Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AJAX ..... *Asychronous JavaScript and XML*
DOM ..... *Document Object Model*
ES ..... *ECMAScript*
GoF ..... *Gang of Four*
JS ..... *JavaScript*
MVC ..... *Model-View-Controller*
MVP ..... *Model-View-Presenter*
MVVM ..... *Model-View-ViewModel*

1 Einleitung

Je größer ein Projekt ist, desto wahrscheinlicher ist es auch darüber den Überblick zu verlieren. Ohne Überblick ist es wesentlich schwerer Fehler zu identifizieren und zu beheben, aber auch neue Inhalte sauber hinzuzufügen. Um ein Projekt übersichtlich zu halten, müssen seine Bestandteile gut strukturiert sein, so auch der JavaScript Code. Die Aufrechterhaltung von wohlstrukturiertem Code kostet viel Zeit, wenn es keine festen Vorgaben gibt. So muss der Entwickler bei jeder Änderung viel Zeit damit verbringen die Lösung für sein Problem sauber zu implementieren. Aus Zeitmangel wird jedoch vereinzelt nur so weit gearbeitet, bis die Funktionalität sichergestellt wurde. In vielen Fällen heißt das, dass auf Sauberkeit und Strukturiertheit des Codes wenig Wert gelegt wird.

Hier kommt eine Architektur ins Spiel. Durch eine feste Vorgabe, wie Probleme zu lösen sind, wird der Zeitaufwand eliminiert, der sonst in die Planung der Lösung fließen würde. Außerdem kann garantiert werden, dass der Code einheitlich strukturiert ist. So können auch bei möglichen Personalwechseln andere Entwickler ohne viel Einarbeitung den Code verstehen.

Da diese Architektur, nachdem sie einmal eingeführt ist, nur sehr unangenehm wieder zu ersetzen ist, sollte sie von Beginn an optimal ausgelegt sein. So ist es Ziel dieser Arbeit eine Architektur zu entwickeln, die in SAP-Commerce-Cloud Systemen optimal einsetzbar ist.

2 Abgrenzung

Im Rahmen dieser Arbeit soll eine einheitliche Architektur für JavaScript (JS) Code entwickelt werden. Diese soll explizit für den Einsatz in SAP-Commerce-Cloud („Hybris“) Systemen optimiert sein. Um dies zu gewährleisten, wird im ersten Kapitel untersucht welche Anforderungen eine JS-Architektur in dieser Umgebung erfüllen muss.

Im nächsten Kapitel werden einige strukturelle Konzepte („Design Patterns“) vorgestellt. Dieses Kapitel dient dem Überblick über existierende Ansätze, welche in der Auswertung weiter betrachtet werden.

In der Auswertung werden die vorgestellten Patterns unter Betrachtung der Anforderungen bewertet. Basierend auf dieser Bewertung wird eine Standard-Struktur formuliert, welche für die Anwendung in Hybris Projekten optimal geeignet ist.

Das letzte Kapitel widmet sich einiger Ausblicke. Zum einen wird die Umsetzbarkeit der entwickelten Struktur in laufenden Projekten betrachtet, zum anderen wird die mögliche Einführung von ECMAScript 6, sowie der damit einhergehenden Veränderungen, kurz angesprochen.

3 JavaScript in Hybris Projekten

Die grundlegenden Herausforderungen in JavaScript Anwendungen unterscheiden sich kaum zu denen anderer Sprachen. Rechenzeit und Speicherauslastung sollen niedrig gehalten werden um das Endnutzergerät nicht unnötig zu beanspruchen. Hinzu kommt, dass JavaScript Code möglichst klein sein sollte um Ladezeiten zu minimieren.

Zusätzlich zu diesen Herausforderungen, stellt die Arbeit in Hybris Projekten spezielle Anforderungen an JavaScript Code. Um diese Anforderungen zu konkretisieren wurden intern Experten befragt, die schon lange mit JavaScript in diesem Umfeld gearbeitet haben. Die Ergebnisse dieser Befragung werden in diesem Kapitel zusammengefasst.

3.1 Übliche Anwendungsfälle

Bei Hybris Projekten wird JavaScript nur in der Entwicklung der Shop-Oberfläche verwendet. Die Anwendung dessen beschränkt sich also auf den Browser und dient eher der optischen Komponente als der Logischen.

Eine Kern-Funktion des JavaScript Codes ist die Veränderung des DOM. Sehr häufig müssen Elemente aus dem DOM-Baum selektiert und mit Klassen oder CSS Stilen versehen werden um die Optik der Seite dynamisch zu verändern. Dies wird in der Regel mithilfe des Frameworks JQuery erreicht, welches sehr häufig in Projekten der dotSource verwendet wird.

```
var select = function() {
        $(“.selector“).addClass(“selected“);
        $(“.selected-paragraph“).show();
};
```

Abbildung 1: Code Beispiel zu DOM Manipulation mit JQuery

Ein Großteil dieser DOM-Manipulationen wird direkt vom Nutzer ausgelöst, indem er beispielsweise einen Knopf betätigt. Für dieses Verhalten müssen EventListener an die entsprechenden DOM-Elemente gebunden werden, welchen dann eine passende Funktion ausführen.

```
$(“.button“).on(“click“, function(_event) {
        $(“#element“).css(“color“, “red“);
});
```

Abbildung 2: Code Beispiel zu Events

An vielen Stellen muss die Seite zudem mit Inhalten aktualisiert werden, die zum Zeitpunkt des Seitenaufrufes noch nicht zur Verfügung stehen, bzw. von Entscheidungen des Nutzers abhängig sind. Hier werden sogenannte *Asychronous JavaScript and XML* (AJAX) Anfragen verwendet. Sobald das Ergebnis der Anfrage vorliegt, wird der entsprechende Code zur Handhabung des Antwortinhaltes ausgeführt.

