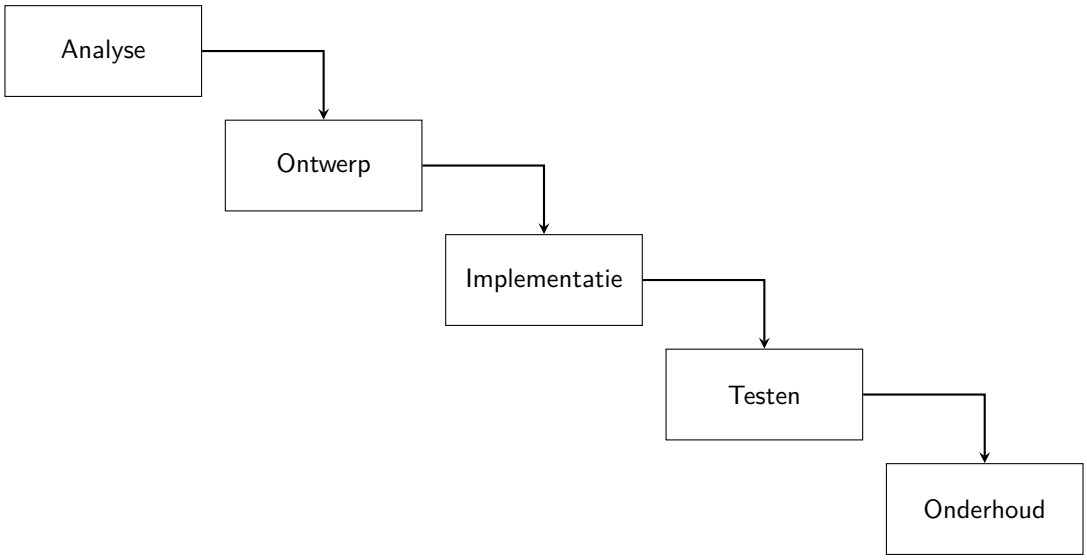
Inhoudsopgave

Introductie

Software engineering is een integraal proces waarbij vanuit een probleemomschrijving een passend product opgeleverd wordt. De grootste uitdaging hierbij is duidelijk krijgen wat er precies ontwikkeld moet worden en daarnaast borgen dat hetgeen dat ontwikkeld is, juist is. Hierbij betekent “juist” niet alleen dat de opgeleverde applicatie werkt maar ook dat deze voldoet aan de eisen van het project: het programma doet wat het moet doen.

In onderstaande afbeelding is het traditionele watervalmodel te zien wat ook vandaag de dag nog vaak gebruikt wordt binnen softwareontwikkeling. Merk op dat deze cyclus vaak meermaals doorlopen wordt: het is geen vereiste om voor de gehele applicatie een complete analyse en ontwerp op te stellen alvorens er begonnen wordt met de implementatie. Doorgaans wordt een applicatie opgedeeld in meerdere iteraties, waarbij per iteratie grofweg dit traject doorlopen wordt.



In de werkelijkheid zijn procesmodellen doorgaans niet zo simplistisch als hier weergegeven. Voor dit document biedt het echter een mooie kapstok om de activiteiten die per fase gedaan worden van een toelichting te voorzien. Het uiteindelijke doel van dit document is het toelichten van de visie op het softwareontwikkelproces. Hierbij wordt context gegeven aan de hand van voorbeelden; deze voorbeelden zijn niet de enig mogelijke of correcte manier om dit te doen: deze dienen puur als beeldvorming.

Analysefase

Het belangrijkste punt in deze fase is helder krijgen welke functionaliteit precies in een project opgenomen wordt. Hier zijn een aantal technieken voor beschikbaar, die allemaal bedoeld zijn om de requirements meer en meer te verduidelijken.

Functionele requirements, beperkingen en kwaliteitseisen

Een functionele requirement beschrijft de handelingen die verricht kunnen worden met de applicatie. Iedere functionele requirement kan kwaliteitseisen en beperkingen bevatten die een afbakening geven. Als voorbeeld, een simpel verkoopsysteem:

FR-01 De gebruiker moet een overzicht van alle verkopen kunnen inzien.

- **B-01.1** Overzichten van verkopen zijn op maanden, kwartalen of jaren.
- **K-01.1** Overzichten dienen oplopend en aflopend gesorteerd te kunnen worden.

FR-02 Er kunnen nieuwe verkopen toegevoegd worden aan het systeem.

- **B-02.1** Voor verkopen wordt een omschrijving, het bedrag en het tijdstip vastgelegd.
- **B-02.2** De omschrijving van een verkoop mag niet leeg zijn.
- **B-02.3** Alleen leidinggevenden mogen de datum van een verkoop aanpassen; anders is deze altijd het moment van invoeren.

FR-03 Het is mogelijk om geld terug te geven aan klanten (voor het ruilen van artikelen).

- **B-03.1** Alleen leidinggevenden zijn in staat om geld terug te geven aan klanten.

K-ALG.01 Bij onjuiste invoer moet een duidelijke foutmelding getoond worden.

