

COOKBOOK UNIT TESTEN



Document
Opdrachtgever
Versie
Datum versie
Datum laatst bewaard
Auteurs

Cookbook Unit Testen
Ordina J-Technologies & Ordina Si&D / RTI / BUTesting
0.29
10 april 2007
10 april 2007
Marc Mooij, Jeroen Resoort

Inhoudsopgave

Inhoudsopgave	3
1. Vooraf	5
2. Inleiding Unit Testen	6
2.1 Begrip Unit test in de praktijk	6
2.2 White Box en Black Box testen	7
2.3 Waarom is Unit test belangrijk?	8
2.4 Basis van Unit test	8
2.5 Testtechnieken	9
3. Testen met JUnit	11
3.1 JUnit Framework	11
3.1.1 JUnit versies	11
3.1.2 Assertions	11
3.2 Andere frameworks	12
3.3 De praktijk	12
3.4 De ontwikkelomgeving	14
3.4.1 Eclipse	14
3.5 Voordelen van unit tests bij refactoring	15
4. Een multi-tier Java EE applicatie testen	17
4.1 Opbouw	17
4.2 Afzonderlijke units testen	18
4.2.1 Stubs	19
4.2.2 Mocks	19
4.2.3 EasyMock	21
4.3 Exceptions	22
4.4 Integratietests	23
4.5 Databases unit testen	24
4.5.1 DbUnit	24
4.5.2 MockRunner	26
4.6 EJB's unit testen	27
4.6.1 In container testen met Cactus	27
4.6.2 MockEJB buiten de container	28
5. Testen in de ontwikkelstraat	29
5.1 Apache Maven2	29
5.1.1 Conventies	29
5.1.2 Gebruikstips	29
5.1.3 Hergebruik test classes	30
5.2 Ordina Smart-Java ontwikkelstraat	31

6.	Code coverage	32
6.1	Plugins	33
7.	Bijlagen	34
7.1	Principe van afleiden testgevallen	34
7.2	Equivalentieklassen	34
7.3	Grenswaardenanalyse	35
7.4	Operationeel gebruik	36
7.5	Beslissingstabellentest / verwerkingslogica	37
7.5.1	Algemeen	37
7.5.2	Werkwijze	37
7.5.3	Definiëren logische testkolommen	37
7.5.4	Specificeren logische testgevallen	42
7.5.5	Specificeren fysieke testgevallen	46
7.5.6	Initiële database	46
7.5.7	Opstellen testscript	47
7.5.8	Uitvoering en beoordeling	47
8.	Referenties	48

1. Vooraf

Het boek “Systeemontwikkeling met J2EE” [9] beschrijft de softwarearchitectuur van webapplicaties die zijn gebaseerd op de J2EE-technologie. In het boek is een aantal uitgangspunten gedefinieerd voor het maken van applicaties volgens Ordina richtlijnen. Mede doordat J2EE technologieën elkaar in een snel tempo opvolgen is er in het boek geen plaats voor te veel detail over verschillende technologieën. Daarnaast is het boek bedoeld als referentiekader voor verschillende disciplines en niet als uitputtend naslagwerk voor ontwikkelaars.

De Ordina J-Technologies cookbooks zijn bedoeld als aanvulling op het boek. In deze serie worden over verschillende onderwerpen extra richtlijnen gegeven welke buiten het bereik van het boek vallen. De cookbooks bieden de technische verdieping die J2EE ontwikkelaars en architecten kunnen gebruiken om hun kennis over deze onderwerpen uit te breiden.

Het Cookbook Unit Testing geeft richtlijnen voor het maken van unit tests. Daarnaast bevat het uitleg over verschillende frameworks die hierbij een rol (kunnen) spelen. Verder worden er ter verduidelijking voorbeelden van programmacode en het gebruik van ontwikkeltools gegeven zodat een ontwikkelaar hiermee meer vertrouwd raakt.

Het cookbook geeft een inleiding in en een overzicht van de belangrijkste technieken. De lezer zou hiermee snel kunnen beginnen met het schrijven van unit tests. Met de aangereikte referenties of wellicht onder begeleiding van een ervaren ontwikkelaar of in een workshop kan de lezer deze technieken verder onder de knie krijgen.

We hopen dat de lezer met behulp van dit cookbook de vaardigheden en de motivatie verkrijgt om bij het ontwikkelen van software unit testen toe te passen en op deze manier een hogere kwaliteit werk weet te leveren.

Tenslotte willen we graag Frank Verbruggen en Martin van den Bemt danken voor de input en feedback die geholpen hebben bij de totstandkoming van dit cookbook.