3.2 Seltene Anwendungsfälle

Vereinzelt müssen auch komplexere Probleme durch JavaScript Code gelöst werden. Bei einem solchen Problem kann es sich zum Beispiel um einen Validator handeln, der die syntaktische und semantische Prüfung von Eingaben vornimmt, bevor diese an den Server gesendet werden. Ein solcher Validator erfordert Zugriff auf den Inhalt des Eingabefeldes, muss teilweise komplexe Logik ausführen und eine entsprechende Ausgabe liefern können.

Besonders auf Galerie- bzw. Listenseiten kommt es vor, dass unbestimmt viele Elemente nach einem Muster geordnet werden müssen. Hier kommt ein Organisator ins Spiel, der zur Laufzeit die vom Server erhaltenen Elemente beispielsweise dem vorhandenen Platz entsprechend anordnet. Organisatoren verwenden meist Schleifen bzw. Iteratoren, müssen ggf. einfache Objekte halten und das DOM stark anpassen.

Eine weitere Funktionalität, die mit komplexerem JavaScript implementiert werden muss, ist das sogenannte *Lazy Loading*. Es beschreibt ein Vorgehen, bei dem das Laden von initial nicht benötigten Inhalten zurückgehalten wird. Verwendet wird Lazy Loading in den meisten Fällen für größere Bilder, die beim Seitenaufruf noch nicht sichtbar sind. Lazy Loading verwendet

zum größten Teil asynchrone Callbacks und muss häufig das Nutzerverhalten in Form von Events auswerten können.

Diese Fälle sind die Üblichsten unter den komplexeren JavaScript Anwendungen. Es ist jedoch nicht auszuschließen, dass ein Projekt weitere Logik erfordert, die hier nicht abgedeckt ist. Generell ist es sinnvoller so viel Logik wie möglich serverseitig auszuführen, um das Gerät des Endnutzers weniger zu belasten.

3.3 Agile Arbeitsweise

In Projekten der dotSource wird agil gearbeitet. Agile Entwicklung stellt besondere Anforderungen an jede Programmarchitektur, selbstverständlich auch in Bezug auf JavaScript.

Die Grundsätze der Agilen Entwicklung sind im *Agile Manifesto* festgehalten. Für eine Programmarchitektur ist daraus besonders der Punkt 4 relevant: „Reagieren auf Änderung mehr als das Befolgen eines Plans“[1]. In der Praxis heißt das, dass der Code auch in kurzer Zeit teilweise drastisch angepasst werden muss.

Um den Entwicklungsprozess unter dieser Bedingung effizient zu halten, muss modular entwickelt werden. In diesem Kontext ist Modularität als die Aufteilung von Code in Sinnabschnitte mit möglichst wenigen Abhängigkeiten zu verstehen. Modularität profitiert auch von flacher Hierarchie, also wenigen Vererbungen.

Durch Modularität wird sichergestellt, dass bei Änderungen nur wenige Anpassungen vorgenommen werden müssen. So werden die Zeitaufwände für die einzelnen Änderungen gering gehalten. Diese lassen sich dann leichter vor dem Kunden rechtfertigen und entsprechend verkaufen.

Um Modularität im Code sicherzustellen, müssen die einzelnen Bestandteile konsequent strukturiert sein. Eine Architektur, wie sie im Laufe dieser Arbeit entworfen werden soll, unterstützt dabei solche Strukturen einzuhalten.

[1] [The01]

4 Gebräuchliche JavaScript Design Patterns

Architekturen sind wie Baukästen. Sie geben einen Rahmen vor, in dem sich frei bewegt werden kann. Dieser Rahmen ist jedoch komplexer als der Begriff vermuten lässt. Er bildet sich aus einer Vielzahl kleiner, festgelegter Vorgehensweisen, die auch als *Design Patterns* bekannt sind.

In diesem Kapitel wird eine Auswahl verbreiteter Design Patterns vorgestellt. Als *„Design Pattern“* (dt. *Entwurfsmuster*) wird ein bewährtes Lösungsschema für wiederkehrende Probleme in der Softwareentwicklung bezeichnet. Diese Patterns werden abstrakt formuliert und lassen sich in den meisten Sprachen adaptieren, so auch in JavaScript.

Ihren Ursprung haben Design Patterns in dem berühmten Buch „*Design Patterns. Elements of Reusable Object-Oriented Software“* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, auch bekannt als die *Gang of Four* (GoF).

4.1 Model-View-Controller

Das Model-View-Controller (MVC) Pattern dient der Verteilung von Zuständigkeiten auf separate Akteure bei der Entwicklung von Software mit einer Nutzeroberfläche. Durch diese Aufteilung können die Klassen einfacher wiederverwendet werden und bei Änderungen sind die betroffenen Stellen im Code leichter zu finden. Folgendes Diagramm beschreibt die Verhältnisse zwischen den Akteuren des MVC Patterns:

Abbildung 3: Strukturdiagramm des MVC Patterns

Das Model beschreibt zugehörigen darzustellenden Daten und muss deren aktuellen Zustand speichern können. Seine Felder müssen durch den Controller veränderbar sein, um unnötige Instanziierungen zu vermeiden.

