Design Principles

aka Object Oriented Programming

July 4, 2017

Why?

- Allow change without redesign.
- Allow reuse on other applications.

Encapsulate what varies.

- Encapsulate . . .
  - Restrict outside access to a things parts.
  - Bundle operations with the things they use.
- . . . what varies.
  - This refers to source code.
  - Source code varies due to changing requirements.
  - Requirements change for a lots of 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, 1977)

Encapsulate what varies . . .

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

// We have encapsulated the calculation of tax.
class TaxCalculator {
    public calculateTax(product: Product): number {
        const tax = 0;
        // Do the calculation of tax,
        // which will likely change over time.
        return tax;
    }
}

class FarmStand {

    private cart: Array<Product>;

    public DisplayGrandTotal(): number {
        // Question: What else might we want to change without redesign?
        const taxCalculator = new TaxCalculator();
        return this.cart.reduce((sum, product) => {
            const tax = taxCalculator.calculateTax(product);
            const productGrandTotal = tax + product.price;
            return sum + productGrandTotal;
        }, 0);
    }
}
```

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

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
- The 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 ()
- The Princple of Least Knowledge

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

- Inversion of Control

A class should have only one reason to change.

- The Single Responsibility Principle ()

Classes should be open to extension and closed for modification.

- The Open-Closed Principle ()

Favour composition over inheritance.

- The Liskov Substitution Principle ()

Strive for loosely coupled designs among objects that interact.

-