Logisch: JavaScript Pattern, oder?

Christof Kaller
Cassini
Düsseldorf

Schlüsselworte
Design Pattern, Service, Singleton, Facade, JavaScript.

Einleitung

Während in der Welt der objektorientierten Programmierprachen Pattern gang und gebe sind, finden wir diese in der JavaScript Welt nur selten. Doch deren Nutzung und gezielte Einsetzung verspricht viele Vorteile. Modularisierung und Kapselung, Vermeidung von Redundanz oder Erhöhung der Flexibilität und Wiederverwendung sind nur einige dieser wichtigen Vorteile. Eine sehr einfache Anwendung zum Verschlüsseln eines Inputfeldes mit einem umfangreichen JavaScript (eine Flatfile) wird in wenigen Schritten refactored. Dabei werden verschiedene Pattern eingesetzt und nebenbei wird die Anwendung sinnvoll erweitert.Ziel ist es, die Anwendung „aufzuräumen“, zu modularisieren und theoretische Grundlagen bzgl. Design Patter praktisch umzusetzen.

Was sind „JavaScript Pattern“ und wie setze ich diese ein?

Design Pattern (Entwurfsmuster)

Design Pattern (deutsch: Entwurfsmuster) stellen abstrakte, bewährte Konzepte und Lösungen für weit verbreitete, wiederkehrende Probleme in der Softwareentwicklung dar. Sie beschreiben, wie Klassen untereinander agieren und in Beziehung stehen sollten. Dadurch werden Flexibilität, Wiederverwendbarkeit, Lesbarkeit, Effizienz, Wartbarkeit und Sicherheit erheblich verbessert. Die Design Pattern wurden im Laufe der Zeit entwickelt und basieren auf bereits gelösten Problemen und den Erfahrungen anderer Softwareentwickler. Wenn man Design Pattern nutzen will, muss man sich an gewisse Regeln halten. Sie geben in Teilen die Architektur von Anwendungen vor. Eine Kombination mit guten Coding Guidelines macht durchaus Sinn. Viele erfolgreiche JavaScript Libraries nutzen und folgen diversen Design Pattern. Viele JavaScript Entwickler nutzen Pattern vielleicht ohne es zu merken. Durch gezieltes Refactoring und Transformation kann man auch „legacy Code“ in eine neue Struktur überführen.

In vielen Projekten gibt es aus „legacy Gründen“ und weil es „historisch so gewachsen“ ist, JavaScript Dateien mit einer „wahnsinnigen“ Anzahl, teils von vielen tausenden von Zeilen und etlichen Funktionen und Methoden. Jeden Tag wächst die Datei ein wenig mehr und wird noch unübersichtlicher, weil immer wieder andere Entwickler daran arbeiten. Das führt dazu, dass nur Experten diese eine Datei und das auch nur noch in Teilen verstehen. Fehlersuche wird erheblich erschwert. Redundanzen sind vorprogrammiert. Abhängigkeiten sind nicht mehr zu überblicken. Wenn man an einer Stelle hineindrückt, fällt auf der anderen Seite etwas heraus. Man muss mit Suchen arbeiten, um sich in der Datei vorzubewegen und an die richtigen Stellen zu springen. Und Änderungen müssen an vielen Stellen vorgenommen werden. Um dem immer schlechter werdenden Code entgegenzuwirken, empfiehlt es sich Design Pattern einzuführen. Dazu müssen gewisse Voraussetzungen geschaffen werden. Eine der wichtigsten ist, dass man sich einen Überblick verschafft. Dazu geht man am besten wie folgt vor:

Debuggen

- Beim Debuggen einer Anwendung kann man wunderbar einzelne zusammengehörende Funktionen erkennen, entsprechend kommentieren und kleinere Coding Guidelines überprüfen. Der Chrome- Browser bietet sich dazu an, die große JavaScript Datei direkt beim Debuggen zu kommentieren. Gruppen und festgestellte Funktionalitäten festzuhalten - Schritt für Schritt - kann später helfen.

Gruppieren - Kategorisieren

- Gruppieren ist ein sinnvoller nächster Schritt. Beim Gruppieren geht es darum, thematisch zusammengehörende Elemente zu erkennen und entsprechend in der großen Datei zusammenzuschieben. Das ist der erste Schritt im Refactoring Prozess. Bisher haben wir noch keinen Code verändert, sondern nur verschoben. Selbst das kann in großen Dateien schon einen erheblichen Vorteil bringen. Diese Kategorien werden später ausgelagert. Mögliche Gruppen wären z.B. Login, User, Warenkorb, Produktdetailseite (PDS), Check Out Vorgang, Sprache, Header, Footer, Menü, Newsletter, Datensynchronisation, Persistenz.