Die View dient der Darstellung der Daten und registriert Änderungen im zugehörigen Zuständigkeitsbereich der Nutzeroberfläche und übergibt sie an den Controller. Bei der Instanziierung der View wird sie mit dem Model verknüpft und bezieht daraus ihre Daten. Die View selbst hat nicht die Autorität das Model zu verändern, kann aber den Controller anfragen dies zu tun.

Der Controller ist für die Steuerungslogik der Komponente zuständig. Er erstellt und verwaltet die View und regelt den Datenfluss von ihr zurück zum Model. Wenn in der View eine Aktion registriert wird, bekommt der Controller die Anweisung diese zu behandeln und anschließend die View entsprechend der Ergebnisse zu aktualisieren. Er kann nur auf Aktionen des Nutzers reagieren, wenn er über diese durch die View informiert wird.

4.2 Model-View-ViewModel

Das Model-View-ViewModel (MVVM) Pattern ist eine Alternative zum üblichen MVC Pattern. Das MVVM Pattern ersetzt den Controller durch das sogenannte ViewModel als zentrale Klasse und definiert ein neues Verhältnis zwischen View und ViewModel.

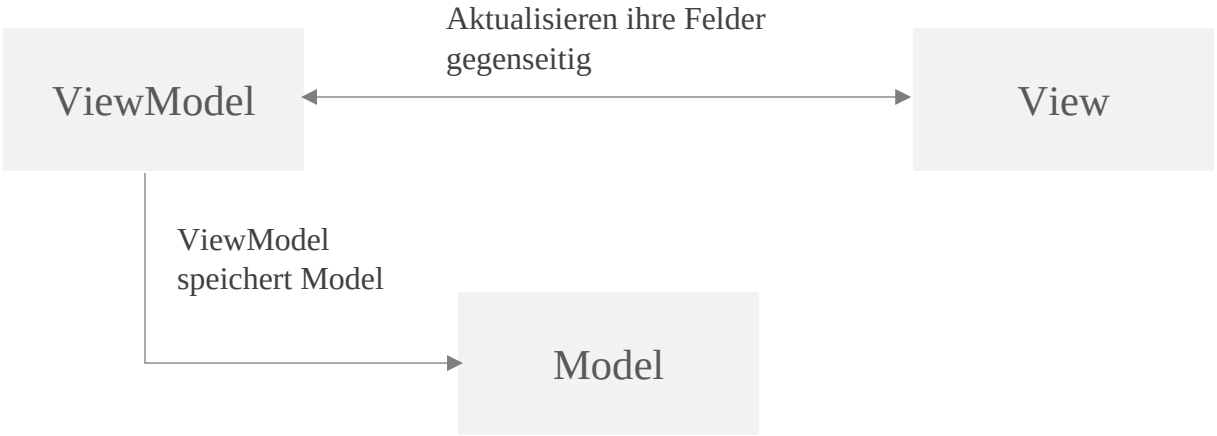

Abbildung 4: Strukturdiagramm des MVVM Patterns

Das Model selbst agiert genau wie in MVC als reiner Container für Daten. Im Gegensatz zum MVC Pattern wird es jedoch nicht im ViewModel referenziert.

Die View dient auch hier der Darstellung der Informationen auf der Nutzeroberfläche. Sie hat jedoch im Gegensatz zu MVC keinerlei Verbindung zum Model, sondern bezieht Informationen aus dem ViewModel. Änderungen werden von der View sofort an das ViewModel übermittelt.

Im ViewModel wird, wie im Controller in MVC, die Steuerungslogik ausgeführt. Das ViewModel hat jedoch eine eigene private Instanz des Models. Das eigentliche Model dient eher als Backup und wird, abgesehen von gelegentlichen Speichervorgängen, nicht aktiv genutzt. Stattdessen werden einzelne Änderungen sofort an die View übermittelt.

4.3 Model-View-Presenter

Als weitere Alternative zu MVC und MVVM bietet sich das Model-View-Presenter (MVP) Pattern. Auch hier liegt der wesentliche Unterschied in den Zuständigkeitsbereichen der einzelnen Klassen.

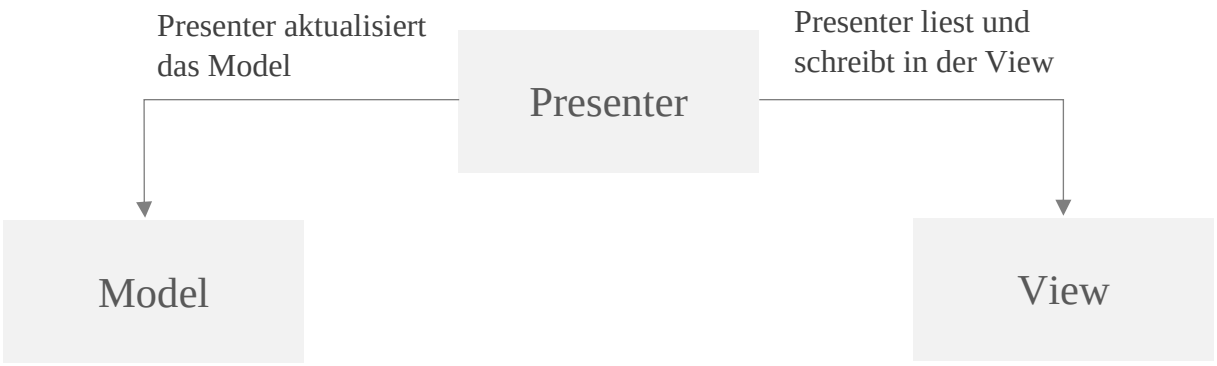

Abbildung 5: Strukturdiagramm des MVP Patterns

Das Model bleibt wie zuvor ein passiver Behälter für Daten. Die View hat jedoch, wie in MVVM, keine Verbindung zu Model.

