Bitte hier Informationen sammeln; Bevor es zu viel durcheinander gibt, sollte zuerst Jeder seinen eigenen Abschnitt (Seiten) in diesem Dokument benutzen, beim nächsten Meeting werden die Infos dann zusammengeführt bzw. schaut natürlich auch etwas was die anderen in ihrer Sektion schon eingefügt haben! :)

Gliederung:

1. MVC

2. MVP

3. MVVM

**Allgemein**

Die Bezeichnung MVC steht für die Aufteilung der Programmkomponenten in Model, View und Controller.

Die drei Komponenten sollen im folgenden Kurz beschrieben werden:

**View**

Hier geschieht die Darstellung der Daten, wobei die View die Komponente(n) der Datenhaltung nicht kennt. Es werden lediglich die Daten dargestellt und die Interkation nach außen vorgenommen.

**Model**

Das Model ist für die Datenhaltung zuständig. Die Form der Datenhaltung wird hierbei nicht vorgeben. Es kann zum einen durch einen simples Array erfolgen aber auch durch komplexere Datenstrukturen wie z. B. einer Datenbank.

**Controller**

Werden Änderungen durch die Interaktion durch den Benutzer in der View durchgeführt, werden diese durch den Controller ausgewertet und gegebenenfalls an das Model weiter gereicht. Im Gegensatz zur View und zum Model kennt der Controller die beiden anderen Elemente.

**Informationen von Buch: “Head First Design Patterns”**

Ab Seite 528 wird ein ganzer Abschnitt MVC gewidmet

Über den Sinn des Controllers wird geschrieben: “The controller does more than just “ send it to the model”, the controller is responsible for interpreting the input and manipulating the model based on that input. But your real question is probably “why can’t I just do that in the view code?”

Die Antwort auf diese Frage:

“You could; however, you don’t want to for two reasons: First, you’ll complicate your view code because it now has two responsibilities:managing the user interface and dealing with logic of how to control the model. Second, you’re tightly coupling your view to the model.”

zusammenfassend kann man sagen: durch den Controller, bleibt für die View das Model nur abstrakt, selbst wenn es eine direkte Schnittstelle zwischen Model und View gibt, bleibt diese durch richtigen Einsatz des Observer-Patterns auf einer abstrakten Ebene.

Controller und View implementiert das “Strategy-Pattern”.

Der Kernunterschied zwischen den Veränderungen in der View ist:

* beim Controller: Die View wird durch Benutzerinteraktionen modifiziert, jeder Controller bewirkt im Grunde ein anderes Verhalten einer View.

* beim Model: durch Änderung der Daten in der Datenhaltung

Sollte der Controller einen völlig anderen Zweck erfüllen als ursprünglich vom Interface vorgesehen kann das Adapter-Pattern eingesetzt werden.

(hier noch Beispiel einfügen)

Wobei die Adaption nur aus meiner Sicht eine unsaubere Lösung ist und zwangsweise einen Design-Fehler darstellt.

Das MVC-Beispiel finde ich persönlich nicht so toll, wie das von Johannes oder das vom WS12 in diesem Buch

Der Controller dominiert das Verhalten der Anwendung:

klassisches “strategy-pattern”, die View wird mit einer Strategie versehen, die der Controller bereit stellt. Der Controller hat sog. die “Reaktions-” Logik für die Präsentation in der View!

Im Grunde kann diese Logik jeder Zeit ausgetauscht werden und durch eine andere (anderer Controller) ersetzt werden.

in diesem Buch findet sich leider kein Beispiel für MVVM oder MVP :(

**Informationen von Buch: “Design Patterns: Elements of Reusable Object-Oriented Software”**

Über das eigentliche Pattern ist wenig zu finden, aber es wird beschrieben, das die Kernkomponenten aus den Observer, Composite und Strategy-Design-Pattern bestehen.

