Cleaner releases

There is one last characteristic of the `monolog/monolog` package that I'd like to discuss here, which again points us in the direction of creating separate packages for each of the specific handlers.

As we saw in the chapter about the *Release/reuse equivalence principle* it is important for a package to be a good software product. One of the important characteristics of good software is that new versions don't cause backward compatibility breaks. However, the `MongoDBHandler` in the `monolog/monolog` package shows clear signs of a struggle for backward compatibility:

```
...

class MongoDBHandler extends AbstractProcessingHandler
{
    ...

    public function __construct($mongo, $database, $collection, ...)
    {
        if (!($mongo instanceof MongoClient
            || $mongo instanceof Mongo)) {
            throw new InvalidArgumentException('...');
        }

        ...
    }
```

```
    ...
}
```

The first constructor parameter `$mongo` has no predefined type at all. Instead the validity of the argument is being checked inside the constructor and this validation step allows us to use two different kinds of `$mongo` objects. It should be either an instance of `MongoClient` or an instance of `Mongo`. Both are classes from the `mongo` PHP extension, but the `Mongo` class is deprecated since version `1.3.0` of the extension.

So now there is an ugly `if` clause inside the constructor of this class which prevents the `$mongo` argument from being strictly typed, even if I have the latest version of the `mongo` extension installed in my development environment. This should not be necessary. I'd like the handler to look like this instead:

```
...

class MongoDBHandler extends AbstractProcessingHandler
{
    ...

    public function __construct(MongoClient $mongo, $database, ...)
    {
        // no need for extra validation

        ...
    }

    ...
}
```

But if we remove the `if` clause, this class would be useless for people who have an older version of `mongo` installed on their system.

The only way to solve this dilemma is to create extra branches in the version control repository of the `monolog/monolog` package: a branch for version ranges of MongoDB that should receive special treatment, e.g. `monolog/monolog@mongo_db_-older_than_1_3_0` and `monolog/monolog@mongo_db`. Of course, this will soon end in

a big mess. The `monolog/monolog` package has many more handlers that may require such a treatment.

Let's fast-forward to the already suggested solution of moving the `MongoDBHandler` to its own package:

```
{
    "name": "monolog/mongo-db-handler",
    "require": {
        "php": ">=5.3.2",
        "monolog/monolog": "~1.6"
        "ext-mongo": ">=1.2.12,<1.6-dev"
    }
}
```

This `monolog/mongo-db-handler` package is hosted inside a separate repository, so it does not need to keep up with the versions of the core `monolog/monolog` package. This means it's possible to add branches corresponding to different versions of the `mongo` extension, for instance you could have a `1.2` branch and a `1.3` branch, corresponding to the version of the `mongo` extension that is supported. Then someone who has version `1.2` of the `mongo` extension installed could add this to their project's package definition file:

```
{
    "require": {
        "monolog/monolog-db-handler": "1.2.*"
    }
}
```

Someone who has the latest version of the `mongo` extension would simply choose `"~1.3"` as the version constraint.

Splitting the package based on its (optional) dependencies is not only advantageous to the user of the package. It will also help the maintainer a lot. They can let someone else maintain the MongoDB-specific handler package, someone who already keeps a close eye on the `mongo` extension releases. This person does not have to be able to

modify the main `monolog/monolog` package. This package automatically becomes a more stable package, because it has less reasons to change (see also the next chapter, about the *Common closure principle*). This is by itself a good thing for its users, who don't need to keep track of every new version that is released because of a change in one of the handlers they don't use.

Bonus features

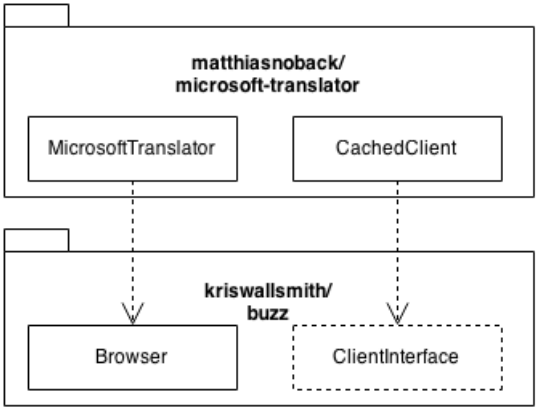
We have looked at obvious and non-obvious feature strata and why these are characteristics of packages that violate the *Common reuse principle*. Sometimes features are not really strata, but single classes which nevertheless don't belong inside a package. In this particular example, it is the `CachedClient` class from the `matthiasnoback/microsoft-translator`[38] package I created myself. It contains the MicrosoftTranslator[39], library which can be used for translating text using the Microsoft (Bing) Translator API. The translator depends on the `kriswallsmith/buzz`[40] package which contains the Buzz[41] browser. Buzz is a simple HTTP client and my package uses its `Browser` class to make HTTP requests to the Microsoft OAuth and Translator APIs, as you may guess by its (stripped) directory structure:

```
.
├── Buzz
├── Exception
├── MicrosoftOAuth
└── MicrosoftTranslator
```

While I was developing this library, I realized that my application might make many duplicate calls to the Microsoft Translator API. For instance it would ask many times to translate the word "Submit" to Dutch. And even though every time this would trigger a new HTTP request, the response from the API would always be the same. In order to prevent these duplicate requests I decided to add a caching layer to the package and I thought it would be a good idea to do this by wrapping the Buzz

[38] https://packagist.org/packages/matthiasnoback/microsoft-translator
[39] https://github.com/matthiasnoback/microsoft-translator
[40] https://packagist.org/packages/kriswallsmith/buzz
[41] https://github.com/kriswallsmith/Buzz

browser client in a `CachedClient`[42]. The `CachedClient` would analyse each incoming request, and look in the cache if it had already made this request before. If so, the cached response would be returned, otherwise the request would be forwarded to the real Buzz client, and afterwards the fresh response would be stored in the cache.



Dependency diagram of the `microsoft-translator` package.

Though at the time I thought the design of this library was pretty good, I would now try to eliminate the dependency on Buzz. There is nothing so special about Buzz that this package would *really* need it. In fact, all it needs is "some HTTP client". So I would define an interface for an HTTP client (`HttpClientInterface`) and then create a separate package, called `matthiasnoback/microsoft-translator-buzz`, which would provide an implementation of my own `HttpClientInterface` that uses Buzz.

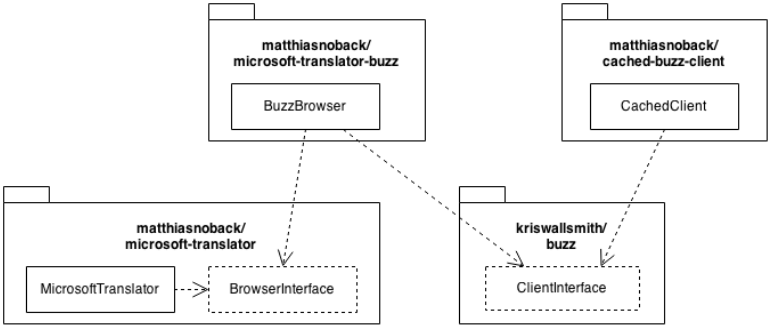
But there is some other thing that's wrong with the design of this library: it contains this highly useful `CachedClient` class. Since it is indeed so very useful, (almost) every user of this package will use this `CachedClient` class together with the other classes in the package. So there is no immediate violation of the *Common reuse principle* here. However, suppose that your project already depends on the `kriswallsmith/buzz` package and you need a cache implementation for the Buzz browser client. The `matthiasnoback/microsoft-translator` package already contains such an implementation so you would simply install it and use only its `CachedClient` class, and no other class from the same package. Now this makes it crystal-clear that the package does indeed violate the *Common reuse principle*: if you use a class from this package,

[42] https://github.com/matthiasnoback/microsoft-translator/blob/master/src/MatthiasNoback/Buzz/Client/CachedClient.php

you will not use *all* the other classes.

Suggested refactoring

As you may have guessed, the solution to this problem would be to extract the `CachedClient` class and put it in another package, `matthiasnoback/cached-buzz-client`, which has nothing to do with the Microsoft Translator and just depends on `kriswallsmith/buzz`. That way, anybody is able to install just this package in their project and use the cache layer for Buzz clients. Even better: this package could evolve separately from the `microsoft-translator` package. Other people may contribute to it by adding features or fixing bugs. These enhancements would become available for everyone who depends on the `matthiasnoback/cached-buzz-client` package.



Dependency diagram of the `microsoft-translator` package after moving the `CachedClient` class out.

Sub-packages

The `CachedClient` class we just discussed was a nice example of a "bonus feature" that consists of just a single class. It seems overkill to install an entire package with many classes in your project, just to use one class. But of course, this is a sliding scale: which number or percentage of classes would be acceptable for a package to remain intact and not be split into multiple other packages?

An interesting example of a package that contains a group of classes that can be reused together, without the rest of the package actually being needed, is the `friendsofsymfony/rest-bundle`[43] package. It contains the `FOSRestBundle`[44], an

[43] https://packagist.org/packages/friendsofsymfony/rest-bundle

[44] https://github.com/FriendsOfSymfony/FOSRestBundle

extension for Symfony which helps you create REST APIs. It has several *feature strata*, but one is particularly interesting: it seems to live its own life, separate from other features of this bundle that are somewhat related to each other. This feature is called the `QueryParamFetcher`[45]. It is used to validate query and request parameters (for instance by asserting that "the `page` query parameter is a positive integer").

```
.
├── Controller
│   └── Annotations
│       ├── QueryParam.php
│       └── RequestParam.php
├── Decoder
├── EventListener
├── Request
│   ├── ParamFetcherInterface.php
│   ├── ParamFetcher.php
│   ├── ParamReaderInterface.php
│   ├── ParamReader.php
├── Response
├── Routing
└── View
```

The `QueryParamFetcher` was added to the bundle because the Symfony framework does not provide such a validation framework. I think the framework should indeed have offered this functionality itself, since many a mistake is being made by trusting query parameters, without sanitizing and validating them sufficiently. So it is very nice that FOSRestBundle provides this functionality. But the problem is: if I'd like to use it in my own application, I need to add the `rest-bundle` package as a dependency, even though I use nothing else from the package. This is bad, because nevertheless I need to load the bundle into my Symfony project and I need to configure it, which may have side-effects on my application (even though the bundle maintainers have minimized that chance).

It would only be too easy to move the group of classes to another package, e.g. `query-param-fetcher`. The `rest-bundle` package itself may then depend on this new

[45] https://github.com/FriendsOfSymfony/FOSRestBundle/blob/master/Request/ParamFetcher.php

package, but also everybody like me who'd like to use this great feature separately would be able to do so.

Conclusion

We have found out many things about the *Common reuse principle* and by now it should be clear that there are some good reasons for splitting packages. Those reasons have advantages for both users and maintainers. A package that adheres to the *Common reuse principle* has the following characteristics:

- It is coherent: all the classes it contains are about the same thing. Users don't need to install the package just to use one class or a group of classes.
- It has no "optional" dependencies: all its dependencies are true requirements, they are mentioned explicitly and have sensible version ranges. Users don't need to manually add extra dependencies to their project.
- They make use of dependency inversion to make dependencies abstract instead of concrete.
- As an effect, they are open for extension and closed for modification: adding or modifying an alternative implementation does not mean opening the package, but creating an extra package.

Guiding questions

In order to help you decide for each class whether or not it should be in the package you are working on, I have created some guiding questions. In practice I tend to just create the class inside the package I am already working on. Afterwards I may decide to move it to another package, based on some of the thoughts that have been made explicit by asking myself these questions:

- Does the class introduce a dependency? If it does, is it an optional/suggested dependency? Then you have to create a new package containing this class, explicitly mentioning its dependency.

- Would the class be useful without the rest of the package? If it would, then you have to create a new package to enable users to depend on the former, and/or the latter package.

Asking yourself these two questions, and following the advice they give, will automatically divide your packages by dependency and functionality. These two concepts are at the same time the main reasons why people will depend on one package, and not on the other: whether or not it introduces too much or too little of the functionality they need, and whether or not its dependencies are compatible with the dependencies of their own projects. If a package's dependencies or features don't match with the requirements of the user, they will not install the package.

When to apply the principle

You can apply the *Common reuse principle* at different moments in the package development lifecycle, for instance when you are creating a new package for existing classes. The first thing you will need to do is group the classes that are *always used together* and put them in a package. When you need class `A` and *it* needs class `B`, it would be a bad idea to put these classes in different packages, `package-a` and `package-b`. Separating these classes would require a developer who would like to use class `A` to install both `package-a` and `package-b`.

But the *Common reuse principle* should also constantly be applied when you are adding new classes to an existing package. You need to check if the new class you are about to add will *always* be used together with all the other classes in the package. Chances are that by adding a new class to a package you are adding features to the package which are not used by everybody who would require the package as a dependency in their project.

When to violate the principle

As we shall see in the next part of this book, which contains many examples of real-life package design, the *Common reuse principle* is a principle that can only be maximized. You can not *always* follow it perfectly. As you can imagine, there are times when you may choose to put two classes in one package, that *can* be used

separately. You would thereby strictly violate the principle, but there may be good reasons to do so. First of all: convenience. Every package you create needs some extra care. It requires time and energy from you as the package maintainer and there is a limit to how many packages you can or want to maintain.

Another reason to violate the principle is that you may know all about your target audience. When you know that almost all of the developers who use your package are using it inside a project built using the Laravel framework you can follow the *Common reuse principle* less strictly and add some classes to your package which may only make sense when someone uses the full-stack Laravel framework. These classes would normally belong in a separate package because they could in theory be used separately, but in practice they will always be used together, which allows you to put them in the same package.

Why not to violate the principle

In most cases however there is the following good reason to *not* violate the *Common reuse principle*: every class that is part of your package is susceptible to change. Maybe one of its methods contains a bug, or some of its functionality needs to be changed. Maybe its interface needs to be modified and a backward compatibility break will be introduced. As a user of the package, you need to follow all the changes and decide if and when to upgrade to a new version. This takes time, because after upgrading a package you need to check all the parts of your project that use classes from the package. Maybe you have some automated tests for this, maybe not.

However, you can not choose *not* to upgrade. You need to take care that your project depends on the latest stable versions of all packages, to prevent (future) problems. If the package you depend upon is big, contains lots of classes and is related to all kinds of things, upgrading it will be quite a problem. There are many points of contact between the code in your project and the code inside the package, which means that changes will have many side effects.

On the contrary, when the package you depend upon is small, it will be easier to track the changes and less painful to upgrade the package. There will be less points of contact, and the chance that an upgrade will break your project is consequently much smaller.

So you greatly help the users of your package when you keep your package small and only put classes in it that they will actually use. Then they will be able to upgrade their dependencies fearlessly.

The Common closure principle

In the previous chapter we discussed the *Common reuse principle*. It was the second principle of package cohesion and it told us that we should put classes in a package that will be used together with the other classes in the package. If a user would want to use a class or a group of classes separately, this would call for a package split.

The third principle of package cohesion is called the *Common closure principle*. It is closely related to the *Common reuse principle* because it gives you another perspective on granularity: you will get another answer on the question which classes belong together in a package and which don't. The principle says that:

> The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.

So "common closure" actually means *being closed* against the same kinds of changes. With regard to the code in a package, this means that when something needs to change, it is likely that the change that is requested will affect only one package. Conversely when a requirement changes and it affects one package, it will likely affect all classes inside that package.

The primary justification for this principle is that we want change to be limited to the smallest number of packages possible. People who have added your package as a dependency to their project will likely keep track of new releases of that package, to keep all the code in their project up-to-date. When a new version of the package becomes available, the user will upgrade their project to require the new version. But they only want to do so if the changes *you* made to the package have something to do with the way the package is used in *their* project, since every upgrade requires them to verify that their code still works correctly with the new version of your package.

As a package maintainer you should follow the *Common closure principle* to prevent yourself from "opening" a package for all kinds of unrelated reasons. It helps you

prevent bringing out new releases that are irrelevant to most of your users. With this goal in mind the principle advises you to put classes in different packages if they have different reasons to change.

These reasons can be divided into several types, each of which we will discuss in the following sections.

A change in one of the dependencies

Consider a package that still makes use of the deprecated PHP `mysql_*` functions for interaction with a MySQL database. You decide that you want to remove all occurrences of these functions in the package and instead make use of the much better alternative, PDO[46]. You start working on this, but it soon becomes clear that of all the classes inside the package only two classes need to be modified. This means that the package violates the *Common closure principle*: the files inside the package are not all closed against the same kinds of changes. Only some classes are affected by the recently changed requirements, while many classes are not. In other words: most classes are closed against a change related to database management, while some are not.

The classes that were changed together should have been in a separate package. This package would contain all the classes that would together be affected by the same kinds of changes. It would contain classes responsible for sending data to and receiving data from a MySQL database. Conversely the other packages would contain all the classes that have nothing to do with the concrete way in which a database is being accessed.

