Mohammadmahdi Farrokhy

Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 06: Objects And Data Structures

Data abstraction

Our focus is on using and manipulating properties, not understanding their implementation. However, defining **private** fields and accessing them through **public getter** and **setter** methods contradicts this principle. It exposes data to all classes, violating data abstraction. True abstraction hides data types, rendering them unknown.

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

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

Object/Data Anti-Symmetry

Domain Classes: Conceal data through abstraction, offering methods to apply operations. Business logics are embedded within, shaping software behavior. Writing unit and acceptance tests for these classes is crucial.

Anemic Classes: Data structures that unveil data with no methods. They're used solely for data transfer, lacking logic. Hence, they're referred to as **Data Transfer Objects (DTOs).**

Procedural Code

```
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(ShapeType shapeType) throws NoSuchShapeException {
        if (isSquare(shapeType)) {
            Square s = (Square) shape;
            return s.side * s.side;
        } else if (isRectangle(shapeType)) {
            Rectangle r = (Rectangle) shape;
            return r.height * r.width;
        } else if (isCircle(shapeType)) {
            Circle c = (Circle) shape;
            return PI * c.radius * c.radius;
        }

        throw new NoSuchShapeException();
    }
}
```

Object/Data Anti-Symmetry

In prior procedural code, there exist three anemic classes or data structures and an **application** class utilizing them to calculate shape areas. Adding a new business logic (e.g., **perimeter()**) affects the application class only, leaving anemic classes untouched. Introducing a new data structure (like shape class **Triangle**) impacts all application class methods, yet anemic classes remain unaffected.

Now, let's rewrite this in an object-oriented manner.

Object-Oriented Code

```
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.isSquare()) {
          shape = new Square();
      } else if (shape.isRectangle()) {
          shape = new Rectangle();
      } else if (shape.isCircle()) {
          shape = new Circle();
      } else if (shape.isTriangle()) {
          shape = new Triangle();
      }

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

Object/Data Anti-Symmetry

This is an Object-Oriented code. Each class implements an interface that determines the business rules. Adding a new business logic involves adding an abstracted method to the Shape interface. So all classes extending this interface must implement the new method.

- Adding new business logic affects all shape classes.
- Adding a new shape influences the main class **(Application Class)** employing these objects.

Procedural Form vs. Object-Oriented Form

Which approach to use for coding?

- Procedural code:
  - Adding new data structures requires changes to all methods of application class.
  - Suitable for systems likely to see more future behaviors and business logics.

- Object-Oriented Code:
  - Adding new methods(business rules) demands modifications to all classes child classes.
  - Suitable for systems likely to introduce new data structures.

The Law Of Demeter

This law is stated in different was:

- A unit should have limited knowledge about others.
- Only units closely tied to the current unit should be familiar with 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

Put simply, the Law of Demeter states that method m() in class C should only be able to call methods from these entities:

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

This implies that sequential **getter** methods cannot be used.

The Law Of Demeter

```java
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;
    }
}
```

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

    public Airport getOriginAirport() {
        return originAirport;
    }

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

```java
public class Order{
    private Flight flight;

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

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

The Law Of Demeter

```java
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;
   }
}
```

```java
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();
   }
}
```

```java
public class Order{
   private Flight flight;

   public Flight getFlight() {
      return flight;
   }

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

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

Data Transfer Object - DTO

Data structures are defined as classes with private variables and no methods. These structures facilitate the transfer of data between different system layers. For instance, we extract data from a database, place it into a DTO, and transmit it to other layers such as domain or application layers. Consider simulating a flight with two classes: **Flight** and **FlightDTO**. The former contains logic methods, while the latter only moves data. Our goal is to distinctly separate business classes from data classes.

In brief we could say:

When we refer to data structures, we mean anemic classes equipped with only getters and setters. These classes store and move data, and are also recognized as DTOs. On the flip side, domain classes execute business logic and lack setter methods. We can only access their data without altering it.

Note that these terms signify the same concept:

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