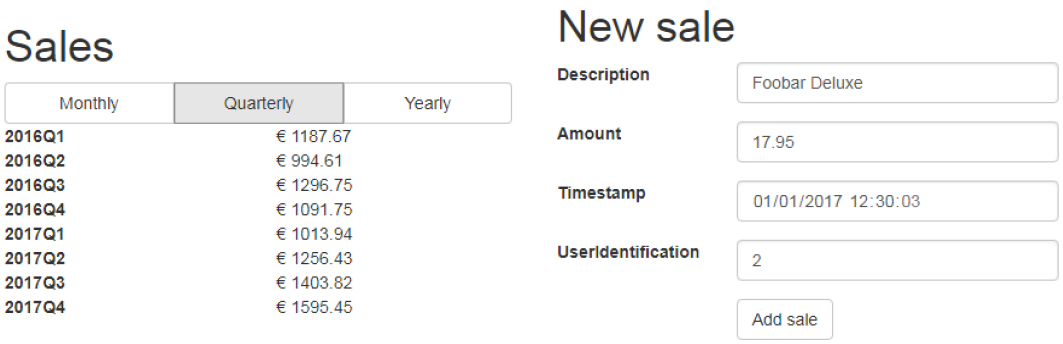
Wat niet getoond is in dit overzicht, is de prioritering van de requirements. In de praktijk kan het voorkomen dat niet alle functionaliteit op het afgesproken tijdstip opgeleverd is. Om toch de meest relevante requirements op te kunnen leveren en dus ook je planning op orde te hebben, wordt iedere requirement aangemerkt via de MoSCoW-techniek: must, should, could of won't. Welke functionaliteit aan welke requirement wordt gekoppeld is iets wat in overleg met de opdrachtgever bepaald wordt. Per iteratie of oplevering kan deze prioritering herzien worden.

Het getoonde overzicht is al aardig uitgewerkt, maar het is prima mogelijk dat de eerste versie van de requirements beknopter of simpeler zijn. In ieder van de opvolgende secties wordt een middel beschreven om deze verfijning te bereiken.

User interface schetsen

Het schetsen van de user interface biedt inzicht in hoe bepaalde aspecten van de applicatie kunnen werken. Iedere knop of element op het scherm biedt de mogelijkheid tot gebruikersinteractie. Dit is ook het punt waarop de haalbaarheid van bepaalde requirements bepaald kan worden. Mogelijk blijkt een requirement bijvoorbeeld complexer dan verwacht en dient deze opgesplitst te worden in meerdere stappen. Zie figuur 1 voor een voorbeeld.

Gevoelsmatig is het opstellen van schetsen voor ieder onderdeel van de te realiseren applicatie vaak niet direct nuttig of relevant. Toch zal op een gegeven moment een beslissing genomen moeten worden over de precieze indeling van een scherm. Het punt is dat de vormgeving van een scherm vaak invloed heeft op de requirements en use cases. Wanneer er dus gewacht wordt



Figuur 1: *Een voorbeeld van UI-schetsen voor een verkoopapplicatie.*

met het vaststellen van de indelingen tot het moment van implementeren, is er een risico dat iets toch onmogelijk blijkt of complexer is dan verwacht. Derhalve loont het de moeite om op voorhand zo volledig mogelijk te zijn bij het opstellen van deze schetsen. Om dubbel werk te voorkomen kunnen de schetsen wel meteen als webpagina of als formulier binnen Visual Studio opgesteld worden.

Use cases

Op basis van de schetsen zijn use cases op te stellen die precies aangeven hoe de interactie met het systeem plaatsvindt. Alhoewel er geen exacte richtlijn te geven is, zijn handelingen die neerkomen op het wisselen van schermen of pagina's typisch geen eigen use case maar stappen hierbinnen. Een use case wordt gebruikt om handelingen die de toestand (gegevens) van het systeem gebruiken of beïnvloeden te documenteren.

In tabel 1 is een voorbeeld te zien van een uitgewerkte use case, overeenkomstig met een van de eerder getoonde UI schetsen. De use case beschrijft hoe de functionele requirement precies vervuld kan worden. Er wordt vanuit gegaan dat er standaard geen problemen optreden bij het uitvoeren van de use case; dit wordt ook wel de "happy flow" genoemd. Mocht er bij bepaalde stappen een probleem op kunnen treden, dan wordt hier een uitzondering genummerd bij aangemerkt. In dit voorbeeld is per uitzondering aangegeven welke beperking ondervangen wordt; ook de use case zelf is voorzien van een identificatie.

Test cases

De volgende stap is het concreet maken van de gegevens die de applicatie moet kunnen verwerken. Door test cases te relateren aan use cases worden enerzijds voorbeelden gegeven van waar het systeem precies mee moet kunnen werken. Anderzijds beschrijft dit ook hoe het systeem reageert wanneer de gegevens onjuist zijn (de uitzonderingen). Aangezien het gebruik van het systeem al gemodelleerd is in de use cases, zullen test cases altijd aan use cases gekoppeld worden. Zie tabel 2 voor een voorbeeld.

