Design Patterns for the Object-oriented and Functional mind

Kiko Fernandez-Reyes

2017-07-27

Contents

1 Prologue

This book was originally written for the course *Advanced Sofware Design*, imparted at Uppsala University by Dave Clarke and Kiko Fernandez-Reyes, and provides a practical hands-on experience on software design. It extends the content of the course with a direct application of design patterns and software principles in the functional and object-oriented paradigms while also considering their type system in a static and dynamic language.

1.1 Who is this book for?

This book is directed to students of any computer science major, young developers, engineers or programmers who have already grasped the idea behind object-oriented and/or functional programming. This book can still be used by those who have knowledge in an object-oriented language or a functional language althought, to get the most out of it, it is recommended to be familiar with both paradigms.

The reader should **already** be familiar with the following concepts from object-oriented programming:

- Classes,
- objects,
- inheritance,
- interfaces,
- abstract classes,
- parametric classes
- mixins and
- traits

and the following concepts from functional programming:

- Anonymous functions / lambdas,
- high-order functions,
- parametric functions,
- algebraic data types,

Paradigm / Type System	Static	Dynamic
Object-oriented	Java	Python
Functional	Haskell	Clojure

Figure 1:

- multi-functions,
- immutability,
- monads and
- arrows

For those who just need a quick reminder, we have included a *recap* chapter that summarises these concepts. If you are not familiar with neither object-oriented programming nor functional programming, you should first understand the ideas behind these paradigms and come back later to this book.

1.2 What is covered in this book?

This book covers most of the design principles and design patterns used today in industry, necessary for building maintable and flexible applications. All the principles and design patterns are written in four languages: Java, Python, Haskell and Clojure. These languages are already used in production in big companies and are well established. For instance, Boeing uses Clojure in their onboard diagnostic system, Facebook uses Haskell to prevent spam and phising attacks in their site, Pinterest uses Python for their backend operations and Java is widely used everywhere.

The mix of these languages covers two different paradigms (object-oriented and functional programming) and two different kinds of type systems (static and dynamic), shown in Table 1.1.

Tabla 1.1 Relation between paradigm and type system

Why multiple languages?

Each language has its strengths and weaknesses. In this book, I show how to apply the same design pattern by fully exploiting the main strengths of each language.

Choosing a language is an important design decision as languages are tied to a paradigm and a type system, and these cannot be changed. For instance, Python has some functional features but, all in all, **Python is an object-oriented language**. Consider using Python for the things it is good for. Should I tell you what it is good for? ummm... Not that fast young padawan, it's only through the journey that one finds its own truth.

In this book, we do not provide an absolute truth nor do we try to do so, we merely **show that the combination of different paradigms and type systems have inherent benefits and drawbacks**. You have to work the examples in the book to find your own truth and comfort zone.

1.3 Organisation

The book is organised in 6 parts. The first part (chapters 1 and 2) puts in perspective what is software design, a recap to object-oriented and functional programming, as well as the main differences between static and dynamic type systems. The second part introduces software principles to create flexible and maintanable software, known as SOLID and GRASP principles. Parts 3 to 5 explain, in depth, core design patterns that are grouped into 3 categories: creational, structural and behavioral patterns. The last part builds a text-based game using concepts and design patterns covered in the book.

Each software principles and design pattern is explained in depth: there is a clear and concise explanation of the design pattern (sometimes ilustrating a simple simil from the real world), its benefits and drawbacks. Afterwards, the pattern is explained in the context of its paradigm and type system, and we include easy to grasp examples to get you familiar with the language and the design pattern. As mentioned above, for a full example that combines software principles and patterns, you should refer to the last part.

1.4 Software requirements

This book contains code written in Java, Python, Clojure and Haskell. This section contains the exact versions of the languages used in this book. All code contained in this book can be found at https://github.com/kikofernandez/asd-book.

1.4.1 Java 8

All examples written in this book has been tested using the Oracle JDK 8u111.

Instructions on how to install Java can be found in http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

1.4.2 Haskell

All examples written in this book has been tested in GHC 7.10.2.

Instructions on how to install GHC can be found in https://www.haskell.org/ghc/.

1.4.3 Clojure

All examples written in this book has been tested for Clojure 1.8.0.

Instructions on how to install Clojure can be found in http://clojure.org/community/downloads.

1.4.4 Python 3.6

All examples written in this book has been tested for Python 3.6.0.

Instructions on how to install Python 3.6 can be found in https://www.python.org/downloads/.

2 Introduction

You worked really hard for over a year in the next one billion dollar project in a small startup: the CEO, Johan, seems happy that the product is (almost) feature complete (it's never feature complete), the designer, Anders, is happy with the UX and graphics and the CTO, Pontus, just wants to press the red button and release the product. You tell them to wait 10 more minutes, the unit tests, integration tests and Xxxx tests are still running (adding tests was a long battle with everyone because it slowed down the project quite a lot, but you are a good engineer and tests will catch many errors now and in the future).

