JavaScript Design Patterns

CS445 Modern Asynchronous Programming

Maharishi University of Management

Department of Computer Science

Teaching Faculty: Assistant Professor Umur Inan

Prepared by: Associate Professor Asaad Saad

What is a Pattern?

A pattern is a reusable solution that can be applied to commonly occurring problems in software design (writing JavaScript web applications).

Patterns are templates for how we solve problems and can be used in a few different situations.

The role of a pattern is to provide us with a solution scheme.

Benefits

Design patterns have three main benefits:

- **Patterns are proven solutions**: They provide solid approaches to solving issues in software development using proven techniques.
- **Patterns can be easily reused**: A pattern usually reflects an out of the box solution that can be adapted to suit our own needs.
- **Patterns are expressive**: Each pattern consists of many properties
  - Context: Where/under what circumstances is the pattern used?
  - Problem: What are we trying to solve?
  - Solution: How does using this pattern solve our proposed problem?
  - Implementation: What does the implementation look like?

Solving Common Problems

Imagine that we have a script where for each DOM element found on a page with class "foo" we wish to increment a counter. There are three different ways this problem could be solved:

1. Select all of the elements in the page and then store references to them. Next, filter this collection and use regular expressions (or another means) to only store those with the class "`foo`".
2. Use a modern native browser feature such as `querySelectorAll()` to select all of the elements with the class "`foo`".
3. Use a native feature such as `getElementsByClassName()` to similarly get back the desired collection.

Anti-Patterns

An **anti-pattern is a bad design** that is worthy of documenting.

- Polluting the global namespace by defining a large number of variables in the global context
- Passing strings rather than functions to either **`setTimeout`** or **`setInterval`** as this triggers the use of **`eval()`** internally.
- Modifying the **`Object`** class prototype
- Using JavaScript in an inline form as this is inflexible
- The use of **`document.write`** where native **`DOM`** alternatives such as **`document.createElement`** are more appropriate.

Knowledge of anti-patterns is critical for success. Once we are able to recognize such anti-patterns, we're able to refactor our code to negate them so that the overall quality of our solutions improves instantly.

Categories Of Design Pattern

Creational Design Patterns

Focus on handling object **creation** mechanisms

Structural Design Patterns

Concerned with object composition and typically identify simple ways to realize **relationships** between different objects.

Behavioral Design Patterns

Focus on improving the **communication** between different objects in a system.

Creational Design Patterns

Module

Prototype

Singleton

Factory

Abstract Factory

Builder

Structural Design Patterns

Decorator

Façade

Proxy

Adapter

Bridge

Composite

Flyweight

Behavioral Design Patterns

Observer
Strategy
State
Visitor
Interpreter
Chain of Responsibility
Command
Iterator
Mediator
Memento

The Prototype Pattern

The prototype pattern as one which creates objects based on a template of an existing object through cloning.

We can think of the prototype pattern as being based on prototypal inheritance where we create objects which act as prototypes for other objects. (we are creating copies of existing functional objects).

When defining a function in an object, they're all **created by reference** (so all child objects point to the same function) **instead of creating their own individual copies**.

Prototype Pattern Example

```
const car = {
    name: "Toyota Camry",
    drive() {
        console.log("I'm driving!");
    },
    honk() {
        console.log("Horn is making sound!");
    }
};

const myCar = Object.create(car);
myCar.drive()
console.log(myCar.name);
```

The Module Pattern

The Module pattern **encapsulates private members**, **state and behavior using closures.** It provides a way of wrapping a mix of public and private methods and variables, protecting pieces from leaking into the global scope and accidentally colliding with another developer's interface.

With this pattern, only a public API is returned, keeping everything else within the closure private.

Module Pattern Example

```
const counterModule = (function () {
  let counter = 0;
  return {
    incrementCounter: function () {
      return counter++;
    },
    resetCounter: function () {
      console.log( "counter value prior to reset: " + counter );
      counter = 0;
    }
  };
})();

counterModule.incrementCounter();
counterModule.resetCounter();
```

The Singleton Pattern

Restricts instantiation of a class to a single object. The Singleton pattern can be implemented by creating a class with a method that creates a new instance of the class if one doesn't exist. If an instance already existing, it simply returns a reference to that instance.

There must be exactly one instance of a class, and it must be accessible to clients from a common access point.

Singleton Pattern Example

```
const Singleton = (function () {
    let instance;

    return {
        getInstance: function () {
            if (!instance) {
                instance = new Object("I am the instance");
            }
            return instance;
        }
    };
})();

const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

console.log("Same instance? " + (instance1 === instance2));
```

The Observer Pattern

The Observer is a design pattern where an object (known as a **subject**) maintains a list of objects depending on it (**observers**), automatically notifying them of any changes to state.

When a subject needs to notify observers about something interesting happening, it broadcasts a notification to the observers (which can include specific data related to the topic of the notification).

When we no longer wish for a particular observer to be notified of changes by the subject they are registered with, the subject can remove them from the list of observers.

Observer Pattern Example

```
function Subject() {
    this.observers = [];
}

Subject.prototype = {
    subscribe: function (fn) {
        this.observers.push(fn);
    },
    emit: function (msg) {
        this.observers.forEach(function (fn) {
            fn.call(null, msg);
        });
    }
}

const subject = new Subject();
subject.subscribe(console.log);
subject.emit('Hello');
```