Diese drei Patterns werden im Buch genauer beschrieben:

**Das Composite-Pattern (Seite 163):**

Das Composite-Pattern wird in der heutigen Zeit mit hoher Wahrscheinlich nur selten “direkt anzutreffen sein, da meist bereits mit fertigen GUI-Frameworks gearbeitet wird.

Im Kern geht es darum, dass man zwischen komplexen GUI-Elementen und einzelnen Basiselementen aus Benutzersicht nicht unterschieden wird, obwohl intern sehr wohl ein Unterschied besteht.

Je nach Komponente besteht diese aus einem einfachen “GUI-Element” oder setzt sich aus mehreren mit einem Bestimmten Verhalten zusammen

**2. Was ist MVP?**

Quellen:

<http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>

<http://www.martinfowler.com/eaaDev/uiArchs.html#Model-view-presentermvp>

<http://www.martinfowler.com/eaaDev/SupervisingPresenter.html>

<http://www.martinfowler.com/eaaDev/PassiveScreen.html>

IEEE-Paper: An Architecture and Implement Model for Model-View-Presenter Pattern

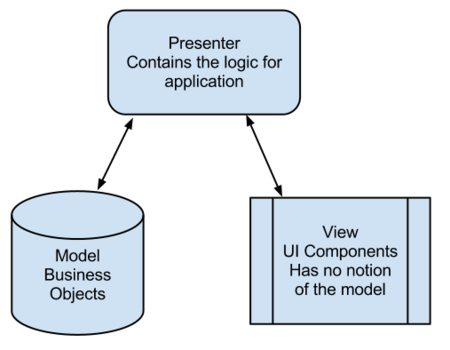
**2.1 Erklärung und Problemstellung**

**2.1.1 Generell**

MVP stellt eine Weiterentwicklung von MVC dar. Da die Komponenten in MVC nicht in jedem aktuellen Anwendungsgebiet eindeutig definiert werden können und der Übergang teilweise fließend erfolgt, ist eine Erneuerung/Verbesserung der Grundidee dieses Patterns notwendig gewesen.

Im Grundsatz ändert sich nur ein zentraler Aspekt im Vergleich zum klassichen MVC-:

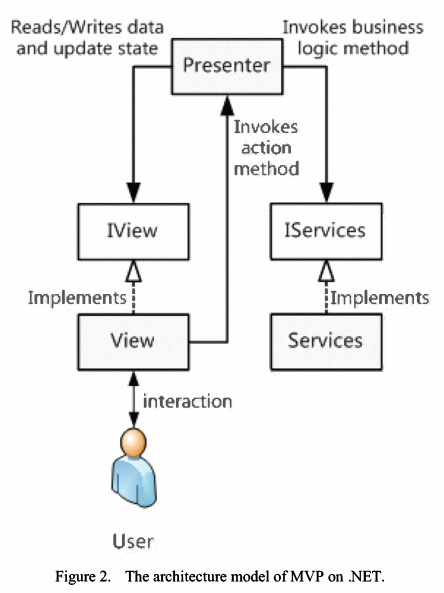
Der Presenter ist das einzige Bindeglied zwischen Model und View. Im Vergleich wird bei diesem Pattern strikt darauf geachtet, dass die View keine Logik bezüglich der Daten mehr enthält und hat lediglich die Funktion die Daten darzustellen. Die zentrale Steuerung der Ansicht erfolgt durch den Presenter, woraus resultiert, dass die View keinen Zugriff auf die Funktionen des Models hat.



Man kann sich View und Model auch als Schichten vorstellen, wobei der Presenter das Bindeglied dieser Schichten ist. Dieser arbeitet konkret mit den Schnittstellen der einzelnen Schichten und stellt somit eine Verbindung her.

