Explain it like I'm 2: Object Orientated Programming

Sam Nolan

May 21, 2018

This paper contains an explanation of object orientated principles as well as some general tips for creating clean code. It requires that you have some knowledge of the Java programming language.

Writing good object orientated code is a massive topic covered by many books. Books I would recommend on writing good code is Clean Code by Robert C Martin and Code complete 2 by Steve McConnell. For something more advanced, try Design patterns (which is colloquially known as GOF)

So how does Object Orientated Programming work?

Classes

Almost all OOP languages use classes. Some would argue that if a language claims to be OOP and does not have classes (For instance, early JS and Lua) that they are not really OOP, and there is definitely a case for that. Classes are at the core of OOP.

A class is simply a description of an object. There are two main things in this description. The instance variables, which are simply variables that are related to an object. This is what the object **has**. The other thing in the description are the methods to the object. These are all the functions that are related to an object and is therefore what the object **does**.

After we have described the object with a class, we can then use the class to **instantiate** the object. Instantiating uses the new keyword in Java.

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

This is telling Java to create a variable of type `Dog` named `dog` which is created from the description of `Dog` with parameter `Bojo` (which I would assume is the name).

We can then reference things about the dog using dot notation. Such as:

```
dog.bark();
String name = dog.getName();
```

Object Orientated design

Object orientated design is similar but different to the design we encounter everyday.

The goal of Object Orientated design is maintainability. That is, how easy is it to add new features to our program if our code is implemented in a particular way. If it's not given too much thought, maintainability will decrease as you keep working on the project.

It's very common to use a building analogy for software engineering, in fact, we are often called *software engineers* and *software architects* for that reason. If the design of the house is not given too much thought beforehand, and error that have been made have not been fixed, it will become harder and harder to build on the house. Any change you make might make it tumble down to the ground.

Keep in mind that a building can look good and work fine (have a good external design) but still have a very fragile internal structure (have a bad internal design). Object Orientated design has nothing to do with the user experience and is instead to do with how easy it is to change.

This is very much like software as it gets built. At first, it is very easy to get started and build what you need. But if you don't consider the design of the internal structure and just keep building, it will become difficult to add to it without breaking it.

Unlike buildings, software design happens throughout the development process. Although it is always a good idea to do some design beforehand, it is also a good idea to improve the design of existing code. The process of improving the design of existing code is called **refactoring**.

Duplicate code

Duplicate code is big no-no in any software project. Why? Having duplicate code makes it harder to make changes to the code, as you may have to make it in 2, 3 or maybe even more places.

Duplicating code is one of the easiest ways to make your project not maintainable. It can also introduce subtle bugs where you remembered to update one part of the code but not the other part.

In general, if you feel like you need to copy and paste, you should think about your design and how you can prevent the duplication.

In the easiest case, duplication is often avoided by simple placing the code in a method and having the two places calling it.

Original

```
private Person[] people = new Person[COMPANY_SIZE];

public void addEmployee(String name, double wage){
  Person employee = new Person(name);
  employee.setWage(wage);
```

```
  for(int i = 0; i < people.length; i++){
    if(people[i] == null){
      people[i] = employee;
    }
  }
}

public void addContractor(String name, double rate, int hoursNeeded){
  Person contractor = new Person(name);
  contractor.setRate(rate);
  contractor.employFor(hoursNeeded);

  for(int i = 0; i < people.length; i++){
    if(people[i] == null){
      people[i] = contractor;
    }
  }
}
```

Refactored

```
private Person[] people = new Person[COMPANY_SIZE];

public void addEmployee(String name, double wage){
  Person employee = new Person(name);
  employee.setWage(wage);

  addPersonToCompany(employee);
}

public void addContractor(String name, double rate, int hoursNeeded){
  Person contractor = new Person(name);
  contractor.setRate(rate);
  contractor.employFor(hoursNeeded);

  addPersonToCompany(contractor);
}

private void addPersonToCompany(Person person){
  for(int i = 0; i < people.length; i++){
    if(people[i] == null){
      people[i] = person;
    }
  }
}
```

Encapsulation

In regards to OOP, to be a valuable member of society, a class should both have something and do something. It must make use of it's resources wisely and not be too concerned in the affairs of others.

Here is an example of a Vector class (a vector for this can be thought of as a point in 2D space). This vector class is not very responsible, and uses the VectorCalculator class to do most of its calculations.

```
class Vector{
  public double x;
  public double y;
}

class VectorCalculator{
  public double distance(Vector vector){
    return Math.sqrt(vector.x * vector.x + vector.y * vector.y);
  }
}
```

The vector class is not considered a valuable member of society because although it has information it doesn't do anything with it. It requires other people to do things for it.