Bij het voorbeeld is dit niet het geval, maar het is prima mogelijk om in een testplan een volgorde aan te brengen. Een voorbeeld hiervan is het testen van het registreren en inloggen van gebruikers. Als eerste kan een account aangemaakt worden, waarbij het registreren getest wordt. Deze aangemaakte account wordt vervolgens gebruikt om het inloggen mee te testen. Op deze

Tabel 1: *Een mogelijke use case voor het toevoegen van verkopen aan een systeem.*

Naam	Nieuwe verkoop toevoegen **UC01**
Samenvatting	Na het ingeven van de vereiste gegevens wordt dit als verkoop aan het systeem toegevoegd.
Actors	Verkoper, Leidinggevende
Aannamen	Geen.
Omschrijving	1. De actor geeft aan een nieuwe verkoop te willen toevoegen. 2. Het systeem toont een pagina waarop de gegevens ingevuld kunnen worden (zie B-02.1). 3. De actor voert zijn gebruikersidentificatie en de gevraagde gegevens in en bevestigt. 4. Het systeem controleert de ingevoerde gegevens en voegt de verkoop toe. [1][2]
Uitzonderingen	1. Niet alle benodigde gegevens zijn ingevuld. Toon een melding en ga terug naar stap 3. (B-02.2) 2. De verkoopdatum is aangepast en de actor is geen leidinggevende. Zet de datum op de huidige datum en tijd en ga verder met de use case. (B-02.3)
Resultaat	Er is een nieuwe verkoop toegevoegd aan het systeem.

manier zijn test cases eenvoudiger uit te voeren omdat er geen afhankelijkheid van bestaande data is. Zorg er wel voor dat in het testplan duidelijk is dat bepaalde cases elkaar dienen op te volgen.

Wanneer is de analyse klaar?

Na het opstellen van de test cases is het mogelijk om een terugkoppeling te maken naar de requirements. Hiervoor kan een test matrix opgesteld worden, waarin per test case aangegeven wordt welke (delen van een) requirement getest wordt. Twee problemen kunnen aan het licht komen: er is een (deel van een) requirement die nog niet getest wordt, of er is een test case waarvoor geen requirement is. In het eerste geval ontbreekt er iets in een UI schets of use case, aangezien er een requirement is waarvoor nog geen interactie beschreven is. In het laatste geval blijkt er uit het analyseproces een nieuwe requirement te zijn ontstaan; dit betekent dat de lijst met requirements uitgebreid zal moeten worden.

Tabel 3 toont de matrix die aansluit bij de voorbeelden die in dit document zijn opgevoerd. Wat hieruit duidelijk wordt is dat requirement FR-01 in zijn volledigheid nog niet getest wordt. Dit impliceert dat er nog geen test cases zijn opgesteld hiervoor. Maar, er is ook nog geen use case voor beschikbaar: dit zal dus eerst uitgewerkt moeten worden. Wanneer dit gedaan is, kunnen tests worden opgesteld en wordt de matrix bijgewerkt.

Wat ook naar voren komt uit de tabel is dat beperking B-02.2 niet getest wordt. Deze beperking kwam neer op het niet leeg mogen zijn van de omschrijving van een verkoop. In het testplan is

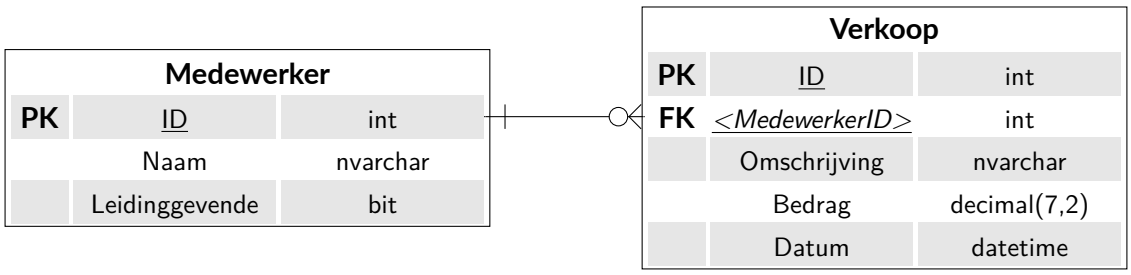
Tabel 2: *Een aantal test cases die het uitvoeren van de eerder getoonde use case beschrijven.*

Test case	Use case(s)	Invoer	Verwachte uitvoer
TC01	UC01	Omschrijving: “Nieuwe verkoop” Bedrag: “€ 12,50” User id: “2”	Verkoop toegevoegd door verkoper.
TC02	UC01	Omschrijving: “Oude verkoop” Bedrag: “€ 12,50” Datum: “2017-01-01 12:00:00” User id: “1”	Verkoop in verleden toegevoegd door leidinggevende.
TC03	UC01	Omschrijving: “Oude verkoop” Bedrag: “€ 12,50” Datum: “2017-01-01 12:00:00” User id: “2”	Verkoop in verleden ingevoerd voor verkoper, datum veranderd naar “nu”.
TC04	UC01	Omschrijving: “Geldteruggave” Bedrag: “€ -25,00” User id: “1”	Geldteruggave geregistreerd door leidinggevende.
TC05	UC01	Omschrijving: “Geldteruggave” Bedrag: “€ -25,00” User id: “2”	Geldteruggave geweigerd voor verkoper.

echter geen case opgenomen die dit valideert. Hiervoor dient dus een testcase toegevoegd te worden; in dit geval is er reeds een toepasselijke use case beschikbaar.

Tabel 3: *Een voorbeeld van een testmatrix voor de getoonde use cases en test cases.*

	FR-01	B-01.1	K-01.1	FR-02	B-02.1	B-02.2	B-02.3	FR-03	B-03.1
TC01				X	X				
TC02				X					
TC03							X		
TC04								X	
TC05									X



Figuur 2: *Een voorbeeld van een databaseontwerp.*

Ontwerpfase

Nu duidelijk is wat precies gemaakt moet worden, kan er voor de opgestelde analyse worden begonnen met het ontwerp. De twee zaken die hierbij het meest relevant zijn, zijn de klassendiagrammen met bijbehorende architectuur en het databaseontwerp.