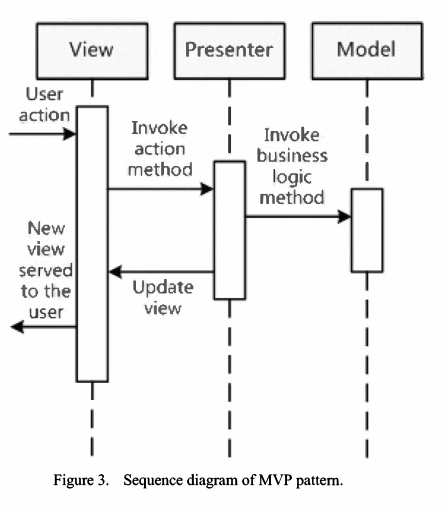
Da es keine direkte Verbindung zwischen View und Model mehr gibt, wird durch dieses weiterentwickelte Pattern die flexibilität weiter erhöht. Da kein Observer-Pattern mehr verwendet wird, ist die Aufgabenteilung im Vergleich zum klassischen MVC noch deutlicher und es passt sich an die gegebenheiten moderner GUI-Frameworks an.

Grundsätzlich ist es nicht schwer weitere Views, sowie Models einem MVP-Programm hinzuzufügen. Es müssen nur die Interfaces für die jeweiligen Rollen berücksichtigt werden.



Das Modell zeigt, wie eine MVP-Anwendung strukturiert sein kann, wobei jede View und Model von einem einheitlichen Interface abgeleitet werden und der Presenter, sowohl View als auch Model kennt.

Der Benutzer interagiert lediglich mit der View. Die Daten bzw. Programmlogik wird vollständig wie auch bei MVC durch das Model gekapselt.



Die Grafik zeigt den prinzipellen Ablauf einer Interaktion mit dem System durch einen Benutzer. Die Anweisung wird durch den Benutzer in der View erstellt. Der zuständige Presenter reagiert auf die Benutzer-Interaktion, wobei er entscheidet wie der Befehl zu interpretieren ist und fragt beispielsweise das Model nach Daten. Dieses übermittelt die Daten an den Presenter, wodurch sie in der GUI dargestellt werden.

Es ist möglich das eine Übersetzung der Daten vom Model zur View erfolgen muss und umgekehrt. Da sich aber beide nicht kennen und nicht kennen sollten, obliegt es dem Presenter diese Übersetzung vorzunehmen. Man darf sich den Presenter hierbei aber, besonders bei komplexeren Anwendungen, nicht als einfache Übersetzungsklasse vorstellen. Ein Presenter kann beispielsweise mehr als 100 Klassen enthalten. Die Darstellung, sowie Verarbeitung von komplexen Datenstrukturen kann sich bei der Adaption zwischen Model und View sehr schwierig erweisen. Dies kann je nach Anwendungsfall dazu führen, dass das Pattern um einige logische Instanzen erweitert werden muss. (hier noch ein Beispiel).

Das IEEE-Paper “An Architecture and Implement Model for Model-View-Presenter Pattern” zeigt eine strukturierte Vorgehensweise bei Implementierung eines MVP-Patterns. Hierdurch können die Interfaces klar spezifiziert werden:

1. Es wird für jedes GUI-Element ein Prototype entwickelt. Dies erleichtert es hinterher das GUI-Interface zu spezifizieren, da grob skizziert ist, welche Komponenten benötigt werden
2. Danach wird die Schnittstelle für die View erstellt. Dabei wird jeder Eingabe, Ausgabe spezifiziert
3. Als nächstes werden die Interfaces das Model und optional für den Presenter erstellt. Hierbei gilt folgendes zu beachten, das für jede Aktion die auf der GUI ausgeführt wird, ein Handle im Presenter erstellt wird.

Es ist zusätzlich noch möglich ein Interface für den Presenter zu erstellen, in welchem die Handler für die GUI-Elemente in Verbindung mit dem Model festgelegt werden. . Dies ist aber nicht zwingend notwendig (nur falls unterschieldicher Handler implementiert werden sollen).

**2.1.2 Speziell**

Zur Veranschaung soll folgendes Beispiel helfen:

Es soll ein Login-Manager für eine kleine Firma geschrieben werden. Jeder Benutzer kann sich über eine Benutzerschnittstelle am System anmelden. An einer Administrator-Schnittstelle können neue Benutzer angelegt und gelöscht werden. Zusätzlich soll eine Benutzerliste mit allen derzeit vorhanden Benutzern angezeigt werden.

