Design Principles

aka Object Oriented Programming

Shaun Luttin

October 1, 2017

Goal

- Become familiar with object-oriented design principles.
- Have a starting point for further research.

Why?

- Modularity
- Allow change of X without changing Y.
- Allow reuse of X without changing Y.

The Principles

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

Encapsulate what varies.

- Encapsulate ...
  - Restrict outside access to a thing's parts.
  - Bundle operations with the data they use.
- ... what varies.
  - This refers to changes to source code.
  - Source code changes due to changing requirements.
  - E.g. A change in government may cause a change in tax law.
- Restrict outside access to parts of the source code that might change due to changing requirements.
- "what [do] you want to be *able* to change without redesign?" (Gamma et al, 1995)

Encapsulate what varies . . .

```
// We have encapsulated the calculation of tax.
class TaxCalculator {

    public calculateTax(product: Product): number {
        // This does then calculation of tax
        return 0;
    }
}

class FarmStand {

    private cart: Array<Product>;

    public CalculateTotalTax(): number {

        const taxCalculator = new TaxCalculator();
        let totalTax = 0;

        for (const product of this.cart) {
            totalTax += taxCalculator.calculateTax(product);
        }

        return totalTax;
    }
}
```

Program to interfaces not to implementations.

- An interface says what requests it will receive.
- An implementation says how it will handle those requests.
- Programming to interfaces adds polymorphism:
  - it lets us change an implementation even at runtime
  - it lets us send the same request to different classes
- A separate, related SOLID principle:
  - Interface Segregation Principle (Martin, 1996)
  - Define lean interfaces that are specific to the client's needs.
  - "Clients should not be forced to depend upon interfaces that they do not use." (Martin, 1996)

Program to interfaces . . .

```
function makeOrangeCarrotJuice(): Array<string> {

    const medley = new Array<string>();

    const orange: Juiceable = new Orange();
    const carrot: Juiceable = new Carrot();

    // The following is programming to Juiceable interfaces
    for (const juicable of [orange, carrot]) {
        const juice = juicable.squeeze();
        medley.push(juice);
    }

    return medley;
}

class Orange implements Juiceable {
    public squeeze = () => "orange juice";
    public peel = () => { /* peel the orange */ }
}

class Carrot implements Juiceable {
    public squeeze = () => "carrot juice";
    public chop = () => { /* chop the carrot */ }
}
```

Depend on abstractions not on concrete classes.

- Aside:
  - Dependency Injection is a mix of two principles:
    - Dependency Inversion
    - Inversion of Control (IoC)
  - IoC containers are a type of Dependency Injection
- See also: https://martinfowler.com/articles/injection.html

Depend on abstractions not on concrete classes.

- "Depend" means make a direct reference.
- "Abstractions" define the interface/type.
- "Concrete classes" define the implementation.
- SOLID: Dependency Inversion Principle (Martin, 1996)
  - Traditionally, high-level modules depend on low-level modules:
  - Higher $\rightarrow$ Middle $\rightarrow$ Lower $\rightarrow$ ...
  - Dependency Inversion inverts that:
  - Higher $\rightarrow$ Abstraction $\leftarrow$ Middle $\rightarrow$ Abstraction $\leftarrow$ Lower ...
- When using dependency inversion,
  - the higher-levels define the abstractions, and
  - the lower-levels implement the abstractions.
- Why? This enables reuse of the higher-level modules.

Depend on abstractions . . .

```
// The higher level module defines the abstraction.
export interface Juiceable {
    squeeze(): string;
}

// The higher level module depends on the abstraction.
export function makeJuice(ingredients: Array<Juiceable>) {

    const medley = new Array<string>();

    // The higher level no longer depends on the lower level conrete classes.
    // const orange: Juiceable = new Orange();
    // const carrot: Juiceable = new Carrot();

    for (const juicable of ingredients) {
        const juice = juicable.squeeze();
        medley.push(juice);
    }

    return medley;
}
```

Depend on abstractions . . .

```
import { Juiceable, makeJuice } from "./depend-on-abstractions-higher";

// The lower level module depends on the abstraction.
class Orange implements Juiceable {
    public squeeze = () => "orange juice";
}

class Carrot implements Juiceable {
    public squeeze = () => "carrot juice";
}

// That lets it plug in to the higher level module.
makeJuice([
    new Orange(),
    new Orange(),
    new Carrot(),
    new Carrot()]);
```

Only talk to your friends.

- The Law of Demeter (Holland, 1987)
- aka The Princple of Least Knowledge
- Why? Promotes loose coupling via encapsulation.
- "Only talk to your friends"
- "Only use one dot"
  - More than one dot is cause for reflection;
  - it is not necessarily a violation of the LoD.
  - E.g. fluent interfaces use many dots.

Only talk to your friends . . .

```
class Farmer {

    private equipment: Array<FarmEquipment>;

    // In the formal definition of the Law of Demeter
    // a method of an object must only call members of...
    public digHole(place: Place) {

        // objects created within the method,
        const shovel = new Shovel();
        shovel.dig(place);

        // the object itself,
        this.decreaseEnergyLevel();

        // direct properties/fields of the object, or
        this.equipment.push(shovel);

        // any argument of the method.
        let placeName = place.getName();

        // BAD: the Farmer now knows too much about the system.
        placeName = place.details.locationDetails.name;
    }

    private decreaseEnergyLevel = () => { };
}
```

A class should have only one reason to change.

