MVC

**M**odel **V**iew **C**ontroller

Auch bekannt unter **MVVM** (**M**odel **V**iew **V**iew**M**odel) oder **MVP** (**M**odel **V**iew **P**resenter)

Dieses Designpattern trennt Business Logik sauber von der Präsentationslogik. Durch diese Trennung können Änderungen leicht vorgenommen werden.

# Ursprung/Motivation von MVC:

In den Anfangsjahren der Informatik wurde nur sogenannter **Spaghetti-Code** produziert, d.h. es erfolgte keine Strukturierung in Methoden, sondern es wurde nur mit Schleifen und Sprünge gearbeitet.

Zusätzlich waren dabei Ausgabe, Eingabe und Verarbeitung völlig ineinander verwoben. Als Abhilfe führte man zuerst die strukturierte Programmierung (1970) und schließlich die objektorientierte Programmierung (1980) ein.

Viele nutzten diese neue Möglichkeit anfangs nur, um Spaghetti-Code in mehrere Segmente aufzuspalten. Doch mit der Zeit setzte es sich durch, dass die Strukturierung auch nach Ausgabe, Eingabe und Verarbeitung vorgenommen wurde.

Zu dieser Zeit war die GUI-Programmierung noch nicht erfunden und die textorientierten Programme konnten eine Interaktion nur entgegennehmen, wenn eine Methode darauf (blockierend) gewartet hat.

Diese Technik wurde ersetzt durch eine Methode, die erst durch ein Ereignis aufgerufen wird und die Verarbeitung dieses Ereignisses übernimmt.

Zusätzlich wird bei der ereignisgesteuerten GUI-Programmierung eine Differenzierung in Model, View und Controller vorgenommen, wobei Model der Verarbeitung, View der Ausgabe und Controller der Eingabe entspricht.

# Ziele des MVC Pattern:

trennt Business und Präsentations Logik voneinander:

Diese Teilung erfolgt durch drei Komponenten:

* Model, View, Controller

Ziele:

* Es können später leichter Änderungen an bestimmten Teilen vorgenommen werden ohne dass das komplette Programm geändert werden muss
* Ebenfalls können im Nachhinein noch Erweiterungen bei einzelnen Komponenten leicht hinzugefügt werden

# Vorteile vom MVC:

Wenn das MVC Design Pattern bei der Implementierung richtig angewendet worden ist, hat mein ein gutes strukturiertes Programm, wo auch im Nachhinein noch Änderungen übernommen werden können.

Durch diese Strukturierung ist es ebenfalls möglich, dass man für jede einzelne Komponente verschieden Testfälle schreibt.

Ein weiterer Vorteil ist, dass während der Entwicklung parallel programmiert werden kann, da zum Beispiel der UI Entwickler an der View entwickeln kann, währenddessen ein anderer Programmierer parallel am Model programmiert. Das erspart einem Viel Zeit in der Entwicklung, wenn gleichzeitig mehrere Developer arbeiten können.

# Nachteile vom MVC:

Ein wesentlicher Nachteil ist bei diesem Pattern, dass man wesentlich mehr Aufwand bei der Planung und auch später bei der Implementierung hat.

Ein weiterer Nachteil ist, dass bei kleineren Projekten dieses Pattern oftmals nicht passend und notwendig ist.

# MVC UML:

A close up of a logo

Description automatically generated

## Model

* keine visuelle Klasse
* **zuständig für die Datenhaltung**
  + z.B. Business Logik
  + Berechnungen
  + Zugriffe auf Datenbanken
* Benachrichtigungen mittels Events

Das Model verwendet das **Observer Pattern**, welches 1-N Beziehung zwischen einem Subject und den Observer zu implementiert.

Das heißt wenn sich das Subject ändert werden **alle Observer benachrichtigt**, dass sich etwas geändert hat. Das Subject enthält eine List von Observer und kann Objekte mit hinzufügen oder löschen. Der Observer enthält ein update Methode, die vom Subject aufgerufen wird, wenn sich etwas ändert.