Assetic

A real-world example of a package which contains classes that are closed against many different kinds of changes is the `kriswallsmith/assetic`[47] package which contains the Assetic[48] library used for managing web assets (e.g. combining, compressing and filtering JS and CSS files).

[46] http://www.phptherightway.com/#databases

[47] https://packagist.org/packages/kriswallsmith/assetic

[48] https://github.com/kriswallsmith/assetic

Looking at its directory structure (I've removed many files to make the picture clearer), you can see that it contains the core classes as well as many classes called "filters", which are used to modify the contents of an asset file (e.g. to compile Less to CSS, compress JS, etc.).

```
.
└── Assetic
    ├── Asset
    ├── Cache
    ├── Exception
    ├── Factory
    ├── Filter
    │   ├── CoffeeScriptFilter.php
    │   ├── CompassFilter.php
    │   ├── CssEmbedFilter.php
    │   ├── CssImportFilter.php
    │   ├── CssMinFilter.php
    │   ├── CssRewriteFilter.php
    │   ├── DartFilter.php
    │   ├── EmberPrecompileFilter.php
    │   ├── FilterCollection.php
    │   ├── FilterInterface.php
    │   ├── GoogleClosure
    │   ├── GssFilter.php
    │   ├── HandlebarsFilter.php
    │   ├── JpegoptimFilter.php
    │   ├── JpegtranFilter.php
    │   ├── JSMinFilter.php
    │   ├── JSMinPlusFilter.php
    │   ├── LessFilter.php
    │   ├── LessphpFilter.php
    │   ├── OptiPngFilter.php
    │   ├── PackagerFilter.php
    │   ├── PackerFilter.php
    │   ├── PhpCssEmbedFilter.php
    │   ├── PngoutFilter.php
```

```
|   ├── RooleFilter.php
|   ├── Sass
|   ├── ScssphpFilter.php
|   ├── SprocketsFilter.php
|   ├── StylusFilter.php
|   ├── TypeScriptFilter.php
|   ├── UglifyCssFilter.php
|   ├── UglifyJs2Filter.php
|   ├── UglifyJsFilter.php
|   └── Yui
└── Util
```

At first glance this package clearly violates the *Common reuse principle*. It is obvious that not all classes in this package will be used together, since many of the filters are not used at the same time by the same user. Maybe each user really requires just two or three of them.

But when we switch our perspective from the user to the package maintainer and try to apply the *Common closure principle*, we should notice that these classes are not all closed against the same kinds of changes. For example if anything changes with regard to the way the Less compiler works, or the type of input that the `RooleFilter` expects, a change will be made in just one or two classes inside the package. Afterwards the package maintainer needs to release a new version of the *entire package* to make the changes available to all users. This will require people to upgrade their project (and probably also bring in many unrelated changes from the repository), which may or may not have unwanted side-effects.

To make Assetic comply with the *Common closure principle* its maintainer should create separate packages for each filter. Each filter-specific package will only be sensitive to specific kinds of changes (namely changes in its own dependencies, like the Less compiler). The main `assetic` package will not be affected by any such change. This means that when one of the specialized filters changes, only a new version of the filter-specific package needs to be released. In such cases there will be no need to release a new version of the main `assetic` package.

Adhering to the *Common closure principle* with regard to changes made in (optional) dependencies will thus prevent unnecessary package releases. This will effectively

lower the burden on your users. They don't need to upgrade their dependencies because of changes that don't apply to them.

A change in an application layer

The first kind of change we discussed was somewhat external to the package: it was caused by a change in one of its dependencies, be it a package, an extension or the programming language itself. The second kind of change is related to something called *architectural layers*.

Layers are a way to apply the *Single responsibility principle* to an application. A well-known method of layering an application is by separating the model from the view and putting controllers in between. The controllers will regulate traffic from the front controller down into the model and back to the view.

Depending on who you to talk to, other divisions are being made between these so-called application layers. There may be a domain layer, but it should not be responsible for storing data. So there is also an infrastructure layer. That layer also handles communication with external services, like a search engine, a job queue or a key/value storage.

However, the general idea of a layer is that it consists of a number of classes that are concerned with the same kind of thing, like storing something, preparing something for presentation to a user, calculating something related to the business domain, etc. This means that layers are basically an organizing principle for classes. But so are *packages*!

Layers are in the first place introduced in an application to prevent you from modifying files all over the place. When you use a layered architecture, it is immediately clear that switching from one persistence tool to another would not require you to modify lots of classes in lots of different places. The classes that need to be modified for such a change are all in the infrastructure layer and you will find nothing database-related in for instance the presentation layer.

Organizing project code into packages has the same goal: you want to concentrate code related to a certain responsibility in one place. The reason is that you want to modify the smallest number of packages possible when requirements change. This is

achieved by following the *Common closure principle* which is equally applicable to packages in general as it is to layers.

As an organizational unit, layers are much less tangible than packages are. Code is written in methods, which are part of classes, which are part of packages. Packages are not part of layers in the same way as classes are part of packages. Nevertheless, packages related to for instance the presentation layer, together constitute the presentation layer.

So the classes that constitute a layer, are placed inside a package, just like the other classes in an application. And the *Common closure principle* says that for classes to be together inside a package means that they should be closed against the same kinds of changes. This is equally applicable to classes that are related to a particular layer. For instance classes related to infrastructure should not be in package together with classes related to the presentation layer, since then they would not be closed against the same kinds of changes: some would be closed against changes in the presentation layer and some would be closed against changes in the infrastructure layer.

This implies that packages that contain classes that are related to the same thing and therefore susceptible to comparable changes, should also be separated according to the application layers they are part of. For example when you have a package that is concerned with calculating financial data (which is a responsibility of the domain layer), that package should not at the same time contain classes related to rendering that data as an HTML table (which is a responsibility that belongs to the presentation layer).

FOSUserBundle

Let's take a look at a real-world example of this: the `friendsofsymfony/user-bundle`[49] which contains the FOSUserBundle[50]. It can be used in a Symfony application as a way to quickly set up user management. Out-of-the-box it provides several useful things that almost every web application needs:

- Persistent users and user groups
- A password reset page

[49] https://packagist.org/packages/friendsofsymfony/user-bundle
[50] https://github.com/FriendsOfSymfony/FOSUserBundle/

- A registration page
- A change password page
- User management from the command-line
- ...

Looking at the directory structure you can roughly recognize these features in the files that are present:

```
.
├── Command
│   ├── ActivateUserCommand.php
│   ├── ChangePasswordCommand.php
│   ├── CreateUserCommand.php
│   ├── DeactivateUserCommand.php
│   ├── DemoteUserCommand.php
│   ├── PromoteUserCommand.php
├── Controller
│   ├── ChangePasswordController.php
│   ├── GroupController.php
│   ├── ProfileController.php
│   ├── RegistrationController.php
│   ├── ResettingController.php
├── Doctrine
│   ├── CouchDB
│   ├── MongoDB
│   └── Orm
├── Document
├── Entity
├── Event
├── EventListener
├── Form
├── Mailer
├── Model
├── Propel
└── Resources
```

```
├── translations
└── views
```

This is another example of a package that violates the *Common reuse principle* because it is quite possible that someone would want to use just the model classes that are provided by this package in their own project, which is not a Symfony application. There is no way to do it without pulling in the entire `user-bundle` package, which is otherwise quite useless to them.

The same objection to the particular lack of cohesion of this package arises when we consider the *Common closure principle* with regard to application layers. It is immediately clear that this package contains code related to all kinds of layers. Therefore a change in requirements related to the user model would result in just a few modifications within this package. Many files will remain untouched. The same goes for a visual makeover of the web pages provided by this package. To make the new templates available for everyone, a new release needs to be issued, which is irrelevant to everyone who uses their own templates. Because some other things may have changed in the new release, users would still need to verify that their application is not broken after upgrading their dependencies.

Following the *Common closure principle* with respect to application layers would thus require the package maintainer to split this package according to the different layers for which it contains code and other resources. That way only relevant changes will cause users to upgrade their dependencies. At the same time it allows users to replace one layer implementation provided by the package maintainer by one of their own, without the need to pull unused code into their projects.

The problem with plugins for frameworks

Most frameworks for web applications offer an all-round solution for each of the traditional application layers (model, view and controller). Although this idea is quite outdated and many people have been experimenting with other types of layers and architectures, still an application can be reduced to these main layers. You always need some model of the domain of your business. You always need a way to show something to a user of your application and you always need something that the framework can invoke in order to put things in motion.

Whenever a new framework starts to attract attention from developers, everything that already exists will be recreated to work well with that particular framework. And since the framework provides a standard way of doing things in each layer, you will end up with packages…

- For Symfony, called "bundles", that use Twig for views and Doctrine ORM for persistence,
- For Laravel, called "packages", that use Blade for views and Eloquent ORM for persistence,
- For Zend Framework, called "modules", that use Zend_View for views and Zend_Db for persistence,
- …

This does not make sense. It means that reuse is actually obstructed, because of the narrowness of the area of reuse. If the maintainers of these packages would care about package design principles, they would split their packages according to responsibilities, with respect to dependencies as well as application layers and subject matter. This way, there would only need to be one package that modeled the domain. There would be several packages that implemented persistence for domain objects using different kinds of database and persistence libraries. And there would also be several packages that provided a presentation layer that works with different templating engines. Separating packages like this would make true reuse possible (at least between projects that use the same programming language).

A change in the business

We have considered layers as a way to partition your packages. Each package should contain code that is related to one application layer. That way, when a requirement changes with regard to other layers, the package will remain untouched.

When you follow the *Common closure principle* with regard to layers, you end up with packages that all have just one of the big responsibilities (like modeling things, presenting things, etc.). Nevertheless these packages would still contain code that is likely to be modified for entirely different reasons. A package that contains all the code of the domain layer, would contain classes that model a person, an article, an address, a payment, etc. Whenever the requirements of the domain layer changes because of business reasons ("we need to support another payment type"), only one file in the domain package will be modified, and the other ones will stay as they were before the requirements changed.

This of course looks like another violation of the *Common closure principle* since the classes in a package should be closed against the same kinds of changes. A package entirely dedicated to the domain layer is closed against all different kinds of changes. When you need to decide how to group classes into packages you therefore need to consider business changes as one kind of change to close classes against.

Sylius

Let me end this section with a *good* example of a project that separates packages by domain. It is the Sylius project[51] which offers one big `sylius/sylius`[52] package containing multiple smaller packages (using so-called Git sub-tree splits), nicely divided by subject matter:

[51] http://sylius.org/

[52] https://packagist.org/packages/sylius/sylius

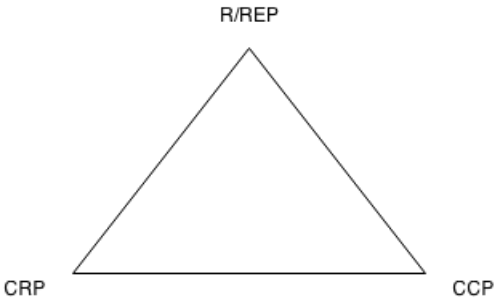
```
SyliusAddressingBundle
SyliusCartBundle
SyliusFlowBundle
SyliusInventoryBundle
SyliusOmnipayBundle
SyliusOrderBundle
SyliusProductBundle
SyliusPromotionsBundle
SyliusResourceBundle
SyliusSettingsBundle
SyliusShippingBundle
SyliusTaxationBundle
SyliusTaxonomiesBundle
SyliusVariableProductBundle
```

Unfortunately each of these packages does not follow the previous guideline to separate responsibilities based on architectural layers. But when it comes to the problem domain these packages are all nicely separated from each other. This way the maintainer will be able to limit modifications necessitated by changed requirements to just one or two packages at a time.

The tension triangle of cohesion principles

Before we continue with the next set of package design principles we will quickly discuss an interesting concept mentioned by Robert Martin in one of his training videos on cleancoders.com[53]. It is called the "tension triangle of cohesion principles".

[53] http://cleancoders.com/codecast/clean-code-episode-16/show

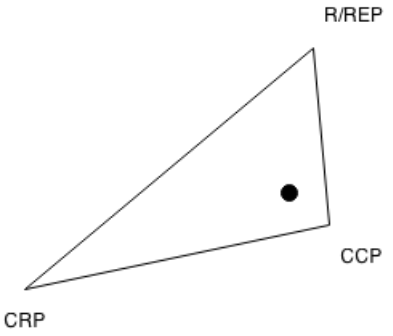


Tension triangle of cohesion principles

You can draw a triangle with in each corner one of the package cohesion principles. Then you can locate any package somewhere within that triangle. Moving a package to one of the corners means that it implements that principle maximally, while it neglects the other principles. Moving to the vertex opposite to one of the corners means that the package definitely does not follow the corresponding principle, but follows the other principles equally well.

What Robert suggests is that a package may move within the diagram. When you first start working on a project, packages may be primarily designed according to the *Common reuse principle*, later on you may choose to follow the *Common closure principle* more strictly because that will ease maintenance. Then in the end you may focus on really making the package reusable so that all the effort you put into it will give you an advantage in your next project (or will help other developers around the world to solve the problems you have solved already quite well before).

I'm not sure if the triangle of the three cohesion principles really is a *tension* triangle. Each of the cohesion principles can be *maximized*, but I don't think that means that the other principles would be *neglected*. They can be maximized at the same time. Following this reasoning it would be more useful to visualize the package as the center of an irregular triangle. The corners still stand for the three cohesion principles. Each corner will be further away from the package, the more that principle is maximized in the given package. For instance a package that requires some attention when it comes to the *Reuse/release equivalence principle*, follows the *Common reuse principle* pretty well, but lacks in common closure could be visualized like this:



Dynamic triangle of cohesion principles

I think such a dynamic triangle is a nice way to estimate the quality of the cohesion of a package at any moment in time. It is easy to notice when a package does not follow the *Reuse/release equivalence principle*, because then it is hard to add it as a dependency to your project and difficult to keep it up-to-date once you have managed to do so. It is also easy to verify how well the *Common reuse principle* is obeyed: when you feel like you have to pull in a lot of code, just to use one or two classes, then something is wrong. And finally if the package contains classes that have responsibilities in many different fields of expertise, the *Common closure principle* is not followed very well.

So the next time you are designing a package, or when you are re-evaluating the design of an existing package, draw a *dynamic tension triangle* to get a quick insight into the current state of the package. Then try to maximize each of the principles. The package will gradually move to the middle of the triangle, because finally none of the principles will be neglected anymore.

Principles of coupling

Coupling

Cohesion ties methods together in a class and it ties classes together in a package. This is a good thing. When you add such a package as a dependency to your project, you will not pull in a large amount of unused or irrelevant code.

Most classes however can not survive on their own: they have some kind of dependency and most likely even multiple dependencies. Maybe they need an instance of another class to delegate some of the work to. Or they *produce* instances of another class. In other words: many classes depend on other classes, which actually couples them to each other.

As we saw in the first part of this book, applying the SOLID principles to class design has a healthy effect on coupling between classes. As you may remember, according to the *Dependency inversion principle* a class should only depend on another class or interface that is abstract, not concrete. It should also not depend on lower-level classes, only on high-level classes. And according to the *Open/closed principle* a class should be open for extension, but closed for modification, which means in particular that its dependencies should be interchangeable without the need for modifying its code.

When we discussed the *Dependency inversion principle*, we already briefly considered the situation in which one class depends on a class in another package. It is likely that you will encounter this situation because by following the cohesion principles, you end up with many small packages. But if there is no package that *uses* those small packages, splitting them will have been a useless exercise altogether.

So class coupling is not limited by package boundaries. A class from one package that depends on a class from another package actually introduces a new level of coupling, namely *package coupling*.

If you have worked with packages and a dependency or package manager you already know that package coupling can go wrong in many ways. Often you get

into trouble because of incompatible dependency versions. Or somehow circular dependencies occur. Maybe instable packages that are liable to change cause your own project to break frequently. And probably some of your dependencies have instable dependencies themselves, the effects of which ripple through to your own code.

Because of these problems, we are in need of some guiding principles which help us design packages that have good dependencies. We need packages that can be trustworthy dependencies of other packages and projects. The relevant package design principles are called "principles of coupling" and we will discuss them in the following three chapters.

The Acyclic dependencies principle

Coupling: discovering dependencies

As I explained in the introduction to the Cohesion principles all programmers develop a sense of "belonging together". But next to this intuition with regard to cohesion, programmers also have a nose for coupling. Looking at a piece of code they will be able to figure out what it is coupled to. As their career progresses they will develop an ever stronger "coupling radar" by figuring out the actual *dependencies* of any piece of code.

Looking at some code, you can ask yourself: on which other *things* does this code rely in order to be successfully executed? Thinking long and hard about this question reveals some obvious dependencies, but probably also some less obvious, or indirect dependencies.