- Ding, ding, ding (sound)

You look at the screen, all 142 tests are green! You tell Pontus to push the red button; deployment scripts start, your software is being installed on all the AWS servers the company could afford, a few database servers with replication for faster reads and a single leader for the writes, load balancer, reverse proxy servers for serving static data and another bunch of servers running your application. After a few minutes, the

deployment is a success, your baby is ALIVE and by tomorrow morning your product will be featured in all the Swedish newspapers (you live in Stockholm of course, the European capital for startups).

The next day comes, you try the service as if you were a customer and observe that the system is just f****** slow, at times it seems that connections time out and you observe in your monitoring tools that the server is not processing that many requests. What the hell is going on!?

The CTO has the intuition that it could be related to that lock that was introduced for the concurrent access to the database and it may be as simple as releasing it. If that's the case, you'll fix the test that should have covered that and be presto in 5 min. There is one problem though, the code doesn't have a well defined structure and the lock somehow was passed inside a lambda – you had to try out lambdas in Java, they are cool – so you have no access to release it directly. More over, you are passing a bunch of closures and there's no easy way to tell which one has the damn lock! What do you do? You decide to create a new function that this time does what you want and somehow you'll fix the issue later. Phew, everyone knew you could solve it!

Congratulations, you are the owner of a big plate of spaghetti code that no one can follow and, as a side effect, you cannot be fired.

How did you end up in this situation? I have the answer for you: you coded a system considering all the functional requirements and the architecture without any solid foundation –you forgot the problem to solve, the non-functional requirements – because it was more important to try out all the new fancy features of the language.

Many books explain software design with focus on one language (and paradigm) and assume that this knowledge can be "easily" extrapolated to other languages and/or paradigms. While it is true that you have to start from some point, it is also true that as soon as you put it to practice in another language their explanations may not be easy follow now or may not even make sense from the language perspective (see Fig. 2 for an example of the author's feelings when reading such books).

Our approach in this book is much more practical and can be extrapolated to other languages. Yes, we am not contradicting the previous paragrah, you heard correctly! We are going to cover software design from the perspective of different languages (paradigms and type systems), and we are going to show different approaches to solving the same problem. You will learn to take advantage of each of the language constructs and benefit from it.

Before we start, let's do a recap on what **software design** means.

2.1 Software Design

Software design is the process of finding a *satisfactory* solution to a problem. Before one can start with an implementation there needs to be a clear specification of the problem to solve, a good understanding of the domain of the system under design (you can draw a domain model to clarify ideas), an (in)formal document that describes the steps necessary to solve the problem (the Unified Modelling Language can help here with class and sequence diagrams, among others) and a well-defined architecture.

Given the definition above, designing software seems to be completely disjoint from agile methodologies (e.g. Scrum) where you break user stories into smaller tasks until everything is clear to the whole team. In Scrum there seems to be no domain model, not a single static and/or behavioural diagram, no architecture, etc, just *code what the post-it says*. However, the post-it tells you **what the problem is** but not **how to solve it!**. The domain modelling, static and behaviour models and architectural diagrams are all tools under your belt that may be used if required, even in agile methodologies.

Software design is all about making decisions. Every problem involves taking small and big decisions and these influence the final outcome of your software. For instance, the "simple" task of choosing a programming language has a tremendous impact on your software. Choose a dynamic language and you'll find quite a lot of errors at runtime (even if you use a test-driven development approach). Other example is choosing between a object-oriented or a functional language. No matter which one you choose, you can always write

Paradigm / Type System	Static	Dynamic
Object-oriented	Java	Python
Functional	Haskell	Clojure

Figure 2: Developer taming code

the same piece of software using one paradigm or the other. However, one of them will bring inherent benefits that have to be otherwise coded in the other approach.

2.1.1 What is a good design?

In general, **a good design is one that can deal with change**. A software engineer has to think much more abstract than other engineers, e.g. civil engineer. For instance, as a civil engineer, you may have to think about building a bridge that can stand `X` number of tons and, under any circunstance, not fall down. As a software engineer, you have to think about a piece of code that today is a car, tomorrow is a submarine and the next day is a tank and this happens because it is just code, anyone can update it and it is not set in stone, ever.

There are many ways to solve a problem and, more often than not, there are common solutions. Because of this, many problems has been studied for many years and they have a well documented solution. These well-documented solutions are called **design patterns** and they exist for all kinds of paradigms. In this book, we will focus on object-oriented and functional design patterns.

2.1.2 Design patterns

A design pattern is a well-desgined solution to a common problem. Design patterns are expressed in a concise and clear format, and they follow a structure similar to the one below:

- *pattern name*, so that experts can refer to a common pattern
- *concise and well-defined problem specification*
- *unequivocal solution*
- *trade-offs of the pattern*

In this book, we follow this structure together with auxiliary UML class diagrams (explained in later chapters).

