Design Pattern - 2

Rujuan Xing

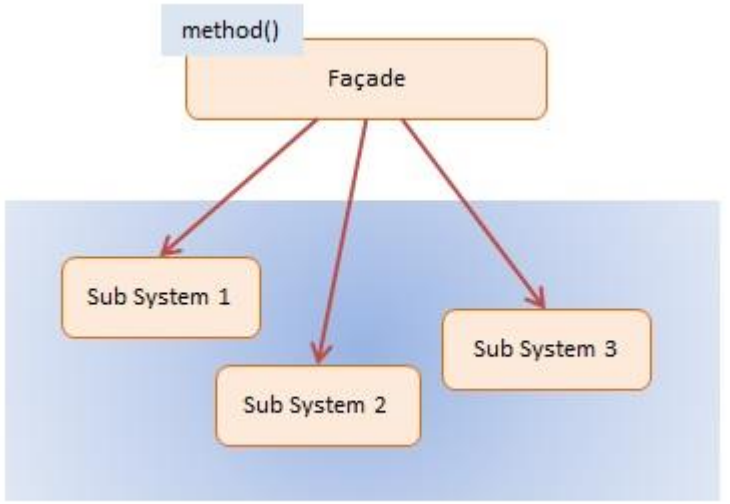
Maharishi International University - Fairfield, Iowa

Main Point Preview

- Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

- Do less and accomplish more

The Facade Pattern



- The Façade pattern provides an interface which shields clients from complex functionality in one or more subsystems. It is often present in systems that are built around a multi-layer architecture.
- The intent of the Façade is to provide a high-level interface (properties and methods) that makes a subsystem or toolkit easy to use for the client.
- Example:
  - On the server, in a multi-layer web application you frequently have a presentation layer which is a client to a service layer. Communication between these two layers takes place via a well-defined API. This API, or façade, hides the complexities of the business objects and their interactions from the presentation layer.
  - Another area where Façades are used is in refactoring. Suppose you have a confusing or messy set of legacy objects that the client should not be concerned about. You can hide this code behind a Façade. The Façade exposes only what is necessary and presents a cleaner and easy-to-use interface.
- Façades are frequently combined with other design patterns. Facades themselves are often implemented as singleton factories.

Façade Pattern Example

The `Mortgage` object is the Facade in the sample code. It presents a simple interface to the client with only a single method: `applyFor`. But underneath this simple API lies considerable complexity.

```
class Bank {
    verify(name, amount) {
        // complex logic ...
        return true;
    }
}

class Credit {
    get(name) {
        // complex logic ...
        return true;
    }
}

class Background {
    check(name) {
        // complex logic ...
        return true;
    }
}
```

```
class Mortgage {
    name;
    constructor(name) {
        this.name = name;
    }

    applyFor(amount) {
        // access multiple subsystems...
        var result = "approved";
        if (!new Bank().verify(this.name, amount)) {
            result = "denied";
        } else if (!new Credit().get(this.name)) {
            result = "denied";
        } else if (!new Background().check(this.name)) {
            result = "denied";
        }
        return this.name + " has been " + result +
            " for a " + amount + " mortgage";
    }

}


let mortgage = new Mortgage("Joan Templeton");
let result = mortgage.applyFor("$100,000");

console.log(result);
```

The Factory Method Pattern

- A Factory Method creates new objects as instructed by the user.
- One way to create objects in JavaScript is by invoking a constructor function with the new operator. There are situations however, where the user does not, or should not, know which one of several candidate objects to instantiate. The Factory Method allows the user to delegate object creation while still retaining control over which type to instantiate.
- The key objective of the Factory Method is extensibility. Factory Methods are frequently used in applications that manage, maintain, or manipulate collections of objects that are different but at the same time have many characteristics (i.e. methods and properties) in common.
- An example would be a collection of documents with a mix of Xml documents, Pdf documents, and Rtf documents.

Factory Method Pattern Example

```
class FullTime {
    constructor() {
        this.hourly = "$12";
    }
}

class PartTime {
    constructor() {
        this.hourly = "$12";
    }
}

class Temporary {
    constructor() {
        this.hourly = "$10";
    }
}
```

```
const employees = [];
const factory = new Factory();

employees.push(factory.createEmployee("fulltime"));
employees.push(factory.createEmployee("parttime"));
employees.push(factory.createEmployee("temporary"));
for (let i = 0, len = employees.length; i < len; i++) {
    employees[i].say();
}
```

```
class Factory {
    createEmployee(type) {
        let employee;

        if (type === "fulltime") {
            employee = new FullTime();
        } else if (type === "parttime") {
            employee = new PartTime();
        } else if (type === "temporary") {
            employee = new Temporary();
        }
        employee.type = type;

        employee.say = function() {
            console.log(this.type + ": rate " + this.hourl
y + "/hour");
        }

        return employee;
    }
}
```

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.
- An example of a decorator is security management where business objects are given additional access to privileged information depending on the privileges of the authenticated user. For example, an HR manager gets to work with an employee object that has appended (i.e. is decorated with) the employee's salary record so that salary information can be viewed.

