Design Pattern - 2

Rujuan Xing

Maharishi International University - Fairfield, Iowa

The Facade Pattern

- When we put up a facade, we present an **outward appearance** to the world which may **conceal a very different reality**.
- This pattern is a structural pattern, it provides a convenient higher-level interface to a larger body of code, hiding its true underlying complexity. Think of it as simplifying the API being presented to other developers, something which almost always improves usability.

Façade Pattern Example

```
const Mortgage = function(name) {
    this.name = name;
}

Mortgage.prototype = {
    applyFor: function(amount) {
        let result = "approved";
        // call subsystems
        // check if account exists
        // check if client has good credit score
        // check if amount is within the approved range
        return `${this.name} has been ${result} for a ${amount} mortgage`;
    }
}

const mortgage = new Mortgage("Asaad Saad");
const result = mortgage.applyFor("$100,000");
```

The Factory Pattern

- The Factory pattern is a creational pattern concerned with the notion of **creating objects**. Where it differs from the other patterns in its category is that it doesn't explicitly require us to use a constructor. Instead, a Factory can provide a generic interface for creating objects, where we can specify the **type of object** we wish to be created.

- Rather than creating an object directly using the **`new`** operator or via another creational constructor, we ask a Factory object for a new object instead. We inform the Factory what type of object is required and it instantiates this, returning it to us for use.

- This is particularly useful if the object creation process is relatively complex, like if it strongly depends on dynamic factors or application configuration.

Factory Pattern Example

```
function Mahler(options) {
    this.clef = options.clef || "treble";
    this.signature = options.signature || "4-flat";
    this.tempo = options.tempo || 75;
}

function Bruckner(options) {
    this.clef = options.clef || "bass";
    this.tone = options.tone || "aria";
}
createMelody = function(options) {
    switch (options.clef) {
        case "treble":
            return new Mahler(options);
        case "bass":
            return new Bruckner(options);
    }
};

const melody = createMelody({
    clef: "treble",
    signature: "1-flat",
    tempo: 102
});
```

The Decorator Pattern

- The Decorator pattern extends (decorates) an object’s behavior dynamically. The ability to add new behavior at runtime is accomplished by a Decorator object which wraps around the original object. Multiple decorators can add or override functionality to the original object.

Decorator Pattern Example

```
const User = function(name) {
    this.name = name;
    this.log = () => console.log("User: " + this.name);
}

const DecoratedUser = function(user, city, state) {
    this.name = user.name;
    this.city = city;
    this.state = state;
    this.log = () => console.log(`User: ${this.name}, ${this.city}, ${this.state}`);
}

const user = new User("Asaad");
user.log();

const decorated = new DecoratedUser(user, "Fairfield", "Iowa");
decorated.log();
```

The Strategy Pattern

- Define a **family of algorithms, encapsulate each one, and make them interchangeable.** Strategy lets the algorithm vary independently from clients that use it.

- The Strategy pattern encapsulates alternative algorithms (or strategies) for a particular task. It allows a method to be swapped out at runtime by any other strategy.

The Strategy Pattern Example

- Given the following three shipping strategies:

```
class UPS {

    calculate(product) { return "$45.95" }

};


class USPS {

    calculate(product) { return "$39.40" }

};


class Fedex {

    calculate(product) { return "$43.20" }

};
```

The Strategy Pattern Example

- We can implement the strategy pattern and encapsulate each one of them, and make them interchangeable.

```
class Shipping {
    shippingCompany = "";

    setStrategy(shippingCompany) {
        this.shippingCompany = shippingCompany;
    }

    calculate(product) {
        return this.shippingCompany.calculate(product);
    }
};
```

The Strategy Pattern Example

- That makes it easier for the client to choose their strategy:

```
const ups = new UPS();
const usps = new USPS();
const fedex = new Fedex();

const shipping = new Shipping();

shipping.setStrategy(ups);
console.log("UPS Strategy: " + shipping.calculate(product));
shipping.setStrategy(usps);
console.log("USPS Strategy: " + shipping.calculate(product));
shipping.setStrategy(fedex);
console.log("Fedex Strategy: " + shipping.calculate(product));
```

Proxy Pattern

- The Proxy pattern provides a **surrogate object** for another object and controls access to this other object.

- The Proxy forwards the request to a target object. The interface of the Proxy object is the same as the original object and clients may not even be aware they are dealing with a proxy rather than the real object

Calling an API for the same results

- Given the following method:

```
function GeoCoder() {
    this.getLocation = function(address) {
        if (address === "Fairfield") {
            return "41.006630 Latitude, -91.965050 Longitude";
        }
    };
}

const geoCoder = new GeoCoder();
geoCoder.getLocation("Fairfield");
geoCoder.getLocation("Fairfield");
```

Proxy Pattern Example

- We can improve its performance using a proxy:

```
function GeoProxy() {
    var geocoder = new GeoCoder();
    var geocache = {};
    return {
        getLocation: function(address) {
            if (!geocache[address]) {
                geocache[address] = geocoder.getLocation(address);
            }
            return geocache[address];
        }
    };
};
const proxy = GeoProxy();
proxy.getLocation("Fairfield");
proxy.getLocation("Fairfield");
```

Memoization

- Design patterns define how more than one relatively complex class/object interact. This involves high level of abstraction.
- Memoization is a **coding/optimization technique** that speeds up applications by storing the results of expensive function calls and returning the cached result when the same inputs are supplied again.
- An **expensive function call** is a function call that consumes huge chunks of these time or memory during execution due to heavy computation.
- A **cache** is simply a temporary data store that holds data so that future requests for that data can be served faster.

How Does Memoization Work?

- The concept of memoization in JavaScript is built majorly on two concepts:
- Closures
- Higher Order Functions

When to Memoize a Function?

- Expensive function calls (functions that carry out heavy computations)
- Functions with a limited and highly recurring input range.
- Recursive functions with recurring input values.
- Pure functions (functions that return the same output each time they are called with a particular input).

Case Study: The Fibonacci Sequence

- Write a function to return the **`nth`** element in the Fibonacci sequence, where the sequence is:
- **`[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …]`**

```
function fibonacci(n) {
    if (n <= 1) {
        return 1
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Analyze the solution

- Looking at the diagram, when we try to evaluate `fib(5)`, we notice that we repeatedly try to find the Fibonacci number at indices `0`, `1`, `2` and `3` on different branches.
- This is known as redundant computation and is exactly what memoization stands to eliminate.