**2.2 Lösung**

Für die Lösung der Aufgabenstellung gibt es grundsätzlich sehr viele Lösungswege bzw. Ansätze. Die Verwendung eines MVP-Patterns kann sich aber besonders unter den Aspekten Erweiterbarkeit und Modularisierung als sehr nützlich erweisen

Zuerst werden die einzelnen Schnittstellen für unsere Anwendung definiert.

1. Die zwei Benutzerschnittstellen werden erstellt
2. Das Interface für die Benutzerschnittstelle wird spezifiziert
3. Es muss ein Model geschaffen werden, das unsere Benutzerdaten speichert
4. Der Presenter muss die interne Benutzer-Logik mit der GUI verknüpfen.

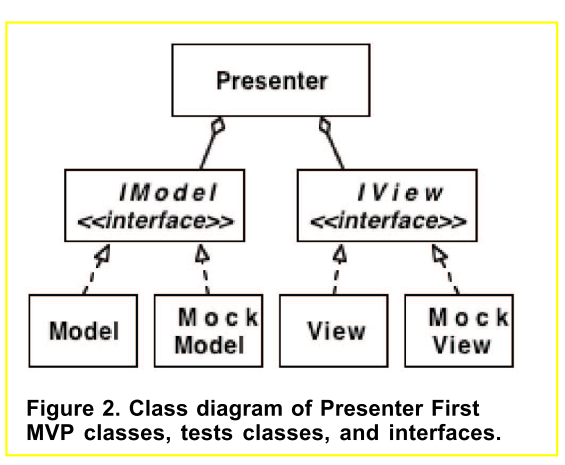
**3 Model-View-Presenter: Presenter First**

**3.1 Erklärung und Problemstellung**

**3.1.1 Generell**

Neben der im vorherigen Kapitel beschriebenen MVP-Implementierung, möchte ich noch eine weitere Methode vorstellen. Das Paper IEEE “Presenter First: Organizing Complex GUI Applications for Test-Driven Development”, zeigt eine Erweiterung/Verbesserung des Patterns, welches besonders bei größeren Anwendung sich als sehr nützlich erweisen kann.

Im Folgenden werde ich dieses modifizierte MVP-Pattern kurz erklären und beschrieben welche Vorteile hierdurch entstehen können.



Grundsätzlich unterscheidet sich das Klassendiagramm vom vorherigem Kapitel nur in einem entscheidend Punkt. Die Kohäsion zwischen den einzelnen Interfaces von Model und View sind stärker an den Presenter gebunden. Der zentrale Gedanke hierbei ist, dass nun nur noch der Presenter seine “Partner” kennt und deshalb alle anderen Komponenten kein Presenter-Objekt mitführen müssen.

Dies erleichtert auch das Testen des Presenters mit sog. Mock-Objekten. Diese sind Platzhalter für echte Objekte, wodurch die einzelnen Komponenten unabhängig voneinander getestet werden können.

**3.1.2 Speziell**

Die Aufgabenstellung bleibt identisch zum Login-Manager in 2.1.2. Doch ist es nun unser Ziel ein paar Dinge mit Hilfe der Presenter-First-Methodik zu verbessern. Hier wäre zum einen das Zusammenspiel der einzelnen Referenzen zu nennen.

… (hier Bild von main-Methode einfügen)…

Es ist relativ umständlich die einzelnen Referenzen zu kopieren. Besser wäre es wenn wir dies leicht verständlich in einer Zeile lösen könnten. Zitat, IEEE-Paper: „We first tried instantiating and wiring together all of the application's MVP triads from the main function. This soon became brittle, tedious, and un-testable”.

**Perty Kommentare:**

MVC is doof, MVVM soll viel toller sein!

Quellen:

<http://www.codeproject.com/Articles/228214/Understanding-Basics-of-UI-Design-Pattern-MVC-MVP>