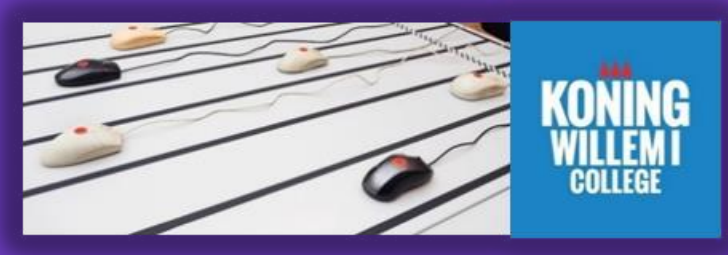
Basis standalone

Realiseren

hoofdstuk

6

Model View Controller + Select



Algemene informatie

Onderwerp	Model View Controller
Leerdoel(en)	1. De student benoemt wat een Design Pattern is 2. De student benoemt de 3 kerncomponenten van Model View Controller 3. De student programmeert een Model 4. De student programmeert een View 5. De student programmeert een Controller 6. De student koppelt deze 3 componenten aan elkaar om een werkende applicatie te krijgen.
Vereiste voorkennis	1. Alle voorgaande readers uit Thema 7 dienen bestudeert te zijn, specifiek de reader betreffende OOP.
Kwalificatiedossier	☐ B1-K1-W1: Plant werkzaamheden en bewaakt de voortgang ☐ B1-K1-W2: Ontwerpt software ☒ B1-K1-W3: Realiseert (onderdelen van) software ☐ B1-K1-W4: Test software ☐ B1-K1-W5: Doet verbetervoorstellen voor de software ☐ B1-K2-W1: Voert overleg ☐ B1-K2-W2: Presenteert het opgeleverde werk ☐ B1-K2-W3: Reflecteert op het werk

Inhoudsopgave

Introductie

Het is ontzettend belangrijk om een applicatiecode te schrijven, die makkelijk *leesbaar*, *goed onderhoudbaar* en *makkelijk* uit te breiden is, ook door andere ontwikkelaars. Het is dus belangrijk om je code vanaf moment één goed gestructureerd, gegroepeerd en via één gedachtenpatroon te schrijven. Dit doen we door gebruik te maken van een programmeer filosofie/ architectuur genaamd **Model View Controller** (afgekort MVC). MVC zullen we in de gehele Realiseren thema 7 en 8 reeks nog gaan gebruiken en de kans is groot dat je MVC gebruikt om je examen in te doen.

MVC is een voorbeeld van een zogeheten Architectural Design Pattern.

Inhoud

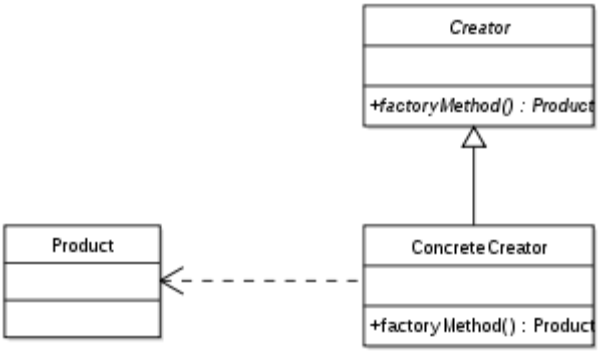
1. Introductie Design Patterns

Goede software bouw je niet door te beginnen met programmeren, zonder structuur, zonder plan en zonder vooraf besproken opzet. Goede software is altijd vanuit een bepaalde opzet of structuur opgezet. Je moet als goede programmeur dus in staat zijn om vanuit een plan of concept kunnen programmeren. In deze hoek kun je de **Design Patterns** vinden.

Een **Design Pattern** is dus een manier van programmeren om vanuit een vast plan je code te schrijven. Dit doen levert gestructureerde en goed onderhoudbare code op. Er zijn vele verschillende Design Patterns beschikbaar, waarvan sommige door elkaar heen gebruikt kunnen worden, die allerlei verschillende programmeer filosofieën aanhangen. Er is een grote kans dat je op een eventuele (technische) vervolgstudie je dieper op de verschillende Design Patterns ingaat.