Die View bleibt im Vergleich zu MVVM auch größtenteils unverändert, agiert jedoch nur noch passiv. Anstatt Änderungen direkt an den Presenter weiterzugeben, stellt die View für jedes Attribut Lese- und Schreibzugriff bereit.

Der Presenter übernimmt im MVP Pattern die respektive Position von Controller bzw. ViewModel. Der Presenter hat volle Autorität über die View und das Model. Im Gegensatz zum MVVM Pattern reagiert der Presenter nicht auf Änderungen, da er über diese nicht informiert

wird. Stattdessen werden bei Bedarf die aktuellen Werte der View gelesen und verarbeitet. Die geschieht meist als Reaktion auf ein Event.

4.4 Proxy

Das Proxy Pattern wurde ursprünglich von der GoF definiert. Das Kernkonzept ist, dass ressourcenlastige Klassen bzw. Funktionen hinter einer Proxy versteckt werden, die Entscheidungen darüber fällt wann Funktionen ausgeführt oder Objekte erstellt werden. Anstatt eine Aktion auszuführen, gibt die Proxy das Versprechen diese Aktion auszuführen, wenn diese tatsächlich nötig ist.

Eine typische Anwendung einer Proxy in JavaScript ist die Kombination von Anfragen. Dadurch wird Zeit in der Summe der Anfragen gespart, weil nur einmal eine Verbindung zum Server aufgebaut werden muss. Serverseitig werden auch Ressourcen gespart, da weniger Anfragen behandelt werden müssen. Der Nachteil ist, dass der Nutzer ggf. unnötig lang auf den benötigten Inhalt warten muss.

Eine weitere Anwendungsmöglichkeit ist das sogenannte *„Lazy Initialization“* (kz. *Lazy Init*, z. Dt. *Verspätete Initialisierung*)[2] . Die Idee ist, die Initialisierungsmethode einer Klasse hinter einer Proxy zu verstecken. Dadurch können unter Umständen sehr viele unnötige Variablen und EventListeners eingespart werden, bis sie tatsächlich benötigt werden.

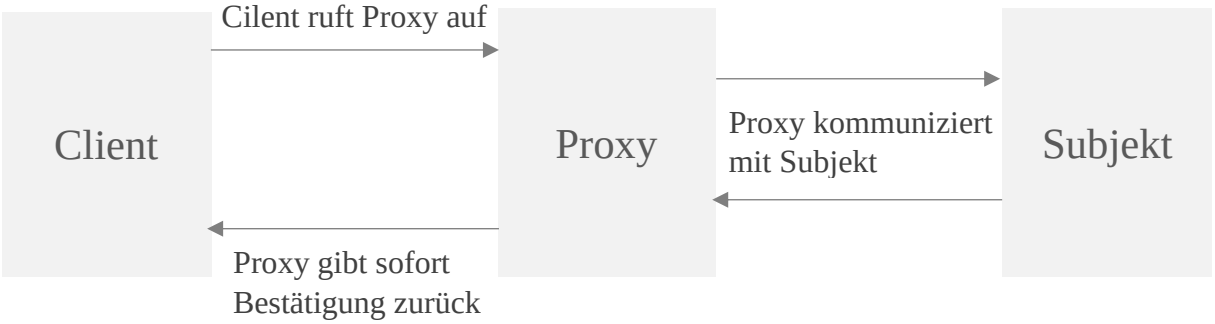

Abbildung 6: Schema des Proxy Patterns

[2] [Ste10], S. 160

4.5 Composite

Das Composite Pattern bietet eine Alternative zur strikten Vererbung. Als Composite wird eine Komponente bezeichnet, die selbst aus einer Vielzahl bekannter Komponenten gebildet wird. Ausschlaggebend ist hier, dass die Komponenten des Composites auch vom selben Typ sein können, wie der Composite selbst. So wird eine tiefe Verschachtelung der Komponenten möglich.

