HO
GENT

Drielagenmodel

Ludwig Stroobant

Inhoudstafel

1. Introductie

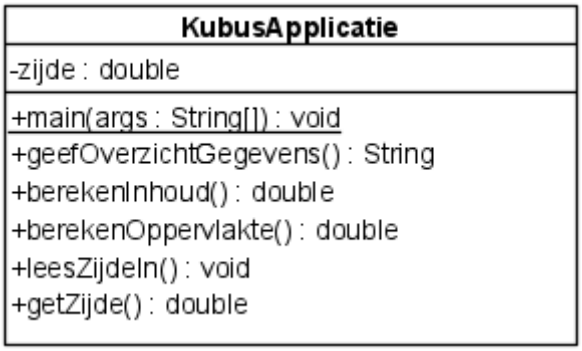
Met alles wat we over programmeren geleerd hebben, kunnen we al een "goede" applicatie schrijven. Maar wat is nu een goede applicatie?

Deze moet:

- aan de verwachtingen voldoen
- herbruikbare onderdelen bevatten
- uitbreidbaar en onderhoudbaar zijn
- performant en stabiel / bugvrij zijn
- gebruiksvriendelijk en veilig zijn

Maar... dat loopt vaak mis! Stel dat we bijvoorbeeld een eenvoudige applicatie maken om kubusvormige doosjes te kunnen maken voor een klant die in de cadeautjestijd snel wil kunnen berekenen hoeveel materiaal hij nodig heeft om de gewenste doos te maken (~ oppervlakte berekenen) en ook wil kunnen zeggen hoe groot de doos dan zal worden (~ volume berekenen).

Ons eerste idee leggen we vast in een DCD:



Bedenk even welke verantwoordelijkheden we terugvinden in deze ene klasse:

- Het opstarten van de applicatie: de `main` methode
- Het UI gedeelte: de communicatie met de klant - het overzicht van de gegevens, het inlezen van de zijden...
- Het business gedeelte: we werken met een kubus, en kunnen hierop berekeningen doen. De kubus heeft mogelijks ook restricties: een zijde kan niet negatief zijn.
- Wat nog niet verwerkt is maar mogelijk in de toekomst wel gevraagd kan worden: het opslaan van een kubus om hem later terug in te laden.

We bedenken ons dat we in deze consoleapplicatie zowel in- en uitvoer als berekeningen aan het doen zijn, maar daar zal onze klant toch niks van merken, zeker?

Alleen... het idee van een consoleapplicatie bevalt onze klant niet! Hij wil een GUI (graphical user interface). We moeten dus herbeginnen of op zijn minst delen van onze ene klasse herschrijven!

Delen herschrijven betekent een risico om bugs te introduceren → alles dient opnieuw getest te worden.

Hadden we van de Kubus een aparte abstractie gemaakt en deze beschreven in een aparte klasse, dan zouden we deze kunnen hergebruiken zonder ze te wijzigen!

Voor een goed object georiënteerd ontwerp moet elke klasse eigen duidelijk afgebakende verantwoordelijkheden krijgen. Dit vloeit voort uit het ontwerp principe: "Single Responsibility". Dat is blijkbaar een beetje misgegaan bij bovenstaand ontwerp, want de verantwoordelijkheden voor in- en uitvoer en verwerking zitten allemaal in dezelfde klasse! Op deze manier gaan we spaghetticode schrijven!

In dit hoofdstuk gaan we dieper in op een model met meerdere lagen en hoe je dit kunt toepassen in je eigen projecten. We zullen stap voor stap uitleggen hoe je een applicatie opbouwt volgens dit model, zodat je een goed gestructureerde, onderhoudbare en uitbreidbare codebase krijgt.

Het drielagen model is een architectuurpatroon dat helpt om de verantwoordelijkheden binnen een applicatie duidelijk te scheiden. Door de applicatie op te delen in drie lagen – de presentatielaag, de domeinlaag en de persistentielaag – zorgen we ervoor dat elke laag slechts één specifieke taak heeft. Dit maakt de code niet alleen overzichtelijker, maar ook gemakkelijker te onderhouden en uit te breiden.

Het idee om je projecten op te bouwen in meerdere lagen is o.a. gebaseerd op het principe van "Single responsibility". Wij gaan werken met een drielagen model, maar in de praktijk zijn we niet gebonden tot slechts drie lagen. Van belang is dat het principe van single responsibility gekend is en het nut ervan wordt ingezien. Hier zullen we het toepassen bij een model met meerdere lagen.

2. Principe van "Single Responsibility"

Het principe van "Single Responsibility" is heel eenvoudig. Het betekent dat elk onderdeel in je programma verantwoordelijk moet zijn voor slechts één ding. Je kan dit bekijken per methode, klasse, module … Met andere woorden, een stukje code, een klasse of een laag in je software moet slechts één taak hebben en mag niet overmatig belast worden met meerdere verantwoordelijkheden.

Het Single Responsibility Principle (SRP) is een van de vijf SOLID-principes in objectgeoriënteerd programmeren en stelt dat een klasse slechts één reden tot verandering mag hebben. Dit betekent dat een klasse slechts één verantwoordelijkheid moet hebben en zich moet richten op één specifieke taak.