Laten we naar een paar voorbeelden kijken (er zijn er nog veel meer):

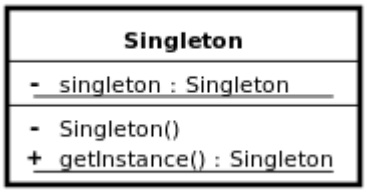
Factory Design Pattern



Een simpel voorbeeld is het **Factory** Design Pattern. Via deze pattern scheid je code die een nieuw **object** maakt en het object zelf van elkaar. Je creëert als het ware een fabriekje die nieuwe objecten kan produceren (uit bijvoorbeeld een database). Dit gebeurt dus door middel van meerdere **classes** te schrijven. Bekijk de code maar eens die in de Wikipedia beschreven staat.

📖 Lees meer over Factory https://nl.wikipedia.org/wiki/Factory_(ontwerppatroon)

Singleton Design Pattern



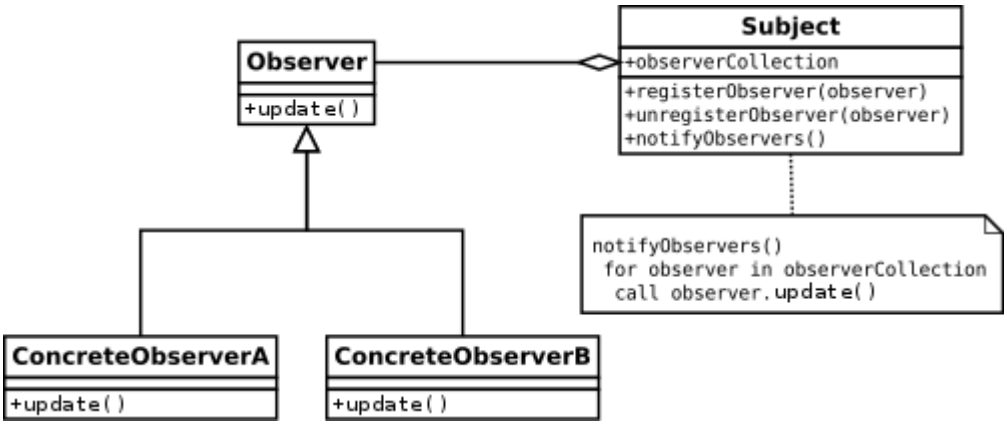
Een van de slechtste, vieste en/of smerigste Design Patterns is de **Singleton**. Dit design pattern zorgt ervoor dat er (door de gehele applicatie) maar één specifiek object van een bepaalde class gemaakt kan worden. Waar dan ook in de applicatie je deze class aanroept, je zult altijd hetzelfde object terugkrijgen. Dit kan handig zijn als een bepaalde programmeur enorm loopt te klungelen met het correct doorgeven van objecten. Het gebruik van een Singleton geeft aan dat de programmeur in kwestie er weinig kaas van gegeten heeft.

Echter moet ik bekennen dat ook ikzelf (ik blijf daarom ook liever anoniem) wel eens zo'n Singleton heb MOETEN gebruiken, aangezien er absoluut geen andere oplossing meer te bedenken was…

📖 Lees meer over Singleton https://nl.wikipedia.org/wiki/Singleton_(ontwerppatroon)

Observer Design Pattern

Ik wil jullie voorstellen aan een wat ingewikkeldere Design Pattern genaamd de **Observer** (ook wel **Observerable**) genoemd. Deze is momenteel erg populair in de MVVM (ook een Architectual Design Patter, net zoals MVC) wereld. MVVM staat voor **Model / View / ViewModel** en is een Design Pattern wat erg in opkomst is.