Het ligt het meest voor de hand om eerst de entiteiten te bepalen die binnen de applicatie gebruikt worden en hoe deze zich tot elkaar verhouden. Hiervoor kan een ERD opgesteld worden (Entity-Relation Diagram); het is echter ook prima mogelijk om direct een databaseontwerp te maken. Dit databaseontwerp toont een aanzienlijke overlap met de modellen die in het klassendiagram opgenomen worden. Naast dit aspect van het klassendiagram worden er nog verdere lagen verwacht die het daadwerkelijke gedrag van de applicatie zullen bevatten.

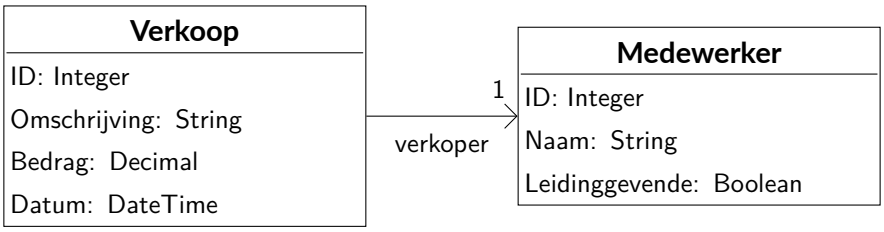
Databaseontwerp

Zoals gezegd kan het nuttig zijn om de entiteiten eerst te schetsen in een ERD. Uiteindelijk is er toch een ontwerp nodig wat daadwerkelijk omgezet kan worden naar een database. Welke weg er ook gekozen wordt, het probleemdomein zal gedocumenteerd moeten worden.

Verder gaande op de requirements van pagina 3 kunnen de volgende onderwerpen hieruit bepaald worden: overzicht, verkoop, verkoopperiode, omschrijving, bedrag, tijdstip, leidinggevende, verkoper. Deze opsomming is de lijst van onderwerpen (of zelfstandige naamwoorden) die in de requirements gevonden zijn. Per element wordt bepaald of het een entiteit is, of een attribuut van een entiteit. Tot slot worden de relaties bepaald tussen deze entiteiten, waarna er een eerste opzet voor het databaseontwerp kan ontstaan. Een simpel voorbeeld hiervan zie je in figuur 2.

Klassendiagram

Waar een databaseontwerp bedoeld is voor de opslag van gegevens, is het klassendiagram bedoeld voor het gebruik van deze gegevens binnen je applicatie. Het klassendiagram is opgedeeld in lagen, waarbij een onderscheid gemaakt wordt tussen de modellen en de architectuur waarbinnen de view-, logica- en datalagen ingedeeld worden.



Figuur 3: *Een mogelijke vertaling van het databaseontwerp naar het klassendiagram.*

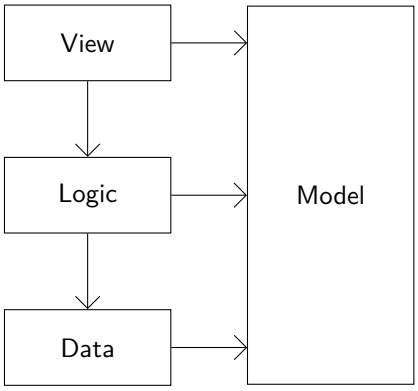
Modellen

De modellen zijn in deze opzet de klassen waarmee de gegevens door de applicatie verplaatst worden. Wanneer er een databaseontwerp opgezet is, kan deze doorgaans redelijk letterlijk vertaald worden naar een klassendiagram voor het model-component. Koppeltabellen komen te vervallen; voor overige relaties moet bepaald worden wat het meestgebruikte scenario is in de applicatie. In het voorbeeld van het verkoopsysteem is er een afweging te maken om de medewerker op te slaan in de verkoop klasse, of juist andersom.

In figuur 3 is gekozen om de relatie vanaf een verkoop naar de medewerker te laten lopen. Het belangrijkste argument hiervoor is dat verwacht wordt dat we vaker een verkoop hebben waar we een medewerker bij willen opvragen, vergelijken met het opvragen van alle verkopen van een medewerker. Overigens is het natuurlijk altijd mogelijk om deze informatie te achterhalen, echter niet rechtstreeks via de relaties tussen de klassen.

Lagen

De hiervoor besproken modellen worden gebruikt voor de communicatie tussen de lagen in een meerlaagse architectuur. Dit is getoond in figuur 4 waar de modellen getoond worden als zijnde “laagoverstijgend”; oftewel, iedere laag gebruikt de modellen. De “echte” lagen bevinden zich aan de linkerzijde van het figuur.



Figuur 4: *Meerlaagse architectuur. Pijlen geven communicatierichtingen aan.*

In de **view**-laag is alle code te vinden die te maken heeft van het weergeven van de informatie die voor de applicatie relevant is. Denk hierbij aan de formulieren uit een desktopappliatie of de pagina’s van een website. Concreet gesteld zijn hier de `Forms` en `Controllers` te vinden. Deze laag communiceert met de logica-laag en heeft kennis van de modellen.

Alle controle op de geldigheid en volledigheid van data wordt afgedwongen in de **logic**-laag. Als voorbeeld: in het verkoopsysteem bestaat de beperking dat alleen leidinggevenden de datum van een verkoop mogen aanpassen. De controle (en eventuele correctie) van ingevoerde gegevens wordt gezien als business logica en wordt derhalve in deze laag geplaatst. Omdat deze laag ook kennis heeft van de modellen kan deze informatie wel doorgegeven worden via een instantie van de `Verkoop`-klasse. Verder communiceert deze laag met de data-laag.