3 Recap

This section covers basic concepts from the object oriented and functional paradigms. More conretely, by the end of the chapter you should be familiar with the following concepts from object-oriented programming in Java and Python:

- Classes,
- objects,
- inheritance,
- interfaces,
- mixins and traits,
- abstract classes and
- parametric classes

and the following concepts from functional programming in Clojure and Haskell:

- Anonymous functions / lambdas,
- high-order functions,
- parametric functions,
- algebraic data types,
- multi-functions
- immutability
- recursion

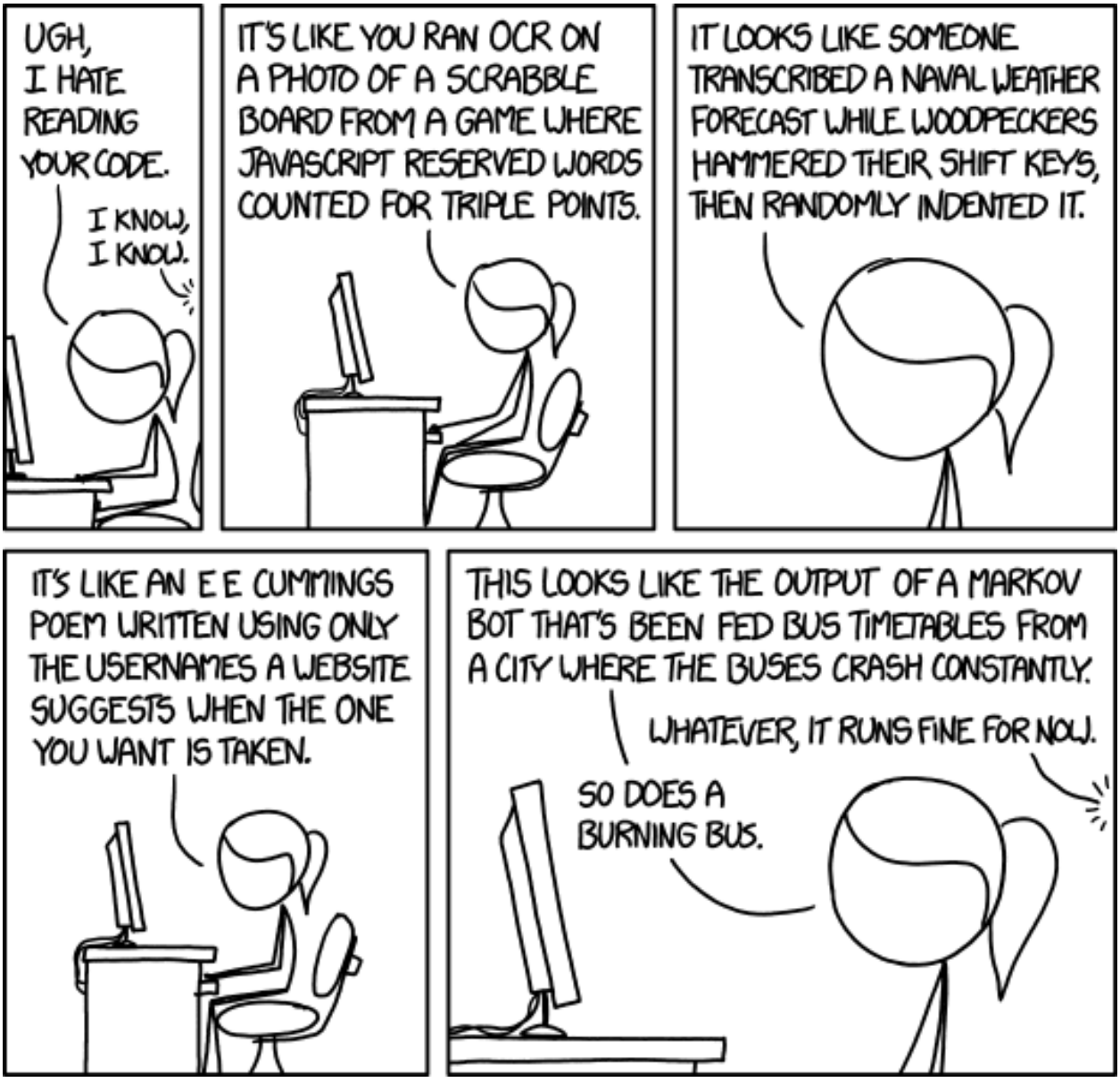

Figure 3: Code quality (http://xkcd.com/1695/)

Keep in mind that some languages have features that do not exist in other languages. For instance, algebraic data types exist in Haskell but not in Clojure and parametric classes exist in Java but not in Python. In the former case, algebraic data types cannot be fully build in Clojure due to its dynamic typing nature – Clojure cannot typecheck the difference between data types and data values (we'll explain what this means later on). In the latter case, parametric functions do not make sense for Python due to, again, its dynamic nature. If you got the feeling that this does not make sense and you cannot follow, everything will be explain with examples soon, when we cover these basic concepts. Let's begin!

3.1 Object-oriented concepts

The concepts covered in this section are shown in Java and Python.

Note: Clojure is an impure language that mixes the functional programming concepts with the object-oriented paradigm. In the remaning chapters, we omit using the object-oriented capabilities of Clojure, since the objective is to learn the main benefits that each language brings to the table.