Sortieren

- Beim Sortieren geht es darum innerhalb der Gruppen eine erste Ordnung zu schaffen. Es bietet sich an nach Daten, Frontend und Backend Funktionalitäten zu sortieren.
    - Daten, die Listen und Variablen sind.
    - Backend Funktionalitäten, wie z.B.:
        - hinzufügen zur Liste, löschen, neue berechnen,
        - synchronisieren
    - Frontend Funktionalitäten, wie das Anzeigen der Daten im Frontend, das Befüllen der div Tags
    - Eventuelle Koordinationsfunktionen holen Daten, veränderen Daten, stellen Daten dar.

  Hierbei erkennt man vielleicht schon Redundanzen, die später aufgelöst werden können.

Schneiden

- Die oben erstellten Gruppen können jetzt in einzelne Dateien ausgegliedert werden. Dieser Prozess kann nach und nach stattfinden, da bisher Funktional keine Änderungen vorgenommen wurden.
- Die einzelnen Dateien kann man anschließend mittels Tools wie z.B. grunt minifizieren und uglifizieren. Das hat den Vorteil, dass nur wenige Dateien übertragen werden müssen und diese platzsparend aufbereitet werden, was insgesamt die Performance steigert.

Wenn man diese Schritte erfolgreich umgesetzt hat, hat man bereits die ersten Schritte in die richtige Richtung vollbracht und wird schon schnell viele der gewonnenen Vorteile bei einer parallelen Weiterentwicklung bemerken. Wichtige Voraussetzungen und erste Schritte für die erfolgreiche Umsetzung von Design Pattern sind jetzt vorhanden. Wir haben eine erste Modularisierung erreicht.

Umbauen Design Pattern nutzen

Wenn man den obigen Empfehlungen gefolgt ist, hat man eventuell eine Kategorie Zahlsysteme (Payment Service Provider), Analysen vielleicht oder vielleicht auch eine Kategorie Warenkorb identifiziert. Im Folgenden gehen wir von einem Shopsystem aus, und haben daher auch einen Warenkorb. Wenn man sich Gedanken über einen Warenkorb im Webshop Umfeld und dessen Funktionen macht, kann man schnell die Frage beantworten, wie viele Warenkörbe und an wie vielen Stellen diese existieren sollten. In der Regel nur einmal. Genauso wie eine Merkliste übrigens.

Woraus besteht so eine Warenkorbfunktionalität? Daten im Warenkorb werden in einer Liste festgehalten, werden mit Backendsystemen synchronisiert, zudem werden Waren werden hinzugefügt und entfernt. Eventuell assoziiert man damit zusätzlich noch die Berechnung der Preise oder Gutscheine. Diese konnten und sollten später auch in eigene Kategorien ausgegliedert werden. Wenn wir nun die reinen Daten des Warenkorbs festhalten wollen und sicherstellen, dass es alles nur einmal gibt, ist das Singleton Design Pattern das Pattern, dass dieser Funktionalität entspricht.

Das erste hier vorgestellte Design Pattern ist daher das Singleton Pattern.

```
var WarenkorbListe = (function () {

          var warenkorbSingletonListe;



      function add(pkId) {
               warenkorbSingletonListe.add(pkId);
          }
      function remove(pkId) {
               warenkorbSingletonListe.remove(pkId);
          }

...

}

      var WarenkorbSingleton = (function () {
          var warenkorbSingletonInstance;

    function createInstance() {

        var object = new WarenkorbListe ("Warenkorbinstance");

        return object;

    }
```

```
    return {
        getInstance: function () {
            if (!warenkorbSingletonInstance) {
                warenkorbSingletonInstance = createInstance();
            }
            return warenkorbSingletonInstance;
        }
    };
})();

function test() {

    var warenkorb1 = WarenkorbSingleton.getInstance();
    var warenkorb2 = WarenkorbSingleton.getInstance();
console.log("Warenkorb 1 = Warenkorb 2? " + (warenkorb1 === warenkorb2));
}

// Die warenkorb1. warenkorbSingletonListe und
// warenkorb2.warenkorbSingletonListe
// sind jetzt immer identisch.
// Jetzt müssen alle Verweise auf die WarenkorbListe angepasst werden.
// Alle Stellen, an denen mit new Warenkorbliste() eine neue Liste
// erstellt wurde, müssen mit dem Aufruf von oben ersetzt werden.
// Die Liste sollte aus den wesentlichen Daten bestehen. Eindeutige
```

```
// ProduktId und Anzahl. Alles andere baut man sich nach und nach
// zusammen. Die Produkte und Details dazu werden in eine eigene Klasse
// ausgelagert und später referenziert.
// Diese Singletonklasse könnte man noch um Warenwert, MwSt., Bildern,
// Links zu Details ausstatten. Diese Informationen sind aber so
// allgemein,
// dass sie auch an anderen Stellen wiederverwendet werden können. Daher
// sollten später weitere neue Objekte eingeführt werden.
```