Stel je voor dat je een kok bent in een keuken. Als je het principe van single responsibility zou toepassen op keukenhulpmiddelen, zou een mes bedoeld moeten zijn om te snijden en niet om ook een deksel los te wrikken. Zo heeft elk gereedschap zijn eigen specifieke taak. Willen we de manier van snijden wijzigen, dan dienen we het mes aan te passen en hoeven we ons geen zorgen te maken dat we per ongeluk ook iets anders wijzigen (aangezien we het mes enkel gebruiken om te

snijden)

Voordelen

- Als je een klasse in de toekomst zou moeten wijzigen, dient er slechts één reden te zijn om dat te doen.
- Dit voorkomt dat een klasse te veel taken krijgt ("God classes" of "Spaghetti code").
- Het verhoogt de leesbaarheid, testbaarheid en onderhoudbaarheid van je code.

Voorbeeld

In softwareontwikkeling betekent het dat als je een klasse maakt, bijvoorbeeld een `Boek` klasse, deze alleen mag zorgen voor boek gerelateerde zaken, zoals het bijhouden van de titel, auteur en het aantal pagina's en de restricties hierop. Een boek moet een titel en auteur hebben en het aantal pagina's mag niet negatief zijn. De `Boek` klasse zou niet ook verantwoordelijk moeten zijn voor het weergeven van die gegevens op een scherm of voor het opslaan van die gegevens in een database; daar zouden andere klassen voor moeten zijn.

Door principe van "Single responsibility" te volgen, wordt je code meestal eenvoudiger te begrijpen, gemakkelijker te onderhouden en flexibeler om aan te passen omdat elke klasse slechts één duidelijke verantwoordelijkheid heeft.

3. Het drielagen model

We kunnen het principe van "Single Responsibility" ook toepassen op een hoger niveau dan enkel op klassen. Het kan ook op een model met meerdere lagen worden toegepast door ervoor te zorgen dat elke laag specifieke verantwoordelijkheden heeft en zich concentreert op één type taak.

In C# mag je een "laag" bekijken als een aparte namespace waar klassen met hetzelfde hogere doel worden samengebracht. Meer zelfs: we gaan die lagen ook in aparte bibliotheken onderbrengen met als doel deze bibliotheken te isoleren qua verantwoordelijkheid: vanuit de ene laag kan je niet zomaar aan alles uit de andere laag. Dit noemen we het principe 'Loose coupling'.

3.1. Principe van "Loose Coupling"

Loose coupling (losse koppeling) is een ontwerpprincipe waarbij componenten in een systeem zo onafhankelijk mogelijk van elkaar worden gemaakt. Dit betekent dat wijzigingen in één component minimale impact hebben op andere componenten. In het drielagenmodel (presentatielaag, domeinlaag en datalaag) wordt dit als volgt toegepast:

Tussen presentatielaag en domeinlaag

- De presentatielaag weet niet hoe de businesslogica precies werkt
- Communicatie verloopt via duidelijk gedefinieerde interfaces
- De presentatielaag roept alleen methodes aan zonder te weten hoe deze intern werken
- Wijzigingen in de GUI hebben geen impact op de onderliggende logica

Tussen domeinlaag en datalaag

- De businesslogica is niet direct afhankelijk van de specifieke database implementatie
- Er worden abstracties gebruikt zoals repositories of data access objects (DAO's)
- De domeinlaag werkt met modellen/entities zonder te weten hoe deze worden opgeslagen
- Bij wijziging van database technologie hoeft alleen de datalaag aangepast te worden

Voordelen van deze losse koppeling

- Betere onderhoudbaarheid doordat lagen onafhankelijk kunnen worden aangepast
- Eenvoudiger testen omdat lagen geïsoleerd kunnen worden getest
- Flexibiliteit bij technische wijzigingen in één van de lagen
- Herbruikbaarheid van componenten in andere contexten

Een praktisch voorbeeld

Een webshop applicatie heeft een productenoverzicht. De presentatielaag toont alleen de producten en roept een methode getProducts() aan. De domeinlaag haalt via een ProductRepository de data op, zonder te weten of deze uit een SQL database, NoSQL database of webservice komt. Als de database later wordt vervangen, werkt de applicatie nog steeds zonder aanpassingen in de andere lagen.

Merk ook hoe 'INKAPSELING' hierdoor wordt toegepast. Inkapseling is één van de vier pijlers van OO.

In deze cursus focussen we op een drielagen model: de presentatielaag, de domeinlaag en de persistentielaag (= data laag). Dit wil niet zeggen dat elk project enkel over deze drie lagen zal beschikken: bedoeling is dat je het concept en nut van een meerlagig model leert kennen en toepassen!

Laten we elk van de lagen bekijken in het kader van het "Single responsibility" principe:

3.2. Presentatielaag

Verantwoordelijkheid

Interactie met de gebruiker.

Voorbeeld

Toont gegevens aan de gebruiker, neemt invoer van de gebruiker aan.

Toepassing van Single Responsibility

De presentatielaag moet niet verantwoordelijk zijn voor het rechtstreeks ophalen van gegevens uit de database of het uitvoeren van complexe bedrijfslogica. De verantwoordelijkheid van deze laag is alleen gericht op het presenteren van informatie aan de gebruiker en het ontvangen van gebruikersinput.

Verboden

Deze laag mag geen domeinregels of andere business logica implementeren en mag ook niet met

de persistentielaag communiceren!

3.3. Domeinlaag

Verantwoordelijkheid

Kernlogica van de applicatie.