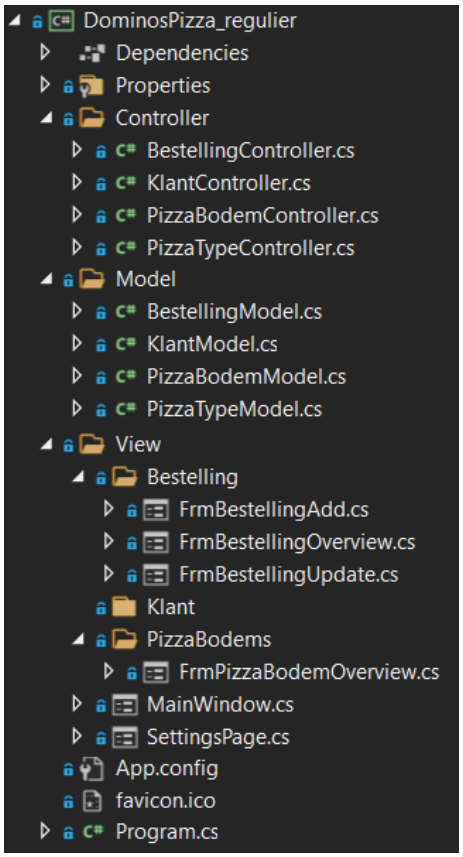
Via een Observer kun je ervoor zorgen dat wanneer een bepaald object of **Property** van waarde veranderd er een soort "update" (*notifyObservers)* melding door de applicatie heen verstuurd wordt. Zo kunnen alle geïnteresseerden stukken code dus automatisch deze update ontvangen en de nieuwe waarde van dat object of property verwerken.

Wellicht dat we deze Pattern nog eens gaan tegenkomen tijdens de opleiding. Schrik dan niet, uiteindelijk is het een lief en makkelijk Design Pattern 😊.

📖 Lees meer over Observer https://en.wikipedia.org/wiki/Observer_pattern

2. Introductie Model View Controller

In software engineering gebruiken we het **Model View Controller** Design Pattern (Officieel moet je het een Architectual Design Pattern noemen) als een bouwstijl waarin enkele verschillende lagen of "afdelingen" van onze code van elkaar gaan scheidden en in lagen plaatsen. Hierdoor scheidden we bijvoorbeeld de **presentatie-laag** (de Forms) van de **Controller-laag** (Logica en SQL-queries). Daarnaast maken we nog een 3e laag, waarin we onze **Modellen** plaatsen (de classes).



De termen MVC-Pattern en MVC-architectuur gebruiken we bewust door elkaar heen, omdat dit in de beschikbare documentatie ook gebeurt.

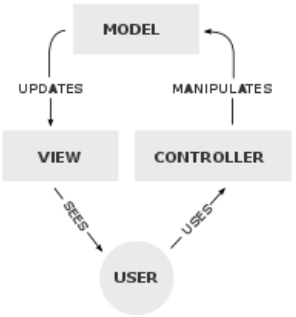
De MVC-architectuur zoals wij hem gaan, maken zien er als volgt uit:

1. **View** (Forms)
2. **Model** (Classes)
3. **Controller** (Logica en SQL-queries)

Je ziet hier (link in het plaatje) dus een duidelijke scheiding van de Forms, Logica en Classes. Door deze scheiding is het veel makkelijker om een wijziging te maken in een Class (Model laag), zonder dat je je Forms (UI laag) meteen ook hoeft te veranderen. Ook zou een wijziging in je database (Controller) niet betekenen dat je je Classes (Model) of Forms (View) direct hoeft aan te passen. Daarnaast maak je code herbruikbaar. Zo is bijvoorbeeld dezelfde Controller in elk formulier te gebruiken, hierdoor schrijf je je SQL queries maar op één plek (Controller) en niet in elk formulier los nogmaals.