3.1.1 Classes

Classes describe the state and behaviour of an object. The state of a class lives in its attributes while the behaviour is expressed via its methods. A class that hides its attributes with a `private` access modifier (remember `public`, `protected` and `private`?) protects its internal state from other classes. Now, the only way to update the internal state is via method calls (**they represent the behaviour of the object**). As a general advice, you should not expose the internal state of your class to others; you should expose your behaviour. This is what makes abstractions great, trust on my behaviour and not on my internal structure. My favorite abstractions are tasks and futures and they hide the internal details of spawning a new thread.

Classes expose their behaviour via its methods. If those methods just get the attributes and set them, we call them *getters* and *setters*.

That's enough for an introduction to something that should you already now, let's see some code! Let's start with a typical example and work on it:

Idea: Let's model a dog, who has as state her alertness and her behaviour is to bark only when she is startled. As owner, you can check if the dog is startled or relaxed.

Java

The class `Dog` has as state a *private* attribute `alert`. (the attribute is not accesible from outside the class). The constructor of the class (`public Dog()`) creates a new `Dog` and sets its state to some default value. To get the state outside the class we use *getters* and *setters* methods. These are preceded by the words `get` and `set` following the attributes name, e.g. `getAlert`. This is represented in the Example 3.1.1.1.

```
class Dog {
  // state of the Dog
  private boolean alert;

  // constructor
  public Dog(){
    this.alert = false;
  }

  // getter method
  public boolean getAlert(){
    return this.alert;
  }
```

```
  // setter method
  public void setAlert(boolean newAlert){
    this.alert = newAlert;
  }

  // behaviour
  public void bark(){
    if (this.alert) {
      System.out.println("Woof Woof");
    }
  }
}
```

Example 3.1.1.1. Introductory example to Java

Python

Python doesn't have access modifiers and uses (by convention) an underscore (or two, let's now go into why) to mean that the attribute is private. The constructor `def __init__(self, alert=False)` method takes explicitly an instance of itself `self` and a default argument, `alert` that, if it's not provided, it is set to `False`. Before we continue disecting the constructor, let's briefly introduce getters and setters.

Python provides a special syntax for getters and setters that wrap the attribute into a function with that very same name. For instance, the getter for the `alert` attribute is created by declaring a method with the name of the attribute and the `@property` on top of it. The body of the method just fetches the attribute.

```
    @property
    def alert(self):
      return self.__alert
```

Setters work in the same way, except that they are annotated with the attribute's name followed by the `setter` word, e.g. `@alert.setter` for the `alert` setter method. From now on, when we call on `self.alert` we are actually calling the getter method and when we assign `self.alert = True`, we are calling the setter method.

Now, if we go back to the `__init__` method from Example 3.1.1.2, we can observe that this is indeed the desire behaviour.

Why would we want to use getters and setters like that? Young padawan, they are an abstraction. Now you only want to retrieve the data but, with this abstraction, you could be returning cached data that otherwise needs to be fetched from the network. Later on, we will learn about the decorator pattern and how it rocks in Python!

```
class Dog:
  def __init__(self, alert=False):
    self.alert = alert

  @property
  def alert(self):
    return self.__alert

  @alert.setter
  def alert(self, new_alert):
    self.__alert = new_alert

  def bark(self):
    if self.alert is True:
      print("Woof Woof")
```

Example 3.1.1.2 Introductory example to Python

3.1.2 Objects

Objects represent instances of the class at runtime. An object gets the default state defined by their constructor and the behaviour defined from the class declaration.

Java

Following the example before, let's create an object:

```
Dog dog = new Dog();
```

This calls on the constructor of the `Dog` class that sets the `alert` state to `false`. If the dog tries to bark nothing will happen:

```
dog.bark();
```

The dog barks when it is startled, let's make her bark:

```
dog.setAlert(true);
dog.bark();
// prints "Woof Woof"
```

Python

Creation of an object is similar to Java:

To make the dog bark:

```
dog.bark()
```

which does nothing because the dog is not startled. Let's use the setter to change its state and make her bark:

```
dog.alert = True
dog.bark()
```

```
## Woof Woof
```

Notice that in this case, we could have set the initial state of the dog and would not have had to use a setter:

```
dog = Dog(True)
dog.bark()
```

```
## Woof Woof
```

3.1.3 Inheritance

A class can inherit methods from other classes, called inheritance. The class that inherits methods is called *subclass* while the class that provides them is called *super class*. Java has single inheritance, meaning that the subclass can inherit only from a single class while Python has a limited form of multiple inheritance.

Idea: A computer has various levels of memory, L1, L2, Last-level cache and main memory. In this example, the class `Computer` inherits operations from memory, so that we can call methods that retrieve objects from main memory. (this is an example and I would not advice any one to create a virtual machine based on this idea)

Java

Since Java can only inherit from a single class, we can model the idea above as shown in *Example 3.1.3 Java inheritance*.

```
class Memory {

  private CacheL1 l1;
  private CacheL2 l2;

  public Memory(){
    int kilobyte = 1024*1024;
    int megabyte = 1024 * kilobyte;
    this.l1 = new CacheL1(512*kilobyte);
    this.l2 = new CacheL2(3*megabyte);
  }

  public Object fetchLocation(Location lo) { // ... }
}

class Computer extends Memory {
  public class Computer(){
    // ...
  }
}
```

Example 3.1.3 Java inheritance

The class `Computer` inherits the methods `Memory` when `Computer extends Memory`. This means that we can fetch objects from memory, `computer.fetchLocation(location);`.

