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 i

The situation ii

Overview of the contents iv

Notes about the code samples v

Thank you vi

Class design 1

Introduction 2
- SOLID principles 2
- Why follow the principles? 3
- Prepare for change 3

The Single responsibility principle 4
- A class with too many responsibilities 4
- Responsibilities are reasons to change 4
- Refactoring: using collaborator classes 5
- Advantages of having a single responsibility 5
- A small peek ahead: common closure 5

The Open/closed principle 6
- A class that is closed for extension 6
- Refactoring: abstract factory 9
- Refactoring: making the abstract factory open for extension 9

Refactoring: polymorphism . . . 9
Conclusion . . . 9
Packages need classes that are open for extension . . . 9

The Liskov substitution principle . . . 10
Violation: a derived class does not have an implementation for all methods 13
Violation: different substitutes return things of different types . . . 13
Violation: a derived class is less permissive with regard to method arguments 13
Violation: secretly programming a more specific type . . . 13
Conclusion . . . 13

The Interface segregation principle . . . 14
Violation: leaky abstractions . . . 14
Refactoring: create separate interfaces and use multiple inheritance . 16
What did we do? . . . 16
Violation: multiple use cases . . . 16
Refactoring: separate interfaces and multiple inheritance . . . 16
Violation: no interface, just a class . . . 16
Implicit changes in the implicit interface . . . 16
Refactoring: add header and role interfaces . . . 16

The Dependency inversion principle . . . 17
Example of dependency inversion: the FizzBuzz generator . . . 17
Making the `FizzBuzz` class open for extension . . . 17
Removing the specificness from the `FizzBuzz` class . . . 17
Violation: a high-level class depends upon a low-level class . . . 17
Refactoring: abstractions and concretions both depend on abstractions 19
Violation: a class depends upon a class from another package . . . 21
Solution: add an abstraction and remove the dependency using composition . . . 21
Conclusion . . . 21
Peeking ahead: abstract dependencies . . . 21

Package design . . . 22

Principles of cohesion . . . 23

Becoming a programmer . . . 23
The hardest part . . . 23
Cohesion . . . 23
Class design principles benefit cohesion . . . 23
Package design principles, part I: Cohesion . . . 23

The Release/reuse equivalence principle . . . 24
Keep your package under version control . . . 25
Add a package definition file . . . 26
Use semantic versioning . . . 26
Design for backward compatibility . . . 28
Rules of thumb . . . 30
Don't throw anything away . . . 30
When you rename something, add a proxy . . . 30
Only add parameters at the end and with a default value 33
Methods should not have side effects . . . 34
Dependency versions should be permissive . . . 36
Use objects instead of primitive values . . . 36
Use private object properties and methods for information hiding . . . 38
Use object factories . . . 40
And so on . . . 41
Add meta files . . . 42
`README` and documentation . . . 42
Installation and configuration . . . 42
Usage . . . 43
Extension points . . . 43
Limitations (optional) . . . 43
License . . . 43
Change log . . . 44
Quality control . . . 46
Quality from the user's point of view . . . 46
What the package maintainer needs to do . . . 47
Add tests . . . 48
Conclusion . . . 49

The Common reuse principle **51**
Signs that the principle is being violated 53
Feature strata 53
Obvious stratification 53
Obfuscated stratification 53
Classes that can only be used when ... is installed 53
Suggested refactoring 53
A package should be "linkable" 53
Cleaner releases 53
Bonus features 53
Suggested refactoring 53
Sub-packages 53
Conclusion 53
Guiding questions 53
When to apply the principle 53
When to violate the principle 53
Why not to violate the principle 53

The Common closure principle **54**
A change in one of the dependencies 55
Assetic 55
A change in an application layer 55
FOSUserBundle 55
A change in the business 55
Sylius 55
The tension triangle of cohesion principles 55

Principles of coupling **56**
Coupling 56

The Acyclic dependencies principle **57**
Coupling: discovering dependencies 58
Different ways of package coupling 58
Composition 58
Inheritance 58
Implementation 58

Usage . . . 58
Creation . . . 58
Functions . . . 58
Not to be considered: global state . . . 58
Visualizing dependencies . . . 58
The *Acyclic dependencies principle* . . . 58
Nasty cycles . . . 58
Cycles in a package dependency graph . . . 58
Dependency resolution . . . 58
Release management . . . 58
Is it all that bad? . . . 58
Solutions for breaking the cycles . . . 58
Some pseudo-cycles and their dissolution . . . 58
Some real cycles and their dissolution . . . 58
Dependency inversion . . . 58
Inversion of control . . . 58
Mediator . . . 58
Chain of responsibility . . . 59
Mediator and chain of responsibility combined: an event system . . . 59
Conclusion . . . 59

The Stable dependencies principle . . . 60
Stability . . . 61
Not every package can be highly stable . . . 61
Unstable packages should only depend on more stable packages . . . 61
Measuring stability . . . 61
Decreasing instability, increasing stability . . . 61
Violation: your stable package depends on a third-party unstable package 61
Solution: use dependency inversion . . . 61
A package can be both responsible and irresponsible . . . 61
Conclusion . . . 61

The Stable abstractions principle . . . 62
Stability and abstractness . . . 62
How to determine if a package is abstract . . . 63

The A metric . . . 63
Abstract things belong in stable packages . . . 63
Abstractness increases with stability . . . 63
The main sequence . . . 63
Types of packages . . . 63
Concrete, instable packages . . . 63
Abstract, stable packages . . . 63
Strange packages . . . 63
Conclusion . . . 63

Conclusion . . . 64
Creating packages is hard . . . 64
Reuse-in-the-small . . . 64
Reuse-in-the-large . . . 64
Embracing software diversity . . . 64
Component reuse is possible, but requires more work . . . 64
Creating packages is doable . . . 64
Reducing the impact of the first rule of three . . . 64
Reducing the impact of the second rule of three . . . 64
Creating packages is easy? . . . 64

Appendices . . . 65

Appendix I: The full Page class . . . 66

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

Thank you

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.