Voordelen van MVC:

1. Dubbele code voorkomen
2. Leesbaarheid code verbeterd
3. Je collega snapt jouw code direct
4. Scheiding van verantwoordelijkheden (iedere laag doet slechts één ding)
5. Gestructureerd werken, hierdoor blijft je applicatie goed onderhoudbaar



1Schematische tekening MVC

📖 Bekijk onderstaande video voor meer achtergrond informatie over MVC:
https://www.youtube.com/watch?v=pCvZtjoRq1I

3. Hoe werkt een Model

In de laag **Model** plaatsen we alle modellen (in de vorm van **Classes**) van onze data die we hebben kunnen vinden via de bekende modeleertechnieken. Op het Koning Willem 1 College gebruiken we de modeleertechniek genaamd "CogNIAM". In het bedrijfsleven zul je waarschijnlijk andere modeleertechnieken gebruiken, maar het einddoel is altijd hetzelfde, namelijk het kunnen beschrijven van een model. Vanuit dit model maak je uiteindelijk weer **Classes** in programmacode.

Voorbeeld classediagram

In de map Model komen dus al onze classes te staan. In ons geval gaan we de onderstaande klasse diagram nabouwen:

2 Classes en hun properties

Voorbeeldcode

Hoe werk je dit uit in C# code? Zie onderstaand voorbeeld.

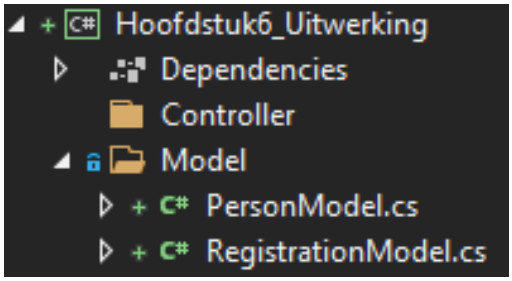
PersonModel.cs:

```
public class PersonModel
{
    // Properties declareren
    public int PersonID { get; set; }
    public string PersonName { get; set; }
    public int PersonAge { get; set; }

    // Constructor
    public PersonModel()
    {

    }
}
```

RegistrationModel.cs:

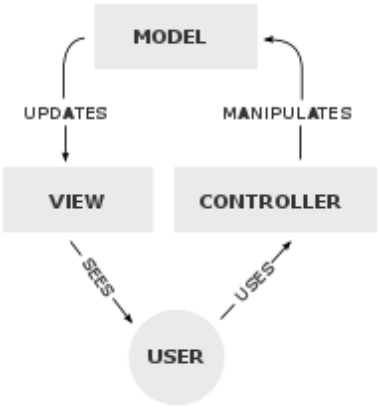
```
public class RegistrationModel
{
    // Properties declareren
    public int PersonID { get; set; }
    public string RegistrationCode { get; set; }
    public DateTime RegistrationDate { get; set; }

    // Constructor
    public RegistrationModel()
    {

    }
}
```

Zodat je dus op volgende structuur uitkomt:



- Je ziet hier hoe je in een MVC applicatie de Model classes opbouwt. Je plaatst ze in een map genaamd **Model**.

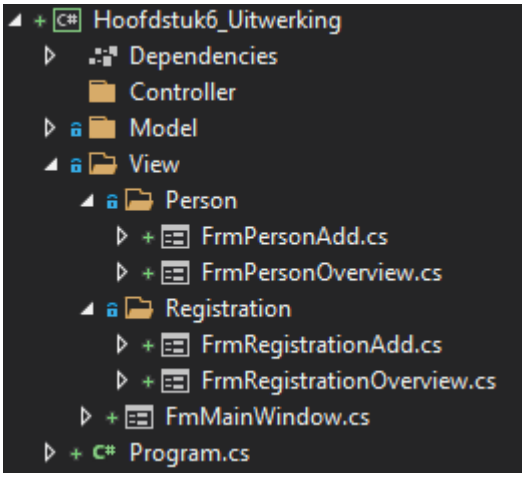
4. Hoe werkt een View



In de View-laag plaatsen we alle GUI elementen die binnen onze applicatie interactie hebben met de gebruiker. Het idee achter de View-laag is dat ze ALLEEN verantwoordelijk is voor de gebruikersinteractie en dus niets "weet" van de databases o.i.d. De View is dus alleen verantwoordelijke voor de interactie met de gebruiker en mag dus niet bijvoorbeeld direct een database aanspreken of kritische beslissingen maken (deze behoren tot de **Controller**).

