Strategy Pattern

Uma descrição do Padrão de Projeto *Strategy*. Sua proposta é encapsular algoritmos de forma que fiquem mais fáceis de reutilizar à medida que o código cresce e mais clientes são criados.

Problema

- Sistemas em constante crescimento: reaproveitamento de código.
- Entidades devem ser abertas para extensão mas fechadas para modificação.
- Maximizar coesão, minimizar acoplamento.

Exemplo: reusabilidade

```
class Pato {
    cor = () => {
        return console.log('Branco');
    }

    quack = () => {
        return console.log('Quack quack quack!');
    }

    voar = () => {
        return console.log('Estou voando!!');
    }
}

class PatoDaCidade extends Pato {
    cor = () => {
        return console.log('Cinza');
    }
}

class PatoDoCampo extends Pato {
    cor = () => {
        return console.log('Marrom');
    }
}
```

```
class PatoDeBorracha extends Pato {
    cor = () => {
        return console.log('Amarelo');
    }

    voar = () => {
        // Não voa
    }
}
```

Nova demanda: Pato de madeira

```
class PatoDeMadeira extends Pato {
    cor = () => {
        return console.log('Marrom');
    }

    quack = () => {
        // Não faz quack
    }

    voar = () => {
        // Não voa
    }
}
```

Problema: código repetido!!

Composição versus Herança

- Herança é uma ferramenta poderosa, mas deixa a desejar em relação ao reaproveitamento do código.
- Bom para modelos cuja árvore de herança é facilmente definida e funciona bem; ruim para modelos em que há repetição de algoritmos para nós de mesma profundidade.

Solução

Conjunto de métodos comportamentais:

Cor

```
const corPadrao = () => {
    return console.log('Branco');
}
const corCinza = () => {
    return console.log('Cinza');
}
const corMarrom = () => {
    return console.log('Marrom');
}
const corAmarelo = () => {
    return console.log('Amarelo');
}
```

Quack

```
const quackPadrao = () => {
    return console.log('Quack quack quack!');
}
const naoQuack = () => {
    // Não faz quack
}
```

Voar

```
const voarPadrao = () => {
    return console.log('Estou voando!!');
}
const naoVoar = () => {
    // Não voa
}
```

Definição das entidades para o meu simulador de patos:

```
class Pato {
    constructor(ICor, IQuack, IVoar){
        this.cor = ICor;
        this.quack = IQuack;
        this.voar = IVoar;
    }
}
```

```
const patoDaCidade = new Pato(corCinza, quackPadrao, voarPadrao);
const patoDoCampo = new Pato(corMarrom, quackPadrao, voarPadrao);
const patoDeBorracha = new Pato(corAmarelo, quackPadrao, naoVoar);
const patoDeMadeira = new Pato(corMarrom, naoQuack, naoVoar);
```

```
patoDaCidade.cor();
patoDaCidade.voar();
patoDeBorracha.cor();
patoDeBorracha.quack();
patoDeMadeira.quack();
```

> Cinza
>
> Estou voando!!
>
> Amarelo
>
> Quack quack quack!