This is analogous to buying an iPhone but when you open the box you realise that you just got shipped the components. The entire iPhone has been disassembled and all of it's internal characteristics exposed. You could technically use the iPhone by building it from scratch, but your not an iPhone builder! That was their job! What if you get it wrong? You might even end up with an iPhone that is slightly different from a normal iPhone and misses every second call, and you just wouldn't know!

Although this Vector class example is a small one, as you get larger programs, this becomes a large issue. This is a classic theme in software development. It might be quicker and easier to write now but if you don't write good code now it will be hard to add new features later.

Back to our iPhone analogy, to solve the issue we create a case for the iPhone. We dictate exactly how you are allowed to interact with the iPhone and what you can and can't do. This doesn't take power from the iPhone user, but it makes their lives exponentially easier.

In programming, we call this case creating **encapsulation** we put our classes is neat cases that dictate what you can do and how you do it. To put a case on our Vector class, we would create the following:

```
class Vector{
  private double x;
  private double y;

  public Vector(double x, double y){
```

```
    this.x = x;
    this.y = y;
  }

  public double distance(){
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
}
```

This vector class now has a case. We use the word `private` to not allow accessing the x and y instance variables from outside the class. This is called information hiding and is like shielding the circuitry of a phone from view, and then we dictate the public methods that operate on the instance variables, which are like the buttons and screens of our phone.

This class is easier to understand than the former with their elements exposed for the same reason that the phone is easier to understand when it's got a case around it. The principles we use in designing products can often be applied to designing classes, because they are in a sense internal products to other classes.

It is always a good practice to make all of the instance variables private. Why? If we go back to our phone analogy, if Apple after shipping their phones in pieces decided that they would like to upgrade the CPU of their disassembled phones, then it is likely that they would now upset customers who relied on the particular architecture or size of the CPU. By exposing all the internal components, you are encouraging people to be dependant on parts of the system that really should be considered none of the user's business.

By hiding the CPU and circuitry of the phone, Apple is able to change those features to remain competitive in the market without upsetting customers.

In a class sense, if we go back to our Vector class, by making the instance variables public we are encouraging people to become dependant on the fact that we are representing a Vector using an x and a y coordinate. However, we might have found that many people were using the distance operation often (which calculates the distance from the point (0, 0)), so thought that they should change the internal representation to polar form. Which would increase performance for this particular case.

Using x and y is called Cartesian form, meaning that a point is represented by 'walk this far in the x direction, then walk this far in the y direction and you should be at the point I'm talking about' Polar form instead uses an angle and a distance. It says 'Face towards this particular direction and walk r meters'. r is the distance we are calculating so in polar form we no longer need to calculate it. Here is the vector class using polar form

```
class Vector{
  private double r;
  private double theta;
```

```
  public Vector(double x, double y){
    // Calculates the direction to (x, y)
    theta = Math.atan2(y, x);

    // Calculates the distance to (x, y)
    r = Math.sqrt(x * x + y * y);
  }

  public double distance(){
    return r;
  }
}
```

Now, any user of Vector wouldn't even notice the difference, except all of a sudden the distance operation would have got much faster.

This wouldn't have been possible if we made the x and y public. As we would have had to change every point for which the class is used, which might be scattered throughout the entire program!

You should always make all the instance variables of a class private. Exceptions to this rule are so extraordinarily few that I can't name any.

Also, you don't want to have too many public methods on your class. This is analogous to a phone with too many buttons and sensors. It simply becomes hard to use!

Interfaces

One thing that's very important in encapsulation is that the users of the class should become dependant on the interface of the class, and not it's implementation. That is, apple users should become dependant on the graphical interface and not the CPU architecture, and our Vector class users should be dependant on the things it does rather than how it's represented.

It's often useful to see what a class looks like from the outside. From the outside, our Vector class looks like this:

```
  public double distance();
```

Other than the constructor, This is what we can see, the externals of the class.

We can call this the class interface. If we would like to, we can make this interface explicit by creating an interface in java.

```
interface Vector{
  public double distance();
}
```

And then we can say that our polar form and Cartesian vectors can **implement** this interface.

```
class PolarVector implements Vector {
    ...
}

class CartesianVector implements Vector {
    ...
}
```

This way we completely separate the interface from it's implementation. We could keep creating new classes and as long as it implements the interface we can interact with it.

What's great about using interfaces is that it becomes extraordinarily easy to swap out different implementations. Say we define our Cartesian vectors like so:

```
Vector vector = new CartesianVector(x, y);
```

If every time we reference a vector we use the Vector interface as a type, and you want to change the implementation of the vector, all you need to do is this:

```
Vector vector = new PolarVector(x, y);
```

Only the point where it's instantiated needs to change! Not even the type, the type can just remain as Vector.

Interfaces add flexibility into your program, they make it easy to swap out one implementation for another. So if you perceive that it might be required to swap out different implementations you should use an interface.

