Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 06: Objects And Data Structures

Data abstraction

We don't want to know the implementation of a property. We just want to use and manipulate them in other classes. Defining **private** fields and accessing them through **public getter** and **setter** violates this rule and it is like there is no data abstraction. Because the data is shown to all classes implicitly. In abstraction we have no idea about the data type.

Data abstraction

```
public interface Vehicle{
   double getFuelTankCapacityInGallons();
   double getGallonsOfGasoline();
}
```

```
public interface Vehicle{
   double getPercentFuelRemaining();
}
```

Object/Data Anti-Symmetry

Domain Classes: Objects that hide their data under abstraction and expose methods to apply operations over them. The business logics are implemented inside these classes and they determine the behavior of the software. Unit and acceptance tests should be written for these classes.

Anemic Classes: Data structures that expose their data and have no methods. No logic is implemented inside these classes and they are only used to transfer data. That is why we call them Data Transfer Objects(DTO).

Object/Data Anti-Symmetry

```
public class Square {
   public Point topLeft;
   public double side;
}

public class Rectangle {
   public Point topLeft;
   public double height;
   public double width;
}

public class Circle {
   public Point center;
   public double radius;
}
```

```
public class Geometry {
   public final double PI = 3.141592653589793;

   public double area(Object shape) throws NoSuchShapeException {
      if (shape instanceof Square) {
         Square s = (Square) shape;
         return s.side * s.side;
      } else if (shape instanceof Rectangle) {
         Rectangle r = (Rectangle) shape;
         return r.height * r.width;
      } else if (shape instanceof Circle) {
         Circle c = (Circle) shape;
         return PI * c.radius * c.radius;
      }

      throw new NoSuchShapeException();
   }
}
```

Object/Data Anti-Symmetry

In the previous code which is written in procedural form, there are three anemic classes or data structures and one application class which uses these classes and implements the business logic of calculating the area for the shapes.

Adding a new business logic(a method called **perimeter()**) affects only the application class and the anemic classes won't be affected.

Adding a new data structure(a new shape class) affects all methods in application class, but none of the anemic classes are needed to be changed.

Now let's write this code in object-oriented form.

Object/Data Anti-Symmetry

```
public interface Shape {
   public double area();
}

public class Square implements Shape {
   public Point topLeft;
   public double side;

   @Override
   public double area() {
      return side * side;
   }
}

public class Triangle implements Shape {
   public double height;
   public double base;

   @Override
   public double area() {
      return height * base / 2;
   }
}
```

```
public class Rectangle implements Shape {
   public Point topLeft;
   public double height;
   public double width;

   @Override
   public double area() {
      return height * width;
   }
}

public class Circle implements Shape {
   public final double PI = 3.141592653589793;
   public Point center;
   public double radius;

   @Override
   public double area() {
      return PI * radius * radius;
   }
}
```

```
public class Main {
 private double shapeArea;
 Shape shape;

   public static void main(){
      if (shape instanceof Square) {
          shape = new Square();
      } else if (shape instanceof Rectangle) {
          shape = new Rectangle();
      } else if (shape instanceof Circle) {
          shape = new Circle();
      } else if (shape instanceof Triangle) {
          shape = new Triangle();
      }

      shapeArea = shape.area();
   }
}
```

Object/Data Anti-Symmetry

This code is an Object-Oriented code. All of the classes implement an interface that defines business rules for us. If we decide to add a new business logic to the software, all we have to do is to add a new abstracted method to **Shape** interface. That will force us implement this method in all classes that extend this interface.

Adding a new business logic affects all of the shape classes.

Adding a new shape affects the main class that uses this objects. (Application Class)

Procedural Form vs. Object-Oriented Form

Which one should I use to write my code?

Procedural code makes adding new data structures difficult. Because all methods have to be changed. OO code makes adding new methods difficult. Because all the classes have be changed.

So if we have a system with a high possibility of adding new behaviors and logics in the future, we should use Procedural Programming.

But if we have a system with a high possibility of adding new classes in the future, we should use Object-Oriented Programming.

The Law Of Demeter

- A unit should have only limited knowledge about other units.
- Only units closely related to the current unit should know about its details.
- Each unit should only talk to its friends.
- Don't talk to strangers.
- Only talk to your immediate friends.

The Law Of Demeter

In other words the law of Demeter says, method **m()** from class **C** should only be able to call the methods of these items:

- Class **C**
- An object created by method **m()**
- An object passed to method **m()** as argument
- An object stored inside an internal variable of class **C**

This means we can't use sequential **getter** methods.

The Law Of Demeter

```
public class Order{
   private Flight flight;

   public Flight getFlight() {
      return flight;
   }
}
```

```
public class Flight{
   private Airport originAirport;
   private Airport destinationAirport;

   public Airport getOriginAirport() {
      return originAirport;
   }

   public Airport getDestinationAirport() {
      return destinationAirport;
   }
}
```

```
public class Airport{
   private AirportCode airportCode;
   private Coordinate coordinate;
   private Location location;
   private String name;

   public AirportCode getCode() {
      return airportCode;
   }

   public Coordinate getCoordinate() {
      return coordinate;
   }

   public Location getLocation() {
      return location;
   }

   public String getName() {
      return name;
   }
}
```

```
public class Application{
   private static void main(String ... args){
      Order order = new Order();
      String originAirportName = order.getFlight().getOriginAirport().getName();
   }
}
```

The Law Of Demeter

```
public class Order{
   private Flight flight;

   public Flight getFlight() {
      return flight;
   }

   public String getOriginAirportName(){
      return flight.getOriginAirportName();
   }
}
```

```
public class Flight{
   private Airport originAirport;
   private Airport destinationAirport;

   public Airport getOriginAirport() {
      return originAirport;
   }

   public Airport getDestinationAirport() {
      return destinationAirport;
   }

   public String getOriginAirportName(){
      return originAirport.getName();
   }
}
```

```
public class Airport{
   private AirportCode code;
   private Coordinate coordinate;
   private Location location;
   private String name;

   public AirportCode getCode() {
      return code;
   }

   public Coordinate getCoordinate() {
      return coordinate;
   }

   public Location getLocation() {
      return location;
   }

   public String getName() {
      return name;
   }
}
```

```
public class Application{
   private static void main(String ... args){
      Order order = new Order();
      String originAirportName = order.getOriginAirportName();
   }
}
```

Data Transfer Object - DTO

Data structures is defined as class with private variables with no methods. These structures are useful for transferring data between different layers of the system. For example we get the data from the database and store it inside a DTO and pass it to other layers like domain or application layers. Let's say we need an object to simulate a flight. We create two classes. **Flight** and **FlightDTO**. The first one has the logic methods and the second one is only used to transfer data. We do all our best to isolate business classes from data classes.

In brief we could say:

By data structure we mean anemic classes with only getters and setters that store and transfer data which are also known as DTOs. In the other hand we have domain class that implement the business logics and have no setter methods. We can only read their data but not to change them.

Note that these words are talking about the same concept:

- Data Structure = Anemic Class = DTO = Data Class
- Object = Domain Class = Business Class