Voorbeeld

Bevat bedrijfsregels, entiteiten en services.

Toepassing van Single Responsibility

De domeinlaag moet niet bezig zijn met het direct manipuleren van de database of het presenteren van informatie aan de gebruiker. De verantwoordelijkheid van deze laag is alleen de logica bevatten die specifiek is voor het domein van de applicatie, zoals het valideren van gegevens, toepassen van bedrijfsregels en het coördineren van acties.

Verboden

Vanuit deze laag mag niet met de gebruiker gecommuniceerd worden. Deze laag mag dus niets op het scherm plaatsen! De verantwoordelijkheid van deze laag is focus op de business logica, deze laag wil dus ook niet op de hoogte zijn hoe zaken worden gepersisteerd. De domeinlaag zal dus ook nooit rechtstreeks met bv. een database communiceren.

3.4. Persistentielaag

Verantwoordelijkheid

Gegevensopslag en toegang tot de database.

Voorbeeld

Verantwoordelijk voor het ophalen en opslaan van gegevens in de database.

Toepassing van Single Responsibility

De persistentielaag moet niet betrokken zijn bij gebruikersinterface-gerelateerde taken of complexe bedrijfslogica. De verantwoordelijkheid van deze laag is zich concentreren op het effectief communiceren met de database en het omzetten van gegevens tussen het domein en de database.

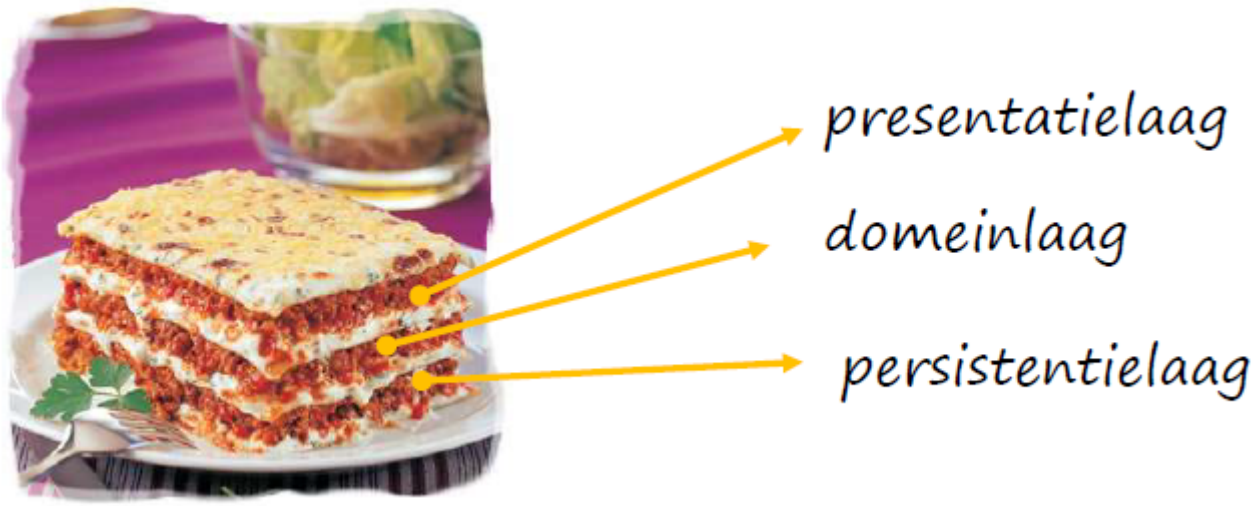
Door ervoor te zorgen dat elke laag zich richt op zijn specifieke verantwoordelijkheden, maak je de codebase modulair en gemakkelijker te begrijpen, onderhouden en uitbreiden. Eventuele wijzigingen in een specifieke laag hebben minimale invloed op de andere lagen, waardoor het systeem flexibeler wordt.

Verboden

Deze laag mag nooit met de gebruiker communiceren noch business logica implementeren.

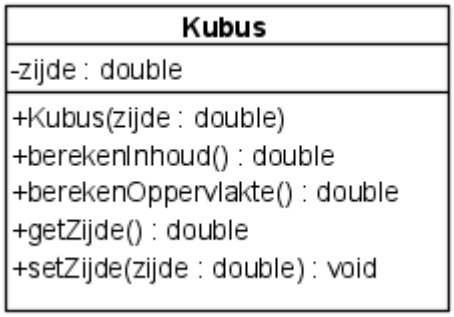
3.5. Lasagne structuur

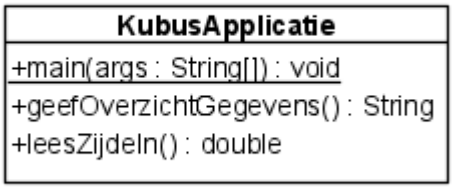
In plaats van spaghetti-code krijgen we op die manier lasagne, een gerecht dat in laagjes wordt bereid:



- De **presentatielaag** voor klassen die instaan voor de voorstelling van gegevens (in- en uitvoer)
- De **domeinlaag** voor klassen die instaan voor de verwerking van gegevens (logica)
- De **persistentielaag** voor klassen die instaan voor de opslag van gegevens (databank-functionaliteit)

Elke laag wordt voorgesteld door een (sub)namespace in het project. Tot nog toe hebben we altijd met 2 lagen gewerkt aangezien we nog geen gebruik hebben gemaakt van een databank (of andere permanente opslagmogelijkheid). We zouden dus onze code uit de KubusApplicatie kunnen herschikken over een domeinklasse en een applicatieklasse die in de presentatielaag thuishoort.