Consider the `Kernel` class defined in the following piece of code:

```
namespace SomeFramework;

class Kernel
{
    public function __construct(EventDispatcher $eventDispatcher)
    {
        ...
    }
}
```

In order to be used successfully in an application, the `Kernel` class depends on:

- The `EventDispatcher` class. It gets an instance of this class injected as a constructor argument.
- It needs the PHP interpreter in order to be run at all. More specifically, the version of the PHP interpreter should be at least 5.3, since the class resides in a namespace, which is not supported by earlier PHP versions.

We could go much, much further in specifying dependencies. A PHP interpreter depends on an operating system, running on a computer, which needs power and should be accessible to you as the user of this software. It should therefore be in this world, this universe (well, maybe your code is even cross-universe compatible, who knows?), and so on.

I agree with you, this is taking things too far. However, merely looking at the classes used in the code and the PHP version that is required to run the code is not sufficient in most cases. Some code may rely on a database server being up and running and reachable from the server on which the code is being executed. Or maybe a certain amount of memory is required to run the code, or the user that runs the software should have certain filesystem rights, etc.

In this chapter and in the next two chapters we will discuss package dependencies. When we discuss the coupling principles for packages, we won't consider the above-mentioned physical dependencies of a package. We only take other units of code - external to the package - into consideration. These external units of code* can be actual packages (with or without a package definition file), language extensions (which are really a special kind of packages themselves), or other conglomerates of code that are necessary to run the code in a given package.

Different ways of package coupling

Let's first settle on the following convention: we call "the given package" the "root package". Starting with any root package, we can enumerate the dependencies of that package. Those dependencies can be any kind of package, or even a language extension, but we just call those dependencies "packages" too, for simplicity's sake.

Now we can make a list of ways in which the code in the root package introduces dependencies on other packages. As a package maintainer we need this list when we are collecting the names of required packages for our package definition file. We

also need this list of dependencies when we are going to apply the package coupling principles. When we don't know exactly in which ways the code in the root package introduces coupling to other packages, we won't know how to draw a dependency graph of the root package as part of a larger system of packages. We also won't know how to fix problems in the dependency graph.

Composition

We already looked at one particular way of coupling: a constructor argument of a class is type-hinted with the name of another class:

```
namespace RootPackage;

use OtherPackage\EventDispatcher;

class Kernel
{
    private $eventDispatcher;

    public function __construct(EventDispatcher $eventDispatcher)
    {
        $this->eventDispatcher = $eventDispatcher;
    }
}
```

The `Kernel` class is in the root package. When the `EventDispatcher` class is in another package (as it should be), this particular piece of code introduces package coupling. For example, the `kernel` package would depend on the `event-dispatcher` package. This is called a dependency *by composition*, because the pattern of storing one object inside another object for later use is called "composition".

Inheritance

Another way in which a class can be coupled to another class is by inheriting from it:

```
namespace RootPackage;

use OtherPackage\Controller;

class LoginController extends Controller
{
    ...
}
```

When the parent class `Controller` (or any of its parent classes) is in another package than the `LoginController`, inheritance introduces package coupling.

Implementation

Very much like inheritance, implementing an interface or extending an abstract class and implementing its abstract methods introduces coupling too:

```
namespace RootPackage;

use OtherPackage\RequestListener;

class IpBlocker implements RequestListener
{
    ...
}
```

Usage

Often coupling between classes occurs when an instance of one class simply *uses* an instance of another class, for instance as one of its method parameters:

```
namespace RootPackage;

use OtherPackage\NewRequestEvent;

class IpBlocker
{
    public function onKernelRequest(NewRequestEvent $event)
    {
        ...
    }
}
```

Creation

In the previous examples dependencies on classes outside the root package were out in the open, because they were part of the public interface (a parent class, an implemented interface and a type-hint of a function parameter). But there are also some more private ways of coupling. For instance, when one object creates other objects of a class inside another package:

```
namespace RootPackage;

use OtherPackage\ServiceContainer;

class Kernel
{
    public function boot()
    {
        $container = new ServiceContainer();

        ...
    }
}
```

Functions

The usual suspects for coupling are classes, but you shouldn't forget to take a look at the functions that are used in a package. Many functions are only available when a particular package or language extension has been installed, for instance the `curl` PHP extension:

```
class HttpClient
{
    public function send()
    {
        $ch = curl_init();

        ...
    }
}
```

Functions are only used in the private parts of a class (i.e. inside the body of its methods). So spotting these dependencies takes a bit more effort.

Not to be considered: global state

Code in a package often relies on a particular global state. The most obvious example is that each package implicitly relies on the presence of an autoloader which is able to (auto)load the classes inside the package.

Depending on global state should otherwise always be avoided, but even when it happens: we don't consider it a package dependency in this book, since we can't make it explicit inside the list of requirements of a package.

Visualizing dependencies

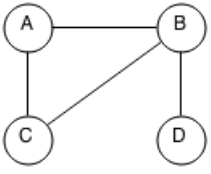
Before we can start to apply the coupling principles to our package design, we need to be able to visualize any current state of package coupling. When we do this, we only consider one system (i.e. application) at a time. We take all the packages inside that system and one by one consider them as a root package. We then use the above

list of coupling types (e.g. composition, inheritance, etc.) to extract a list of all the dependencies of classes in this package. As soon such a class dependency goes beyond the boundaries of the package itself, it should be marked as a *package dependency*.

The result of such an exercise is a list like this:

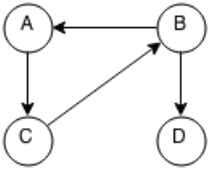
- Package `A` depends on package `C`
- Package `B` depends on package `A` and `D`
- Package `C` depends on package `B`

Packages and their dependencies, when written down like this, form a recipe for a graph, where each package is a *vertex* (node) and each dependency is an *edge* (line):



A graph with vertices as packages and edges as dependencies

Since dependencies have directionality (a package depends on another package, not automatically the other way around), we should convert this graph into a directed graph by simply adding some arrows to the lines:

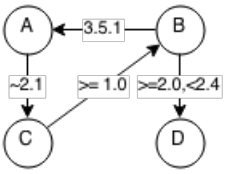


A directed graph

What's left out of these diagrams are self-referencing packages. These aren't of interest at this point, though I will briefly get back to this subject later. Almost all packages are self-referencing actually, because often a class in a root package uses another class or a function from the same package.

When it comes to package dependencies there is another important aspect that we need to take into consideration when we draw the dependency graph of packages

within a system: there are probably some version constraints with regard to the dependencies. For instance package `C` may require at least version `1.0` of package `B`. We can write these constraints as annotations in the graph.



A directed graph with annotations for version constraints

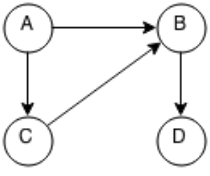
The *Acyclic dependencies principle*

With all the above acquired knowledge about package coupling it is only one small step towards the explanation of the *Acyclic dependencies principle*. The principle states that:

> The dependency structure between packages must be a directed acyclic graph, that is, there must be no cycles in the dependency structure.

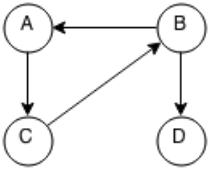
We already discussed how you can figure out the actual dependencies of all the packages in a system, then draw a *directed graph* of the outcome. The only missing piece of information is what an *acyclic* directed graph is.

A directed graph has no cycles if, starting from any vertex, there is no path that via any number of vertices leads back to the original vertex. Such a directed graph is called an *acyclic* graph. Translated to the language of dependency graphs: whichever package you take as the root package, by following the dependency arrows, you will not be able to return to the root package.



An acyclic directed graph: no cycles

Conversely, if a directed graph *has* cycles it means there is *at least one* vertex for which you can find a path that leads back to that same vertex. In terms of a dependency graph this means: there is at least one package which is the beginning of a path of subsequent dependencies which leads back to that same package.



A cyclic directed graph: one cycle

The *Acyclic dependencies principle* just says that when you draw your dependency graph it should look like an *acyclic directed graph*, meaning: it has no cycles.

Nasty cycles

As a programmer you are likely familiar with cycles, often caused by mistakes easily made, like this one:

```
$somethingThatIsAlwaysTrue = ...;

while ($somethingThatIsAlwaysTrue) {
    ...
}
```

Or something similar, like this:

```
for ($i = 0; $i < 100;) {
    // we forgot to increment...
}
```

But also:

```
class Node
{
    private $parent;

    public function getParent()
    {
        if ($this->parent) {
            return $this->parent;
        }

        // hmm
        return $this->getParent();
    }
}
```

An example of a common type of cycle that's not really a mistake:

```
class Desk
{
    private $programmer;

    public function __construct(Programmer $programmer)
    {
        $this->programmer = $programmer;
    }
}

class Programmer
{
    private $desk;

    public function __construct(Desk $desk)
    {
        $this->desk = $desk;
    }
```

```
}

$desk = new Desk($programmer);
$programmer = new Programmer($desk);
// ??
```

No matter how hard we try, we won't get this straight. It's a real circular dependency: We first need to instantiate a `Desk`, but in order to do that we need a `Programmer`, which needs a `Desk`, etc.

We commonly fix issues like these by first constructing an object, then injecting the dependency that caused the cycle:

```
class Desk
{
    private $programmer;

    public function setProgrammer(Programmer $programmer)
    {
        $this->programmer = $programmer;
    }
}

class Programmer
{
    private $desk;

    public function __construct(Desk $desk)
    {
        $desk->setProgrammer($this);

        $this->desk = $desk;
    }
}

$desk = new Desk();
$programmer = new Programmer($desk);
```

When you use this solution, you can be sure that at one point in time *at least one* of the objects is in an invalid state. In this case it's the `Desk` object. It has no associated `Programmer` until it has been assigned in the constructor of the `Programmer` itself.

As the Wikipedia entry on circular dependencies[54] states:

> Circular dependencies may [...] cause memory leaks by preventing certain very primitive automatic garbage collectors (those that use reference counting) from deallocating unused objects.

PHP luckily does not have such a "very primitive automatic garbage collector" anymore. Yet, circular dependencies can definitely cause memory problems, also known as *memory leakage*. Usually a garbage collector takes note of something called a `refcount` for variables. When a part of your application starts to use a variable, the `refcount` for that variable will be incremented with `1`. When the variable is not used by this particular part of the application anymore (because it is copied or moves out of scope), its `refcount` is decremented again. When the garbage collector notices a `refcount` of `0` for a variable, it will free the memory used to store that variable (probably not immediately though).

When two objects are dependent on each other, i.e. have circular dependencies, the problem is that the `refcount` will never be `0`: the two objects will always be referring to each other. Before PHP 5.3 the memory that was used to store such objects could never be freed (instead, this was only done at the end of a request or when the script died). In order to make PHP support long-running processes, this has been fixed[55].

Still, cycles are known to cause memory leakage from time to time, especially when objects are self-referencing:

[54] http://en.wikipedia.org/wiki/Circular_dependency

[55] http://php.net/manual/en/features.gc.collecting-cycles.php

```
class ServiceContainer
{
    private $services;

    public function __construct()
    {
        $this->services['service_container'] = $this;
    }
}
```

This example is taken from real life too - it was the way the Symfony service container made itself available as one of its services. This has been fixed[56]). It caused memory leakage because of that `refcount` that never became `0`.

Cycles in a package dependency graph

So, yes, cycles have some downsides. But the problems described above usually don't travel across package boundaries. In a way you could say that these problems are privately held by the package and should be fixed there. However, when a circular dependency between classes goes beyond the boundary of their containing package, it does becomes a *circular dependency between packages* and then some other, bigger problems arise.

Dependency resolution

In the first place, *resolving* circular dependencies can be really hard. When you have worked with *Dependency Injection* (DI) or *Inversion of Control* (IoC) *containers*, you know that they tend to give up when they encounter a cycle in the service definition configuration. If service `b` is needed to instantiate service `a`, but eventually it turns out service `a` also needs service `b` to be instantiated, DI containers don't have another option than to report the problem and stop trying to resolve the dependency hierarchy.

Apparently this problem of resolving a circular dependency also occurs when package managers try to resolve the dependencies of a given package. Once they

[56] https://github.com/symfony/symfony/commit/440322effc9e482241bc552b2cc6a682c6063d27

encounter a cycle, some of them will just fail. Others will still try to resolve the dependency graph and find the right versions, skipping or postponing the process for the cycles which it encounters.

Release management

So *resolving* circular package dependencies may not always be a problem. A bigger problem turns out to be *release management*.

Ask yourself: why do you create a package in the first place? You probably want to *reuse* some code. This means: you want several projects and/or libraries to be able to depend on that piece of code. The package is the best unit of reuse, since it allows you to share more than one class at the same time. It also enables you, using a combination of version control and package management techniques (as described in the chapter about the *Release/reuse equivalence principle*), to manage new releases of the code. Using semantic versioning to support your backward compatibility promise and corresponding feature and patch branches, you should be able to offer support for older versions of a package, while working on newer versions of the package. You want to add features (with the release of new minor versions), or do things in an entirely different manner (with the release of new major versions).

Say you have developed some code as part of a project. You are planning to extract that code into a reusable unit, a package. You move this particular piece of code to its own project. Then you release it as a package, allowing you to add the code back into the original project as one of its dependencies. From then on the package can start to live its own life. By creating new branches and bringing them under version control you can start working on a new version, without disturbing people who use the older version. You can also fix any bugs you encounter in the branch containing the older version, just like you would fix those bugs when the code was still part of the original project. Simply updating the dependency of that project to the latest patch version would prevent anything from breaking.

Often one of the reasons to start working on a new *major* version of your package is that one of the packages on which your package depends is going to release a new major version, which offers many new and better ways of doing things. This kind of thing happens all the time (think of packages that need to keep up with new major releases of a framework).

Let's say the current version of your package, package `A`, is `1.2.3`. `A` depends on package package `B`, which is currently at version `1.4.0`. The team of package `B` is working on a new version with all kinds of cool, new and therefore better ways of doing the same old thing. Now, as it happens, package `B` depends on package `C`, which provides some useful classes, but only in its next major version. Now, you can guess that releasing a new major version of `A` may take quite some time, because of the cascading dependencies. Well, the plot thickens: package `C` depends on package `A` again, and word has it that they are releasing a new version, which keeps `B` waiting to release their new major version too…

What a story. They are all waiting for each other! There are two solutions in this scenario. They have to coordinate their movements and release the new major versions of all the interdependent packages at the same time. This means there is a time when they have to force-release their package, without definitive knowledge that together they will work well. The other option is to only move in little, backward compatible steps, towards a new version and never make any leaps. But, to quote Anthony Ferrara[57]:

> […] every release adds more cruft for you to maintain. Over time this creates a halting effect on the code base involved that makes it nearly impossible to clean up and "make things better".

Is it all that bad?

It *will* be difficult to coordinate releases when a dependency graph has cycles, although this can be overcome by doing "kamikaze" releases, adding some backward compatibility measures and providing friendly version constraints for the package dependencies involved. In the end this may not cause you too much pain. The problem of circular package dependencies is actually much bigger with programming languages that have a build process. When dependencies of the build process are being resolved, a circular dependency may even prevent the entire build process from succeeding.

Still, there is the programmer's intuition that something is wrong about cycles in your software, in particular cycles that transcend the boundaries of one package. So

[57] http://blog.ircmaxell.com/2013/06/backwards-compatibility-is-for-suckers.html

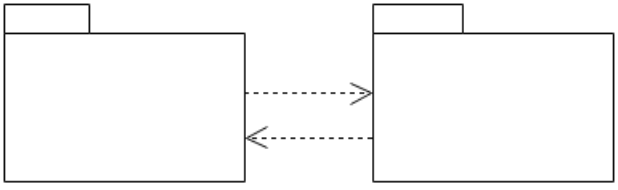
you may want to fix them anyway. The good news is: there are some easy solutions to remove cycles from a dependency graph.

Solutions for breaking the cycles

In the first place, some cycles are "more real" than other cycles.

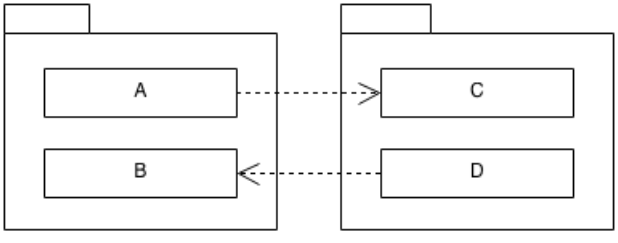
Some pseudo-cycles and their dissolution

Consider these two packages that have a direct dependency on each other (although it doesn't really matter for this example whether or not the cycle consists of more packages):



Two packages that are dependent on each other

When a dependency is considered a "package dependency", this means that part of the code in the *root* package depends on a part of the code in the *other* package. Most often the root package contains a class which uses a class from the other package in one of the ways described at the beginning of this chapter. When zooming in on both packages we discover in this case that, indeed one class in the root package (class `A`) depends on a class in the other package (class `C`), which explains one direction of the package dependency.

