

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?

- Allow change (of system capacities) without redesign.
- Allow reuse in other applications.

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 {
        // Do the calculation of tax.
        return 0;
    }
}

class FarmStand {

    private cart: Array<Product>;

    public CalculateGrandTotal(): number {
        let total = 0;
        const taxCalculator = new TaxCalculator();
        for (const product of this.cart) {
            const tax = taxCalculator.calculateTax(product);
            total += product.price + tax;
        }

        return total;
    }
}
```

Program to interfaces not to implementations.

- an interface says only what requests it will receive
- an implementation says how it will handle those requests
- programming to interfaces helps because it
 - lets us easily change an implementation, even at runtime
 - allows applications to send the same request to different classes
- SOLID: 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 ...

```
class Orange implements Juiceable {
    public squeeze() {
        return new Juice("orange juice");
    }
}

class Carrot implements Juiceable {
    public squeeze() {
        return new Juice("carrot juice");
    }
}

// The juicer is programming to interfaces.
// The following only cares that it is dealing with Juiceables.
function orangeCarrotJuice(juiceGarden: JuiceGarden): Array<Juice> {

    const orange: Juiceable = juiceGarden.pickOrange();
    const carrot: Juiceable = juiceGarden.pickCarrot();

    let ingredients: Array<Juiceable> = [orange, carrot];

    return ingredients.map((j: Juiceable) => j.squeeze());
}
```

Depend on abstractions not on concrete classes.

- interfaces and abstractions are similar: neither can exist
- concrete classes can exist (i.e. can become objects)
- to depend on something means a direct reference to it
- 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 layering, higher-levels define the abstractions
- and lower-levels implement the abstractions.
- Why? Enable reuse of higher-level modules.

Depend on abstractions ...

```
// Both the higher-level juicer and the lower-level components
// depend on an abstraction.
namespace HigherLevel {

  export function juicer(ingredients: Array<Juiceable>): Array<string> {
    // Dependency inversion leverages programming to interfaces.
    return ingredients.map((i) => i.juice());
  }

  // The higher level module owns the abstraction on which it depends.
  export interface Juiceable {
    juice(): string;
  }
}

namespace LowerLevel {

  export class Orange implements HigherLevel.Juiceable {
    public juice() {
      return "orange juice";
    }
  }

  export class Carrot implements HigherLevel.Juiceable {
    public juice() {
      return "carrot juice";
    }
  }
}
```

Only talk to your friends.

- The Law of Demeter (Holland, 1987)
- aka The Principle 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>;  
  
    private energyLevel: number;  
  
    // A method of an object may only call methods of:  
    public DigHole(place: Place) {  
        const shovel = new Shovel();  
        while (this.energyLevel > 0) {  
  
            // 1. The object itself.  
            this.decreaseEnergyLevel();  
  
            // 2. Any argument of the method.  
            const target = place.getHighestPlaceWithin();  
  
            // 3. Any object created within the method.  
            shovel.dig(target);  
        }  
  
        // 4. Any direct properties/fields of the object.  
        this.equipment.push(shovel);  
    }  
  
    private decreaseEnergyLevel() {  
        this.energyLevel = this.energyLevel - 1;  
    }  
}
```

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 ...

```
class RaisedBed {  
  
    public addCompost() { }  
  
    public addMulch() { }  
  
    public addWater() { }  
  
    public addSeeds() { }  
  
    public harvestProduce() { }  
  
    public pullWeeds() { }  
}  
  
/*  
 * 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.  
 */
```

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
 - 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 ...

```
// TODO: Add a template method example.
```

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 contrait.
 - 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 principles.
- When loosely coupled, we can ...
- ... change X without needing to change Y, and
- ... use X without needing to bring along Y.