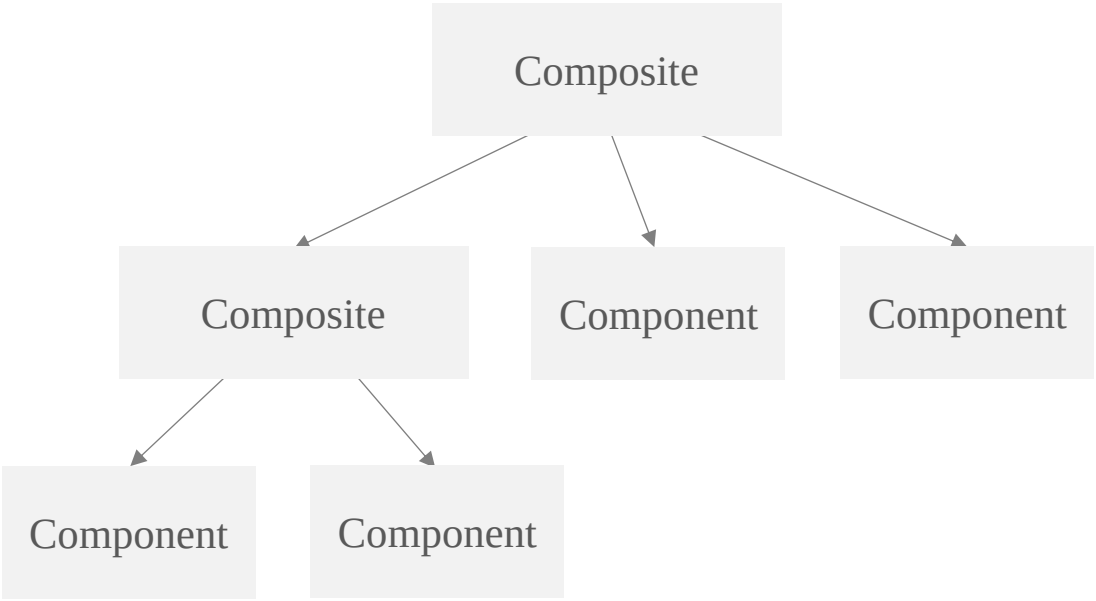

Abbildung 7: Beispielhafter Aufbau eines Composites

4.6 Template Method

Das Template Method Pattern stammt auch ursprünglich aus dem Werk der GoF. Es beschreibt die Praktik eine „Vorlage“ zu erstellen, auf Basis derer leicht unterschiedliche Objekte erzeugt werden können, indem Methoden und Attribute der Vorlage überschrieben werden.

Die Adaption dieses Patterns in JavaScript kann über den Prototypen erfolgen. Die Vorlage wird als eigenes Objekt erstellt und jedem der Kind-Objekte als Prototypen zugewiesen. Alle Attribute des Prototypen können direkt von dem Kind überschrieben werden und auf intakte Attribute des Prototypen kann direkt zugegriffen werden, als wären sie Teil des Kindes.

4.7 Observer

Auch das Observer Pattern stammt ursprünglich von der GoF. Es dient dazu Änderungen zentral zu überwachen und Reaktionen von dort auszulösen. Das Pattern definiert 2 Akteure: Den Observer, sowie den Subscriber. Das zu beobachtende Feld wird als Observable bezeichnet.

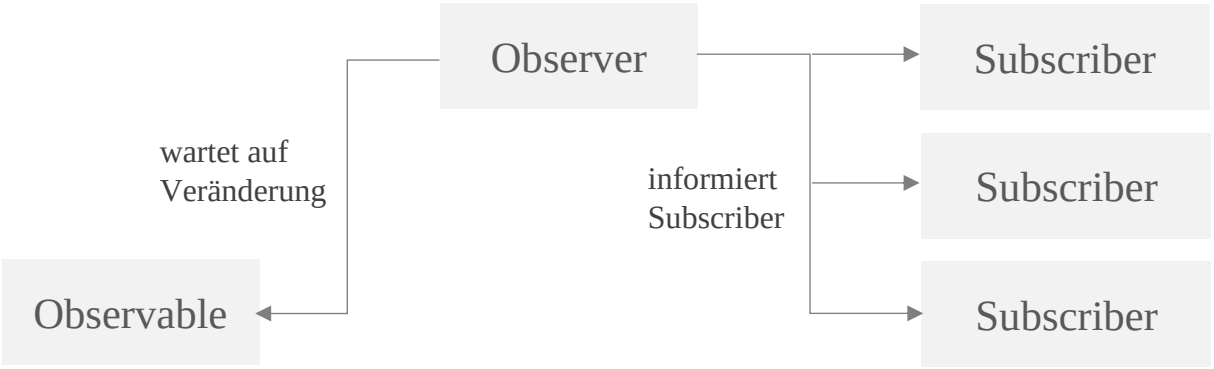

Abbildung 8: Schema des Observer Patterns

Subscriber sind üblicherweise Funktionen, die daran Interessiert sind, wann Änderungen vorgenommen werden. Ohne dieses Pattern stünde jeder Subscriber selbst in der Verantwortung aus Änderungen zu hören.

Der Observer ist eine externe Methode, die Subscriber über Veränderungen informiert. Um informiert zu werden, müssen sich die Subscriber beim Observer anmelden. Dieser kann dann im Falle einer Änderung durch seine Subscriber iterieren und sie ausführen.

Dieses Pattern wird sehr häufig in Verbindung mit MVVM verwendet, um die Änderungen in View und ViewModel besser überwachen zu können.

5 Auswertung

Um mithilfe der vorgestellten Patterns eine JavaScript-Architektur zu entwickeln, müssen diese im Kontext eines SAP-Commerce-Cloud Projektes bewertet werden. So kann entschieden werden, welche Patterns Teil der Architektur werden sollen. Die Bewertung erfolgt anhand der in Kapitel definierten Anforderungen, wobei zunächst auf die allgemeinen und danach auf die spezifischen Anforderungen eingegangen wird.

5.1 Allgemeine Anforderungen

5.1.1 Performanz und Größe des nötigen Codes

Das MVC Pattern benötigt an sich am wenigsten Code, da nicht wie in MVP oder MVVM Zugriffsmethoden für den Datenaustausch zwischen View und ViewModel bzw. Presenter vorausgesetzt werden. In MVVM müssen diese Zugriffsmethoden auf Seiten der View und des ViewModels existieren, im Gegensatz zu MVP, bei welchem diese nur in der View benötigt werden. So ist der nötige Code in MVVM noch um ein wesentliches größer als in MVP.

Das Observer Pattern kommt besonders in Verbindung mit MVVM zum Tragen. Durch einen Observer wird eine überwachte Ressource nur noch von einer Quelle abgerufen und nicht von einer Vielzahl Interessenten. Obwohl in der Praxis nur wenige Ressourcen von mehreren Interessenten betrachtet werden, ist es an sich nicht schädlich trotzdem einen Observer zu kreieren, da die Anfragen nur verschoben werden und sich keine großen Leistungseinbußen ergeben.