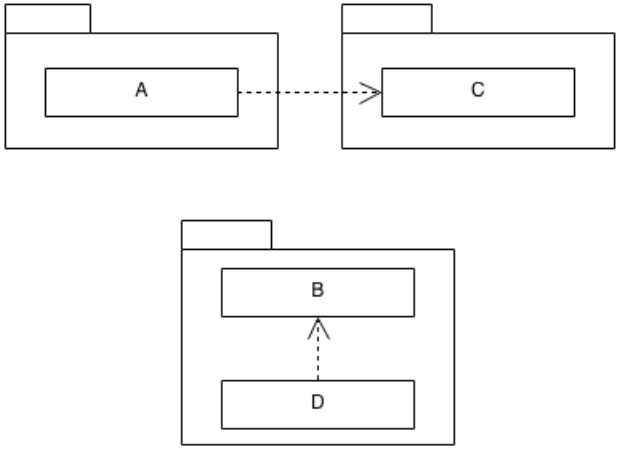


Zooming in on the actual dependencies

Trying to explain the reason why this package dependency is *circular*, we notice that the reciprocal dependency is totally unnecessary, because it concerns two other, unrelated classes, class `B` and `D`.

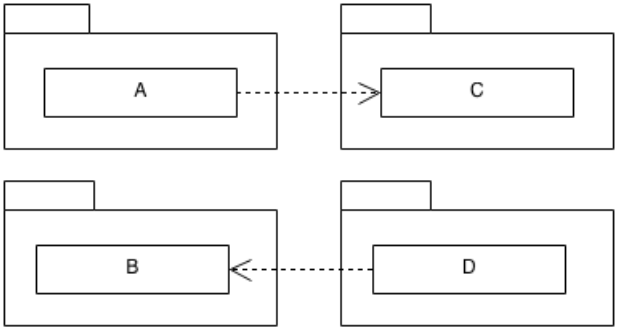
This means that these two packages are used in two different, unrelated ways and that the classes have not been divided correctly among the two. These packages simply violate the *Common reuse principle*: not all of their classes are reused together at the same time. This is also reminiscent of the *Interface segregation principle*, but applied to the package itself instead of just one class: apparently there are multiple different *clients* for this package, and only some of them cause a dependency cycle.

I call this type of dependency cycle a *pseudo-cycle*. It can easily be dissolved by rearranging the code and creating one or two new packages. For instance you can put classes `B` and `D` together in one package, which makes the dependency internal to that package:



A solution for dissolving the pseudo-cycle

Or you can put `B` and `D` in their own separate packages, which leaves the package dependency as it is, but at least removes the cycle from the dependency graph:

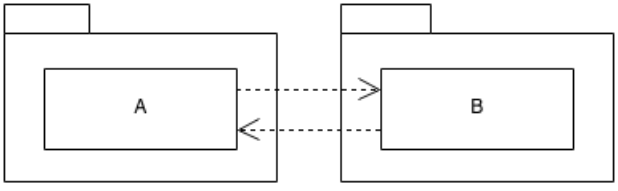


Another solution for dissolving the pseudo-cycle

You should choose the second solution when other parts of the system rely on just class `B` or just class `D` which would mean that according to the *Common reuse principle* these classes should be in separate packages.

Some real cycles and their dissolution

We've discussed a nice pseudo-cycle, but what does a real cycle look like? It might be something like this:



A real cycle

Again, there may be any number of packages between these two, as long as there is an actual cycle.

To make this a bit more concrete, let's say class `A` is used to process web forms and class `B` is a generic validator. Forms and validation are very much related, but at least validation is not limited to forms. So the validation package should be available for separate use in different scenarios (for instance to validate query parameters or deserialized objects).

In the form package we find the following code:

```
use Validator\Validator;

class Form
{
    private $validator;

    public function __construct(Validator $validator)
    {
        $this->validator = $validator;
    }

    public function isValid()
    {
        $this->validator->validateForm($this);

        return count($this->errors) === 0;
    }

    public function addError($error)
    {
        $this->errors[] = $error;
    }
}
```

In the validator package we find the following code:

```
use Form\Form;

class Validator
{
    public function validateForm(Form $form)
    {
        ...

        $form->addError(...);
```

```
    }
}
```

The above code snippets expose a circular dependency between `Validator` and `Form` of type "usage" (see the list of coupling types at the beginning of this chapter).

To break this type of cycle it won't suffice to merely move code around (like it did when we discussed a pseudo-cycle in the previous section). We must do some actual programming to fix this problem. We will refactor the code, i.e. change its structure, not its behavior, by applying some design patterns to it. There are many conceivable solutions and I will show some of the most used ones.

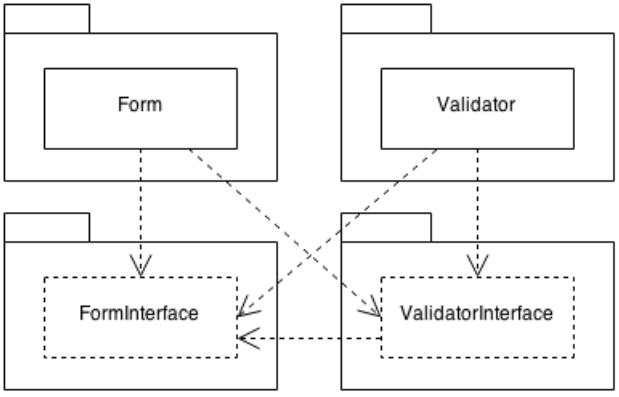
Dependency inversion

The first thing we can do is remove the hard dependency on a class in another package. Like explained in the chapter about the *Dependency inversion principle* we can easily revert dependency directions by depending on something *abstract*, i.e. an interface, instead of something *concrete*, i.e. a class. If we then move the interfaces to separate packages, we are saved.

```
interface FormInterface
{
    public function isValid();

    public function addError($error);
}

class Form implements FormInterface
{
    public function __construct(ValidatorInterface $validator)
    {
        ...
    }

    public function isValid()
    {
```

```
        ...
    }

    public function addError($error)
    {
        ...
    }
}

interface ValidatorInterface
{
    public function validateForm(FormInterface $form);
}

class Validator implements ValidatorInterface
{
    public function validateForm(FormInterface $form)
    {
        ...
    }
}
```

We introduce interfaces and we make the existing classes implement those interfaces. Then we type-hint all arguments to interfaces, not classes. When we put the interfaces in separate packages, we have successfully diverted some of the problematic dependencies and there are strictly no circles left in the dependency graph:



Removing the cycle with dependency inversion

The real trick here is that even though `Form` originally depended on an instance of `Validator` (injected as a constructor argument), `FormInterface` doesn't, which means there is no dependency arrow from `FormInterface` to `ValidatorInterface`.

So using dependency inversion we have actually dissolved the cycle in the dependency graph. However, it's not really the best solution there is.

Inversion of control

The design problem with the `Form` and `Validator` class which caused them to be tightly coupled is: they know too much about each other which results in lots of communication back and forth between them. Remember: the reason for having a separate package for data validation was that data validation is not necessarily limited to validation of data submitted using a web form. So it is surprising to say the least that the `Validator` class actually has a `validateForm()` method. This clearly violates the *Interface segregation principle* since only a portion of the *clients* of the `Validator` class will use that method.

When an object communicates with another object by calling methods on it, it really exercises control over it. Calling a method triggers an action in the other object. When objects call each other's methods, i.e. *communicate with each other*, this should thus be seen as *exercising control* over each other. But communication that goes back and forth creates a cycle. To resolve that cycle, we must break the communication lines between objects and let some other object(s) do the talking. This would basically

invert the direction of the controlling behavior of these objects. Hence this technique is known as *inversion of control*.

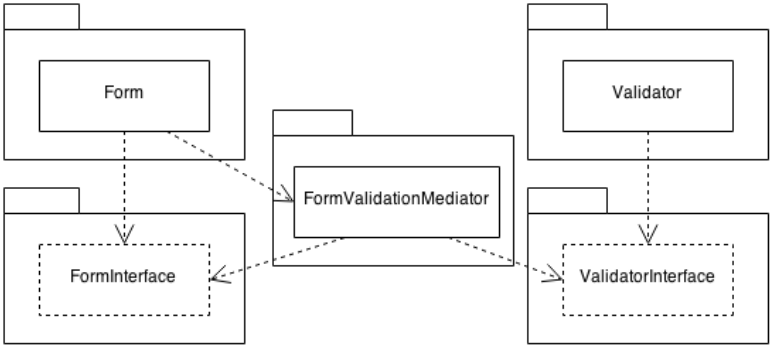
There are quite a lot of options when you want to refactor code that is too much "in control". In fact, all the design patterns known as "behavioral patterns" (see also the famous "Gang of Four" book: *Design Patterns: Elements of Reusable Object-Oriented Software*) are suitable for this purpose. It's still up to you to judge if they apply to your situation. Also, you don't necessarily need to follow the exact patterns.

Mediator The first and easiest scenario would be to introduce a *mediator*. The `Form` object then shouldn't make any direct calls to a `Validator` object anymore. Instead, it may only call the mediator, which on its turn will make any form-specific calls to the `Validator`:

```
class FormValidationMediator
{
    private $validator;

    public function __construct(ValidatorInterface $validator)
    {
        $this->validator = $validator;
    }

    public function validate(FormInterface $form)
    {
        $formValues = ...;

        $this->validator->validate($formValues);
    }
}

class Form implements FormInterface
{
    private $formValidator;

    public function __construct(
```

```
        FormValidationMediator $formValidator
    ) {
        $this->formValidator = $formValidator;
    }

    public function isValid()
    {
        $this->formValidator->validate($this);

        ...
    }
}
```

This basically removes the dependency from the validator package to the form package. `Validator` is now truly stand-alone. The mediator which runs validation on forms should be in its own package, having dependencies on both the form and the validator interface packages:



Removing the dependency from `ValidatorInterface` to `FormInterface` by introducing another package for the mediator between the two

Naming pattern-inspired classes

Even though some believe it's a best practice to incorporate the name of the design pattern you used in the name of the class itself, I really think that `FormValidationMediator` in this case is a strange name. In my experience it makes a lot more sense to just name the class whatever is most appropriate in the context of your application and to make sure it reads well. Then in the docblock of the class you could mention the pattern you used, if it will help the reader understand what is going on and why you implemented the solution using this particular pattern:

```
/**
 * Mediator for validating Form objects using a generic
 * Validator object
 */
class FormValidation
{
    ...
}
```

A mediator package is also known as a *bridge* package. A bridge connects two packages that are highly useful when used together, but shouldn't know about each other's existence or inner workings.

In some cases it even makes sense to call such a mediator package an *adapter* package. An adapter package provides one specific implementation of an interface defined in another package. Having adapter packages implies that there can be many parallel implementations of that interface. In the case of the `FormValidation` mediator, this is not very likely since this class should be the "one and only" implementation of a form validator that uses this particular validation package.

If at any time you'd like to use a validator from another vendor to validate forms from this particular form package, the main characteristic of being a mediator class changes to that of being an adapter, which, following the definition of the *Adapter* design pattern, makes the interface of one object compatible with that of another one.

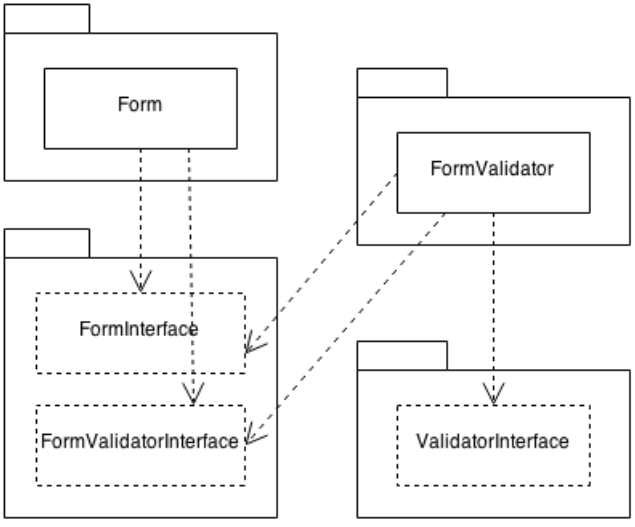
Chain of responsibility Another useful pattern in our quest to break a dependency cycle is the *Chain of responsibility*. You can use it to allow other parts of the application to hook into a certain process and let them do whatever they like:

```
interface FormValidatorInterface
{
    public function validate(FormInterface $form);
}

class Form implements FormInterface
{
    private $validators = [];

    public function addValidator(FormValidatorInterface $validator)
    {
        $this->validators[] = $validator;
    }

    public function isValid()
    {
        foreach ($this->validators as $validator) {
            $validator->validate($this);
        }
    }
}
```

Any class implementing `FormValidatorInterface` can be added to the stack of validators. This is a nice example of applying the *Open/closed principle* to the `Form` class: it is possible to change the behavior of the class with regard to validation by just injecting other objects into it, instead of modifying its code.

Please note that the original recipe for a *Chain of responsibility* includes a separate object for the request and each of the candidates for this request needs to pass the request object to the next candidate explicitly. In most cases such an implementation is way too complex. A simple loop over the candidates makes much more sense and is definitely better readable too.

Using a chain of objects that have the same responsibility is a great way to decouple packages. The package may contain just one interface, which other packages can depend on (i.e. implement). The project that uses these packages then only needs to configure the object graph correctly.



A chain of responsibility used to break the dependency cycle

Mediator and chain of responsibility combined: an event system There's one last solution that's a very common and appropriate way of dissolving dependency cycles. It's a combination of a *Mediator* and a *Chain of responsibility*. It is often known as an *event dispatcher* or an *event manager*.

An event manager is an abstract mediator: the names and types of the messages are not predefined. It simply passes messages (events) to delegates (event listeners). Below you will find a simple implementation of such an event manager. It allows for event listeners to be registered using the name of the event they listen to and a callable that should be called whenever the event occurs:

```
class EventManager
{
    public function registerListener($eventName, callable $listener)
    {
        $this->listeners[$eventName][] = $listener;
    }

    public function triggerEvent($eventName, $eventData)
    {
        foreach ($this->getListeners($eventName) as $listener) {
            /*
             * Call the listener;
             * the first and only argument is $eventData
             */
            call_user_func($listener, $eventData);
        }
    }

    private function getListeners($eventName)
    {
        if (isset($this->listeners[$eventName])) {
            return $this->listeners[$eventName];
        }

        // no listeners are defined for this event
        return [];
    }
}
```

In the `Form` class we can use the event manager to trigger an event whenever the form has been submitted; let's call this event `form.submitted`:

```
class Form
{
    private $eventManager;

    public function __construct(EventManager $eventManager)
    {
        $this->eventManager = $eventManager;
    }

    public function submit(array $data)
    {
        // create the event object, provide the right context
        $event = new FormSubmittedEvent($this, $data);

        $this->eventManager->triggerEvent('form.submitted', $event);
    }

    ...
}
```

The `FormSubmittedEvent` class is quite simple. It is merely used to carry some contextual data about the event that occurred. In this case it allows event listeners to inspect (and modify) the form object itself and the data that was submitted:

```
class FormSubmittedEvent
{
    private $form;

    public function __construct(FormInterface $form, array $data)
    {
        $this->form = $form;
    }

    public function getForm()
    {
```

```
        return $this->form;
    }

    public function getData()
    {
        return $this->data;
    }
}
```

Now we only need to implement an event listener which validates the form based on the submitted data. It listens to the `form.submitted` event and unpacks the `FormSubmittedEvent` object. When some of the submitted data from the event object is invalid, the listener adds an error to the form object.

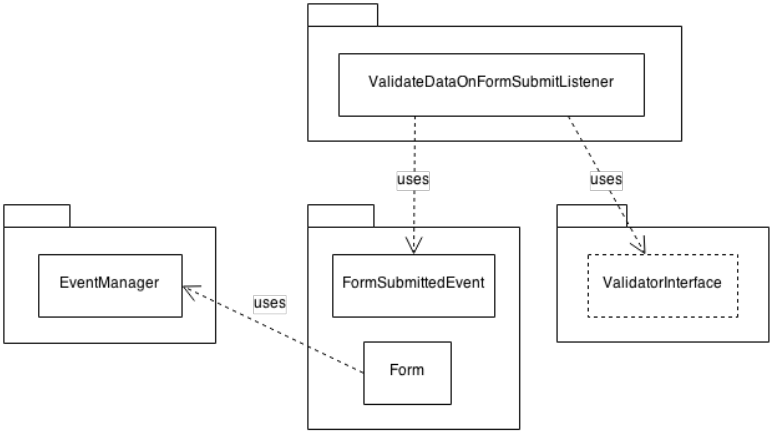
```
class ValidateDataOnFormSubmitListener
{
    public function __construct(ValidatorInterface $validator)
    {
        $this->validator = $validator;
    }

    public function onFormSubmit(FormSubmittedEvent $event)
    {
        $form = $event->getForm();
        $submittedData = $event->getData();

        if (!$this->validator->validate(...)) {
            // part of the submitted data is invalid
            $form->addError(...);
        }
    }
}
```

To make all of this work, you need to set up the event manager and register the form validation listener, then provide the event manager as the constructor argument of the `Form` object.

```
$eventManager = new EventManager();
$validationListener = new ValidateDataOnFormSubmitListener();
$eventManager->registerListener(
    'form.submitted',
    // array notation for a callback:
    [$validationListener, 'onFormSubmit']
);

$form = new Form($eventManager);

// will trigger the event listener
$form->submit([...]);
```

The `EventManager` is a proper *Mediator*: a `Form` never talks to a `Validator` directly, but always by means of the `EventManager`. Inside the `EventManager` the listeners form a *Chain of responsibility*. Each of them gets a chance to respond to the `form.submitted` event.