Door deze opdeling in 2 klassen, hebben we het volgende bereikt:

- **GEEN in- en uitvoer** in de domeinklasse!
- **GEEN logica** in de applicatie!
- Consoleapplicatie kan makkelijk vervangen worden door een GUI of webapplicatie, **zonder dat het domein moet gewijzigd worden**

Als we nu nog willen bijhouden welke dozen er al gemaakt zijn, kunnen we aan deze applicatie ook een database koppelen. In de persistentielaag komt dan een (of meer) mapperklasse(n) die de

vertaalslag maken van klassen en objecten naar tabellen en records.

4. Scheiding der lagen

Om de verantwoordelijkheden van elke laag te garanderen en spaghetti-code nog meer te vermijden gaan we een strikte scheiding toepassen tussen de verschillende lagen (= 'loosely coupled').

Om de scheiding der lagen te garanderen passen we INKAPSELING toe:

- De presentatielaag heeft een specifiek toegangspunt tot de domeinlaag om info op te vragen of door te geven. Dit toegangspunt voorzien is een verantwoordelijkheid van het domein en wordt typisch voorzien door een klasse: de `DomeinController` klasse (zie ook het "GRASP controller" principe bij Ontwerp).
- De domeinlaag heeft een specifiek toegangspunt tot de persistentielaag om info op te vragen of door te geven. De CRUD operaties nodig vanuit de domeinlaag worden in een contract vastgelegd (= een interface) en via "Dependency Injection" wordt de implementatie van zo een interface geïnjecteerd in het domein.

De verantwoordelijkheid van de `DomeinController` klasse is gericht op de interactie met de presentatielaag:

- het delegeren van taken vanuit de presentatielaag binnen het domein
- het vertalen van domein informatie naar de presentatielaag (meestal in de vorm van een 'Data Transfer Object' (DTO))
- het bijhouden van één of meer centrale domein objecten (bv. de aangemelde spelers van een spel en het spel zelf)

Let ervoor op om binnen de `DomeinController` geen business logica te implementeren!

De scheiding der lagen wordt gegarandeerd door het toepassen van INKAPSELING:

- De presentatielaag mag de interne werking van de domeinlaag niet kennen: Het is verboden om vanuit de domeinlaag referenties naar domeinobjecen door te geven naar de presentatielaag. De presentatielaag mag de types uit het domein nooit kennen!
- Door het gebruik van een interface zal ook de domeinlaag nooit de interne implementatie van de persistentielaag kennen → dit is net wat we willen bereiken!

4.1. Data Transfer Objects (DTO's)

Een Data Transfer Object (DTO) verwijst naar een ontwerppatroon dat wordt gebruikt om gegevens te encapsuleren en over te brengen tussen verschillende lagen van een applicatie of tussen applicaties.

Het belangrijkste doel van een DTO is om een gestandaardiseerde manier te bieden om gegevens te representeren, onafhankelijk van de onderliggende architectuur, en om de overdracht van gegevens tussen lagen of systemen te vergemakkelijken.

4.1.1. Kenmerken van een DTO

Encapsulatie van Gegevens

Een DTO bevat velden (properties) die de gegevens vertegenwoordigen die moeten worden overgedragen. Deze velden kunnen variabelen zijn die een specifieke staat of informatie representeren. Vaak wordt hiervoor een `record` gebruikt.

Overdracht van Gegevens tussen Lagen

DTO's worden vaak ingezet in architectuurmodellen zoals een drielagenstructuur, waarbij ze fungeren als tussenliggende objecten tussen de presentatielaag, domeinlaag en persistentielaag. DTO's zorgen voor een gestructureerde manier om gegevens heen en weer te sturen zonder de directe koppeling tussen lagen.

Immutabiliteit

DTO's zijn vaak ontworpen als `immutable` objecten, wat betekent dat de gegevens die ze bevatten na creatie niet kunnen worden gewijzigd. Dit draagt bij aan de voorspelbaarheid en veiligheid van gegevensoverdracht.

Serializatie en Deserializatie

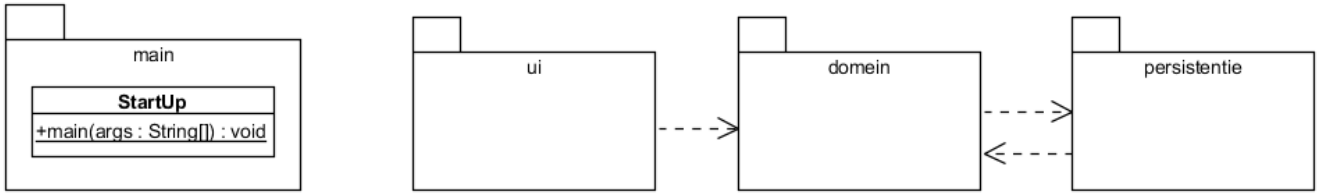
Bij het verplaatsen van gegevens worden DTO's vaak gebruikt bij het serialiseren, wat betekent dat de gegevens worden omgezet in een formaat dat gemakkelijk kan worden verplaatst of opgeslagen. Bij ontvangst kunnen deze gegevens vervolgens worden gedeserialiseerd om hun oorspronkelijke vorm terug te krijgen.