Die Größe einer Proxy ist definiert durch ihre Komplexität. Im Regelfall sollte eine einfache Methode mit Konditionsblock genügen. Es ist jedoch nicht außer Acht zu lassen, dass manche Proxys auf komplexere Logik und Events zurückgreifen müssen. Eine sinnvoll eingesetzte Proxy verzögert die Ausführung von intensivem Code, dementsprechend muss abgewägt werden, ob der Einsatz der Proxy tatsächlich vorteilhaft ist. Eine schlecht eingesetzte, also überflüssige, Proxy kann auch zu Einbußen in der Performanz führen.

Da es häufig vorkommt, dass einzelne Komponenten in mehreren Kontexten verwendet werden, ist es an diesen Stellen sinnvoll vom Composite Pattern Gebrauch zu machen. Von

einer Reduktion des nötigen Codes kann jedoch nicht immer gesprochen werden, da jede Sub-Komponente dennoch in voller Länge implementiert werden muss. Vielmehr bietet Composite eine angenehmere Möglichkeit mit Vererbung umzugehen.

In der Theorie kann durch das Template Pattern viel Code gespart werden, indem ähnliche Objekte ihre Methoden von demselben Template erben. In der Praxis fallen jedoch nur wenige Situationen an, in denen zwei Objekte verschiedener Funktion so ähnlich sind, dass der Einsatz eines Templates gerechtfertigt ist. Dennoch kann dieses Pattern, eingesetzt in den richtigen Situationen, die Ladezeit merklich reduzieren.

5.1.2 Wiederverwendbarkeit von Code

Um optimal von den MV* Patterns zu profitieren, sollten diese segmentiert auf verschiedene Komponenten der Nutzeroberfläche angewandt werden, wie z.B. ein bestimmtes Formular. Dieser Code kann jedoch, ohne die saubere Segmentierung zu sprengen, nicht in einem stark verschiedenen Formular wiederverwendet werden. Um den Code trotzdem wiederverwendbar zu machen, müssen die entsprechenden Methoden sehr generalisiert verfasst werden, was im Kontext einer bestimmten Komponente nicht sinnvoll ist. In dieser Situation ist es eher angebracht die Methode in einem generellen Formular-Kontext, oder einer Art Toolbox zu implementieren.

Dieses Problem lässt sich besonders sauber durch das Template Pattern lösen. Indem eine Formular-Vorlage erstellt wird, kann den bestimmten Formularen eine generelle Methode bereitgestellt werden, die diese bei Bedarf auch überschreiben können. Um Code entsprechend dieses Patterns auch über Sinnabschnitte hinweg wiederverwenden zu können, müsste eine Vorlage entworfen werden, die für jede Komponente gültig ist. Hierdurch werden die Hierarchiestufen innerhalb des Codes jedoch stark vervielfältigt, sodass schnelle Anpassungen an bestimmten Komponenten entweder zu einer Vielzahl unvorhergesehener Komplikationen an anderen Komponenten führen, oder durch den Einsatz unzähliger Überschreibungen der Sinn des Patterns verfehlt wird.

Das Pattern, welches mit Hierarchien hervorragend umgeht, ist Composite. Es ermöglicht die Definition einer Komponente außerhalb ihres ursprünglich vorgesehenen Kontexts. So kann die Komponente reibungslos an anderen Stellen eingebunden werden. Vorausgesetzt ist, dass auch

in diesen Fällen immer die exakt selbe Komponente benötigt wird. Composite bietet an sich keine Möglichkeit eine Komponente für ihre unterschiedlichen Verwendungen zu dynamisieren.

Indem die Überwachung an einem zentralen Ort verlegt wird und sich eine unbegrenzte Anzahl Interessenten für einen Aufruf anmelden können, bietet das Observer Pattern eine sehr saubere Code-Wiederverwendung. Da die Überwachung eines Feldes jedoch häufig nur sehr wenig Code erfordert, muss im Kontext abgewägt werden ob die Erstellung eines dedizierten Observers profitabel ist.

In der Regel werden Proxys für einen ganz bestimmten Zweck entworfen, und lassen sich nicht wiederverwenden. Sollte es jedoch vorkommen, dass mehrere Komponenten vereinheitlichbare Proxys haben, ist es mit diesem Pattern vereinbar und problemlos möglich eine erweiterte Proxy zu erstellen, die diese Komponenten gemeinsam behandelt.

5.2 Anwendungsfälle in Hybris Projekten

5.2.1 DOM Manipulierung, Events & AJAX

Die üblichen Fälle von JS in Hybris Projekten lassen sich einfach auf die Akteure der MV* Patterns aufteilen. DOM Manipulationen benötigen Referenzen auf die Elemente im DOM, auf die nur die View zugreifen sollte. Demnach steht die Ausführung der Manipulationen in der Verantwortlichkeit der View. Der Aufruf zur Ausführung stammt wiederrum vom Controller, Presenter oder ViewModel.

Auch zum Binden von nativen Events müssen Elemente direkt angesprochen werden. Da wieder nur die View Referenzen auf das DOM beinhalten sollte, fällt auch dies in die Verantwortlichkeit der View. Das Initiale Handling der Events wird auch von der View übernommen, allerdings nur soweit, dass dem Controller davon mitgeteilt wird. Von dort übernimmt der Controller, Presenter, oder das ViewModel und führt entsprechende Berechnungen aus, bzw. gibt der View Anweisungen.