Looking at the dependency diagram, we see no circles:



The new dependency diagram including the event manager

What is missing here is some connection between the event manager and the event listener. In a way, the `EventManager` depends on the `ValidateDataOnFormSubmit-Listener` class. This is however a runtime dependency. If we look at the code there

is no *strict* dependency on the `ValidateDataOnFormSubmitListener` class. You can use the `EventManager` class without this particular event listener. Hence it should not result in a dependency between packages.

So an event manager is a nice way to decouple things, remove dependencies and even dependency cycles. However there is a downside to it. Because event managers are *highly abstract mediators*, it can sometimes be difficult to read and understand code that uses events to do important things like form validation. When you feel that this is the case, you should consider to pick just the dependency inversion solution explained above, possibly combined with a *Chain of responsibility*.

Conclusion

In this chapter we discussed many aspects of coupling. First we looked at different types of dependencies between classes, which can lead to package dependencies when the dependency of one class on another transcends the boundary of the package that contains the class.

When following the path from dependency to dependency, you sometimes return to the package from which you started. In that case you have a cycle in your dependency graph. The *Acyclic dependencies principle* told us to not have cycles in our dependency graph. Thinking about this, we concluded that cycles indeed cause a lot of trouble and towards the end of this chapter we learned that they are also not that hard to overcome.

Breaking all the cycles in our dependency graph (which is then an *acyclic directed graph*) makes it possible for us to easily create branches in the history of the packages. This means we can work on new minor and even major versions of a package, without preventing other packages from making progress or ever releasing their next major version at all. Having no cycles therefore means that a change in one package affects only the smallest number of packages possible.

Being susceptible to changes in other packages, or necessitating other packages to change when your package changes is the subject of the following two chapters.

The Stable dependencies principle

In the previous chapter we discussed the *Acyclic dependencies principle*, which helps us prevent cycles in our dependency graphs. The greatest danger of cyclic dependencies is that problems in one of your dependencies might backfire after they have travelled the entire cycle through the dependency graph.

Even when your dependency graph has no cycles, there is still a chance that dependencies of a package will start causing problems at any time in the future. Whenever you upgrade one of your project's dependencies, you hope that your project will still work as it did before. However there is always the risk that it suddenly starts to fail in unexpected ways.

When your project still works after an upgrade of its dependencies, the maintainers of those dependencies are probably aware that many packages *depend on their package*. In each release they merely fixed bugs or added some new features. They never made any changes that would cause failure in a dependent package.

If however something is broken in your project after an upgrade, the package maintainers made some changes that are not backward compatible. These kind of changes bubble up through the dependency graph and cause problems in dependent packages.

When a dependency of your project suddenly causes failures, you must first rethink your choice of dependencies instead of blaming the maintainers. Some packages are highly volatile, some are not. It can be in the nature of a package to change frequently, for any reason. Maybe those changes are related to the problem domain, or maybe they are related to one of its dependencies.

Likewise, before adding a dependency to your project you need to decide: is it likely that this dependency is going to change? Is it *easy* for its maintainers to change it? In other words: can the dependency be considered *stable*, or is it *unstable*?

Semantic versioning and stability

As we discussed in the chapter about the *Reuse/release equivalence principle* the word "stable" is also used in the context of semantic versioning. A package is considered stable if it has a version that is at least `1.0.0`, and is not in a development (or alpha, beta, RC) branch. Such a stable version promises to have a public API that does not change in backwards incompatible ways.

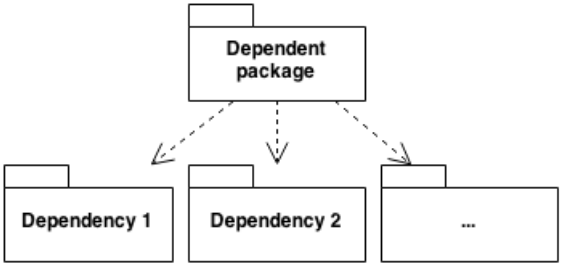
The *Stable dependencies principle* is also about the stability of a package, but has nothing to do with semantic versioning. In this chapter "stable" means "not easy to change". A stable package in this context is a package on which many other packages depend while it does not depend on other packages itself (or only on a small number of other stable packages).

Stability in the sense of semantic versioning is obviously also *related* to "not easily being changed" since the package maintainer promises backward compatibility for all versions that have the same major version.

Stability

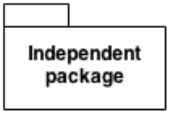
The stability of a package is all about how easy it is to change something in its code. This is not about clean code, or if the code can be easily refactored. It is about how *responsible* the package is with respect to other packages and if the package is susceptible to changes in any one of its dependencies.

As I already mentioned, changes in the dependencies of a package are likely to bubble up to the package itself. You will often need to make changes to your own package to accommodate for changes in its dependencies. If you have a lot of dependencies it is much more likely that an update of your dependencies will require you to modify your own package. If you have no or just a small number of dependencies, chances are that an update of your dependencies will cause no problems at all.



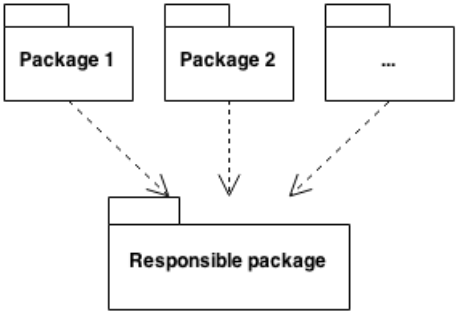
A highly dependent package.

When a package needs to be changed often to accommodate for a change in one of its dependencies, it can be considered highly *instable*. If it is not very susceptible to changes in its dependencies, because it has none, or only a few, the package can be considered highly *stable*.



An independent package.

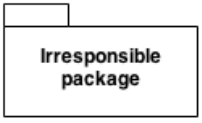
There is another direction in the dependency graph that needs to be considered: the direction *towards* a package. In other words: how many other packages depend on a given package? If the number is high, it will be difficult to make changes to the package, because so many other packages are depending on it, and those local changes may require many modifications *elsewhere*.



A responsible package with many packages depending on it.

On the other hand, if the number of incoming dependencies is low or even lacking,

it will be very easy for the package maintainer to make changes, since those changes will have little impact on *others*.



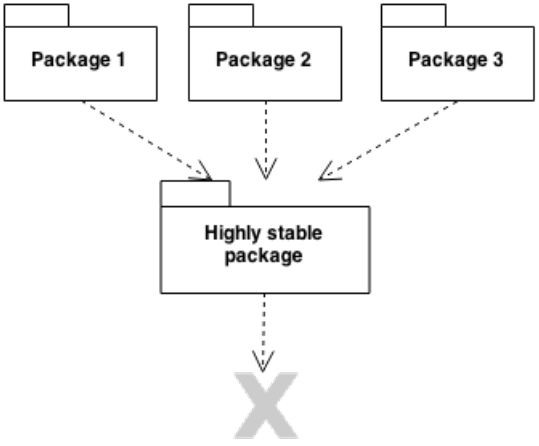
An irresponsible package with no packages depending on it.

We call a package with no other packages depending on it an "irresponsible" package, because it will not be held responsible for any changes that are made to it. In other words: the package maintainer is free to change anything they like. On the contrary a package with many dependents can be called "responsible" since its maintainer can not just change anything they want. Any change should be expected to have an impact on depending packages.

Not every package can be highly stable

Of course, if no package would depend on another package, no package would depend on *it* too, so it would be at the same time irresponsible and independent. This would make such a package totally useless; merely an island of code. Most packages however are somewhere between *highly independent and responsible* and *highly dependent and irresponsible*.

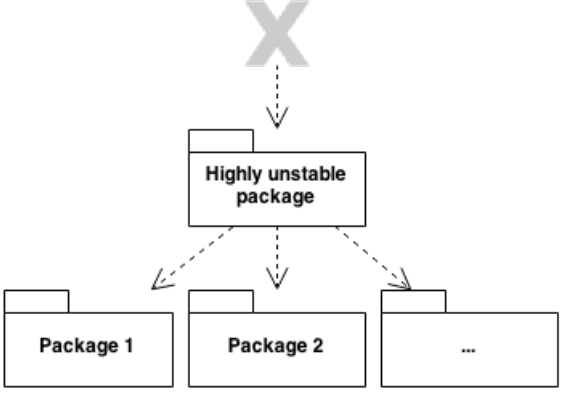
Packages that are more independent and responsible should be considered *highly stable*. Those are packages that don't need to change because of a change in one of their dependencies, but they also can't easily change themselves because other packages heavily depend on them.



A highly stable package: no dependencies, only dependents.

These highly stable packages are usually small libraries of code which implement some abstract concepts which are useful in many different contexts.

On the other side of the scale, packages that are more dependent but at the same time very irresponsible, should be considered *highly instable*. These packages are susceptible to changes in any of their dependencies, but they are not depended on by any other package, so it is no problem for them to change because a change would not ripple through.



A highly unstable package: many dependencies, no dependents.

These highly unstable packages are likely to contain concrete implementations which are for example coupled to a specific persistence library, or they may contain detailed

implementations of business rules that are liable to change. Code that is only useful within the context of a certain application framework is also likely to be inside an unstable package, since a framework is itself highly unstable according to the definition used in this chapter.

Unstable packages should only depend on more stable packages

Intuitively it would be alright for an unstable package to depend on a stable package. After all, the stable package is unlikely to have negative effects on an already unstable package. However the other way around - a *stable* package that depends on an *unstable* package - would not be acceptable. The volatility of an unstable package would pose a thread to the stability of the stable package and would in fact make it less stable.

To prevent package designers from introducing "bad" dependencies, the *Stable dependencies principle* tells us that:

> The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than *it* is.

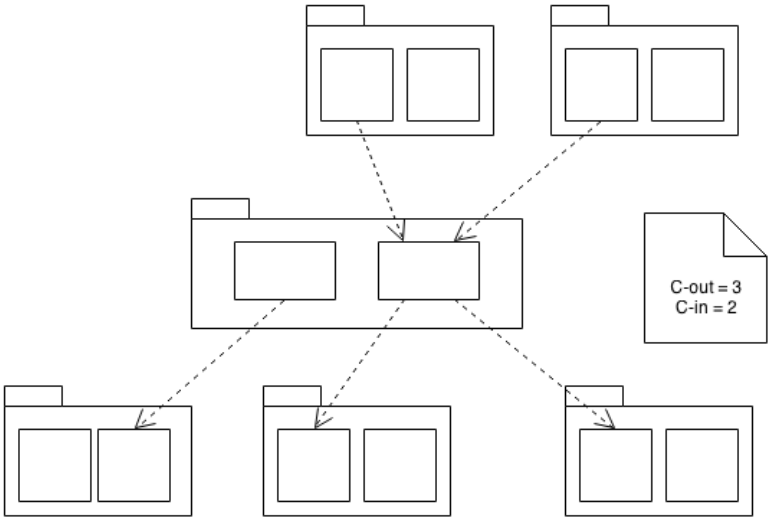
In other words: less stable packages may depend on more stable packages. Stable packages should not depend on unstable packages.

Measuring stability

Stability is actually a quantifiable unit, which we can use to determine if any package in a dependency graph satisfies the *Stable dependencies principle*.

The conventional way of expressing stability is by calculating the `I` metric for packages. First you need to count the number of classes *outside* a package that depend on a class *inside* the package. We call this value `C-in`. Then you need to count the

number of classes *outside* the package that any class *inside* the package depends upon. We call this `C-out`.



Calculating `C-in` and `C-out` for the package in the center.

You can then determine the `I` metric for the package by calculating `C-out` divided by `C-in + C-out`. This means that `I` will be between `0` and `1` where `1` indicates that the package is maximally unstable and `0` indicates that it is maximally stable.

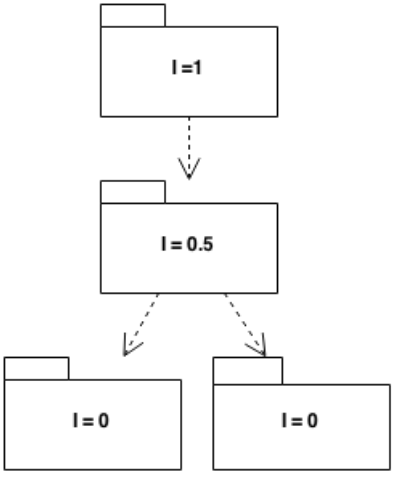
A highly *stable* package is *responsible*: it has many dependents, so `C-in` is a high number. At the same time it is *independent*: it has no dependencies, so `C-out = 0`. This means that `I = 0` since `C-out / (C-in + C-out) = 0`.

A highly *unstable* package is very *dependent*: it has many dependencies, so `C-out` is a high number. But it's also *irresponsible*: it has no other packages depending on it, so `C-in = 0`. Then `I = 1` since `C-out / (C-in + C-out) = 1`.

Of course these are very extreme examples. Most packages have an `I` that is not `0` nor `1` but somewhere in between. For example, the package in the center of the previous diagram has a `C-out` of `3` and a `C-in` of `2`, so the value of `I` for that package is `3/(2+3) = 3/5`or `0.6`.

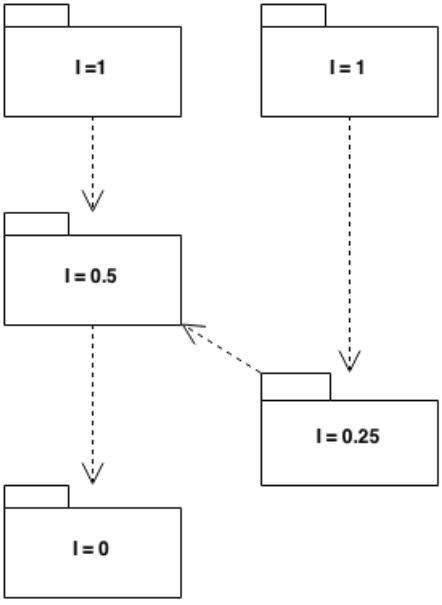
Decreasing instability, increasing stability

According to the *Stable dependencies principle*, the dependencies between packages in a design should be "in the direction of the stability of the packages." In other words, each step we take in the dependency graph should lead to a more stable package. More stable also means less instable, so we are only allowed take steps in the dependency graph which lead to packages with a lower value for `I`.



An example of packages that all depend in the direction of stability.

When you draw such a diagram for your packages it's useful to put packages with a low `I` near the bottom and packages with a high `I` near the top. Then every dependency arrow should point downward since that is the direction of stability. If an arrow would point upward, the *Stable dependencies principle* has been violated (we will later discuss your options to force the arrow in the right direction again).



An example of packages that do not all depend in the direction of stability.

In the following sections we will discuss some violations of the *Stable dependencies principle* and how you can fix them (if you have the power to do so!).

Violation: your stable package depends on a third-party unstable package

In the following example of a violation of the *Stable dependencies principle* I make use of the Gaufrette library[58] which offers an abstraction layer for filesystems. It allows you to switch from a local filesystem to an in-memory filesystem, or even to Dropbox or Amazon storage without the need to make changes to your own code.

The `FileCopy` class below is part of my own package. Its naive implementation of a copy mechanism allows you to copy files between any two filesystems. It depends on the `Filesystem` class offered by the Gaufrette library.