Der nächste Schritt der sich anbietet ist das Extrahieren der Synchronisationsfunktion, der Kommunikation mit dem Backendsystem und dem Befüllen des WarenkorbSingletons. Wobei das Hinzufügen und Entfernen daraus über eine dazwischen geschoben Klasse stattfinden sollte. Ein weitere Klasse für das Darstellen und Updaten der Darstellung sollte gebaut werden.

Die verschiedenen so entstandenen Services können dann in einer Facade „zusammengefasst“ werden, umso die Komplexität der einzelnen Klassen und Methoden zu verstecken oder sinnvoll zu bündeln.

Eine mögliche Facade zum Interagieren mit dem Warenkorbsystem, welches entstanden ist, könnte in etwa wie folgt aussehen:

```
var WarenkorbListe = (function () {

  var warenkorbSingletonListe = WarenkorbSingleton.getInstance();


function add(pkId) {

        warenkorbSingletonListe.add(pkId);

    }

function remove(pkId) {

        warenkorbSingletonListe.remove(pkId);

    }

...

}
```

```
var WarenkorbFacade = (function () {
    var warenkorbSingletonInstance = new WarenkorbListe();

    return {
        add: function (pkId) {
            return warenkorbSingletonInstance.add(pkId);
        }
        remove: function () {
            return warenkorbSingletonInstance.remove(pkId);
        }
        repaintAll: function(){
        //... add more code
        return warenkorbView.repaintAll();
        }
        //... add more functions

    };
})();

function test() {
     var warenkorb = new WarenkorbFacade();
    warenkorb.add("123");
    warenkorb.synchronizeToServer();
    warenkorb.repaintAll();
```

```
console.log("Warenkorb 1 = " + (warenkorb1.list(“asc“)));

}
```

Im letzten Schritt in dieser Kategorie braucht man noch einen Controller. Er ist selber wieder eine Facade, die man aus anderen Bereichen der Anwendung heraus ansprechen kann. Er abstrahiert den Warenkorb nach außen. Natürlich können die in den Facade zusammengefassten Funktionen immer noch manuell angesprochen werden.

Inzwischen hat man fünf neue Dateien erstellt und es bietet sich an, diese in einen eigenen Ordner auszulagern. Die Produktdetailbeschreibung kann natürlich später wiederverwendet und noch einmal verschoben werden.

Das Kommentieren beim Debugging und beim Refactoring mit den Daten zu beginnen haben sich als eine gute Strategie bewährt. Von da aus kann man schnell Funktionen in Facaden und Services kapseln. Man geht dabei Schritt für Schritt vorgehen. Bei neuen Funktionalitäten plant man natürlich bevor man das ganze umsetzt und analysiert, ob sich weitere Funktionen vereinheitlichen oder wiederverwenden lassen. Im Internet findet man noch diverse gute Quellen, die die verschiedenen Pattern anschaulich darstellen. Beim Refactorn muss man Schritt für Schritt vorgehen und immer wieder von vorne anfangen bei Refaktoring Prozess. Kleine Schritte verringern dabei das Fehlerrisiko. Das Ergebnis ist ein sauberer Code, der effizient, leicht zu warten und widerverwendbar ist.

Wenn man diese Schritte oft genug wiederholt, werden die große Datei immer kleiner und es entsteht ein Quellcode mit dem man wieder gerne arbeitet.

Nebenbei setzt man infolge der obigen Empfehlungen bereits ein neues Pattern um. Das Model View Controller (MVC / MVP) Pattern, bei dem es darum geht, das Model, View und Controller voneinander getrennt gehalten und entwickelt werden um eine lose Kopplung zu erhalten und es möglichst einfach ist einzelne Teile der Anwendung schnell zu ersetzen. Das Model, welches die Daten hält, wird dabei vom Controller mit der View zusammengehalten. Das Model weiß nicht und muss auch nicht wissen, wie und ob die Daten dargestellt werden. Der Controller weiß es auch nicht. Er weiß aber, wen er wann aufrufen muss. Die View weiß, wo sie etwas ändern und hinschreiben muss.

Kontaktadresse:
Christof Kaller
Cassini GmbH
Bennigsen-Platz, 1
D- 40474 Düsseldorf

Telefon:	+49 (0) 211 65854133
Mobile:	+49 (0) 12-345 6788
E-Mail	Christof.Kaller@cassini.de
Internet:	www.cassini.de