De **data**-laag voorziet in de persistentie van gegevens. Dit zal vaak neerkomen op het wegschrijven naar en ophalen van gegevens uit een database, maar tegelijkertijd kan hier ook een tijdelijke geheugenopslag voor testdoeleinden in bestaan. De zaken die opgeslagen worden, worden wederom aangeboden als instantie van klassen uit de model-laag. Tegelijkertijd worden opgehaalde gegevens gebruikt om instanties uit de model-laag te vullen zodat deze als resultaat van methodes terugvloeien naar bovenliggende lagen.

Vertaling van use cases

Zoals eerder gesteld kan het databaseontwerp redelijk letterlijk omgezet worden naar een klassendiagram voor de model-laag. Daarnaast dient echter het beoogde gedrag van de applicatie gemodelleerd te worden. Het resultaat van deze omzetting komt voornamelijk terecht in de logica-laag.

In iedere use case zijn handelingen te vinden die de gebruiker kan verrichten. Iedere handeling komt op zijn beurt in aanmerking om als methode opgenomen te worden. Om het voorbeeld uit de use case uit tabel 1 uit te werken, zouden daarvoor de hiernavolgende elementen gehaald kunnen worden.

Ten eerste is er het aangeven dat er een verkoop toegevoegd moet worden. Dit is exemplarisch voor een methode in de view-laag, aangezien er een ander scherm getoond moet worden. Aangezien het hier om een nieuwe verkoop gaat, is er ook geen methode benodigd voor het ophalen van een specifieke verkoop.

Daarna worden de ingevoerde gegevens bevestigd en dient er een nieuwe order aangemaakt te worden. Op basis van de use case kunnen we stellen dat we informatie over de verkoop nodig hebben. Daarnaast is er ook een controle benodigd op de medewerker (of het een leidinggevende is of niet). De medewerker is echter beschikbaar via de verkoop; zie het model (figuur 3) waar de relatie van `Verkoop` naar `Medewerker` gedefinieerd is. De resulterende methode zou er dan (in UML-notatie) zoals hieronder uit kunnen zien, waarbij de tweede uitzondering uit de use case (het aanpassen van de verkoopdatum) in deze methode afgehandeld kan worden.

```
VerkoopToevoegen(verkoop: Verkoop): Boolean
```

De uitzondering betreffende het invullen van alle benodigde gegevens, is iets wat niet alleen voor deze use case zal gelden. Het is denkbaar dat ook in andere gevallen het ongewenst is dat bijvoorbeeld het bedrag leeg is. Deze gevallen kunnen voortkomen bij het ophalen van gegevens uit de database of wanneer verkopen via een andere bron worden aangemaakt. In al deze gevallen willen we dus dat specifieke zaken voor een verkoop gegeven zijn.[1] Hiervoor zou dan ook een methode aan de klasse `Verkoop` toegevoegd worden; in dit geval een constructor. Merk op dat onderstaande code in de model-laag terecht zal komen:

```
Verkoop(medewerker: Medewerker, omschrijving: String,
        bedrag: Decimal, datum: DateTime)
```

Hiermee zijn alle onderdelen van deze use case verwerkt in het klassendiagram. Voor de logica- en data-laag worden nieuwe, losstaande klassendiagrammen gemaakt die de handelingen in die laag beschrijven. Tussen de logica- en data-laag is verder vaak een duidelijke parallel te vinden:

[1] Zie de functionele requirements op pagina 3, beperking B-02.1 voor wat in ons voorbeeld verwacht wordt.

wanneer er een `VerkoopToevoegen`-methode opgenomen is in de eerste laag, zal deze ook vaak in de opvolgende laag te vinden zijn. Het verschil zit dan in de implementatie: hier wordt verderop in dit document aandacht aan besteed.

Dataopslag

Code in de data-laag vertoond vaak een sterke overeenkomst met de methodes die de bovenliggende logica-laag te vinden zijn. Het grootste verschil is dat er in deze laag vaak de aanname gemaakt wordt dat gegevens correct zijn. Om door te gaan op het voorbeeld uit de vorige sectie, zal de `VerkoopToevoegen`-operatie in de data-laag doorgaans geen controle bevatten op de juistheid van de invoer: aangenomen wordt dat dit correct is. Dit betekent natuurlijk niet dat er geen checks in de database bestaan die bijvoorbeeld een niet-toegestane waarde blokkeren; de opmerking is met name van toepassing op de code in deze laag.

Als voorbeeld van een afwijkende implementatie in deze laag zou hiervoor bijvoorbeeld requirement FR-01 en specifieker, beperking B-01.1 bekeken kunnen worden (zie pagina 3). Overzichten op maand, kwartaal of jaar zijn waarschijnlijk drie verschillende methodes in de logica-laag. Tegelijkertijd is het eveneens aanmaken van drie methodes in de data-laag wat redundant. De data-laag heeft dan slechts een enkele methode:

```
GeefAlleVerkopen(): Verkoop[*]
```

De logica-laag zou dan deze methodes kunnen bevatten:

```
MaandOverzichtVerkopen(): Verkoop[*]
KwartaalOverzichtVerkopen(): Verkoop[*]
JaarOverzichtVerkopen(): Verkoop[*]
```

