Design Patterns

aka Object Oriented Programming

June 28, 2017

Outline

Fundamentals

Principles

A class should have only one reason to change.
Classes should be open to extension and closed for modification.
Depend on abstractions not on concrete classes.
Don’t call us, we’ll call you.
Encapsulate what varies.
Favour composition over inheritance.
Only talk to your friends.
Program to interfaces not to implementations.
Strive for loosely coupled designs among objects that interact.

Section 1

Fundamentals

Abstraction

Delineates a simplified, context-specific representation of a thing.

- Ignores contextually irrelevant details.
- Includes contextually relevant details.

Encapsulation

Restricts outside access to a things parts.
Bundles a things state with the routines that use that state.

Inheritance

Grants one thing the capabilities of another thing.
Is not the same as though often agrees with subtyping.
Includes prototypal and class-based inheritance.

Polymorphism

Pluralizes the numbers of types on which a routine can operate.

- Ad hoc / static polymorphism (aka method overloading)
- Parameteric polymorphism (aka generics)
- Subtype polymorphism

Section 2

Principles

Subsection 1

A class should have only one reason to change.

What does it mean?

A reason to change is a responsibility or an axis-of-change.
Change refers to changes in source code not to variables at runtime.
Example responsibilities: print an invoice, calculate tax.

Why does it matter?

- Clients can consume individual responsibilities.
- Responsibilities can change without breaking each other.
- Responsibilities can be recompiled independently.

The art is to balance rigidly and needless complexity.

```
interface Rectangle {
    calculateArea(): number;
    draw();
}

function computationalApp(rectangle: Rectangle) {
    rectangle.calculateArea();
}

function graphicalApp(rectangle: Rectangle) {
    rectangle.draw();
}
```

```
interface GeometricRectangle {
    calculateArea(): number;
}

interface Rectangle {
    draw();
}

function computationalApp(rectangle: GeometricRectangle) {
    rectangle.calculateArea();
}

function graphicalApp(rectangle: Rectangle) {
    rectangle.draw();
}
```

Subsection 2

Classes should be open to extension and closed for modification.

What does it mean?

Change the way a module behaves,
without changing its source code.
How? Use abstraction.
E.g. A draw() method ought to be closed against

- new shapes (via a Shape abstraction),
- new ways to order shapes (via an ordering abstraction), though
- cannot be closed against all changes.

Why does it matter?

Protect existing code against breaking changes.

```
enum ShapeType {circle, square};

class Shape {
  itsType: ShapeType;
}

class Circle extends Shape {
  itsType: ShapeType;
  itsRadius: number;
}

class Square extends Shape {
  itsType: ShapeType;
  itsSide: number;
}

function drawSquare(square: Square) { }
function drawCircle(circle: Circle) { }

function drawAllShapes(shapes: Shape[]) {
    shapes.forEach((s) => {
        switch (s.itsType) {
            case ShapeType.circle:
              DrawCircle(s as Circle);
```

```
interface Shape {
  draw();
}

class Circle implements Shape {
  public draw() { }
}

class Square implements Shape {
  public draw() { }
}

function drawAllShapes(shapes: Shape[]) {
  shapes.forEach((s) => s.draw());
}
```

Subsection 3

Depend on abstractions not on concrete classes.

What does it mean?

Also known as the Dependency Inversion Principle.
It inverts the traditional, procedural dependency hierarchy.

Why does it matter?

Without inversion, changing the utility layer may effect the policy layer.

```
// this programs to an interfaces
// but it depends on concrete classes
function doSomething(packageType: string) {
    let package: IPackage;
    if (packageType === "roofing") {
        package = new RoofingPackage(); // concrete lower layer
    }
    if (packageType === "flooring") {
        package = new FlooringPackage(); // concrete lower layer
    }
    package.tender();
    package.ship();
}
```

```
// this programs to an interfaces
// and it depends on abstractions
function doSomething(packageType: string, packageFactory: IPackage
    let package: IPackage;
    if (packageType === "roofing") {
        package = packageFactory.createRoofingPackage();
    }
    if (packageType === "flooring") {
        package = packageFactory.createFlooringPackage();
    }
    package.tender();
    package.ship();
}
```

Subsection 4

Don’t call us, we’ll call you.

What does it mean?

Inversion of Control:

- Application code defines a routine (operation/function),
- plugs that routine into the framework, and
- the framework controls when to execute it.

Whereas libraries tend to provide normal control,
in which we call the library code,
frameworks tend to provide inversion of control (IoC),
in which the framework calls us.

Inversion of Control is a key part of what makes a framework different to a library. Martin Fowler (26 June 2005)

Why does it matter?

Binding to events:

```
<p id="foo">Foo</p>

var foo =
 document.getElementById('foo');

foo.addEventListener(
  'click',
  () => alert('They called us.'));

foo.addEventListener(
  'click',
  () => alert('They called us too.'));
```

Implementing interfaces:

```
// OnInit is an Angular lifecycle hook interface
export class PeekABoo implements OnInit {
  ngOnInit() {
    alert('They called us.");

    // we could now call a library
    d3.select("body")
      .style("color", "black")
      .style("background-color", "white");
  }
}
```

Subsection 5

Encapsulate what varies.

What does it mean?

”What varies” means source code that might change.
”Encapsulate” means restrict outside access to a things parts.
Therefore: Restrict outside access to source code that might change.
This is a principle of procedural, functional, and OO programming.

What causes change?

- Changing requirements causes change.
- Refactoring causes change.
- Performance improvements cause change.
- Changing business needs cause change.
- Improved understanding of the problem causes change.
- Change in government policies causes change.

Why does it matter?

Why? We can change code later in as few places as possible.

```
if (pet.type() == dog) {
    pet.bark();
} else if (pet.type() == cat) {
    pet.meow();
} else if (pet.type() == duck) {
    pet.quack();
}

// The number of animals that our app supports might change.

pet.speak();
```

```
class Course
{
    public LinkedList<Student> Students;
}

// The way we represent a collection of students might change.

class Course
{
    private LinkedList<Student> _students;
    public bool Register(Student s)
    {
        _students.Add(s);
    }
}
```

```
// SomeFile01.cs
this.Tax += this.Price * 0.07;

// SomeFile02.cs
product.Tax += product.Price * 0.05;

// SomeFile03.cs
If (product.Type === "accommodation") {
    Product.Tax -= product.Price * 0.07;
    product.Tax += product.Price * 0.08;
}

// The way we calculate tax in British Columbia might change.

product.Tax =
    _taxService.calculateTax(product);?
```

Subsection 6

Favour composition over inheritance.

What does it mean?

Why does it matter?

Subsection 7

Only talk to your friends.

What does it mean?

Why does it matter?

Subsection 8

Program to interfaces not to implementations.

What does it mean?

Why does it matter?

Subsection 9

Strive for loosely coupled designs among objects that interact.

What does it mean?

Why does it matter?