Decorator Pattern Example

```
class User {
    constructor(name) {
        this.name = name;
    }

    log() {
        console.log("User: " + this.name);
    }
}
```

```
class DecoratedUser {

    constructor(user, street, city) {
        this.user = user;
        this.name = user.name; // ensures interface stays the same
        this.street = street;
        this.city = city;
    }

    log() {
        console.log("Decorated User: " + this.name + ", " +
            this.street + ", " + this.city);
    };
}
```

```
var user = new User("Kelly");
user.log();

var decorated = new DecoratedUser(user, "Broadway", "New York");
decorated.log();
```

The Strategy Pattern

- 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 method (strategy) without the client realizing it.
- Essentially, Strategy is a group of algorithms that are interchangeable.
- Say we like to test the performance of different sorting algorithms to an array of numbers: shell sort, heap sort, bubble sort, quicksort, etc. Applying the Strategy pattern to these algorithms allows the test program to loop through all algorithms, simply by changing them at runtime and test each of these against the array. For Strategy to work all method signatures must be the same so that they can vary without the client program knowing about it.

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

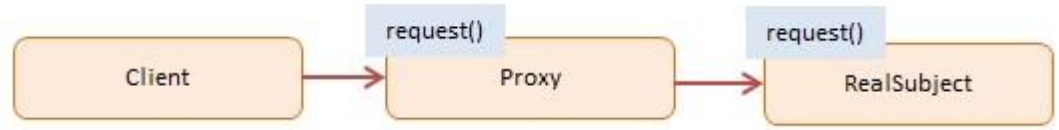
- That makes it easier for the client to choose their strategy:

```
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



RealSubject

- defines the real object for which service is requested

```
class GeoCoder {
    getLatLng(address) {
        if (address === "Fairfield") {
            return "41.006630 Latitude, -91.965050 Longitude";
        } else if (address === "London") {
            return "51.5171° N, 0.1062° W";
        } else {
            return ""
        }
    };
}
```

Proxy

- provides an interface similar to the real object
- maintains a reference that lets the proxy access the real object
- handles requests and forwards these to the real object

```
class GeoProxy {
    constructor() {
        this.geocoder = new GeoCoder();
        this.geocache = {};
    }
    getLatLng(address) {
        if (!this.geocache[address]) {
            this.geocache[address] = this.geocoder.getLatLng(address);
        }
        console.log(address + ": " + this.geocache[address]);
        return this.geocache[address];
    }
}

let geo = new GeoProxy();
// geolocation requests
console.log(geo.getLatLng("Fairfield"));
console.log(geo.getLatLng("Fairfield"))
console.log(geo.getLatLng("London"));
```

Memoization

- Memoization is an optimization technique used primarily to speed up computer programs by **storing the results of expensive function calls** and returning the cached result when the same inputs occur again.
- **Memoizing** in simple terms means **memorizing** or storing in memory. A memoized function is usually faster because if the function is called subsequently with the previous value(s), then instead of executing the function, we would be fetching the result from the cache.
- 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

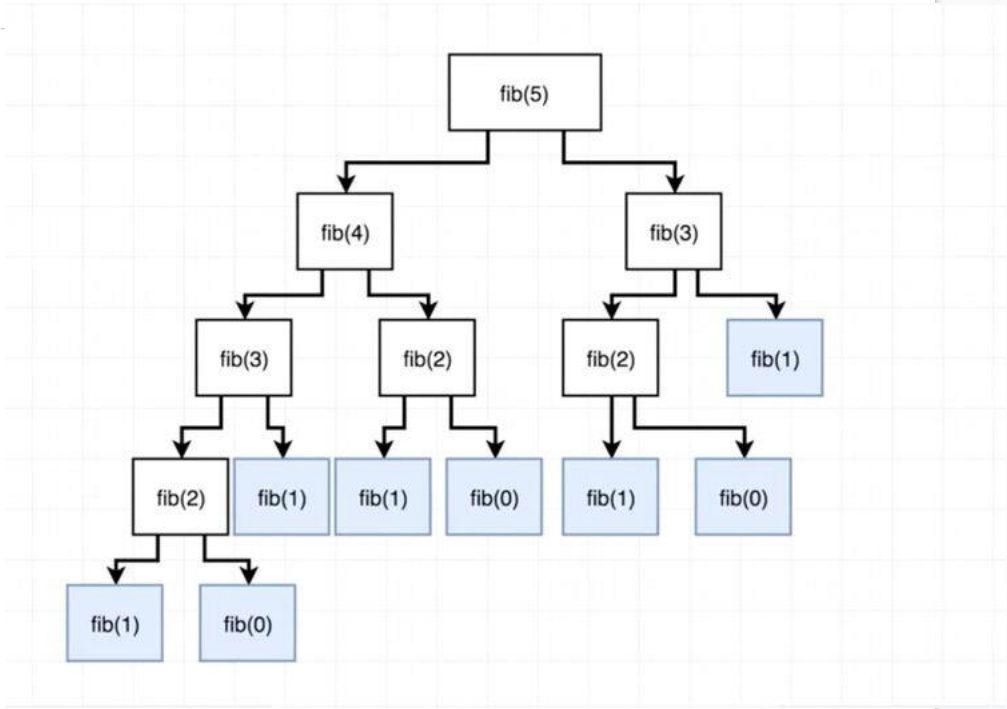
- Write a function to return the **`nth`** element in the Fibonacci sequence, where the sequence is:

- **`[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.
- For example, to compute the 50th Fibonacci number, the recursive function must be called over 40 billion times (40,730,022,147 times to be specific)!
- This is known as redundant computation and is exactly what memoization stands to eliminate.



References

- https://www.dofactory.com/javascript/design-patterns
- https://scotch.io/tutorials/understanding-memoization-in-javascript

Main Point

- Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

- Do less and accomplish more