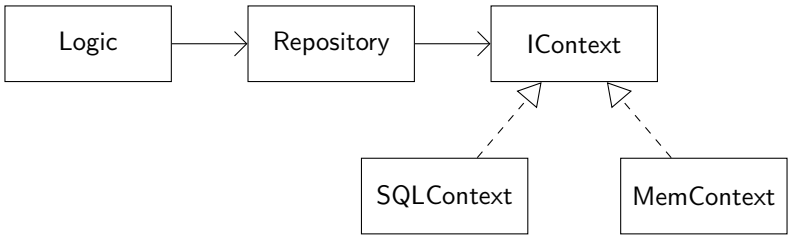
In deze drie methodes wordt een en dezelfde methode uit de data-laag aangeroepen. Het resultaat van deze aanroep wordt dan in de logica-laag per methode anders verwerkt om het uiteindelijke resultaat te verkrijgen. Merk op dat dit één mogelijke implementatie is: wat de “juiste” keuze is, is verschillend per project, de complexiteit en ook persoonlijke voorkeuren. Het belangrijkste is dat de uiteindelijke keuze onderbouwd en verantwoord is.

Het repository-pattern

Alhoewel er veel redenen zijn om het repository-pattern te gebruiken, zijn de voornaamste argumenten vanuit het perspectief van dit document de (automatische) testbaarheid van de applicatie en een duidelijke scheiding tussen de logica- en data-lagen in de applicatie.[2] Het laatste punt zorgt voor een verbeterde robuustheid van de applicatie. Wanneer lagen die met elkaar communiceren een zogenoemde *loose coupling* vertonen en dus weinig kennis van elkaars implementatie bevatten, dan leiden wijzigingen in een laag doorgaans niet tot wijzigingen in een andere laag.

Een voorbeeld van *loose coupling* is te vinden in een eerder besproken methode, *VerkoopToevoegen*. Hierbij zit een medewerker gekoppeld aan een verkoop, maar vanuit de aanroepende kant hoeven we ons niet druk te maken wat een leidinggevende wel mag en een medewerker niet. Aangezien deze logica geëncapsuleerd is in deze methode, heeft de view-laag geen verdere

[2] Voor gedetailleerde achtergrondinformatie over het repository-pattern, zie https://msdn.microsoft.com/en-us/library/ff649690.aspx.



Figuur 5: *Een abstracte weergave van het repository-pattern. De klassen uit de logica-laag communiceren met een of meerdere repositories, die op hun beurt met een SQL- of memory-realisatie van de betreffende context communiceren.*

kennis van de implementatie. Wanneer er een wijziging ontstaat in de requirements waardoor deze methode veranderd, is er geen wijziging benodigd in de afhankelijke lagen.

De verhoogde testbaarheid wordt bereikt doordat de logica laag met een conceptuele dataopslag communiceert: de repository. Dit is een abstracte dataopslag, waarbij geen aanname gemaakt wordt over de exacte opslagwijze. Vaak is dit achter de schermen een SQL-database, maar voor testdoeleinden is een tijdelijke opslag in het geheugen veelgebruikt. In figuur 5 is de algemene structuur van het repository-pattern weergegeven.

Implementatiefase

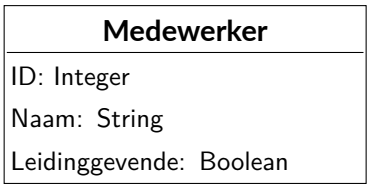
Wanneer de eerste versie van alle ontwerpen afgerond is, kunnen deze omgezet worden naar code. Iedere laag uit de architectuur zal een eigen project worden binnen de gehele solution voor de te realiseren applicatie. De volgorde van de implementatie wordt doorgaans ingegeven door de afhankelijkheid tussen de lagen. Afgaande op de afbeelding in figuur 4 betekent dit dat eerst de modellen geïmplementeerd worden, aangezien deze geen afhankelijkheden bevat. Hierna kan de data-laag opgezet worden, gevolgd door de logica-laag en tot slot de view-laag.

Binnen lagen kan eenzelfde opzet aangehouden worden. Bij het klassendiagram in figuur 3 zou dus eerst de `Medewerker`-klasse geïmplementeerd worden; daarna de `Verkoop`-klasse. Op deze manier kan het gehele diagram doorlopen worden totdat alle klassen omgezet zijn.

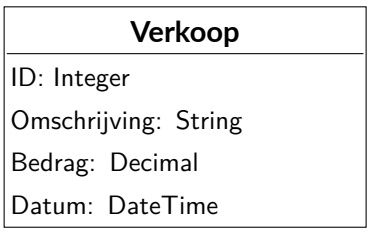
> Merk op dat in figuur 4 de *communicatie* tussen lagen weergegeven is. Het is mogelijk (en eventueel zelfs gewenst) dat de *afhankelijkheiden* van de verschillende lagen omgekeerd zijn. Conceptueel komt het neer op waar de interfaces voor de communicatie tussen lagen gedefinieerd worden. Legt de logica-laag vast hoe de structuur van een repository eruit ziet, of wordt de structuur vastgelegd in de data-laag? In het eerste geval loopt de *afhankelijkheid* van de data-laag naar de logica-laag; in het tweede geval is het omgekeerd. Welke optie ook gekozen wordt, in beide gevallen is de *communicatie*richting hetzelfde.