[58] https://github.com/KnpLabs/Gaufrette

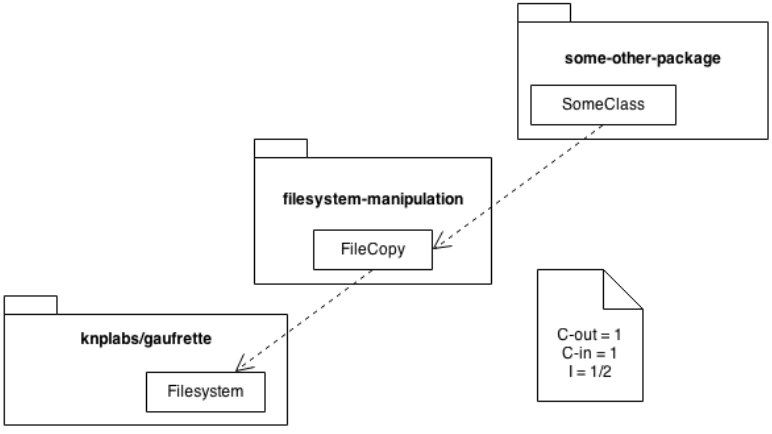
```
use Gaufrette\Filesystem as GaufretteFilesystem

class FileCopy
{
    private $source;
    private $target;

    public function __construct(
        GaufretteFilesystem $source,
        GaufretteFilesystem $target
    ) {
        $this->source = $source;
        $this->target = $target;
    }

    public function copy($filename)
    {
        $fileContents = $this->source->get($filename);

        $this->target->write($filename, $fileContents);
    }
}
```

The package which contains the `FileCopy` class, let's call it `filesystem-manipulation`, has an explicit dependency on the `knplabs/gaufrette` package which contains the `Filesystem` class:

```
{
    "name": "filesystem-manipulation",
    "require": {
        "knplabs/gaufrette": "~0.1"
    }
}
```

Currently `FileCopy` is the only class in this package. It has one dependency on a class of another package, which causes the `C-out` of this package to be `1`. In the

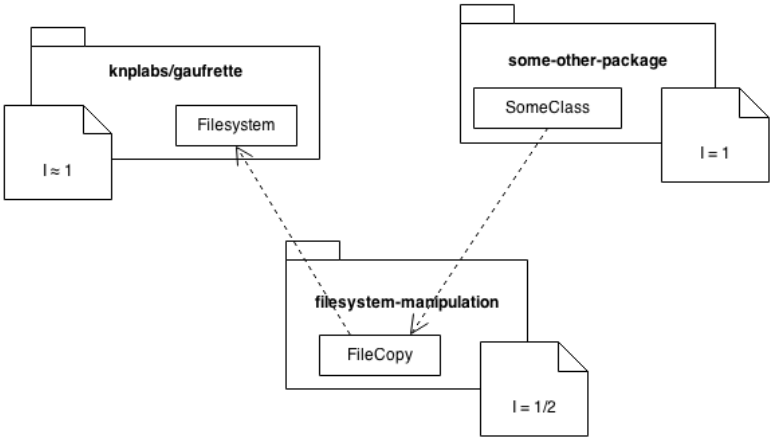
project in which the `filesystem-manipulation` package is being used, there is one package which makes use of the `FileCopy` class, so `C-in` is also `1`, which causes `I` to be `1/(1+1) = 1/2`.



Calculating `I` for `filesystem-manipulation`

When we make the same calculation for the `knplabs/gaufrette` package, we need to count the number of classes outside that package that are depended on by classes inside the package, which is `54`. So `C-out = 54`. Within the current project, only the `FileCopy` class depends on one of the classes of `knplabs/gaufrette`, so `C-in = 1`. This results in a fairly high value for `I: 54/(54+1) = 54/55`, which is almost `1`.

So `knplabs/gaufrette` is actually a highly unstable package. Much more unstable than the `filesystem-manipulation` package. Nevertheless, the `filesystem-manipulation` package depends on the entire `knplabs/gaufrette` package. So we clearly violate the *Stable dependencies principle*, since our packages do not all depend in the direction of stability. Instead our package depends in the direction of instability. This becomes even more clear when we order the packages according to their stability and then draw the dependency arrows:



The `filesystem-manipulation` package depends on a less stable package.

The reason why `knplabs/gaufrette` is such an unstable package is that it contains many concrete filesystem adapters for Dropbox, Amazon S3, SFTP, etc. These adapters are not used by everyone at the same time. So according to the *Common reuse principle*, they should have been in separate packages.

The `filesystem-manipulation` package does not need all those specific filesystem adapters, it only needs the `Filesystem` class, which provides generic methods for communicating with any specific filesystem.

Solution: use dependency inversion

In order to fix the dependency graph and force the arrows to point in the direction of stability we would very much like to take the `Filesystem` class (which is the *actual* filesystem abstraction layer) and put it inside a separate package: `knplabs/gaufrette-filesystem-abstraction`. Then the adapter classes should be put inside other packages, like `knplabs/gaufrette-amazon-adapter`, `knplabs-/gaufrette-sftp-adapter`, etc. We could then change the dependency on `knplabs/gaufrette` to `knplabs/gaufrette-filesystem-abstraction` and this would do the trick.

However, we can't do this, since we are not the maintainers of `knplabs/gaufrette`. So we need to resort to another solution, previously described in the chapter about the *Dependency inversion principle*. First, instead of depending on the `Gaufrette\-`

`Filesystem` class which is still inside a highly unstable package, we define our own `FilesystemInterface` inside our `filesystem-manipulation` package:

```
interface FilesystemInterface
{
    public function read($path);

    public function write($path, $contents);
}
```

Then we make the constructor of `FileCopy` accept objects that implement this new `FilesystemInterface`:

```
class FileCopy
{
    ...

    public function __construct(
        FilesystemInterface $source,
        FilesystemInterface $target
    ) {
        ...
    }

    ...
}
```

Now we can actually remove the dependency on `knplabs/gaufrette` from the package definition of our `filesystem-manipulation` package. As a matter of fact, the package has become independent at once: it has no dependencies at all. This means that it now has an `I` of `0` and it is to be considered highly stable.

As already mentioned, we'd still want to make use of the Gaufrette library. Therefore we need to bridge the gap between `FilesystemInterface` and the `Gaufrette\-Filesystem` class. We may accomplish this by introducing a new class, `Gaufrette-FilesystemAdapter`.

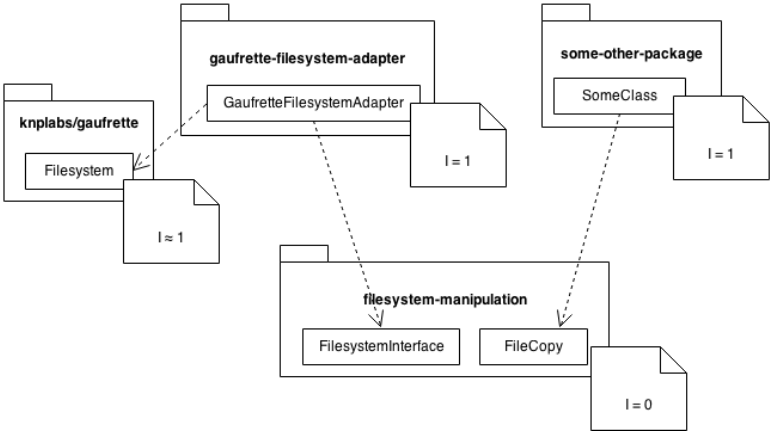
```
use Gaufrette\Filesystem as GaufretteFilesystem;

class GaufretteFilesystemAdapter implements FilesystemInterface
{
    private $gaufretteFilesystem;

    public function __construct(
        GaufretteFilesystem $gaufretteFilesystem
    ) {
        $this->gaufretteFilesystem = $gaufretteFilesystem;
    }

    public function read($path)
    {
        return $this->gaufretteFilesystem->get($path);
    }

    public function write($path, $contents)
    {
        $this->gaufretteFilesystem->write($path, $contents);
    }
}
```

This class uses the Gaufrette filesystem object by composition and still is a proper substitute for `FilesystemInterface`. We put this class in a new package, `gaufrette-filesystem-adapter`. Since the class needs both the `FilesystemInterface` and the `Gaufrette\Filesystem` class, it depends on both `knplabs/gaufrette` and `filesystem-manipulation`:

```
{
    "name": "gaufrette-filesystem-adapter",
    "require": {
        "knplabs/gaufrette": "1.*"
        "filesystem-manipulation": "*"
    }
}
```

The `C-out` of this new `gaufrette-filesystem-adapter` package is `2`, because it uses two classes outside the package. Its `C-in` is `0`, since no other package uses a class from this package. This means `I = 2/(2+0) = 1`. It is highly unstable (i.e. easy to change), which is totally fine for an adapter package.

Let's see what all this did for the dependency graph and the arrows in it:



Each package depends in the direction of stability.

The packages are sorted in the direction of stability, and all dependency arrows are pointing downward, which means that no package in this system violates the *Stable dependencies principle* anymore.

All of this was accomplished without making any changes to third-party code. We applied the *Dependency inversion principle* to the `FileCopy` class by letting it depend on something abstract instead of something concrete. This automatically makes the `FileCopy` class easily extensible: others can implement their own adapters for

`Filesystem` and make it compatible with for instance the `Flysystem` filesystem abstraction library[59]. It also makes the `filesystem-manipulation` better maintainable, since changes in `knplabs/gaufrette` will not affect it anymore.

Staying unaffected by external changes makes the `filesystem-manipulation` package very stable: it is unlikely to change because of its dependencies (since it has no dependencies anymore). Its previous instability is pushed away to the more concrete `gaufrette-filesystem-adapter` package, which from now on is susceptible to changes in `knplabs/gaufrette`. But even though the code inside the `gaufrette-filesystem-adapter` package is likely to change, it poses no thread to other parts of the system, since no other package depends on it.

A package can be both responsible and irresponsible

As I already quickly pointed out, the `knplabs/gaufrette` package has some design issues. It contains classes that would not be used by everyone who uses the package in their project, so it violates the *Common reuse principle*. It also contains classes (the same classes actually) that are not closed against the same kinds of changes, so the package violates the *Common closure principle*.

Now that we are looking at the `knplabs/gaufrette` package from the perspective of stability, it becomes clear that grouping those classes that actually don't belong together is the reason why this package has become very unstable. It introduces many external dependencies, which makes it no longer safe for other packages to depend on it.

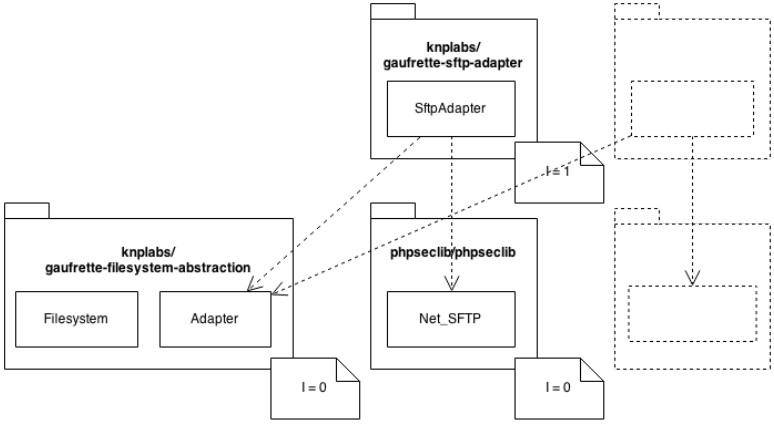
Not being safe to depend on is not a good property for packages that are supposed to be highly reusable. In fact, a reliable package should be very safe to depend on: it should be stable. In other words: it should be independent and responsible.

In the previous section we discussed a solution for this stability problem. It entailed the introduction of an interface and an adapter to rearrange the dependency directions. We needed to resort to this solution because we could not do what was really necessary: to split the package into a package containing the more generally reusable

[59] https://github.com/thephpleague/flysystem

parts (like the `Gaufrette\Filesystem` class and the `Gaufrette\Adapter` interface) and one or more packages containing the more specific and concrete parts (like the filesystem adapters for SFTP, Dropbox, etc.).

The first package would have no dependencies, only dependents, which would make it independent and responsible, i.e. very stable. We would call it `knplabs/gaufrette-filesystem-abstraction`. The other packages would be named after the specific filesystems they provided an implementation for, like `knplabs/gaufrette-sftp-adapter`. Each of those packages could then have as many dependencies as needed by the specific filesystem implementation. And of course each of them would depend on `knplabs/gaufrette-filesystem-abstraction` because that package will contain the interface that each filesystem adapter needs to implement. It would make those adapter packages dependent and irresponsible (i.e. no other package is depending on it).



`knplabs/gaufrette-*` packages after refactoring

The great thing is that in such a constellation of packages, `knplabs/gaufrette-filesystem-abstraction` would be very stable, and the `filesystem-manipulation` package containing the `FileCopy` class could safely depend on it. The `filesystem-manipulation` package itself has an `I` of `1/2`, while `knplabs/gaufrette-filesystem-abstraction` has an `I` of `0`, which is lower. All package dependencies would follow the direction of stability, so the *Stable dependencies principle* would not be violated.

Conclusion

After we moved the `Gaufrette\Filesystem` class and the `Gaufrette\Adapter` interface out of the main package and into a small and very stable package, it became a lot safer to use that class and/or interface in another package because the freshly created `knplabs/gaufrette-filesystem-abstraction` package is very stable.

The lesson to be learnt here is that when you have the ability to split a package into multiple packages with different levels of stability, you should do it, You will make it easier for people to use *your* code in *their* projects, without introducing any more instability. If you don't have that ability because the project is not maintained by yourself (or it doesn't accept pull requests), it is still very easy to introduce the adapter design pattern, which helps you fix any stability issues right-away.

The Stable abstractions principle

We have reached the last of the design principles related to package coupling, which means we have in effect reached the last of *all* the package design principles. This principle, the *Stable abstractions principle*, is about stability, just like the *Stable dependencies principles*. While the previous principle told us to depend "in the direction of stability", this principle says that packages should depend in the direction of *abstractness*.

Stability and abstractness

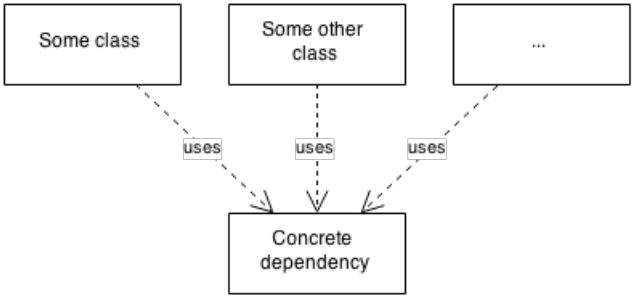
The name of the *Stable abstractions principle* contains two important words: "stable" and "abstract". We already discussed stability of packages in the previous chapter. A stable package is not likely to change heavily. It has no dependencies so there is no external reason for it to change. At the same time other packages depend on it. Therefore the package should not change, in order to prevent problems in those depending packages.

In the previous chapter we learned that you can calculate stability and that you can verify that the dependency graph of a project contains only dependencies of increasing stability, or decreasing instability. In this chapter we learn that we also have to calculate the abstractness of packages and that the dependency direction should be one of increasing abstractness, or decreasing concreteness.

The concept of abstractness is something we also encountered in previous chapters. For example in the chapter about the *Dependency inversion principle* we learned that our classes should depend on abstractions, not on concretions. We discussed several ways in which a class can be abstract. The most obvious way is when a class has abstract (also known as virtual) methods. These methods have to be defined in a subclass. This subclass is a concrete class because it is a full implementation of the

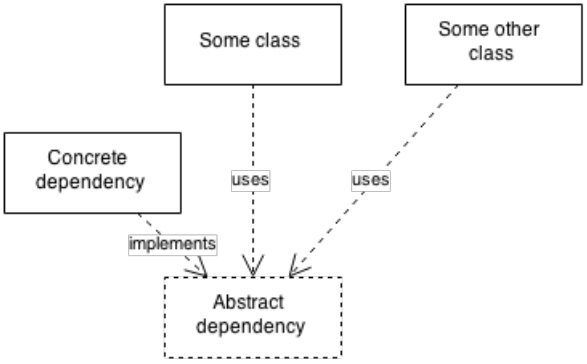
type of thing that the abstract class tries to model. When a class only has abstract methods, we usually don't call it a class, but an interface. Classes that implement the interface eventually have to provide an implementation for all of the abstract methods defined in the interface.

The *Dependency inversion principle* told us to depend on abstractions, not on concretions. The reason was, like always, that we need to be prepared for change. A class that depends on a concrete thing is likely to change whenever some implementation detail of the concrete thing changes. Besides, if at some point we want to replace the concrete thing with another concrete thing, we have to modify the class to understand and use that new concrete thing. And in that situation it's probably not the only class that needs to be modified.



Depending on concrete things

If instead we define something abstract, like an abstract class or preferably an interface, and we depend on it, we are much better prepared for change. Most changes occur in concrete things, i.e. in fully implemented classes. The abstract things, like interfaces, will remain the same over a longer period of time. So if we depend on an abstract thing, it is likely that we will not be negatively influenced by it: it is supposed to be very *stable*.



Depending on abstract things

And this is where the two concepts, stability and abstractness, meet. If we consider stability to be the likeliness that something is going to change, then what is true for classes is also true for packages. As we know now it is better to depend on stable packages than to depend on unstable packages. Stable packages are less likely to change, so a depending package won't be negatively influenced by changes in its dependencies. In the same way it is safe to depend on abstract classes or interfaces because they are less likely to change.

We can follow the same reasoning while we apply the concept of abstractness to packages: it would be better to depend on an abstract package than on a concrete package. For the same reason: an abstract package would contain no particular implementation details that would be susceptible to change. Over a longer period of time it will stay the same.