Python

Python supports two forms of multiple inheritance, known as old-style and new style. In the old-style form, the multiple inheritance works in depth-first and left-to-right order. For instance:

```
class L1(object):
  def send_info():
    print "Sending from L1 to bus"

class L2(object):
  def send_info():
    print "Sending from L2 to bus"

class Cache(L1, L2):
  pass

class CPU(object):
  def send_info():
    print "Sending info from CPU to bus"

class Computer(CPU, Cache):
  pass

c = Computer()
c.send_info()
```

Example 3.1.3 Python old-style

In this case, the `c.send_info()` method prints `Sending info from CPU to bus`. If we were to change the

order of the inherited classes, `class Computer(Cache, CPU)`, then the printing method would have been `Sending from L1 to bus`.

The multiple inheritance new style kicks in when classes do not inherit from `object` (as opposed to what `L1` and `L2` do in *Example Python old-style*). The new style considers breath-first and left-to-right order.

```
class L1:
  def send_info():
    print "Sending from L1 to bus"

class L2:
  def send_info():
    print "Sending from L2 to bus"

class Cache(L1, L2):
  pass

class CPU:
  def send_info():
    print "Sending info from CPU to bus"

class Computer(Cache, CPU):
  pass

c = Computer()
c.send_info()
```

Example 3.1.3 Python new style

In *Example 3.1.3 Python new style*, the `c.send_info()` method would have printed `Sending info from CPU to bus` given that `Cache` does not provide a suitable `send_info` method explictly.

(More details in here: http://www.python-course.eu/python3_multiple_inheritance.php)

3.1.4 Interfaces

TODO: missing default methods and static methods in interfaces (they can have implementations!)

Interfaces declare a contract of the expected behaviour of classes that implement the interface. An interface *merely* defines methods. Classes that would like to be compatible with the interface need to implement those methods. The real purpose of interfaces is to enable polymorphism, e.g. being able to call methods on different classes because they implement the same interface. Java has interfaces but Python completely lacks them. Dynamic languages such as Python cannot express interfaces because they don't have a static type system, the fact of not having static types inhibits interfaces.

Java

In Java an interface is declared similar to how classes are declared, except that you use the word `interface`.

```
interface MemOperations {
  public Object fetchLocation(Location loc);
  public void flush();
  public void sendToBus();
}

public class CacheL1 implements MemOperations {
```

```
  public CacheL1(int size){
    //...
  }

  public Object fetchLocation(Location loc){
    // ...
  }

  // ...

  public void flush(){
    System.out.println("Flushing from CacheL1");
  }

}

public class CacheL2 implements MemOperations {
  // ...

  public void flush(){
    System.out.println("Flushing from CacheL2");
  }
}
```

Example 3.1.4 Java interfaces

If you forget to implement one of the methods declared in the interface, the type system will throw an error and complain that you forgot to implement the missing methods. You won't get unexpected crashes at runtime because the method doesn't exist.

From the *Example 3.1.4 Java Interfaces* we can create different cache levels and treat them as if they are just `MemOperations`. Calling methods on each one prints a different `String`.

```
CacheL1 c1 = new CacheL1(512);
CacheL2 c2 = new CacheL2(2048);

ArrayList<MemOperations> a = new ArrayList<MemOperations>();
a.append(c1);
a.append(c2);

for(cache: a){
  cache.flush();
}

// prints:
// "Flushing from CacheL1"
// "Flushing from CacheL2"
```

As of Java 8, interfaces can declare a *default* implementation for methods. This is great from the design point of view: you can provide a default implementation as long as you rely only on the known behaviour (other public methods of the interface). This means that you can create a clean design that does not rely on internal state and/or methods.

TODO: Template pattern fits here perfectly!

TODO: Should I explain polymorphism here? It's GRASP...

TODO: Should I show a code example?

TODO: I believe I am explaining super basic stuff and should level up...

3.1.5 Abstract classes

Abstract classes can be seen as a mix between interfaces and classes: they can contain attributes, define methods and declare abstract methods. Abstract methods provide a declaration of the behaviour but not a definition (implementation). An abstract class cannot work on its own – cannot be instantiated alone – and needs to be subclass by another class, which implements the abstract methods.

Java

It may seem as if Interfaces (as of Java 8) and Abstract Classes serve the same purpose. However, there is a subtle difference: an interface can only create public methods and constant declarations (*static* and *final* attributes) while an abstract class can declare and define the internal state of the class.

As an advice, I would recommend to use interfaces whenever you have various subclasses that need to implement the same behaviour. I would consider the use of an abstract class in limited cases and only when there is a need to share the same private state among objects that are quite similar.

