Mohammadmahdi Farrokhy

Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 05: Objects And Data Structures

Data abstraction

True abstraction hides data types and applies operations on them as public methods.

```
// BAD CODE: Reveals details of private fields
public interface Vehicle {
   double getFuelTankCapacityInGallons();
   double getGallonsOfGasoline();
}
```

```
// GOOD CODE: Uses private fields for a public behavior
public interface Vehicle {
   double getPercentFuelRemaining();
}
```

Object/Data Anti-Symmetry

Anemic Classes: *Data structures* that contain data, but no methods. They're used only to move data between classes and layers. That's why we call them *Data Transfer Object(DTO)*.

Domain Classes: Hide data through abstraction, offering methods that implement *Business logics* and define software behavior.

Procedural Code

In the code below, there are three anemic classes and a client class that uses them. Adding a new business logic or new data structures affects only the client.

```
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(Shape 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-Oriented Code

*Here is an Object-Oriented code with **interface**s that define **business rule**s. Adding a new business logic means defining a new abstract method in the interface and implementing it in concrete classes. So, adding new business logic affects all of them. Adding a new concrete class affects the client class.*

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 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;
   }
}
```

Object-Oriented Code

```
public class ShapeService {
   Shape shape;

   public ShapeService(Shape shape){
      this.shape = shape;
   }

   public static double calculateArea() {
      return shape.area();
   }
}
```

```
public enum ShapeType {
   Rectangle,
   Circle,
   Square
}
```

```
public class ShapeServiceFactory {
   private ShapeServiceFactory() {
   }

   public static ShapeService createShapeService(ShapeType shapeType) {
      Shape shape;

      switch (shapeType) {
         case ShapeType.Square:
            shape = new Square();
            break;

         case ShapeType.Rectangle:
            shape = new Rectangle();
            break;

         case ShapeType.Circle:
            shape = new Circle();
            break;
      }

      return new ShapeService(shape);
   }
}
```

Object-Oriented Code

```
public class ClientClass {
    private ShapeType shapeType;
    private ShapeService shapeService;

    public ClientClass(ShapeType shapeType) {
        this.shapeType = shapeType;
        this.shapeService = ShapeServiceFactory.createShapeService(shapeType);
    }

    public double calculateArea() {
        return shapeService.calculateArea();
    }
}
```

Procedural Form vs. Object-Oriented Form

Which approach to use for coding?

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

- Object-Oriented Code:
  - Adding new methods(business rules) demands modification of all concrete 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 next to the current unit should be familiar with its details.
- Each unit should only talk to its neighbors.
- Don't talk to strangers.

The Law Of Demeter

Simply put, ***Law of Demeter*** states that method *m()* in class *C* should only be able to call methods from:

- Class *C*
- An object from return type of method *m()*
- Parameters of method *m()*
- A field object of class *C*

This implies that sequential ***getter*** methods violate this principle.

The Law Of Demeter – BAD CODE

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

   public Airport getOriginAirport() {
      return originAirport;
   }

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

```
public class Order {
   private Flight flight;

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

```
public class Client {
   private void operation() {
      Order order = new Order();
      String originAirportName = order.getFlight().getOriginAirport().getName();
   }
}
```

The Law Of Demeter – GOOD CODE

```
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 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 Order {
   private Flight flight;

   public Flight getFlight() {
      return flight;
   }

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

```
public class Client {
   private void operation() {
      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. Consider simulating a flight with two classes: Flight, contains logic methods and FlightDTO that 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 with only getters and setters. These classes store and move data, and are also recognized as DTOs. On the other side, domain classes execute business logic and lack setter methods. We can only access their data without changing it.