Design Patterns

aka Object Oriented Programming

July 2, 2017

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

```
export interface Juiceable {
    juice(): string;
}

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

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

function juicer(oranges: Array<Orange>, carrots: Array<Carrot>): Array<string> {

    // This is programming to interfaces.
    // The following only cares that it is dealing with Juiceables.
    let ingredients: Array<Juiceable> = oranges.concat(carrots);
    return ingredients.map((j: Juiceable) => j.juice());
}

// run
const juice = juicer(
    [new Orange(), new Orange()],
    [new Carrot()]
);

console.log(juice);
```

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.

```
// This is dependency inversion.
// Both the higher-level juicer and the lower-level components
// depend on an abstraction.
export function juicer(ingredients: Array<Juiceable>): Array<string> {
    // Dependency inversion leverages programming to interfaces.
    return ingredients.map((i) => i.juice());
}

export interface Juiceable {
    juice(): string;
}

// orange and carrot are lower-level modules
export class Orange implements Juiceable {
    public juice() {
        return "orange juice";
    }
}

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

// run
const juice = juicer([new Orange(), new Carrot()]);
console.log(juice);
```

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.

Strive for loosely coupled designs among objects that interact.