TODO: Favour composition over inheritance

```
abstract class Entity {
  private int life;
  private Equipment equipment;

  public Entity(int life, Equipment Equipment){
    this.life = life;
    this.equipment = equipment;
  }

  public int getLife() { return this.life; }
  public void setLife(int life) { this.life = life; }

  abstract void attack(Enemy e);
  abstract void escape();
  // ...
}

interface Attackable {
  public int getPower();

  default void attack(Enemy e){
    e.setLife(e.getLife() - getPower());
  }
}

public class Hero {
  public void attack(Enemy e){
    // Filter equipment by things that I can attack with
    Attackable attackable = this.equipment.attackable();

    // attack
    attackable.attack(e);
```

```
  }
}
```

TODO: equipment may use the null pattern

Python

Test [1]

3.1.6 Parametric classes

3.2 Functional concepts

3.2.1 Anonymous functions / lambdas

3.2.2 Immutability

3.2.3 High-order functions

3.2.4 Parametric functions

3.2.5 Algebraic data types

3.2.6 Multi-functions

4 SOLID principles

TODO: Explain SOLID design principles. TODO: Put examples in Java and Python (easy) TODO: Put examples in Clojure and Haskell TODO: Fill out all principles

4.1 Single Resposibility

Single Resposibility Principle

4.2 Open/Close Principle

Open/Close Principle

4.3 Liskov Substitution

Liskov Substitution

4.4 Interface Segregation

Interface Segregation

[1] Python allows creation of abstract classes via meta classes. These concept is a bit more advanced and may be covered upon receiving some initial feedback.

4.5 Dependency Inversion

Dependency Inversion

5 GRASP principles

Before you learn advanced design patterns, it's useful to look at common principles / recommendation rules to guide your design. By following these principles, you will design and write code that's easy to understand, maintain, and refactor.

By the end of this chapter you will have grok these principles and they will become second nature to you. Next, we introduce these principles (which can be applied to any object-oriented language): - low coupling, - high cohesion, - creator, - information expert, - controller, - polymorphism, - indirection, - pure fabrication, and - protected variation.

There are exercises for each principle at the end of the chapter to help you understand the difference between them.

5.1 Low Coupling

This principle states that a class should only depend on the minimum and required amount of classes, no more and no less. A low coupled class is easy to maintain and refactor because it interacts with a minimum amount of objects. A high coupled class has many dependencies to other objects and has low cohesion, i. e. it is not focused, has too many responsibilities and does too many things.

A low coupled design is easy to change and the changes do not spread across multiple classes. In the beginning of your journey to become a better programmer, it's difficult to acknowledge this principle until you deal with its counterpart, a high coupled bowl of spaghetti code. For this reason, it is much easier to spot a highly coupled design than to identify a low coupled one.

We define coupling as any code that satisfies items in the following list: - the class has an attribute to another object, - calls on methods of other objects, - inherits from another class, - implements one or more interfaces, and - inherits mixins

The reader should notice that there will always exist coupling but, dependencies to stable items are not problematic – standard library – because these have been well designed and do not change often. The problem is not coupling per se, but creating a highly coupled design to unstable objects.

Example: (Example of high coupled design)

Better alternative is the following code:

(Example of low coupled code)

Exercise based on case study

5.2 High Cohesion

Cohesion refers to the property of staying focus on the responsibilities of the class. A cohesive class does one thing and does it well. Highly cohesive code is focused and, as side effect, doesn't talk to too many objects.

Often you'll see code that contains many methods, attributes and seems to contain a lot of logic and magic. From now on, please know that this is the antipattern known as"The Blob", result of a high coupling and low cohesion.

(Image of the Blob)

Example of The Blob antipattern:

(Code, better if it's from a real example)

You can identify highly cohesive code if: - code doesn't mix responsibilities, - has few methods, - doesn't talk too many objects.

Dependence relationship: *Coupling* and *cohesion* usually go hand in hand, since code with low cohesion is code that does too many things, hence it relies on too many objects.

One way to look at it is to understand coupling as the relation between subsystem and cohesion as the relation within a subsystem. For instance, the following code shows these two weaknesses in Figure X.

(Listing of high coupled and low cohesion code)

(Figure X)

These figure further shows how a change in one class affects many other classes thus, the code becomes more complex and difficult to maintain. If we refactor this code, it ends up as listing Yyy, Figure Yy6y.

(Listing Yyy)

(Figure Yy6y)

Exercise: given the following code, add the following functionality: - feature a - feature b - feature c *Reflection*: - How many classes did you change? - does it seem like such a design is flexible and easy to refactor?

Exercise: Refactor the baseline code from the above exercise to achieve low coupling and high cohesion. Add more features a, b, and c. *Reflection*: does it seem like this design is flexible and easy to refactor?

5.3 Creator

This principle is likely overlooked but it plays a crucial role in object-oriented programming; the creator *establishes who is responsible* for creating an object. If you defined it well, your code will exhibit loose coupling and high cohesion; err an you will begin your journey to maintaining spaghetti code.

Application: a class `X` is the creator of an object `Y` if any of the following statements are satisfied: - class `X` initialises object `Y` - class `X` contains object `Y` - class `X` closely uses and depends on class `Y`

For instance, a point of interest (POI) has comments, hence POI is the creator of instances of the `Comment` class.