Bei der Betrachtung von AJAX-Aufrufen sind die Patterns etwas mehr zu differenzieren. In MVC werden diese vom Controller ausgeführt und Ergebnisse werden direkt in das Model

geschrieben. In MVVM wird AJAX auch vom ViewModel ausgeführt, aber die Ergebnisse werden im internen Model des ViewModels festgehalten und erst zum gegebenen Zeitpunkt im eigentlichen Model gespeichert. Der Presenter in MVP aktualisiert primär die View mit den erhaltenen Daten, da diese keine Anbindung an das Model hat. Das Model kann der Presenter bei Bedarf aktualisieren.

Durch das Template Pattern können besonders einheitliche DOM-Manipulationsschritte und Event Handlers in einer Vorlage zusammengefasst werden. AJAX-Anfragen sind jedoch bei unterschiedlichen Komponenten immer verschieden und erfordern so auch bei jeder Implementierung theoretisch eine Überschreibung. Daher ist das Template Pattern für Ajax nicht sehr profitabel.

5.2.2 Komplexere Logik & künstliche Performance-Optimierung

Komplexere Logik, wird bei den MV* Patterns grundsätzlich im Controller, Presenter bzw. ViewModel ausgeführt. Als eine solche Anwendung führen Organizer, abgesehen von der Sortierungslogik, hauptsächlich DOM Manipulationen aus. Diese Funktionalität lässt sich besonders gut in MVC umsetzen, da die drastischen Änderungen des Markups meist einen vollen Austausch der View rechtfertigen. Organizer werden jedoch recht selten, etwa 1-2 mal pro Projekt, verwendet und sollten die Entscheidung nicht zu stark beeinflussen.

Ein Proxy ist an sich kein Lazy Loading, kann aber in manchen Implementationen eine solche Funktionalität enthalten. So werden Methodenaufrufe und die Speicherzuweisung innerhalb eines Kontextes, welcher zum Zeitpunkt keine Rolle spielt, zurückgehalten. Lazy Loading kann jedoch auch innerhalb der Komponente benötigt werden, also dort wo der Zuständigkeitsbereich des Proxys aufhört.

Validatoren hingegen sind nicht zwangsläufig im Controller, Presenter, oder ViewModel anzulegen. Es ist z.B. in MVC möglich die Validierung in der View durchzuführen und so nur valide Daten an den Controller zur Verarbeitung weiterzugeben. Auch MVP bietet eine aktive Variante, die diese Vorgehensweise erlaubt[3]. In MVVM hingegen macht dies wenig Sinn, da das ViewModel sofort die Inhalte der View zur Verfügung hat und die Validierung dort

[3] [Tim16], S. 160

theoretisch sogar in Echtzeit durchgeführt werden könnte. Auch Gründen der Performanz ist davon jedoch abzuraten.

5.3 Konkretisierung der Architektur

Im Vergleich mit den anderen MV* Patterns, bietet MVC am meisten Flexibilität und hat am wenigsten Overhead, da die vielen Zugriffsmethoden nicht benötigt werden. Demnach werden Komponenten auf die 3 Akteure: Model, View und Controller, aufgeteilt. Es kann auch vorkommen, dass eine Implementierung kein Model, bzw. keine View benötigt. Diese dann auszulassen ist akzeptabel.

Um eine gewisse Code-Wiederverwendung zu ermöglichen, sollen MVC Komplexe als Composites aufgebaut werden. So können auch Views/Controller/Models aus anderen Quellen reibungslos kombiniert werden. Dafür ist es wichtig die einzelnen Akteure getrennt zu behandeln, es sollte also für jedes Model, jede View und jeden Controller eine eigene Datei angelegt werden.

Das Template Pattern wird nur für Ausnahmefälle reserviert, da in den meisten Implementationen die Vorlagen-Methoden überschrieben werden müssten und das Risiko von Anpassungen an Basis-Vorlagen recht groß ist.

Da das MVVM Pattern nicht verwendet wird, bietet das Observer Pattern sehr wenig Nutzen. Mögliche Change-Events sind in der Praxis so selten, dass sich das generelle Einsetzen von Observern nicht lohnt.

Vor jedem Composite, nicht vor den Views oder Controllern, sollte ein Proxy sitzen, der bestimmt, ob die Instanziierung dieses Composites nötig ist. Selbst bei Komponenten die kaum Ressourcen kosten, sollte der Proxy dennoch bestehen und schlichtweg keine Prüfung durchführen. In diesem Sinne kann der Proxy wie eine Weiterleitung betrachtet werden.

6 Ausblicke

6.1 Umsetzbarkeit

Viele Projekte der dotSource, auch im Hybris Bereich, sind bereits so weit fortgeschritten, dass der eigene JavaScript Code tausende Zeilen umfasst. Der Aufwand, der also nötig wäre um diese Struktur in einem bestehenden Projekt umzusetzen ist kaum vor einem Kunden vertretbar. Es kann sich hierbei um Wochen, ggf. Monate handeln, da der umgeschriebene Code sich auch unerwartet verhalten kann und dadurch Probleme entstehen, die vor der Umsetzung noch nicht bestanden.

Es wäre möglich eine Struktur wie diese in einem neuen Projekt von Beginn durchzusetzen, dies ist jedoch etwas riskant bei eine Struktur, die in der Praxis noch nicht getestet wurde. So können z.B. wichtige Aspekte außer Acht gelassen, oder der Standard nicht ausgiebig genug definiert worden sein um alle Fälle abzudecken.