Doelgerichtheid

Een DTO is specifiek ontworpen voor een bepaald doel of een reeks gerelateerde doelen. Een DTO bevat alleen die gegevens die relevant zijn voor de overdrachtstaak.

```
1 public class OrderDTO(int orderId, string customerName)
2 {
3     public int OrderId { get;  } = orderId;
4     public string CustomerName { get;  } = customerName;
5 }
6
7 public record ProductDTO(int ProductId, string ProductName, decimal Price);
```

5. Opbouw van het drielagen model



5.1. Klasse `StartUp`

Aangezien er in grote projecten in elke laag al gauw een tiental klassen kunnen zitten en het de bedoeling is dat het programma de use cases in een bepaalde vastgelegde volgorde afwerkt, willen we ervoor zorgen dat het volledige project maar op één manier kan gestart worden. Er mag dus in heel het project maar één `Main`-methode zijn.

Soms gebruiken een aparte klasse, vaak `StartUp` genoemd, die deze `Main`-methode bevat.

De verantwoordelijkheid van de `StartUp` klasse bestaat uit het configureren van de applicatie o.a. door het maken van een object van de eerste UI-klasse met als parameter een DomeinController-object. Aan de hand van dit object wordt dan een methode uit deze klasse aangeroepen die de verdere controle van het systeem overneemt: typisch is dit een methode `Run`.

5.2. De presentatielaag

De ui kan een consoleapplicatie (cui) of een grafische applicatie (gui) zijn. De constructor van de eerste klasse krijgt een `DomeinController`-object mee vanuit de StartUp of maakt deze zelf aan en zal aan de hand van dit object gegevens kunnen opvragen uit de domeinlaag die dan op het scherm kunnen getoond worden.

Gegevens die door de gebruiker worden ingevoerd kunnen, eveneens via het DomeinController-object, worden verwerkt in de domeinlaag. Wanneer een volgende ui-klasse wordt aangeroepen, krijgt de constructor hiervan opnieuw de DomeinController mee zodat daarin op dezelfde manier kan verder gewerkt worden.

- Wanneer gegevens uit de database moeten geraadpleegd worden of er data moet worden weggeschreven, kan dit **NIET** rechtstreeks vanuit de presentatielaag gebeuren. Alle communicatie van de presentatielaag naar de persistentielaag gebeurt steeds via de domeinlaag!
- Er worden in de presentatielaag geen objecten uit de domeinlaag (behalve het DomeinController-object) gebruikt, behalve DTO objecten.

5.3. De domeinlaag

De domeinlaag vormt het hart van de applicatie. Via de DomeinController wordt informatie uit de user interface doorgesluisd naar de betreffende domeinklasse die dan validaties en berekeningen hierop kan doen. De DomeinController heeft daartoe public methodes die overeenkomen met de systeemoperaties uit het SSD.

Indien de applicatie te groot wordt, kan besloten worden om nog verdere UseCaseControllers te gebruiken die ervoor zorgen dat de operaties logisch opgedeeld kunnen worden per use case. De DomeinController "delegeert" dan naar de UseCaseControllers, die dan zelf weer de andere domeinklasses aanspreken.

In de domeinlaag kan er ook communicatie plaatsvinden met de persistentielaag. Dit gebeurt typisch via een repository interface.

5.4. De persistentielaag

In de persistentielaag wordt de link gelegd met de databank. Maar... in de databank zit data, geen objecten! Het is dus de taak van de persistentielaag om van de data in de database, op aanvraag van het domein, objecten te maken en omgekeerd, om op aanvraag van het domein, objecten op te slaan als data in de database (persistent maken).