Exercise: what kind of benefits and drawbacks do I get if `POI` is not the creator of comments? Solution: if comments are always the same, just text, then it makes sense that `POI`is the creator. However, what if I want to support multiple kinds of comments, i.e. video comments, text, or images as à comment? In that case, you are better off injecting the comments as an extra argument to the POI object. – does this makes sense? POI is always created first and comments are added later on, it's not an initialiser but a method that adds comments.

5.4 Information Expert

Classes have methods that define their responsibility and behaviour. A class exposes its behaviour via methods and its internals should always be opaque. With this basic idea, this principle helps you to identify the behaviour of each class.

A class is responsible for a behaviour if that class contains all the information to carry it out. For example, which object should be responsible for updating the description of a guide? a. `User` object b. `POI` (point of interest) object c. `LatitudeLongitude` object d. `Guide` object

This one is easy. Lets consider each option: a. If we update the guide description via `this.user.setGuideDescription(newDes` it implies that each user has a single guide, otherwise we would not know which guide we are updating. It could also mean that each user may be uniquely identified by its email and its guide. This is a really bad design from the relational point of view, database-wise. When representing the data layer from a NoSQL database this is allowed and I have seen it in use. I do not think this is a good model since, in the database, each record with the the same email is treated independently. b. A point of interest, such as a view, may be linked to multiple guides. For this reason, similar to above, such a design is non-maintainable. c. This really does not make sense. d. We have a winner! The guide instance has all the information to update the guide.

The information expert assigns responsibilities to the classes that have all the information to fulfill the action.

Exercise: A user wants to submit a few pictures with a review of her favourite restaurant in Málaga, Spain. Which of the following classes should be the information expert and provide that behaviour? a. `User` object b. `Guide` object c. `Review` object

5.5 Controller

Controller

5.6 Polymorphism

This principle is one of the most important ones in object-oriented programming and the one that makes OOP great at dividing responsibilities between classes.

As we saw in the recap section, this polymorphism refers to classes that implement an interface or inherit from a top class and not to parametric classes, which is also polymorphic on the (opaque or bounded) type variable.

This principle allows classes to specify the same responsibilities via an interface but decouples the behaviour for each type. For instance, in our application, we want to show a special logo on top of the pictures of famous users who have confirmed their identity. A valid design, that does not scale, has a single `User` class with an attribute named `confirmedIdentity` which sets the flag to true when the user has confirmed its identity. This design works for 2 users, the normal and confirmed users. Tomorrow, Johan (CEO) wants to add a new kind of user who represents a company instead of a person, companies cannot create accounts and they have confirmed its identity. Creating a company's profile as a confirmed user seems wrong and error prone, it makes no sense the attribute `confirmedIdentity` for a company's profile because we know that this will always be true. The current code looks like listing Xxxx.

(Listing Xxxx)

Another design choice is to represent this distinction of different users using an enum attribute. Based on this value, the `displayImage()` method adds logic to check which kind of user you are and how to display the image. You go home thinking that this is a good design, all the logic it's kept in a single method.

The problem with this approach, quite often used by beginners or as a shortcut, is that different classes are encoded within the same single one. This design is not maintainable in the long run because the same class encodes behaviour for different objects. Your design is abstracting at the wrong level.

A better approach is to create a class for each kind of user and dispatch dynamically. This design is easier to maintain because the behaviour is not encoded solely on the method, but on the type. The Figure Yyy shows how to dynamically dispatch based on the type.

Exercise Write the code depicted on Figure Yyy.

Exercise Write the main class and show that the method performs a dynamic dispatch based on the classes.

Exercise Write the code relying on inheritance and another version relying on interfaces. What are the benefits and drawbacks of the design and implementation decisions?

(Note: this is distinct for Python because there's no interface etc. Think how would you explain it in python)

Python is a dynamic language with a weak type system. This means the language assigns types at runtime, but you as a developer won't get any type error at compilation time. In Python polymorphism is always given and you as developer are in charge of ending that the methods you call exist or you will get a runtime error. As we said, the are no interfaces that we are bound to and python supports what's called duck typing, which is an idea much closer to how things work in real life. Let me show this with an example: you can read articles, news, cereal boxes and anything that has text. In Java you are bound by the type of object you are but in Python you are bound to the behaviour you have. A method call on `newspaper. hasText()` and `cerealBox.hasText()` works in Python independent from their type; Java would make this work only when they belong to the sand type or interface. For this reason, Python is more expressive in this regard, although it sacrifices static typing guarantees and forces the developer to either handle exceptions or crash and burn at runtime if the object on which the program calls the method `hasText()` does not provide such behaviour.

Exercise write the code depicted on Figure Yyy in Python. Do not forget to handle exceptional cases.

Exercise Write the main class and show that the method performs a dynamic dispatch based on the classes.

5.7 Indirection

Domain objects may end up being coupled to other objects in the first iteration of your design. This principle states that you should create an intermediate object that mediates between these two, hence reducing coupling.

To the untrained eye, it may seem as if two objects that where coupled, after applying the principle, are still coupled but to other object. This reasoning is right, except that this indirection breaks the idea of once object having a direct relation to the other one. The layer in between breaks the coupling direction and creates a more flexible design, since the two initially coupled objects do not need to know anything about each other anymore.