Sollte diese Architektur erfolgreich eingesetzt werden, muss sie auch eingehalten werden. Dadurch können Code-Review Schleifen sehr in die Länge gezogen werden, besonders wenn verschiedene Personen an dem Projekt nur leihweise arbeiten und mit der Architektur nicht bekannt sind. Durch die klaren Zuständigkeiten und vordefinierten Strukturen wird jedoch auch viel Zeit beim Entwurf einer Komponente gespart, was unter Umständen die verlängerte Zeit für den Code-Review ausgleichen könnte. Die Einführung einer Architektur kann also nur profitabel sein, wenn sie bereits zu Beginn eingesetzt wird.

6.2 ECMAScript 6

ECMAScript, entworfen von Ecma International, dient als Sprachstandard für JavaScript und verwandte Scripting Sprachen. Die 9. Edition von ECMAScript ist zum Verfassungszeitpunkt dieser Arbeit die neueste öffentliche Version dieses Standards und stammt vom Juni 2018.

Mit jeder Weiterentwicklung des Standards wird ECMAScript (ES) um besondere Syntax und neue Funktionalitäten erweitert. Als Meilenstein dieser Entwicklung gilt ECMAScript 6 (auch ES6), da in dieser Version sehr viele nützliche Neuerungen eingeführt wurden. Jede Version ist zwar abwärts kompatibel, aber nicht aufwärts. Aus diesem Grund bestehen häufig Probleme

bei der Kompatibilität von ES6 mit veralteten Browsern, insbesondere dem Internet Explorer, welcher trotz seines Alters noch eine nicht zu vernachlässigende Rolle spielt.

In ES6 wurden der Sprache besonders viele neue Syntax Formen hinzugefügt, die als sogenannter „Syntactic Sugar“ einzuordnen sind. Dabei handelt es sich um Operanden und Schlüsselwörter, die entweder neue Funktionen der Sprache bloß simulieren, oder bestehende in einer ansprechenderen Art verwendbar machen. Um diese Ausdrücke verwenden zu können, ohne die Kompatibilität mit alten Browsern zu verlieren, muss der neue Syntactic Sugar in altes JS umgeschrieben werden. Diese Aufgabe übernehmen sogenannte Transpiler, die als Präprozessor ES6 Code lesen und diesen in ES5-verständlichen Code übersetzen.

Eine solche Neuerung ist das Schlüsselwort *class*. JavaScript ist an sich eine klassenlose Sprache, jedoch können Klassen durch das prototypenbasierte Vererbungssystem von JavaScript zumindest angenähert werden. Hinter dem Schlüsselwort class steckt im Kern immer noch diese Vorgehensweise[4]. Es handelt sich hier also um *Syntactic Sugar*. Dennoch erleichtert die Verwendung das objektorientierte Arbeiten enorm, so auch die Verwendung der MV* Patterns, die in dieser Arbeit behandelt wurden.

Um nicht zu weit hinter die aktuellen Technologien zu fallen, ist es empfehlenswert ES6 zu adaptieren. Dabei unterstützen die Neuerungen in der Sprache dabei effizienter intuitiven Code zu schreiben. Dies setzt jedoch voraus, dass jeder Entwickler mit ES6 vertraut ist und die neue Syntax auch anwenden kann. Ohne diese Kompetenz geschriebener Code ist zwar in ES6 noch valide, macht aber die Einführung des Standards hinfällig. Weiterhin muss bei der Einführung auch ein Transpiler, wie Babel, in die Projekt Pipeline eingeführt werden um die Kompatibilität mit alten Browsern zu wahren.

Schlussendlich ist es Abwägungssache ob der Schritt gemacht wird. Hier muss der Kunde konsultiert werden, besonders da ggf. auch größere Aufwände entstehen wenn der bestehende Code mit den Neuerungen umgeschrieben werden soll.

[4] [Sim16], S. 135

7 Schlussteil

Eingesetzt werden soll eine Architektur basierend auf dem MVC Pattern. Die einzelnen Akteure sollen durch Composites kombiniert werden. Es soll Pauschal vor jedes Composite eine Proxy gehängt werden, damit sie bei Bedarf mit Konditionen versehen werden kann. Diese Proxy kann so die Zuweisung unnötigen Speichers unterbinden. Templates und Observer sollten nur in Spezialfällen verwendet werden.

Sollte sich dazu entschieden werden diese Architektur einzusetzen, wird für die Umstellung eine signifikante, aber vertretbare Zeit benötigt. Es zahlt sich im weiteren Verlauf eines Projektes aus eine feste Architektur als Leitfaden zu haben.

Um die Architektur klarer darzulegen, könnte weiterhin ein Dokument mit konkreten Beispielen angefertigt werden. Möglicherweise auch ein umfängliches Beispielprojekt, das bei Unklarheiten nach konkreten Beispielen durchsucht werden kann.

In Zukunft sollte es in Betracht gezogen werden, das verfügbare JavaScript Toolset durch die Neuerungen in ES6+ zu erweitern. Hierzu müssen ggf. Entwickler geschult werden und es sollte ein Transpiler eingebunden werden.

Literaturverzeichnis

[Sim16] Simpson, K.: „ES6 and Beyond“, O'Reilly Media. Sebastopol, CA, 2016

[Ste10] Stefanov, S.: „JavaScript Patterns“, O'Reilly. Sebastopol, Calif., 2010

[The01] The Agile Alliance: „Manifesto for Agile Software Development“, 2001. https://agilemanifesto.org/ Abruf: 2019.04.19

[Tim16] Timms, S.: „Mastering JavaScript Design Patterns“, Packt Publishing. Birmingham, UK, 2016