Abstraction

Now that we know that it's better to be dependant on an interface rather than an implementation, what makes a good interface? That's the question of abstraction. An abstraction is a metaphor to how the system actually works in order to make our understanding easier.

One example of an abstraction is a file system, data is not actually stored in folders and files, on a computer it's either stored on a metal platter or a computer chip. The file system is an abstraction of what's actually happening, and allows the user to interface with the system without needing to understand exactly how the system works.

Computer systems are quite famous for having layers and layers of abstraction. When you create an account on a website, you are either:

1. Creating an account
2. Pressing a button on the web page
3. Sending a POST request to a web server

4. Creating a record in a database
5. Saving your details to a file
6. Saving your details to a hard disk

Every time we add a layer of abstraction, the process becomes easier to use but more specialised.

This is very similar to how human language works. We start off by understanding very simple terms like 'car' and 'cat', and then we use those concepts to provide larger concepts such as 'traffic' and 'species'. Our understanding keeps building on what we already know.

The aim of this language job is to as succinctly as possible express what you are trying to express with as few words as possible. In programming, it's very similar, the high level abstractions talk in the language of the problem, and the low level abstractions talk in the language of the implementation.

When it comes to building classes, we have classes that have a high level of abstraction such as:

```
UserAccount account = new UserAccount("username", "password", "display name");
```

Or we could have something that is at a low level of abstraction such as

```
File file = new File("filename");
```

The things that are at the high level of abstraction are simply a collection of the things at the lower level, but they are much easier to use.

Our vector class is an abstraction for 2 floats.

When creating a class, you should consider what level of abstraction your class is, and sticking to that level. For instance, a class with the following interface:

```
class AccountList{
  public void addAccount(Account account);
  public Account getAccountById(String id);
  ...
  // More account related features
  ...
  public void writeToFile(String filename, byte[] contents);
}
```

Violates the abstraction because of the `writeToFile` method. This method is on a much lower level of abstraction than the rest of them, and as of such should not be included in this class.

Abstract classes

An abstract class is a mixture between a class and an interface. An abstract class has some methods that are implemented but then also provides an interface to implement other methods.

As there are some methods that are left unimplemented, it cannot be instantiated unless a subclass of an abstract class gives an implementation for them.

Here is an example of a shape abstract class with subclasses

```
abstract class Shape{
  private String name;
  public Shape(String name){
    this.name = name;
  }

  public String getName(){
    return name;
  }

  public abstract double getArea();
}

class Square extends Shape{
  private double sideLength;

  public Square(double sideLength){
    super("Square");
    this.sideLength = sideLength;
  }

  public double getArea(){
    return sideLength * sideLength
  }
}

class Circle extends Shape{
  private double radius;

  public Circle(double radius){
    super("Circle");
    this.radius = radius;
  }

  public double getArea(){
    return Math.pi * radius * radius;
```

```
  }
}
```

The Shape class here is abstract, as it has the feature of remembering it's name, which is a common implementation among all shapes, but also says that it's children need to define a `getArea()` method.

the `getArea()` method works like an interface, and the `getName()` works like a normal class.

An abstract class cannot be instantiated, you cannot call `new Shape("Squircle")` as it will come up with a compiler error. This is because it does not yet know how to implement the `getArea` method without subclassing.

Abstract classes are useful for mixing classes with interfaces, as well as having a superclass call a method of a subclass, for instance, you could add to your Shape class the following method:

```
public String getDetails(){
  return "Area: " + String.valueOf(getArea());
}
```

This would get the value of the area from whatever implementation the shape and return it in a standard format.

It only makes sense to make a class abstract if it has subclasses. For our Shape example, the question to ask is 'Is it useful to have generic Shapes in our program?' If not, make the class abstract, if so, keep it concrete (opposite of abstract)

Single responsibility principle

There is so much in object orientated design that we could touch on, but I'll finish this paper with a very important concept popularised by Robert C Martin in his book Clean Code.

This important principle is that every method or class should do one thing, and exactly one thing.

If you could argue that the function you are making does two things or has two steps. You should split one step them into private function and call them from the first function.

This ensures that every method is small and easy to understand. It makes it an absolute breeze to find errors in a method that's only 10 lines long.

Furthermore, each class should do exactly one thing. Or in Robert's words, 'A class should have only one reason to change'. If you could easily split the class into different things that it does or represents, you should consider creating two or more classes for the different features.

In practical terms, it's a good idea to keep methods below 30 lines, there's no practical limit for classes, but in well written frameworks the average class size is around 80 lines.

If you want to see more examples on how to write clean code, I would recommend the books mentioned in the introduction.

I hope that this was useful! If you have any feedback, please let me know at s3723315@student.rmit.edu.au or check out the GitHub repository for these papers at github.com/Hazelfire/EILI2. If you feel like there's a topic in computer science that needs a good explanation, just let me know!