6. Drielagen applicatie - Stappenplan

6.1. Solution opzetten

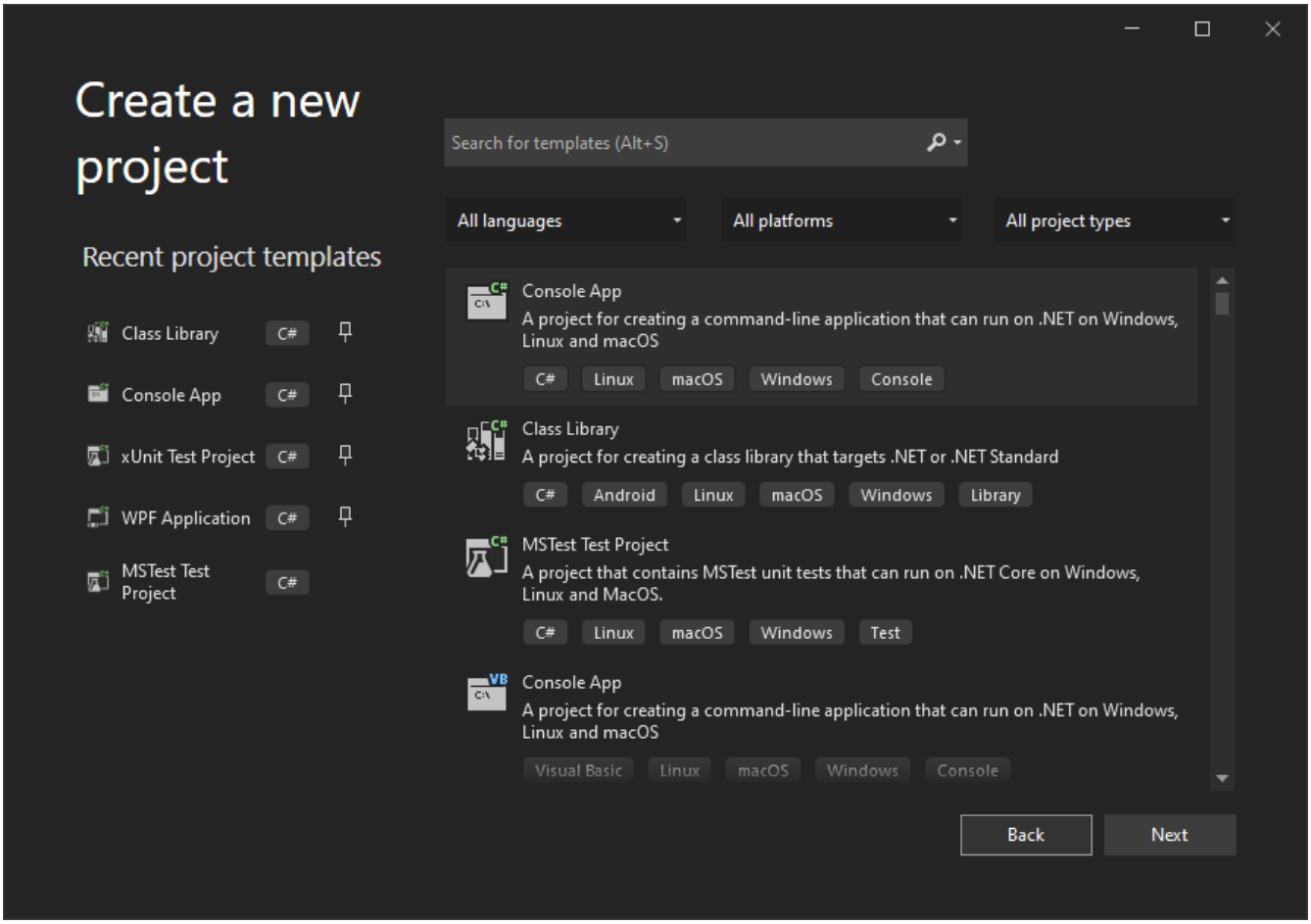
6.1.1. Startup project

We beginnen het opzetten van de solution door een naam te bedenken, deze zullen we gebruiken doorheen de volledige solution. Zorg dus dat je een goede naam kiest die je achteraf niet meer moet veranderen. In volgende stappen wordt hiernaar verwezen als {app_naam}. In de screenshots zal je zien dat hier voor `SetupGuide` gekozen is als projectnaam.

Een naam van een solution en/of project komt soms wel overeen met de naam van een toekomstige klasse, wat een probleem is. Kies de naam van je solution en projecten dus wijs zodat je later niet in de problemen komt.

We beginnen met het aanmaken van het startup project, dit is geen volwaardige laag binnen het drie lagen model. De startup klasse in dit project breekt sommige afspraken die gelden voor de rest van de solution daarom zetten we deze apart.

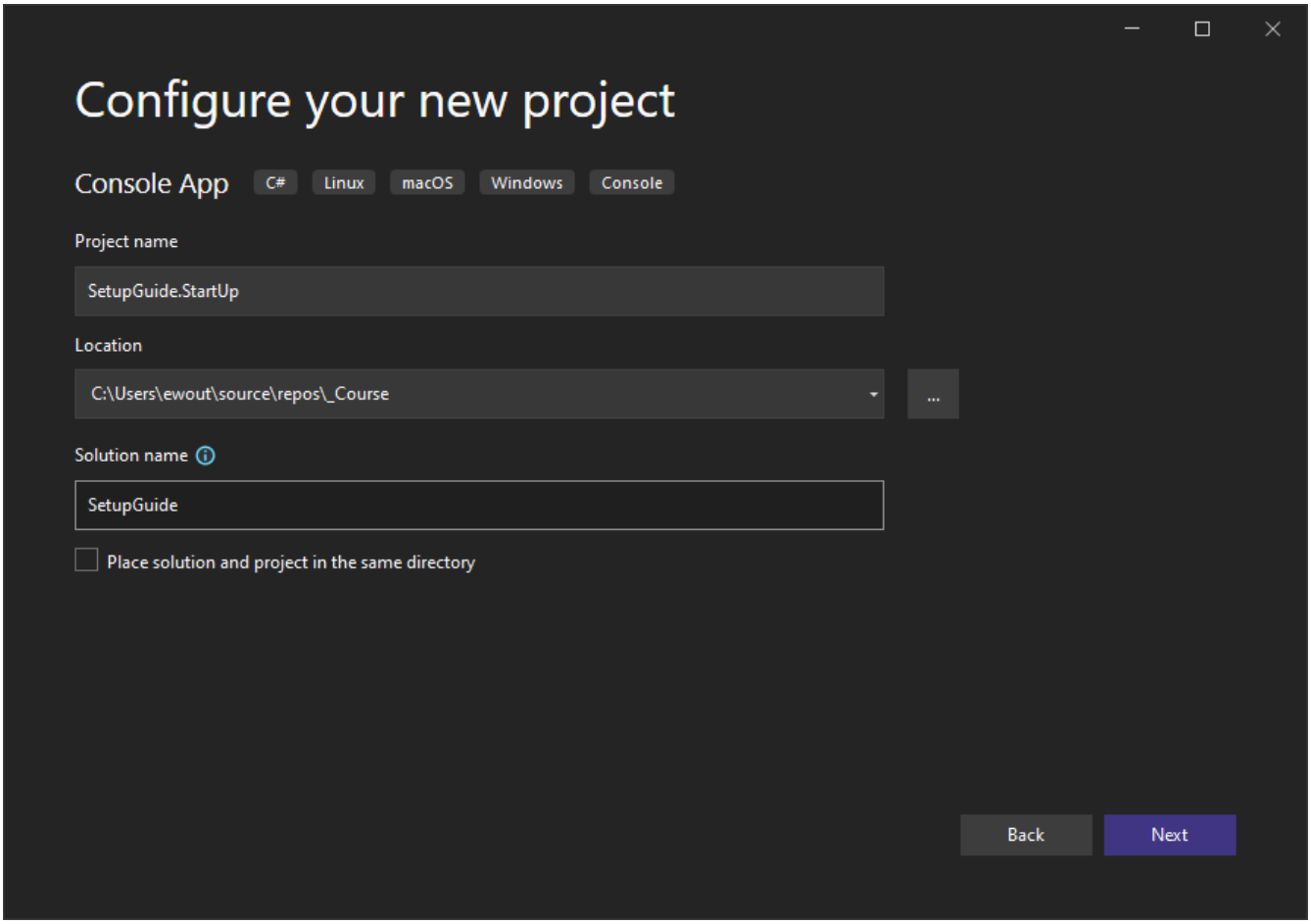
Maak dit project aan met de template “Console App”.