How to determine if a package is abstract

The question is: is it possible to mark a package as either abstract or concrete? It is possible, even though "being abstract" is not a boolean value. There are many degrees of abstractness. We might consider a class to be abstract if it contains *at least one abstract method*. Then a class is concrete if it has no abstract (or virtual) methods.

We can determine the abstractness of packages in a similar way. A package is abstract if it contains no regular classes, only interfaces and abstract classes. And a package is concrete if it has at least one fully implemented, concrete class.

Still we'd need a little nuance here. According to this definition of abstract and concrete packages, a package with 10 interfaces and 1 concrete class would count as a concrete package, even though it contains much more abstract things than concrete things. Hence, we should also take the total number of classes and interfaces into account.

The A metric

The suggested way to find an indication of the abstractness of packages is to calculate the number of abstract classes and interfaces in a package, then to divide that number by the total number of concrete classes, abstract classes and interfaces in that package. The resulting thing would be a quotient with a value somewhere between `0` and `1`. We call this number the `A` metric for packages:

```
A = C-abstract / (C-concrete + C-abstract)
```

When the value of the `A` metric for a package is equal to or near `0`, it is a highly concrete package. It contains (almost) no interfaces, only concrete classes. Hence it's full of implementation details, and therefore liable to change.

When on the other hand the `A` metric is equal to or near `1`, it is a highly abstract package. It contains (almost) no concrete classes, merely abstract classes and interfaces. It is likely that these abstract things will stay the same over time. After all, only concrete classes and consequently concrete packages that are liable to change.

Abstract things belong in stable packages

So abstract packages are stable too. Or at least, they should be. This is where the *Stable abstractions principle* steps in:

> Packages that are maximally stable should be maximally abstract. Instable packages should be concrete. The abstraction of a package should be in proportion to its stability.

We already knew that a package should depend only on packages that are more stable. Now we also know that abstract things are likely to be more stable, i.e. they change less often and less dramatically. Hence concrete classes can safely depend upon them. But what if an interface is part of a highly unstable package? Then it is consequently not safe to depend on that package. The unstable package is likely to change. The interface inherits the instability of its containing package.

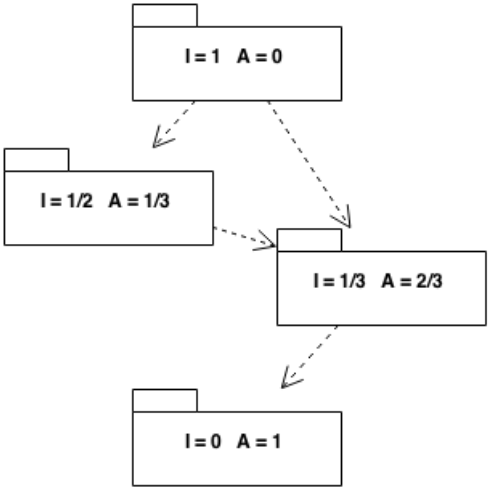
Interfaces and abstract classes are better off in a stable package. The stability of the package itself will be beneficent for the abstract things it contains. At the same time, the abstract things are good for the stability of the containing package. Packages that apply the *Dependency inversion principle* start to depend on it because of the abstract things it contains. This turns it into a more responsible package and will thereby force it to become more stable.

The opposite is also true: concrete classes are better off in unstable packages. The implementation details of the classes are likely to change anyway and this would better happen in an unstable package, which has less responsibility towards depending packages. If however a concrete class would be inside a highly stable package, it would make that package more unstable because a concrete class is liable to change.

Abstractness increases with stability

The *Stable abstractions principle* adds one extra requirement. It wants to unify abstractness and stability into this simple rule: a package should be as abstract as it is stable.

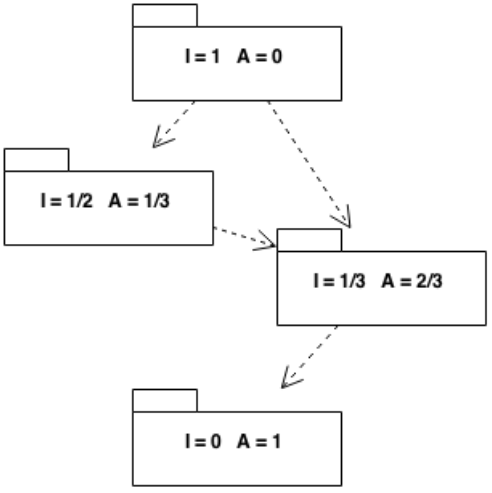
I assume you have successfully applied the *Stable dependencies principle* as explained in the previous chapter: you have calculated the `I` values for all the packages. When you draw them in a diagram, the packages with the highest value for `I` are at the top and those with the lowest value for `I` are at the bottom. When you travel from a package to its dependencies you encounter packages that only have decreasing values for `I`.



All dependencies go in the direction of stability

To figure out if you have also applied the *Stable abstractions principle* correctly, you also need to calculate the values for `A` (by dividing the number of abstract classes and interfaces by the total number of classes and interfaces). Then add the resulting `A` values to the dependency diagram.

Now you only need to verify that each dependency arrow leads to a package with a higher value for `A`. In other words: dependencies should have an increasing abstractness.



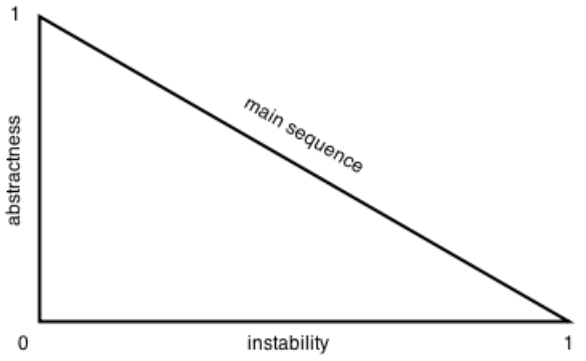
All dependencies go in the direction of abstractness

Strictly speaking the values for `I` and `A` added together should be exactly `1`. This would mean that all packages are *as abstract as they are stable.* But this is completely unrealistic. There is always some margin to this. However `I + A` should not be too far away from `1`. In general, highly abstract packages should be highly stable, and concrete packages should be unstable.

The main sequence

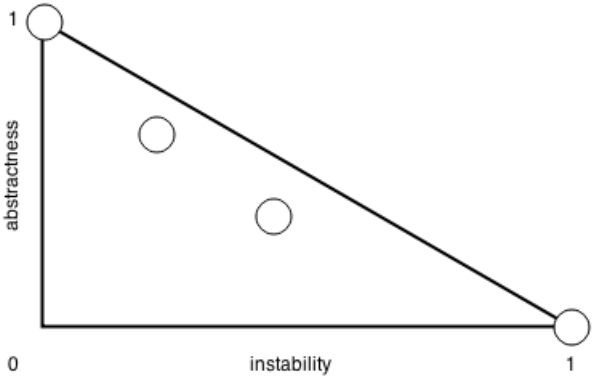
It is possible that stable packages contain concrete classes and unstable packages contain abstract classes or interfaces. These packages might be easy to spot, but there are many degrees of stability and abstractness. To find out which packages have imbalanced values for `I` and `A` we can plot the packages in a diagram called "the main sequence diagram".

We first draw a vertical ax going from `0` to `1`. It stands for the degree of abstractness of a package (expressed by `A`). Then we draw a horizontal ax going from `0` to `1`. This represents the degree of instability of a package (expressed by `I`). Finally we draw a diagonal line from the top-left corner to the bottom-right corner. This line is called "the main sequence".



The main sequence diagram template

Now we plot each package at the right spot in the diagram, depending on its values for `I` and `A`.



The main sequence diagram for the packages in the previous dependency diagram

You will know when you have applied both the *Stable dependencies principle* and the *Stable abstractions principle* correctly if all the packages are near or on the diagonal, i.e. the main sequence. According to this rule the previously shown main sequence diagram looks pretty good.

If you find any package that lies quite far from the main sequence, you should take a closer look at it. Consider its surrounding packages too and try to make some changes to its dependencies or its dependents in order to achieve the right amount of stability. It may also be necessary to change the abstractness of the package by relocating some

abstract classes or interfaces.

Once you have fixed the biggest issues with stability and abstractness, you should not forget to regularly come back to see if packages have not started to drift away from the main sequence. After some time the nature of existing packages may change because of new features being added to them and this may eventually cause an imbalance.

Types of packages

Concrete, instable packages

When you travel down the main sequence from the top-left to the bottom-right corner, you will first come across highly concrete, highly unstable packages. These should all be application-level packages. They are full of implementation details, specific to the actual project you are working on. They are allowed to be concrete, because no other package depends on them. Hence, they are unstable packages bound to change as often as the business changes.

Taking some more steps on the main sequence you will find in the middle packages that are somewhat abstract and somewhat stable. They are much less effected by external changes, but to a certain degree they are allowed to change themselves without bringing the whole project in danger.

Abstract, stable packages

When the journey on the main sequence ends, you will be in the realm of abstract, stable packages: the foundational blocks of your application. These contain lots of interfaces and abstract classes, which is why many classes (and consequently packages) depend on them. These classes apply the *Dependency inversion principle* correctly in order to be less susceptible to change. Hence, these abstract packages should be stable too. And they are, because they are responsible and have no dependencies themselves.

Interface packages

When you want to create highly abstract, stable packages, you may end up simply extracting the interfaces from existing packages and putting them all in one big *interface package*.

This is not the best thing you can do for your project. By putting all the interfaces in one package you are going to violate the *Common reuse principle*: some clients need just one or two interfaces from the package, still they would have to depend on the entire package.

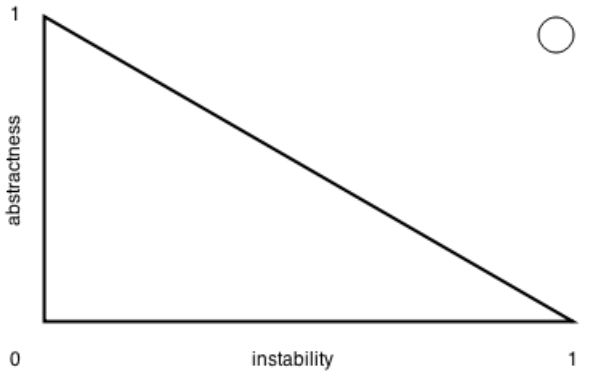
Besides, not all interfaces are meant to be depended on outside of a given package. You may have lots of interfaces that are only for private use within the same package (depending on your programming language you may be able to mark an interface as private).

One last thing to be aware of: before moving interfaces to a separate project, make sure that you have properly applied the *Interface segregation principle* to all of them. That way, clients won't be forced to depend on interfaces with methods they do not or should not want to use.

If you properly apply these rules, go ahead and create lots of small interface packages, each in support of one specific feature.

Strange packages

What happens in the corners that lie far away from the main sequence? What misbehaving packages can be found there?

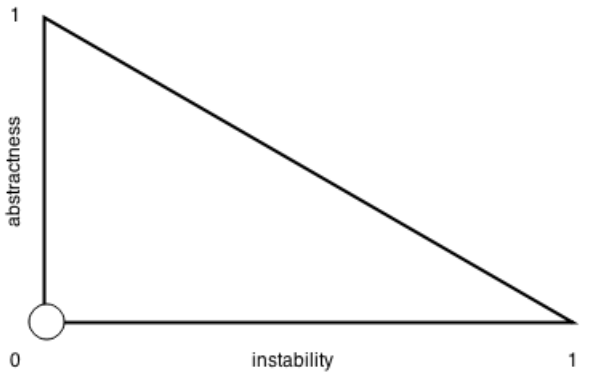


A package in the top-right corner

In the top-right corner we will find packages that are both highly abstract and highly unstable. These packages can thus be characterized as:

- Irresponsible (no other packages depend on them)
- Dependent (they depend on lots of other packages)
- Abstract (they contain only abstract classes or interfaces)

This is a very strange kind of package. It's very unlikely that such a package can be found in your project, because it would most likely contain *dead code*: it is never used by any part of the project, yet the code is abstract, meaning that it can not be used stand-alone - someone has to provide an implementation for it. In other words: packages like these are useless and you should try to get rid of them.



A package in the bottom-left corner

On the opposite side of the diagram, in the bottom-left corner, you will find packages that are highly concrete, yet highly stable. Such packages are:

- Responsible (many packages depend on them)
- Independent (they have no dependencies)
- Concrete (they contain only concrete classes)

This kind of package would be heavily used all over the project. Because it doesn't need any other packages to do its work, this is probably some kind of low-level library. However, it doesn't offer any abstractions for the things it does. This means that it's hard for classes in other packages to naturally comply to the *Dependency inversion principle*: they have to depend on the concrete classes from this package instead of interfaces.

The solution for getting this package back in shape is to apply the *Dependency inversion principle* to *depending* packages. They should not depend on the concrete classes from this package anymore, but instead depend on their own interfaces or interfaces defined in some more stable and abstract package.

Conclusion

In this chapter we have discussed the intimate relation between stability and abstractness. We learned that the more stable a package is, the more abstract things

it should contain. The counterpart of this is that the more unstable a packages is, the more concrete things it should contain.

We looked at the main sequence diagram to get a grip on different kinds of packages and where they are on the sliding scale between concrete/unstable and abstract/stable. Using this diagram we are able to spot packages with extraordinary characteristics.

Now that we are done discussing the actual design principles of package design, we are in a position to overlook the scene and formulate some general conclusions, remaining questions and suggestions.

Conclusion

Now that we have discussed so many design principles, you may feel that it's time to get a little bit more practical. You just want to finally start creating wonderful packages and share them with your coworkers, or even with the international open source software audience. I definitely think you should do that. Before you do, I want to give you some last words of advice.

Creating packages is hard

Reuse-in-the-small

In his book *Facts and Fallacies of Software Engineering*, Robert Glass mentions some interesting "facts" about reusable software components (or "packages"). First of all:

> Reuse-in-the-small (libraries of subroutines) [...] is a well-solved problem.

There are all kinds of little shared functions that are very useful and help you to quickly solve ever-recurring problems, like sorting an array, reading bytes from a file, etc. These are often distributed as "standard libraries" along with the runtime of your programming language.

Combining basic functions like these, you can produce some new low-level functions. As long as these functions are sufficiently small and unspecific, there should be no problem in reusing them in other projects. You only have to make sure that you release the code properly (as we discussed in *The Release/reuse equivalence principle*).

Reuse-in-the-large

When we keep combining and restructuring these low-level functions, we eventually produce complex and advanced software components. If you want to *reuse* entire software components in other projects, it's a whole different story. As Robert Glass puts it:

> Reuse-in-the-large (components) remains a mostly unsolved problem […].

It is much harder to prepare these bigger components for reuse than it is to write small, reusable functions. The main reason is that components usually fulfill an application-specific goal. Most components in a project are the result of actual requirements for that project.

Even though there are still lots of projects in the world that share some or a big part of their requirements (probably because they are part of the same business domain), no two applications are the same. This means that there is always some aspect of a component that wouldn't be useful in another project, or that would contradict some of the other project's requirements, etc.

When you are working on a component that you want to make reusable, it can be very hard, maybe impossible, to think of all the ways in which other developers might want to use it. Often you only recognize room for improvement when someone points it out to you.

Embracing software diversity

Robert Glass attributes the fact that reuse-in-the-large is so hard to something called *software diversity*:

> If, as many suspect, the diversity of applications and domains means that no two problems are very similar to one another, then only those common housekeeping functions and tasks are likely to be generalized.

Creating reusable components (or "packages") may only be possible if:

1. The domain of two projects is the same, or
2. The component offers general-purpose functionality.

In the first case, components model some part of the shared domain in a reusable way, leaving some details to the client. For example, it may be possible to reuse an online payment component in several e-commerce applications. Or if you write flight control software like NASA does, you will likely be able to reuse some components in the next project. As Robert Glass mentions, NASA reports the amount of code reuse to be around 70%, which should be mainly attributed to the fact that almost all of their projects have a shared problem domain.

In the second case, components offer some generally useful features, like logging things, talking to a server over HTTP, processing web requests, converting Markdown to HTML, etc. These types of things are useful in large subsets of all applications. As long as they offer lots of configuration options and apply the SOLID and package design principles correctly, these components or packages have a good change of being easily reusable.

Component reuse is possible, but requires more work

This leaves us with two situations in which it may be possible and useful to turn a component that you are developing into a reusable component. You can do so when you target clients in the same business domain. And you can do it if your component offers some really useful, somewhat low-level, general-purpose tools. However, even if the starting position is great, it may still be quite the job to actually make your component widely reusable.

When you intend your code to be reusable, you should be constantly concerned with extensibility, readability, automated tests, code quality, etc. If you want your reusable code to be successful in lots of projects, you should be aware of any environmental differences between those projects (different versions of the programming language, different operating systems, etc.). You have to provide some level of care for the package, like offering support or providing bug fixes. You need to offer some kind of a product experience, e.g. by writing a bit of documentation and providing usage examples. All of the effort you put into it leads to the following rule of thumb, as proposed by Robert Glass:

> It is three times as difficult to build reusable components as single use components [...].