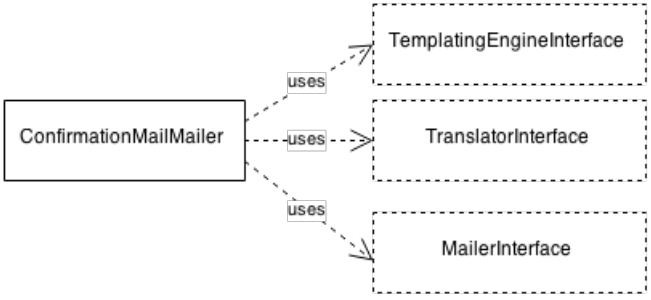
Prepare for change

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

A class with too many responsibilities



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.

Refactoring: using collaborator classes

Advantages of having a single responsibility

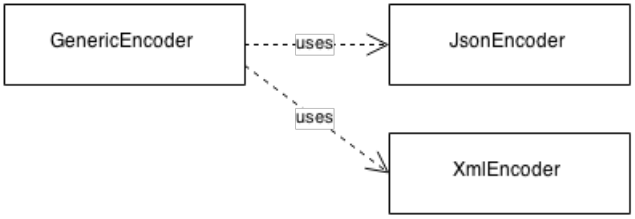
A small peek ahead: common closure

The Open/closed principle

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

Refactoring: making the abstract factory open for extension

Refactoring: polymorphism

Conclusion

Packages need classes that are open for extension

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

Violation: different substitutes return things of different types

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

Violation: secretly programming a more specific type

Conclusion

The Interface segregation principle

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[5] folder (Dropbox is a cloud storage service):

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

```
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.

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[6]. In it he mentions the *Law of leaky abstractions*:

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

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

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

What did we do?

Violation: multiple use cases

Refactoring: separate interfaces and multiple inheritance

Violation: no interface, just a class

Implicit changes in the implicit interface

Refactoring: add header and role interfaces

The Dependency inversion principle

Example of dependency inversion: the FizzBuzz generator

Making the `FizzBuzz` class open for extension

Removing the specificness from the `FizzBuzz` class

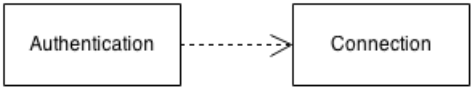
Violation: a high-level class depends upon a low-level class

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[7], which is based on PDO[8]). 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?*

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

> 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 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:

```php
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`

Violation: a class depends upon a class from another package

Solution: add an abstraction and remove the dependency using composition

Conclusion

Peeking ahead: abstract dependencies

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.

The hardest part

Cohesion

Class design principles benefit cohesion

Package design principles, part I: Cohesion

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[9]).

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

[9] 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:

```
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[10]. 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"[11] delivered by the Symfony framework team. It may become a good guide for you too.

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

[11] 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[12], 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.

[12] 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[13] 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.

[13] 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[14] 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.

[14] 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[15], 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

[15] 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

Feature strata

Obvious stratification

Obfuscated stratification

Classes that can only be used when … is installed

Suggested refactoring

A package should be “linkable”

Cleaner releases

Bonus features

Suggested refactoring

Sub-packages

Conclusion

Guiding questions

When to apply the principle

When to violate the principle

Why not to violate the principle

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

Assetic

A change in an application layer

FOSUserBundle

A change in the business

Sylius

The tension triangle of cohesion principles

Principles of coupling

Coupling

The Acyclic dependencies principle

Coupling: discovering dependencies

Different ways of package coupling

Composition

Inheritance

Implementation

Usage

Creation

Functions

Not to be considered: global state

Visualizing dependencies

The *Acyclic dependencies principle*

Nasty cycles

Cycles in a package dependency graph

Dependency resolution

Release management

Is it all that bad?

Solutions for breaking the cycles

Some pseudo-cycles and their dissolution

Chain of responsibility

Mediator and chain of responsibility combined: an event system

Conclusion

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*?

Stability

Not every package can be highly stable

Unstable packages should only depend on more stable packages

Measuring stability

Decreasing instability, increasing stability

Violation: your stable package depends on a third-party unstable package

Solution: use dependency inversion

A package can be both responsible and irresponsible

Conclusion

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.

How to determine if a package is abstract

The A metric

Abstract things belong in stable packages

Abstractness increases with stability

The main sequence

Types of packages

Concrete, instable packages

Abstract, stable packages

Strange packages

Conclusion

Conclusion

Creating packages is hard

Reuse-in-the-small

Reuse-in-the-large

Embracing software diversity

Component reuse is possible, but requires more work

Creating packages is doable

Reducing the impact of the first rule of three

Reducing the impact of the second rule of three

Creating packages is easy?

Appendices

Appendix I: The full `Page` class