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 createJuiceMedly(): Array<string> {

    let ingredients: Array<Juiceable> = [
        new Orange(),
        new Carrot()
    ];

    // This is programming to interfaces.
    // The call to `map` only cares that it is dealing with Juiceables.
    return ingredients.map((j: Juiceable) => j.juice());
}

// run
const juice = createJuiceMedly();
```

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
- aka The Dependency Inversion Principle
  - traditional high-level classes depend on concrete classes
  - we invert this
  - the high-level defines the abstractions
  - the low-level implements the abstractions
  - both depend on an abstractions

```
// This is dependency inversion.
// The higher level component defines the interface,
// thereby allowing reuse of the high-level component.
export namespace HighLevel {

    export interface Juiceable {
        juice(): string;
    }

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

// orange and carrot are defined in the lower-level
export class Orange implements HighLevel.Juiceable {
    public juice() {
        return "orange juice";
    }
}

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

// run
const juice = HighLevel.createJuiceMedly([new Orange(), new Carrot()]);
```

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.