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.

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.
  - Requirements change for many reasons.
  - 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 . . .

```
class Product {
    public price: number;
}

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

    public calculateTax(product: Product): number {
        // This does complex, involved 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 helps because it
  - lets us change an implementation, even at runtime
  - allows applications to send the same request to different classes
- A separate, related SOLID principle:
  - Interface Segregation Principle (Martin, 1996)
  - Define an interface that is specific to the needs of the client.
  - "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 only knows about Juiceables.
    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.

- To depend means to make a direct reference.
- Abstractions commit to a interface/type.
- Concrete classes commit to an 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 high 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) {
        medley.push(juicable.squeeze());
    }

    return medley;
}
```

Depend on abstractions . . .

```
// The lower level module depends on the abstraction.
// Thereby letting it plugin to the higher level module.
import { Juiceable } from "./depend-on-abstractions-higher";

export class Orange implements Juiceable {
    public squeeze = () => "orange juice";
}

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

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>;

    // A method of an object may only call methods of:
    public digHole(place: Place) {

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

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

        // The object itself.
        this.decreaseEnergyLevel();

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

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

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

A class should have only one reason to change.

- SOLID: Single Responsibility Principle (Martin, 2003)
- "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.
- Why?
  - (Re)use feature X without bringing feature A-Z.
  - Change feature X without breaking/recompiling what depends on feature A-Z.

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

```java
/*
 * This is NOT single responsibility.
 * There are several responsibilities here:
 *
 * 1. preparing the raised bed before planting
 * 2. maintaining it after planting
 * 3. harvesting
 *
 * It's likely that our watering system will change
 * independently of our harvesting system.
 */

class RaisedBed {

    public addCompost() { }

    public addMulch() { }

    public addSeeds() { }

    public addWater() { }

    public harvestProduce() { }

    public pullWeeds() { }
}
```

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

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

class RaisedBedPreparationService {

    public addCompost() { }

    public addMulch() { }

    public addSeeds() { }
}

class RaisedBedMaintenacenService {

    public addWater() { }

    public pullWeeds() { }
}

class RaisedBedHarvestService {

    public harvestProduce() { }
}
```

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

- "Hollywood Principle" (Sweet, 1983)
- "Inversion of Control" (Johnson and Foote, 1988)
  - Dependency injection is a type of Inversion of Control (TODO: Check this)
  - IoC containers are a type of Dependency Injection
- Dependency Inversion - who owns the abstraction?
- Inversion of Control - when do things happen?
- "coordinating and sequencing application activity"
- "makes a framework different from a library":
  - 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)

Favour composition over inheritance.

- Composition means a has-a relationship.
  - It is often more semantically natural.
  - It lets us swap implementations at runtime.
- Inheritance means an is-a relationship.
  - Tall class heirachies are brittle.
  - Changing an implementation is limited to compile time.
  - It is harder to do correctly.
- SOLID: Liskov Substitution Principle (Liskov and Wing, 1994)
  - A consumer that is expecting A,
  - should have no surprises on receiving a child of A.
  - Compilers do not help: this is a semantic syntactic contraint.
  - e.g. class Hemlock should probably not inherit class Vegetable.

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.