An example of this pattern (Figure zzz) is the indirection introduced between a guides and the images that belong to the guides. The idea here is that the same guide is shared among friends and each one of them may potentially see a different cover image. Prior to this principle, your initial solution was to duplicate the guide for each friend and recalculate the cover image for each of them. This solution is redundant, consumes memory, and duplicates data, so an update on one description involves updating all data in all copies of the guide. A better solution is to add a new level of indirection between a guide and its cover image, 'CoverManager', that knows how to retrieve the best cover image for the calling object. Internally, the manager may have to call the AI algorithm from time to time to update the image, cache it if the same user keeps coming to the same guide and even persist this mapping of guide-cover image in the database if the cover image doesn't change that often.

(Figure zzz)

Exercise Following the principles of these chapter, how would you design data persistence of an object from the case study? That is, would you add CRUD (Create, Read, Update and Delete) methods to all domain objects? If so, how would you avoid the coupling introduced by this design?

Exercise How would you design the offline mode of your application? That is, how do you deal with low connectivity or no connectivity at all? Solution: you never assume that there is a connection and instead create an intermediate layer that handles the communication. If the server is unreachable, this layer handles how to proceed. An advanced design pattern that users this idea is the circuit breaker (explained in later chapters)

5.8 Pure Fabrication

The term pure fabrication means to make something up. Use this principle whenever you observe that your domain classes are getting too overloaded, i.e., the start to exhibit high coupling and low cohesion.

The principle adds a new indirection between two objects that would otherwise be directly connected (coupled). The indirection means adding a new object that mediates the communication between two other entities. This indirection object is not part of the domain and will be a made software concept. Examples of this principle are: object pools, database classes, and pretty much any object that sits in between two domain objects. In terms of design patterns, this principle is observe in the adapter pattern, shown in Figure xyz.

(Figure xyz)

Explain figure.

Exercise:Wwhere could this principle be applied in the case study? Why? (There are many valid examples) Solution: the AI algorithm that inject images to guides. If this was not there, every guide would have the same image for all guides, tying the guide to the images. With the algorithm, the same guide share with friends shows different images based on other friends and connections. **Exercise**: let's assume that you would like to add a notification system to the mobile app of the case study whenever a friend posts a new comment on one of her guides. What classes would you need to create? Solution: at the very least, you would need a `Notification` class that contains the text and image of the notification and some kind of notification manager that schedules and send notifications to the appropriate parties.

5.9 Protected Variation

This principle is easy to apply and, in practice, requires you to be good at forecasting future pivot points or changes of direction.

The core idea of this principle is to shield your code in places where you expect changes, let that be via interfaces or other means. For example, if we were to provide special guides that users can buy, we would need to integrate with a payment platform. If you are not sure which one is best, you'll end up picking one and calling their methods where necessary. However, what happens if you see that platform Z has low commissions? You would want to change. ReFactoring may not be so easy because both payment solutions have different libraries with different API. So, you need to change all those specific calls by whatever the new API and workflow mandates.

One way of solving this problem is via an interface and different classes that implement those methods (Figure XYZ). In this design, you are protecting yourself from future changes. Your concrete classes that implement the interface maintain the workflow and your only job is to create a new concrete class that implements your interface.

You should apply this pattern to instability points and, specially, when using third party libraries that you don't have previous experience with.

5.10 Pure functions

This principle is not part of the common and well known GRASP principles but, in my experience, you should consider it. Lambdas, closures and pure functions are ubiquitous nowadays. My advice is that you stop using closures that captures mutable state. The main reason is that, at one point or another, you'll benchmark your application and start using parallel capabilities of your computer: task, futures, actors or whatever is the next parallel abstraction. If you encapsulate mutable state in closures, you are building your own coffin for the not so far away future. Things like processing an recognitive computation cannot be used anymore if the object the closure captures is shared between two threads – unless you can guarantee data race freedom. Data race freedom code is hard to write, maintain and make fast, so using locks may get you out of troubles at the cost of losing horizontal scalability [2].

[2] explain

6 Design Patters: Creational

Covers creational design patterns TODO: Text is copied from https://sourcemaking.com/design_patterns, do not copy this!

6.1 Abstract Factory

Creates an instance of several families of classes

6.2 Builder

Separates object construction from its representation

6.3 Factory Method

Creates an instance of several derived classes

6.4 Object Pool

Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

6.5 Prototype

A fully initialized instance to be copied or cloned

6.6 Singleton

A class of which only a single instance can exist

7 Design Patters: Structural

Covers structural design patterns

7.1 Adapter

7.2 Bridge

7.3 Composite

7.4 Decorator

7.5 Facade

7.6 Flyweight

7.7 Proxy

8 Design Patters: Behavioral

Covers behavioral design patterns

8.1 Chain of responsibility

8.2 Command

8.3 Interpreter

8.4 Iterator

8.5 Mediator

8.6 Memento

8.7 Null object

8.8 Observer

8.9 State

8.10 Strategy

8.11 Template method

8.12 Visitor

9 Cheatsheet

Add tables and images that show the difference between GRASP and SOLID, and different design patterns.