After all of that, you would still be in the position where *only you* have used the component in a project. You don't know yet how it will behave in other projects; if it will live up to any other developer's expectations. Therefore a second rule of thumb is introduced:

> [...] a reusable component should be tried out in three different applications before it will be sufficiently general to accept into a reuse library.

Creating packages is doable

Reducing the impact of the first rule of three

I agree with the first rule of thumb. In my experience creating reusable software takes definitely more time than creating non-reusable software. I don't know exactly how much more, but three times more sounds about right. However, you can certainly influence this amount of time (and effort), by considering these factors:

1. The amount of features that your package implements
2. The area of the domain that your package covers
3. The level of extensibility of your package

If your package has too many features because you want to satisfy a group of users that is as large as possible, it is likely that you will spend ever increasing amounts of time on maintaining that package. All those features will start to get in each other's way. Fixing a bug in one feature, might break another feature. Besides, each of those features has its own dependencies, which makes the package quite unstable (see also *The Stable dependencies principle).

If your package tries to cover a big area of the problem domain, you will most likely be spending lots of time trying to implement all imaginable details that *can* be part of the domain. A business domain usually has so many aspects that may be slightly

or even largely different for distinct projects in the same domain that it's impossible to cover them all with your package. Users of your package will always be able to point out some more details that you overlooked.

These two factors for the amount of time and effort required to create a package are both related to the *scope of the package*. And our conclusion based on the above discussions should be that we *always need to limit scope*.

There is one other thing that we need to do in order to influence the amount of time that we need to invest: we need to make sure that our packages are highly extensible. If users of a package are not able to change the behavior of its classes without modifying or overriding the actual code, they will come to you to complain about that. Instead of asking you to add feature "foo", you should enable them to add that feature themselves. This should save you a lot of time and really make your package attractive for its users (if you want to make your code extensible, look at the SOLID principles again).

Reducing the impact of the second rule of three

The second rule of thumb said that we should try our reusable component out in at least three different applications. In my experience, this isn't necessary at all. I know that many people have been successful while applying this rule. But to me it sounds like an easy way to procrastination. Maybe you have written some very nice piece of code, yet you don't know if it works in other applications, so you keep it for yourself.

In reality you don't need to extensively test your package in other applications than the one you're currently working on. You can just think about your code and how it would be used in other projects. This would already provide you with some great ideas to make the code more generally useful and extensible. Still, there will always be some use cases that you didn't think of. And *wouldn't* think of, even if you tried your component out in six different applications. So I think that you should skip the second rule of thumb and just release your package. Its users will sooner or later (or never) give you some feedback about opportunities to improve your package.

Creating packages is easy?

In the above sections we changed our mind from "creating packages is hard" to "creating packages is doable". Could we take the next step and conclude that creating packages is *easy*? On the one hand, yes, I think that it can be very easy. In fact, you may already write your code as if it was going to be reused, because that makes your code in general much better (contrary to what people say about it not being pragmatic or something). It's just a couple of small steps to turn such well-written code into a reusable package.

On the other hand, it can be very hard as well to create reusable packages. In particular if your code was written with a single use case in mind and offers no easy ways to be extended or generalized. In that case I'd say you will have a hard time maintaining that code anyway, let alone turning it into a reusable package.

In general my advice when writing code is to write it as if it was going to be reusable. Every piece of your application should be limited in scope and be extensible too. It doesn't matter whether or not you're going to make a piece of code reusable. Write it well, and you will always be better off.

Appendices

Appendix I: The full Page class

```php
<?php
class Page
{
  public $uri = null;
  public $assigns = array();
  public $page = array();
  public $parent_node = 0;
  public $site_title = '';
  public $breadcrumbs = array();
  public $auto_include_dir = '';
  /* @public $smart Smarty */
  public $smarty = null;
  public $default_template = '';
  public $template = '';
  public $cms_login = null;
  public $user_login = null;
  public $available = true;
  public $is_user = false;
  public $is_admin = false;
  public $menu_items = array();
  public $caching = 1;
  public $cache_id = null;

  protected $_extra_request_parameters = array();

  /**
   * @param array $parameters
   */
  public function setExtraRequestParameters(array $parameters)
  {
```

```
    $this->_extra_request_parameters = array_values($parameters);
  }

  /**
   * @return array
   */
  public function getExtraRequestParameters()
  {
    return $this->_extra_request_parameters;
  }

  public function __construct($uri)
  {
    $this->connect_db();
    header('Content-Type: '.HEADER_CONTENT_TYPE);
    $this->smarty = new Smarty;

    if (isset($_GET['clear_cache']))
    {
      $this->smarty->clear_cache();
    }

    if (DEBUGGING)
    {
      $this->smarty->caching = false;
      if (trusted_ip())
      {
        $this->smarty->debugging = true;
      }
    }

    if (trusted_ip())
    {
      ini_set('display_errors', '1');
      error_reporting(
```

```
      E_ERROR | E_PARSE | E_WARNING
      | E_USER_ERROR | E_USER_NOTICE | E_USER_WARNING);
  }
  else
  {
    $this->smarty->debugging = false;
    ini_set('display_errors', '0');
    error_reporting(0);
  }

  $this->smarty->template_dir = ROOT.'/site/templates';
  $this->smarty->compile_dir = ROOT.'/site/templates_c';
  $this->smarty->use_sub_dirs = true;
  $this->default_template = DEFAULT_TEMPLATE;

  if (!table_exists('content'))
  {
    require(ROOT.'/includes/install.php');
    install();
  }

  $this->add_title_part(SITE_TITLE);

  $this->cms_login = new LoginClass('admins', 'cms_login');
  $this->user_login = new LoginClass('users', 'user_login');

  if ($this->cms_login->isLoggedIn())
  {
    $this->is_admin = true;
  }

  if ($this->user_login->isLoggedIn())
  {
    $this->is_user = true;
  }
```

```
    $parsed_url = parse_url($uri);
    $relative = ROOT;
    $url = $parsed_url['path'];
    $this->assign('header_content_type', HEADER_CONTENT_TYPE);
    $this->determine_page($url);
    if ($this->smarty->is_cached('', 'page_'.$this->page['id']))
    {
      $this->smarty->display(
        $this->default_template,
        'page_'.$this->page['id']);
      exit;
    }

    $this->smarty->register_function(
      'translate',
      'smarty_function_translate'
    );
    $this->smarty->register_modifier(
      'translate',
      'smarty_modifier_translate'
    );
    $this->smarty->register_function(
      'url_for',
      'smarty_function_url_for'
    );

    $this->open_page();
  }

  function get_page_info($id)
  {
    $result = mysql_query(
      "SELECT c.id, c.uri, t.title, t.menu_name, " .
      "c.node, c.skip_to_first_subpage FROM content c ".
```

```
      "WHERE c.id='$id';"
    );
    if ($result && mysql_num_rows($result))
    {
      return mysql_fetch_assoc($result);
    }
    return false;
  }

  function determine_page($url)
  {
    $url_parts = explode('/', trim($url, '/'));

    $page_ids = array();
    $uri_prefix = '';
    $parent_id = 0;

    foreach($url_parts as $key => $part)
    {
      if ($key == 0 || empty($part))
      {
        unset($url_parts[$key]);

        continue;
      }

      $result = mysql_query("SELECT id, skip_to_first_subpage ".
        "FROM content c WHERE t.uri='".addslashes($part)."' ".
        "AND c.parent_id='$parent_id';");
      if (!$result)
      {
        throw new RuntimeException('MySQL error');
      }

      if (mysql_num_rows($result))
```

```
    {
      $page_id = mysql_fetch_assoc($result);
      $parent_id = $page_id['id'];
      $page_ids[] = $page_id;
      unset($url_parts[$key]);
    }
    else
    {
      break;
    }
  }

  // remaining URL parts are extra request parameters
  $this->setExtraRequestParameters($url_parts);

  while(empty($page_ids)
      || $page_ids[count($page_ids)-1]['skip_to_first_subpage'])
  {
    $page_id = $this->find_first_subpage(
        $page_ids[count($page_ids)-1]['id']);
    $result = mysql_query("SELECT id, skip_to_first_subpage ".
    "FROM content WHERE id='$page_id';");
    if ($result && mysql_num_rows($result))
      $page_ids[] = mysql_fetch_assoc($result);
    else
      break;
  }

  $page = array();
  foreach($page_ids as $id)
  {
    $page = $this->get_page_info($id['id']);
    if ($page)
    {
      $uri_prefix .= '/'.$page['uri'];
```

```
        $this->breadcrumbs[] = array('id' => $page['id'],
        'uri' => $page['uri'], 'href' => $uri_prefix,
        'menu_name' => $page['menu_name'],
        'title' => $page['title']);
        if ($page['node']) $this->parent_node = $page['id'];

        $this->add_title_part($page['title']);
      }
      else
        break;
    }
    $this->page_url = $uri_prefix;
    $this->page = $page;
    $this->assign('breadcrumbs', $this->breadcrumbs);
    return true;
  }

  function redirect($page_id)
  {
    if ($this->page['id'] != $page_id && $page_id > 0)
    {
      session_write_close();
      header('HTTP/1.1 301 Moved Permanently');
      header('Location: '.$this->get_url($page_id));
      exit;
    }
  }

  function open_page()
  {
    $result = mysql_query("SELECT * FROM content c ".
      "WHERE id='{$this->page['id']}';");
    if ($result && mysql_num_rows($result))
    {
      $this->page = mysql_fetch_assoc($result);
```

```
        $this->page['contents'] = plain_text($this->page['contents']);

        if ($this->is_admin)
        {
          if (!$this->page['available_for_admins'])
          {
            $this->page['contents'] =
                TPL_NOT_AVAILABLE_FOR_ADMINS;
            $this->available = false;
          }
          else if ($this->cms_login->user['id'] != 1
              && !$this->page['available_for_guests']
              && !$this->page['available_for_users']
              && !$this->has_permission(
                  $this->cms_login->user['id'],
                  $this->page['id'])
                  )
          {
            $this->page['contents'] =
                TPL_NOT_AVAILABLE_FOR_SPECIFIC_ADMIN;
            $this->available = false;
          }
        }
        else if ($this->is_user)
        {
          if (!$this->page['available_for_users'])
          {
            $this->page['contents'] =
                TPL_NOT_AVAILABLE_FOR_USERS;
            $this->available = false;
          }
        }
        else
        {
          if (!$this->page['available_for_guests'] && !$this->is_user)
```

```
        {
          $this->page['contents'] =
              TPL_NOT_AVAILABLE_FOR_GUESTS;
          $this->available = false;
        }
      }

      if (!$this->page['show_contents'])
      {
        $this->page['contents'] = TPL_INVISIBLE;
        $this->available = false;
      }
    }
    else
    {
      $this->page['contents'] = TPL_NOT_FOUND;
    }
  }

  function has_permission($admin_id, $page_id)
  {
    $result = mysql_query("SELECT id FROM permissions WHERE ".
    "admin_id='$admin_id' AND page_id='$page_id';");
    if ($result && mysql_num_rows($result))
      return true;

    return false;
  }

  function find_first_subpage($parent_id = 0)
  {
    $result = mysql_query("SELECT id FROM content WHERE ".
    "parent_id='$parent_id' ORDER BY priority ASC, id ASC;");
    if ($result && mysql_num_rows($result))
        return mysql_result($result, 0, 0);
```

```
  }

  function show_page()
  {
    if ($this->available)
    {
      $this->cache_id = 'page_'.$this->page['id'];
      if ($this->page['include_file'] != ''
          && file_exists(ROOT.'/site/'.$this->page['include_file']))
      {
        include_once(ROOT.'/site/'.$this->page['include_file']);
      }
    }
    $this->menu = $this->load_menu();
    $this->assign('menu', $this->menu);
    $this->assign('page_id', $this->page['id']);
    $this->assign('site_title', $this->get_site_title());
    $this->assign('contents', $this->page['contents']);
    $this->assign('description', $this->page['description']);
    $this->assign('keywords', $this->page['keywords']);
    $this->assign('page_title', $this->page['title']);
    $this->assign('page_url', $this->page_url);

    $this->assign('subnavigation', $this->subnavigation());
    $this->assign('main_navigation', $this->main_navigation());
    $this->assign('is_admin', $this->is_admin);
    $this->assign('is_user', $this->is_user);
    $this->assign('parent_node', $this->parent_node);

    $this->assign('meta_title', $this->get_page_title(' - ', true));

    $this->assign('timers', sfTimerManager::getTimers());

    $this->smarty->caching = $this->caching;
    $this->smarty->display(
```

```
        ($this->template != '' ?
            $this->template
            : $this->default_template),
            $this->cache_id);
  }

  function add_title_part($title_part)
  {
    $this->title_parts[] = $title_part;
  }

  function get_title_parts()
  {
    return $this->title_parts;
  }

  function get_page_title($separator = ' - ', $reverse = false)
  {
    $title_parts = $this->get_title_parts();

    if ($reverse)
    {
      $title_parts = array_reverse($title_parts);
    }

    return implode($separator, $title_parts);
  }


  function menuitems($parent_id=0, $uri_prefix='')
  {
    $timer = sfTimerManager::getTimer('navigation');
    $timer->startTimer();

    $menu_items = array();
```

```
    $result = mysql_query("SELECT c.id, t.uri, t.menu_name ".
    "FROM content c "
    "LEFT JOIN content_translations t ON c.id = t.content_id ".
    "WHERE ".
    "c.parent_id='$parent_id' AND c.show_in_menu='1' AND (".
    ($this->is_admin ? "c.available_for_admins='1' OR ":'') .
    ($this->is_user ?
        "c.available_for_users='1'"
        : "c.available_for_guests='1'").
        ") ORDER BY c.priority ASC, c.id ASC;")
            or $this->trigger_error(mysql_error());
    if ($result && mysql_num_rows($result))
    {
      while ($item = mysql_fetch_assoc($result))
      {
        $item['href'] = $uri_prefix.$item['uri'];
        $menu_items[$item['id']] = $item;
      }
    }
    $timer->addTime();

    return $menu_items;
  }

  function load_menu()
  {
    $menu = array();
    $menu[0] = $this->menuitems(0, '/');
    foreach($this->breadcrumbs as $item)
    {
      $menu[$item['id']] = $this->menuitems(
          $item['id'],
          $item['href'].'/'
            );
    }
```

```
    return $menu;
  }

  function subnavigation()
  {
    if ($this->page['id'] != $this->parent_node
        && !empty($this->menu[$this->page['id']]))
      return $this->menu[$this->page['id']];
    else if ($this->page['parent_id'] != $this->parent_node
        && $this->page['parent_id'] != 0)
      return $this->menu[$this->page['parent_id']];
  }

  function main_navigation()
  {
    return $this->menu[$this->parent_node];
  }

  function get_uri_prefix($page_id)
  {
    $uri_prefix = '/';
    if ($page_id > 0)
    {
      foreach ($this->breadcrumbs as $item)
      {
        $uri_prefix .= $item['uri'].'/';
        if ($item['id'] == $page_id) break;
      }
    }
    return $uri_prefix;
  }

  public function get_url($page_id)
  {
    $timer = sfTimerManager::getTimer('get_url');
```

```
    $timer->startTimer();

    $url = '';
    while ($page_id > 0)
    {
      $result = mysql_query("SELECT c.id, c.parent_id, t.uri ".
      "FROM content c WHERE c.id='$page_id';");
      if ($result && mysql_num_rows($result))
      {
        $page = mysql_fetch_assoc($result);
        $url = '/'.$page['uri'].$url;
        $page_id = $page['parent_id'];
      }
      else {
        break;
      }
    }

    $timer->addTime();

    return $url;
  }

  function get_site_title()
  {
    $site_title = SITE_TITLE;
    foreach($this->breadcrumbs as $crumb)
    {
      if ($crumb['id'] == $this->page['id'])
        $crumb['title'] = $this->page['title'];
      if ($crumb['title'] != '')
        $site_title .= ' - '.$crumb['title'];
    }
    return $site_title;
  }
```

```
  public function connect_db()
  {
    $this->db_connection = @mysql_connect(
        MYSQL_HOST,
        MYSQL_USER,
        MYSQL_PASSWORD);
    if ($this->db_connection)
    {
      $this->db = @mysql_select_db(MYSQL_DB);
      if (!$this->db)
      {
        ?><p class="warning">Geen database.</p><?
        exit;
      }
    }
    else
    {
      ?><p class="warning">Geen verbinding.</p><?
      exit;
    }
  }

  function assign($name, $value)
  {
    $this->smarty->assign($name, $value);
  }

  function trigger_error($message, $error_type=E_USER_WARNING)
  {
    $this->smarty->trigger_error($message, $error_type);
  }
}
```