- “A class should have only one reason to change”
  - Recall from “encapsulate what varies.”
  - This refers to changes to source code.
  - Source code changes due to changing requirements.
- SOLID: Single Responsibility Principle (Martin, 2003)
- Why?
  - Use feature X without bringing feature Y
  - Change feature X without breaking/recompiling feature Y.

A class should have only one reason to change . . .

```
/*
 * This is NOT single responsibility.
 * There are several responsibilities here:
 *
 * 1. preparing the raised bed before planting
 * 2. maintaining it after planting
 * 3. harvesting
 *
 */

class RaisedBed {

    public addCompost() { }

    public addMulch() { }

    public addSeeds() { }

    public addWater() { }

    public pullWeeds() { }

    public harvestProduce() { }
}
```

A class should have only one reason to change . . .

```
/*
 * This is a better segregation of responsibilities.
 *
 * E.g. A client can use the harvesting component independently.
 */

class RaisedBedPreparationService {

    public addCompost() { }

    public addMulch() { }

    public addSeeds() { }
}

class RaisedBedMaintenanceService {

    public addWater() { }

    public pullWeeds() { }
}

class RaisedBedHarvestService {

    public harvestProduce() { }
}
```

Don't call us, we'll call you.

- This principle is also known as
  - "Hollywood Principle" (Sweet, 1983)
  - "Inversion of Control" (Johnson and Foote, 1988)
- Inversion of Control (IoC) is about when things happen.
- It differs from Dependency Inversion,
  - which is about who owns the abstraction.
- IoC "makes a framework different from a library" (Fowler)
  - library: "a set of functions you can call"
  - framework: "insert your behavior into various places"
- How? subclassing, implementing interfaces, binding/events

Don't call us, we'll call you . . .

```
abstract class FertalizeGardenProgram {

    private pourFertalizerOnSoil() {
        // some implementation
    }

    // this is a hook
    protected abstract roughUpTheSoil(): void;

    public run() {
        // this is WHEN the roughUpTheSoil routine happens
        this.roughUpTheSoil();
        this.pourFertalizerOnSoil();
    }
}

export class LooseSoilProgram extends FertalizeGardenProgram {

    protected roughUpTheSoil(): void {
        // define loose soil routine
    }
}

export class RockySoilProgram extends FertalizeGardenProgram {

    protected roughUpTheSoil(): void {
        // define rocky soil routine
    }
}
```

Don't call us, we'll call you . . .

```
import {
    LooseSoilProgram,
    RockySoilProgram,
} from "./inversion-of-control";

const program1 = new LooseSoilProgram();
program1.run();

const program2 = new RockySoilProgram();
program2.run();
```

Classes should be open to extension and closed for modification.

- SOLID: Open-Closed Principle
- Once it is shipped, the source code is sacrosanct.
- Rather than change the source code and risk breaking it,
- extend the source code via inheritance or wrapping.
- E.g. the Decorator Pattern (Gamma et al, 1977)

Open-Closed Principle . . .

```
// we have shipped this, clients love it, zero bugs!
export default class GardenWateringSystem {

    // some operations

    public detectMoistureLevel() { }
}
```

Open-Closed Principle ...

```
// version2 without breaking version1
import GardenWateringSystem from "./open-closed";

// composition / decorating / wrapping
class FilteringWateringSystem implements GardenWateringSystem {

    constructor(private baseSystem: GardenWateringSystem) { }

    public detectMoistureLevel = (): void =>
        this.baseSystem.detectMoistureLevel();

    public filterChemicalsFromWater() { }
}
```

Open-Closed Principle . . .

```
// version2 without breaking version1
import GardenWateringSystem from "./open-closed";

// inheritance
class SolarWateringSystem extends GardenWateringSystem {

    public chargeFromSolarPower() { }
}
```

Favour composition over inheritance.

- Composition means a has-a relationship.
  - It is often more semantically natural.
  - It lets us swap implementations at runtime.
  - It is good for code-reuse.
- Inheritance means an is-a relationship.
  - Tall class heirachies are brittle.
  - Changing an implementation is limited to compile time.
  - Is it good for defining taxonomies. (e.g. a String is an Object)
  - Note: inheritance means subclassing not subtyping.
- https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

Favour composition over inheritance.

- SOLID: Liskov Substitution (Liskov and Wing, 1994)
  - A consumer that is expecting type X,
  - should have no surprises on receiving a child XX of type X.
- Compilers do not help: this is semantic not syntactic.
  - E.g. Even though a Square is a Rectangle,
  - a Square class should not inherit from a Rectangle class,
  - because a client with a Rectangle expects,
  - the ability to set the height and width to different values.

Favour composition over inheritance . . .

```
// TODO: Add an example of when inheritance is more appropriate

// TODO: Add an example of when compisition is more appropriate
```

Strive for loosely coupled designs among objects that interact.

- This is the summary statement for all the princples.
- When loosely coupled, we can:
    - change X without needing to change Y, and
    - use X without needing to bring along Y.
- Modular architecture!