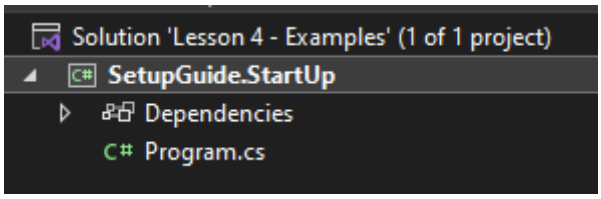
Geef je solution de naam van je applicatie en het project de naam {app_naam}.StartUp.

Pas op bij het ingeven van je Project name, dit veld is verbonden aan het Solution name veld.

Geef dus eerst de volledige naam van je project in alvorens je de naam van de solution ingeeft.



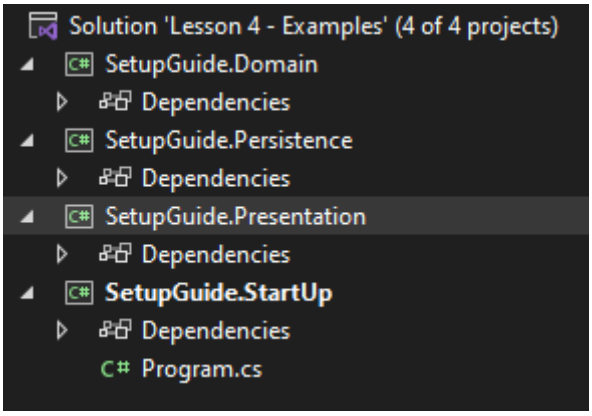
We hebben nu een solution die één project bevat.



6.1.2. Drie lagen toevoegen

Nu voegen we drie lagen toe aan de solution, rechtsklik op je solution > add > new project. Voeg hier drie projecten toe die elk gebruik maken van de “Class Library” template. De drie lagen krijgen volgende namen:

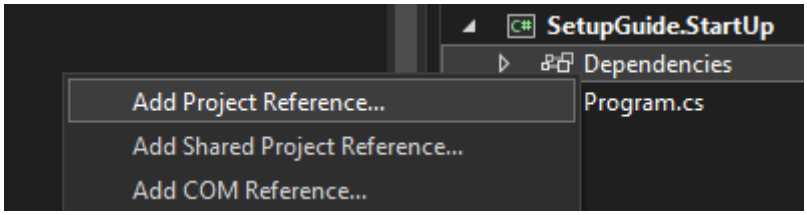
1. {app_naam}.Presentation
2. {app_naam}.Domain
3. {app_naam}.Persistence



6.1.3. Dependencies

Dependencies op voorhand vastleggen

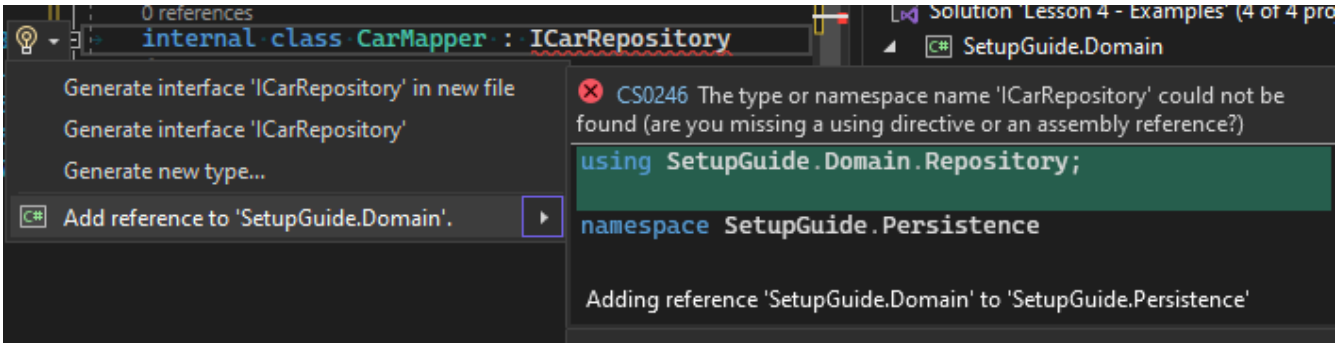
Voeg in het Startup project dependencies toe naar de andere drie projecten. Dit doen we door in het startup project rechts te klikken op Dependencies en dan Add Project Reference…