Klassendiagram naar code

Ter referentie zijn hiernaast de eerder ontworpen klassen nogmaals getoond. Deze kan stap voor stap omgezet worden naar code. Eerst definiëren we de klasse; vervolgens worden de attributen toegevoegd. Onderstaande code is het resultaat van deze omzetting.




```
class Medewerker
{
    private int ID;
    private string naam;
    private bool leidinggevende;
}
```



De verkoopklasse wordt op eenzelfde manier omgezet. Twee verschillen zijn hierbij op te merken: zoals in de sectie aangegeven is er een constructor benodigd. Deze zou in het diagram als extra compartiment naast de methodes van de klasse weergegeven worden. Daarnaast heeft deze klasse een relatie met de hiervoor opgezette `Medewerker`-klasse. De *multipliciteit* van deze relatie was `1`; de *rol* was `verkoper`. In de uiteindelijke code is er dus een enkele `Medewerker`-instantie gedefinieerd. Mocht de multipliciteit "meer" zijn geweest (aangegeven met een `*`), dan zou dit een *List<Medewerker>* geworden zijn.

In deze fase is het voldoende als de structuur opgezet wordt. Het is dus geen probleem als er nog geen duidelijke invulling voor methodes en constructors is. Om deze onvolledige blokken

wel compilerend te krijgen, kan er `throw new NotImplementedException()` in de methode geplaatst worden. Hierdoor ontstaat een foutmelding als de code uitgevoerd wordt, als herinnering dat die methode nog niet compleet is.

```
class Verkoop
{
    private int ID;
    private string omschrijving;
    private decimal bedrag;
    private DateTime datum;
    private Medewerker verkoper; // Relatie van Verkoop naar Medewerker

    Verkoop(medewerker: Medewerker, omschrijving: String,
            bedrag: Decimal, datum: DateTime)
    {
        this.verkoper = medewerker;
        this.omschrijving = omschrijving;
        this.bedrag = bedrag;
        this.datum = datum;
    }
}
```

De logica- en data-laag (met repository-pattern)

Voorgaande code komt terecht in de model-laag, maar een belangrijk deel van de code komt in de logica- en data-laag terecht. Alhoewel in dit document geen ontwerp gegeven is hiervoor, wordt ter illustratie in deze sectie een voorbeeldimplementatie beschreven. Deze implementatie is gebaseerd op de eerder beschreven use case op pagina 5 en geeft eveneens een voorbeeld van een MemoryContext binnen een repository pattern.

Kijk ook nog eens naar figuur 5 voor het schematische overzicht van deze implementatie. Aangezien er veel afwegingen mogelijk zijn bij het bepalen van welke klasse in welke laag geplaatst wordt, is geen indicatie gegeven waar alles geplaatst wordt: het gaat met name over het concept van het pattern.

Bij de context beginnende: in deze interface staat beschreven welke operaties er mogelijk zijn voor de dataopslag. Zoals eerder toegelicht zal hier vaak een *SQLContext* bij gebruikt worden: dit voorbeeld geeft echter een *MemoryContext* ten behoeve van de unit testen die verderop besproken worden. Op basis van de use cases die we beschreven hebben, ziet onze context er als volgt uit:

```
interface IVerkoopContext
{
    bool VerkoopToevoegen(Verkoop verkoop);
    IEnumerable<Verkoop> GeefAlleVerkopen();
}
```

Dit beschrijft de structuur waar alle verschillende contexten aan moeten voldoen. Een voorbeeld van de implementatie van een MemoryContext is de volgende:

```
class VerkoopMemoryContext : IVerkoopContext
{
    private List<Verkoop> verkopen = new List<Verkoop>();

    public bool VerkoopToevoegen(Verkoop verkoop)
    {
        verkopen.Add(verkoop);
        return true; // In dit voorbeeld lukt toevoegen altijd
    }

    public IEnumerable<Verkoop> GeefAlleVerkopen()
    {
        return verkopen;
    }
}
```

Vervolgens hebben we een repository die gebruikt wordt als ontkoppeling tussen de logica- en data-laag. Deze heeft doorgaans dezelfde structuur als de eerder aangemaakte interface. Het belangrijkste aspect hiervan is dat we aan deze klasse aan kunnen geven welke context gebruikt wordt voor de opslag.

```
class VerkoopRepository
{
    private IVerkoopContext context;

    public VerkoopRepository(IVerkoopContext context)
    {
        this.context = context;
    }

    public bool VerkoopToevoegen(Verkoop verkoop)
    {
        return context.VerkoopToevoegen(verkoop);
    }

    public IEnumerable<Verkoop> GeefAlleVerkopen()
    {
        return context.GeefAlleVerkopen();
    }
}
```

Bovenstaande klasse wordt tot slot gebruikt door de logica-klassen van de applicatie. Op het moment dat een repository geïnstantieerd wordt, kan bepaald worden welke context gebruikt moet worden voor de opslag. Voor nu worden simpelweg dezelfde methodes hierin geplaatst: later in het document worden deze van een concrete invulling voorzien.