Auf das MVC Pattern angewendet ist das **Model das Subject** und die **Views** übernehmen die Rolle des **Observers**.

## View:

* ist ein visuelles Element
* zuständig für die Struktur und Darstellung
  + Datum richtig formatieren
  + Selbe Daten verschieden Darstellen (Kreisdiagramm, Balkendiagramm, …)
* referenziert den Controller
* führt Zustandsabfragen an dem Model durch um an die Daten zu gelangen

## Controller:

* keine visuelle Klasse
* enthält Instanz von Model
* koordiniert Interaktionen von der View mit dem Model
  + ruft z.B. die jeweiligen Methoden von dem Model auf

Bei der Beziehung von Controller und View verwendet man das **Strategy Design Pattern**:

Die View verwendet den Controller um bestimmte Responsearten zu implementieren. Jedoch kann der Controller zur Laufzeit geändert werden damit die View anders auf Eingaben reagiert.

Somit kann man sich die View wie ein Context vorstellen und der Controller ist ein Strategy Objekt.

# Grafisches Verwaltungssystem für ein Autohaus:

Das Back-End ist in Java implementiert und bietet eine REST Schnittstellen für die CRUD Operationen an. Dabei können Kunden, Arbeiter, Autos, Städte und Rechnungen gelesen, hinzugefügt, geändert oder gelöscht werden.

Das Front-End ist in .Net WPF implementiert und verwendet dabei die üblichen Elemente und Interfaces dafür:

* Pages welche die View beim MVC repräsentieren
  + Die Views enthalten die jeweiligen Controller im **DataContext**
  + Verwenden **DataBinding** an **Properties** vom Model
  + Die Buttons **binden** an die **Commands** im Controller
* Controller
  + welche eine Instanz von den jeweiligen Models enthalten
  + Objekte von einer Klasse, welche das ICommand Interface von .Net implementieren
    - Diese Commands rufen dann die jeweilige Methode im Model auf
* Model
  + Enthalten ObservableCollection mit den jeweiligen Klassen (welche INotifyPropertyChanged implementieren)
  + Sowie die jeweiligen Methoden für Berechnungen und für die CRUD Operationen
* Data Access Layer
  + Ist ein Singelton und enthält einen httpClient
    - Greift auf die Rest Schnittstelle vom Back End zu

Wenn sich etwas im Model ändert, wird ein neues **PropertyChangedEventArgs** Event ausgelöst und somit werden alle Views benachrichtigt, dass sich etwas geändert hat.

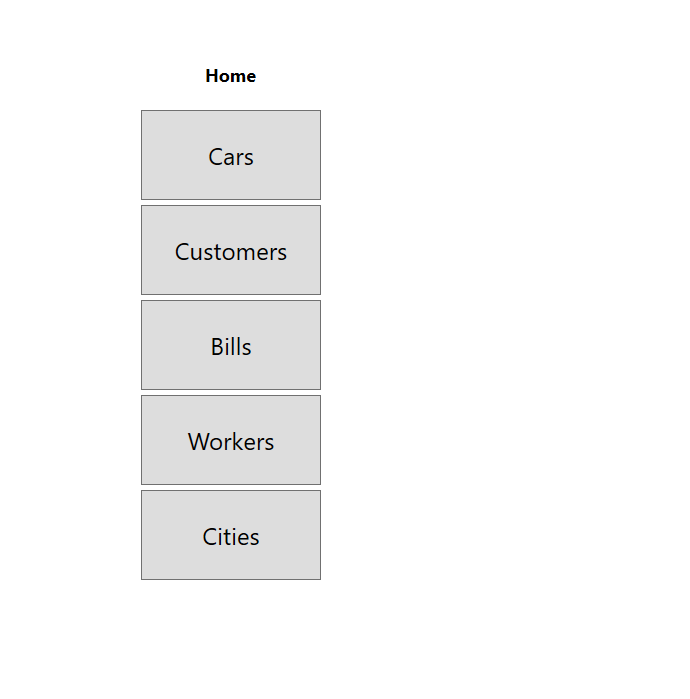
Durch die **Commands** auf den Buttons, werden dabei Events ausgelöst und die definierte Methode vom Model wird vom Controller aufgerufen.