Voeg op dezelfde manier in het Persistence project een dependency toe naar het Domain project. Ook in het Presentation project voegen we een dependency toe naar het Domain project. Normaal zou je de references ook moeten zien verschijnen als je de dependencies van een project open klikt en kijkt onder projects.Eenmaal deze vijf dependencies gelegd gaan we door met de opbouw van ons project.

Dependencies achteraf instellen

Bovenstaande manier om dependencies vast te leggen is de gemakkelijkste, zeker in het begin. Maar er is ook nog een andere manier: je kan de verbindingen ook leggen tijdens het schrijven van je applicatie. Let dan wel op dat je de references in de juiste richting legt.



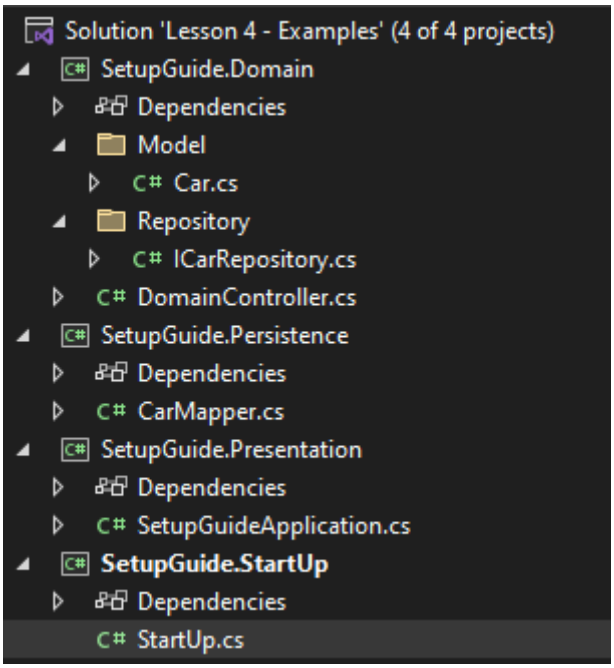
6.2. Structuur schrijven

6.2.1. Klassen aanmaken

Maak in de verschillende projecten volgende klassen aan (of hernoem de bestaande):

- Presentation
  - {app_name}Application
- Domain
  - DomainController
  - Een of meerdere data klassen/models (eventueel in een subfolder)
  - Repository interface per data klasse (eventueel in een subfolder)
- Persistence
  - Repository klasse per data klasse/model
- Startup
  - Startup

In dit voorbeeld doen we alsof onze SetupGuide applicatie met data van auto's werkt, in de praktijk verandert dit natuurlijk afhankelijk van de oefening of applicatie die je maakt.



6.2.2. Public en internal access modifiers

Vanaf nu moet je beginnen opletten met het gebruiken van het keyword `internal` als access modifier. Deze waren tot nu toe identiek, onze projecten bestonden bijna uitsluitend uit één project in één solution. Als je deze access modifier gebruikt in plaats van public zal je die klasse, property of methode niet kunnen gebruiken buiten het project waar het in zit.

Ga daarom zeker niet alles standaard public zetten, het is nog steeds het beste om de access scope zo klein mogelijk te houden. Daarmee wordt bedoeld dat alles wat `internal` kan staan best `internal` blijft, enkel wat buiten het project (of de laag van je applicatie) toegankelijk moet zijn mag je public zetten.

6.2.3. Namespaces

Zorg er voor dat de namespaces van alle klasses ook kloppen. Een namespace moet bestaan uit de naam van je solution + de naam van je project + eventuele subfolders.

Persistence

Zorg ervoor dat elke repository klasse zijn respectievelijke interface uit de Domain layer implementeert. Let er op dat methodes om de data aan te spreken hier en in de geïmplementeerde `IRepository` interface gedefinieerd worden.

Domain

De constructor van de DomainController verwacht een instantie van elke IRepository interface mee te krijgen via zijn constructor. Deze instanties worden bijgehouden in private (readonly) fields.

Presentation

De constructor van de {app_name}Application verwacht een instantie van de DomainController mee te krijgen via zijn constructor. Deze instantie wordt bijgehouden in een private field.

Startup

De `Startup` klasse bouwt laag voor laag onze applicatie op in zijn Main methode: het drielagen model wordt geconfigureerd:

- In de Persistence layer wordt elke `IRepository` interface geïmplementeerd dmv een overeenkomstige repository klasse die de respectievelijke interface implementeert. Hier worden hiervan objecten geïnstantieerd.
- De Domain layer wordt aangemaakt door de `DomainController` te instantiëren en in zijn constructor dependency injection toe te passen van elke IRepository interface.
- De Presentation layer wordt aangemaakt door de {app_name}Application klasse aan te maken en de instantie van de `DomainController` mee te geven in zijn constructor.