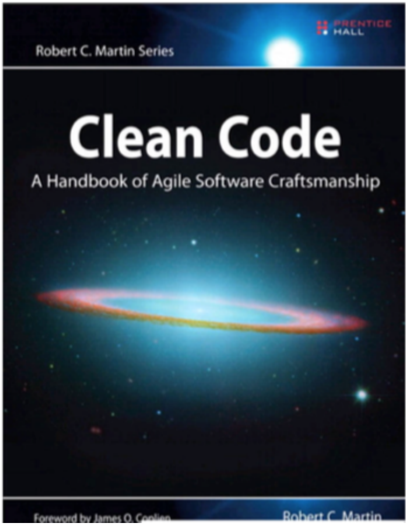
Clean Classes



PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Data Abstraction with Classes

Consider a concrete `Point` data type:

```java
public class Point {
  public double x, y;
}
```

and an abstract `Point` data type:

```java
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

- The concrete `Point` exposes its implementation, the abstract `Point` hides it.
- Abstract `Point` expresses that it take two elements to define a point, concrete `Point` allows `x` and `y` to be set independently.

Data Abstraction with Classes

```java
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

- The abstract `Point` class is truly an absraction - its interface expresses the essence of pointness and hides its implementation.

Data abstraction isn't just making instance variables private and providing getters and setters.

Classes as Data Structures

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

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

```java
class Geometry {
  public double area(Object shape) throws NoSuchShapeException {
    if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle) shape; return r.height * r.width;
    } else if (shape instanceof Circle) {
      Circle c = (Circle)shape; return Math.PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

- Adding a shape requires adding a new shape class and then touching every function in `Geometry`.
- Adding a function only requires adding it to `Geometry` and coding it to work with each shape.

Object-Oriented Classes

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

```java
class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;
  public double area() {
    return height * width;
  }
}
```

-

```java
class Circle implements Shape {
  private Point center;
  private double radius;
  public double area() { return
       Math.PI * radius * radius; }
}
```

```java
class Square implements Shape {
  private Point topLeft;
  private double side;
  public double area() { return
       side*side; }
}
```

- Adding a class requires only creating a class that implements each of the functions in `Shape`.
- Adding a function requires adding its declaration to `Shape`, and then adding a defintion to every class that implements `Shape`

Data/Object Anti-Symmetry

The observations above lead to two complementary general rules:

> *Using objects as data structures makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.*

and

> *Using objects as data structures makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.*

Clean design requires knowing when to apply each style (will you be more likely to add new functions or new classes?). Don't drive every nail with the same hammer.

The Law of Demeter

A module should not know about the internal structure of an object it uses. Consider:

```java
final String outDir =
    ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Code like this is a “train wreck” because it looks like a train of method calls on objects returned from a succession of methods.

Is this better?

```java
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outDir = scratchDir.getAbsolutePath();
```

Maybe, but probably not.

- Internal structure is still exposed and relied upon.
- A protocol-ish interface is a design smell - the client of the `ctxt` object is trying to do something - give that something a name and represent it as a method.

Hiding Internal Structure

What is it that the client is doing with an absolute path?

```
String outFile = outDir + "/" + className.replace''(., ''/) + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

- First, this code smells: multiple levels of abstraction are mixed together.
- But ultimately the client code is using the absolute path of the scratch directory to create a file in that directory.

Better OO design to let the `ctxt` object do this for us:

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

- Now the internal structure of the `ctxt` object is no longer exposed and is free to change without affecting client code.
- Client code is much cleaner: several messy lines replaced with one method call whose intent is clear.

Data Transfer Objects and Active Data Objects

Data Transfer Objects (DTOs) are simple data structures useful for passing data between clients and servers, into and out of databases.

```java
public class Person {
  private String name, email;
  public Person(String name, String email) {
    this.name = name; this.email = email;
  }
  public String getName() { return name; }
  public String getEmail() { return email; }
}
```

- Other than meeting the JavaBean spec, no need for private instance variables and getters. (This is one of Java's warts.)
- Sometimes a DTO will include methods like save and find that operate on the database in which the DTOs are stored. These are called active data objects (ADOs).
- Don't put business logic in an ADO. Create a separate class to hold business logic and let the ADO have a single responsibility: transferring data to and from a database.

Class Organization

A class should folow the standard Java organization:

- public static constants
- private static variables
- private instance variables
- public functions
- private helper functions right after the functions they serve (stepdown rule/newspaper metaphor)

Should nearly never have public instance variables, but they'd go right after the private instance variables.

Only valid reason to break encapsulation is to facilitate unit testing. Do this by giving protected or package access – the unit test should be in the same package as the class it tests.

Closing Thoughts

Use classes for data abstraction.

- Expose a public interface via methods.
- Encapsulate the implementation.
- Don't require or rely on knowledge of internal implementation (Law of Demeter).

Think about whether your classes represent data structures or objects that define coherent sets of behavior (OOP).

- Using objects as data structures makes it easy to add new functions, hard to add new data structures (classes).
- Using OO objects makes it easy to add new classes, hard to add new functions.