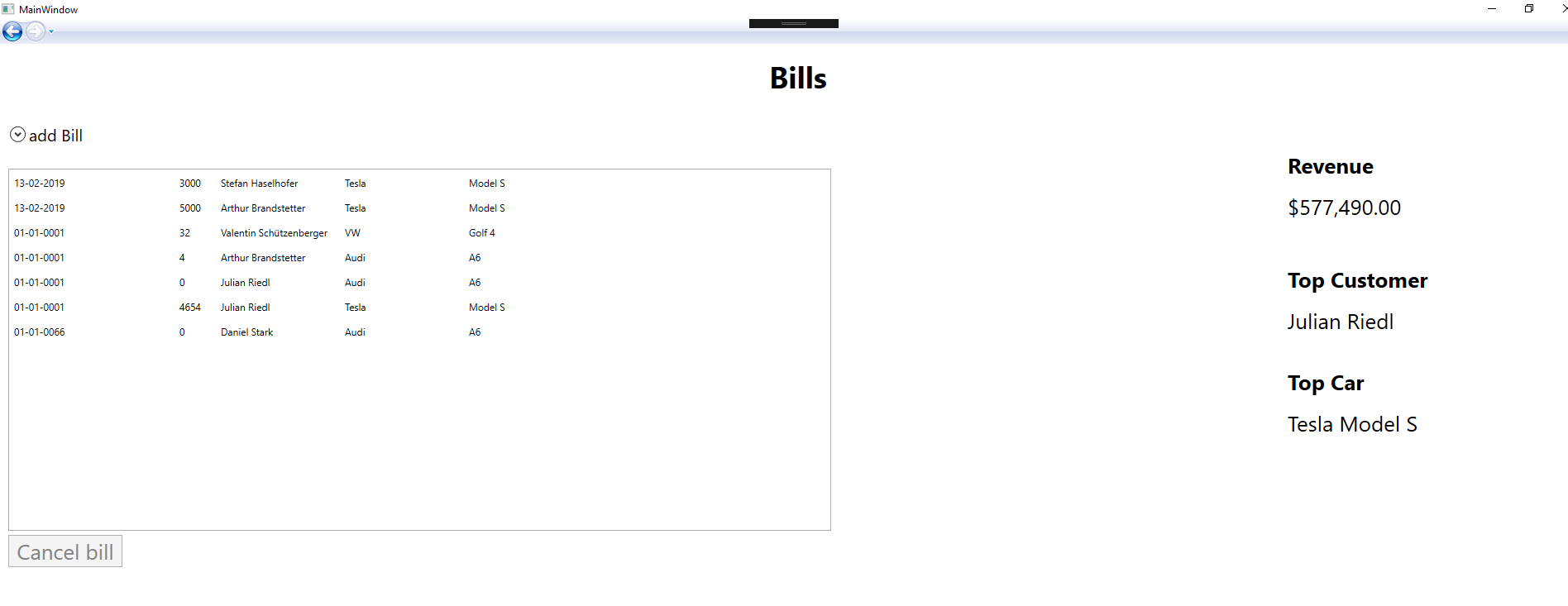
Wichtige **Interfaces** oder **Klassen** verwendet für MVC in WPF:

* **INotifyPropertyChanged** für Änderungen bei Properties
* **ObservableCollection<T>** für Änderungen bei Collections
* **ICommand** für Commands bei Buttons

Hilfreiche **Bibliotheken**:

* **JSONConvert**
* **PRISM**, vereinfacht das MVC Pattern in WPF





**Quellen:**

<https://de.wikipedia.org/wiki/Model_View_Controller>

<https://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod>

<http://www.fim.uni-linz.ac.at/diplomarbeiten/Diplomarbeit_stoiber/Dietmar_Stoiber_-_Model-View-Controller_Paradigma.pdf>

<https://martinfowler.com/eaaDev/uiArchs.html>

<http://prismlibrary.github.io/docs/wpf/legacy/Implementing-MVVM.html>