In ons geval maken we de UI via **Windows Forms**, maar je kunt je voorstellen dat in een ander type applicatie hier .HTML of .XAML-bestanden zullen zitten. Binnen de programmeertaal Java is bijvoorbeeld Swing heel populair.

Voorbeeld:



Hier links zie je de indeling die we binnen onze Solution gemaakt hebben. Binnen de map "View" hebben we per **Model** (*Person* en *Registration*) een aparte map gemaakt, waarbinnen de Forms behorende tot dit Model geplaatst zijn.

Het is denkbaar, dat je hier nog een hele hoop extra Forms uiteindelijk bij gaat plaatsen zodat je een mooi gevulde **View** gaat opbouwen.

5. Hoe werkt een Controller

De **Controller-laag** is de nog de meest complexe laag van de MVC-architectuur. De Controller is namelijk verantwoordelijk voor de zogeheten logica (de beslissingsstructuur in je applicatie) en de toegang tot de data. Een Controller beheert dus niet alleen de toegang tot de database, maar mag ook specifieke beslissingen die een applicatie soms dient te nemen (logica noemen we dit). Je kunt hierbij denken aan, of een bepaalde afspraaksregistratie (bijvoorbeeld bij een tandarts) wel gemaakt mag worden.

In de basis bestaat een Controller altijd uit de methodes **Create(), Read(), Update() en Delete()**. Deze zogeheten **CRUD**-acties zijn de 4 handelingen waarmee je kunt interacteren met een stuk data (database, bestand, API, etc). Daaromheen ga je een Controller uitbouwen, om de logica van de applicatie in te verbouwen.

Belangrijk om te weten is dat de **Create()**, **Update()** en **Delete()** methodes een **Integer** waarde returnen. Deze Integer geeft aan hoeveel **Records** (in de database) zijn bewerkt (zogeheten **Rows Affected**). Via de Rows Affected waarde kun je bepalen of ene bepaalde databaseactie geslaagd is (of dus niet). Zodra je een Rows Affected waarde van 0 terugkrijgt, weet je dat de database actie niet gelukt is.

Rows Affected binnen SQL Management Studio

Binnen de SQL Management Studio krijg je bij iedere INSERT / UPDATE / DELETE (dus niet bij een SELECT) de Rows Affected waarde ook al terug. Zie onderstaand voorbeeld.

```
DELETE FROM Bestelling WHERE BestelNr = 63;
```

```
Messages

(1 row affected)

Completion time: 2022-03-18T12:56:58.0722987+01:00
```

De Rows Affected waarde binnen SQL Management Studio

Voorbeeldcode

Bekijk in onderstaande voorbeeldcode goed naar de **Methods**. Welke parameters hebben ze en welke datatype returnen de 4 CRUD methodes?