Onderstaand voorbeeld heeft een tekortkoming op het vlak van ontkoppeling en encapsulatie aangezien in deze de *MemoryContext* rechtstreeks wordt aangemaakt. Om het voorbeeld beknopt te houden is deze implementatie gegeven: in de praktijk worden hier fraaiere oplossingen voor verwacht. Denk hierbij aan het gebruik van enumeraties om de context aan te geven; ook een zogeheten “factory” is een mogelijkheid.

```
class VerkoopLogic
{
    private VerkoopRepository repo = // Zie toelichting hierboven
            new VerkoopRepository(new MemoryContext());

    public bool VerkoopToevoegen(Verkoop verkoop)
    {
        throw new NotImplementedException();
    }

    public IEnumerable<Verkoop> GeefAlleVerkopen()
    {
        throw new NotImplementedException();
    }
}
```

Use cases en test cases

Nadat het skelet van de applicatie opgezet is, kunnen de methodes gevuld worden met een daadwerkelijke implementatie. Om snel de invulling van de methodes te kunnen testen, kan voordat er concrete code geschreven is al begonnen worden met de unit testen. Deze unit testen zijn zeer snel en herhaaldelijk uitvoerbaar wat ervoor zorgt dat we tijdens het schrijven van de methodes direct feedback kunnen krijgen over de correctheid van de implementatie.

Ook al ontbreekt de invulling van de methodes nog, toch kunnen we al test cases opstellen. In de analysefase zijn deze test cases al beschreven (zie pagina 6). Om case TC01 als voorbeeld te nemen: deze is gebaseerd op use case UC01 (“nieuwe verkoop toevoegen”), geeft concrete invoerdata en daarnaast de verwachte uitvoer.

We gaan er in deze vanuit dat er een werkende *MemoryContext* beschikbaar is zoals beschreven in de voorgaande sectie en dat de gegeven repository gebruikt wordt. Daarnaast is de aanname dat de `Medewerker`-klasse een constructor heeft waar de naam- en leidinggevende-attributen mee ingesteld kunnen worden. Verder zijn waar relevant properties aangemaakt om attributen mee uit te kunnen lezen. De eerste implementatie van de testcase zou er als volgt uit kunnen zien:

```
[TestClass]
public class VerkoopTest
{
    [TestMethod]
    public void TC01_VerkoopToevoegen()
    {
        VerkoopLogic vLogic = new VerkoopLogic();
        Medewerker m = new Medewerker(1, "Medewerker", false);
        Verkoop v = new Verkoop(m, "Nieuwe verkoop",
                        12.50M, DateTime.Now);

        Assert.IsTrue(vLogic.VerkoopToevoegen(v));
        Assert.AreEqual(1, vLogic.GeefAlleVerkopen().Count);
    }
}
```

Deze test case zal nu nog als resultaat aangeven dat deze mislukt is. Wel is de test uitvoerbaar en kan dus gebruikt worden bij de ontwikkeling van de code. Om de test te laten slagen zouden we de methodes uit de `VerkoopLogic`-klasse van de volgende implementatie kunnen voorzien:

```
public bool VerkoopToevoegen(Verkoop verkoop)
{
    return repo.VerkoopToevoegen(verkoop);
}

public IEnumerable<Verkoop> GeefAlleVerkopen()
{
    return repo.GeefAlleVerkopen();
}
```

Hiermee slaagt de test case. Maar, de implementatie is nog niet compleet. Om de use case volledig te implementeren, zullen alle uitzonderingen ook geïmplementeerd moeten worden. Hiervoor zijn ook test cases opgesteld. Verder gaande met de test cases voor aankopen in het verleden (TC02 en TC03):

```
[TestClass]
public class VerkoopTest
{
    // De eerder gemaakte methode is niet getoond

    [TestMethod]
    public void TC02_VerkoopToevoegenInVerledenDoorLeidinggevende()
    {
        VerkoopLogic vLogic = new VerkoopLogic();
        DateTime d = new DateTime(2017, 1, 1, 12, 0, 0);
        Medewerker m = new Medewerker(2, "Leidinggevende", true);
        Verkoop v = new Verkoop(m, "Oude verkoop", 12.50M, d);

        Assert.IsTrue(vLogic.VerkoopToevoegen(v));
        Assert.AreEqual(1, vLogic.GeefAlleVerkopen().Count);
        Assert.AreEqual(d, vLogic.GeefAlleVerkopen().First().Datum);
    }

    [TestMethod]
    public void TC03_VerkoopToevoegenInVerledenDoorMedewerker()
    {
        VerkoopLogic verkoopLogic = new VerkoopLogic();
        DateTime d = new DateTime(2017, 1, 1, 12, 0, 0);
        Medewerker m = new Medewerker(1, "Medewerker", false);
        Verkoop v = new Verkoop(m, "Oude verkoop", 12.50M, d);

        Assert.IsTrue(vLogic.VerkoopToevoegen(v));
        Assert.AreEqual(1, vLogic.GeefAlleVerkopen().Count);
        Assert.AreNotEqual(d, vLogic.GeefAlleVerkopen().First().Datum);
    }
}
```

Wanneer deze test cases uitgevoerd worden, zullen ze niet slagen. Vervolgens wordt wederom de implementatie aangepast: als de test slaagt weten we dat aan dat deel van de requirements

voldaan is.

```
public bool VerkoopToevoegen(Verkoop verkoop)
{
    if (!verkoop.DoorLeidinggevende())
    {
        verkoop.ZetDatumOpNu();
    }
    return repo.VerkoopToevoegen(verkoop);
}
```

Op deze manier kunnen meer en meer test cases worden toegevoegd. Hierdoor neemt de omvang en inhoud van de logica-laag toe, maar tegelijkertijd kan dit ook nieuwe code opleveren in andere lagen. De beperking dat de omschrijving van de verkoop niet leeg mag zijn (B-02.2) kan bijvoorbeeld in het model afgedwongen worden. Aangezien iedere test case te maken heeft met de implementatie van een use case, wordt op deze manier *test driven* de applicatie stap voor stap completer.