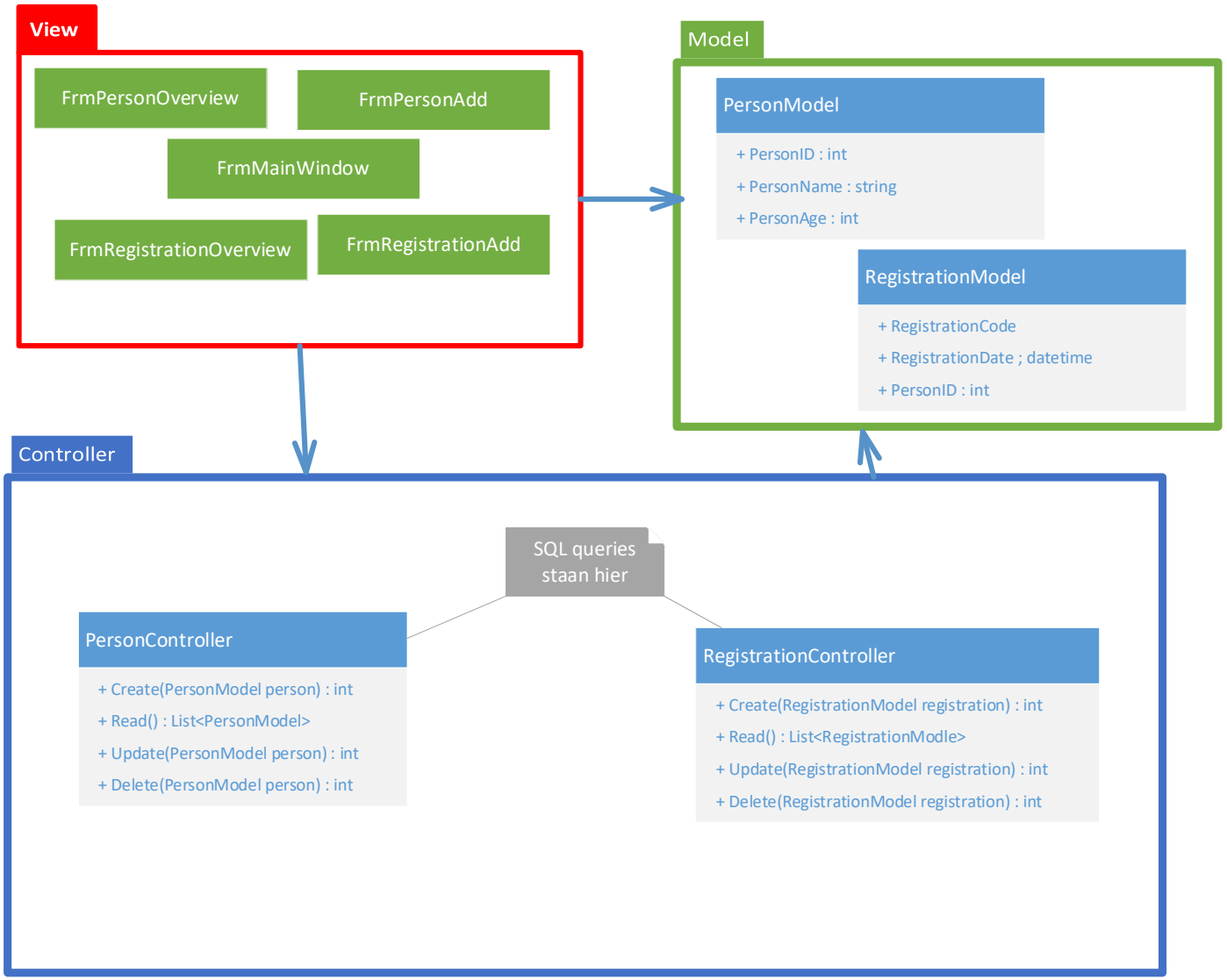
RegistrationController.cs

```
class RegistrationController
{
    public int Create(RegistrationModel registration)
    {
        // Code hier
    }

    public List<RegistrationModel> Read()
    {
        // Code hier
```

```
        }

        public int Update(RegistrationModel registration)
        {
            // Code hier
        }

        public int Delete(RegistrationModel registration)
        {
            // Code hier
        }
    }
```

6. Totaaloverzicht MVC

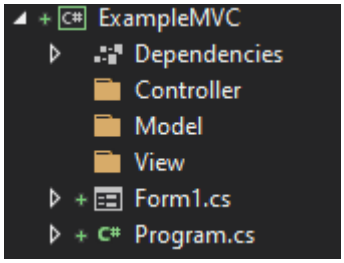
Hieronder zie je een Class Diagram, waarin de MVC-architectuur is uitgewerkt met slechts 2 modellen (*Person* en *Registration*). Uiteraard groeit MVC mee, zodra er meer modellen gevraagd worden.



7. En nu jij: Programmeren!

Projectopstart maken

1. Maak een nieuw Windows Forms project aan genaamd "ExampleMVC".
2. Maak binnen deze solution de mappen Model, View en Controller aan.



Plottwist: De rest van de uitleg ga je in de lessen krijgen 😊

Je kunt nu **oefening 6.2** maken, waarin je een MVC opzet gaat maken!