2. Inleiding Unit Testen

Testen werd tot voor kort door ontwikkelaars als een tijdverspilling gezien. “Wat heeft het voor nut en het kost alleen maar tijd”, waren vaak gehoorde uitspraken. Maar in de laatste jaren heeft het Unit Testen zijn nut wel bewezen. De code wordt vaak eerder en met minder fouten worden opgeleverd voor volgende testfasen. Want Unit Testen is belangrijk voor het controleren van de eigen code van de ontwikkelaar, maar ook voor het verder doordenken wat er nu precies gebouwd moeten worden. Door eerst testen – voor zowel een goed als foutsituatie - te schrijven en dan pas coderen, wordt de ontwikkelaar min of meer gedwongen om de mogelijkheden en onmogelijkheden van de functie, unit, component te begrijpen en te maken.

De ontwikkelaars hoeven echter niet bang te zijn dat ze het werk van de testers uit handen nemen en dat ze alleen maar testen. Testers kijken naar heel nadere aspecten van een applicatie; vaak naar het geheel, of de samenhang en samenwerking van en tussen verschillende componenten. Met andere woorden zij kijken veel meer naar de gehele functionaliteit en controleren of het systeem voldoet aan de verwachtingen van de opdrachtgever.

In Nederland wordt veelal getest met behulp van TMap (Test Management Approach), een gestructureerde testaanpak voor informatiesystemen. De laatste jaren is de internationale testaanpak via ISEB (Information Systems Examinations Board)¹ ook in opkomst; die veel meer ingaat op de technische testen en meer handvatten biedt aan de ontwikkelaars. De testen die veelal door ontwikkelaars geschreven en uitgevoerd worden, worden ook wel white box testen genoemd. De (functionele) testen die door testers worden gedaan worden veelal black box testen genoemd. Bij black box testen geldt dat dit testen zijn zonder de inhoudelijke c.q. specifieke kennis van de techniek. Veelal vergeleken met een auto; een tester weet wel hoe hard een auto moet rijden en dat deze binnen 15 seconden op de 100 km moet zijn (specificaties), maar weet niet welke motor er onder de motorkap zit om dit voor elkaar te krijgen. Een ontwikkelaar weet dit wel en dan spreekt men ook van white box testen; omdat hij precies kan testen wat de motor wel en niet zou moeten kunnen (hij weet de cilinderinhoud, het koppel en weet dus hoe snel de auto op zich is en in het bijzonder hoe lang het duurt voordat deze 100 km / h rijdt (bijv. op 11,9 seconden). Hoewel glass box een beter passende naam is, wordt veelal de term white box gebruikt. In dit document houden we dan ook white box testen aan.

2.1 Begrip Unit test in de praktijk

Uit het voorgaande blijkt dat de Unit Testen zijn gericht op het technisch testen van kleine softwareonderdelen, units, classes en niet op de totale functionaliteit. Nadat is vastgesteld dat deze delen van het systeem van goede kwaliteit zijn, worden tijdens de *integratietest* grotere delen van het systeem integraal getest.

¹ De Information Systems Examinations Board (ISEB) is een internationaal erkende certificatie instelling die zich richt op het certificeren van IT-professionals. Een ISEB certificaat betekent markterkenning, een maatstaf voor bepaalde bekwaamheden op vele terreinen binnen ICT. ISEB heeft ook een certificatie-programma ontwikkeld voor testprofessionals. In November 2002 is de International Software Testing Qualification Board (ISTQB) opgericht met onder andere als doel een verdere internationalisatie en harmonisatie van de testcertificatieprogramma's tot stand te brengen. Inmiddels bestaat er een volledig wereldwijd geharmoniseerd en erkend Foundation certificatieschema.

Unit testen zijn (relatief) eenvoudig te automatiseren en de testresultaten geven een goede indicatie over de kwaliteit van de ontwikkelde software.

In TMap wordt veelal het begrip *programmatest* gebruikt, waar anderen het begrip unit test hanteren.

In TMap staan de volgende definities van *programmatest/unit test* en *integratietest*.

De *programmatest* is een door de ontwikkelaar in de laboratoriumomgeving uitgevoerde test, die moet aantonen dat een programma aan de in de technische specificaties gestelde eisen voldoet.

De *integratietest* is een door de ontwikkelaar in de laboratoriumomgeving uitgevoerde test, die moet aantonen dat een logische serie programma's aan de in de technische specificaties gestelde eisen voldoet.

Er zijn dus verschillende termen voor Unit Test en integratietest, maar het komt op hetzelfde neer; het testen van een zelfstandige class / unit van de code,

2.2 White Box en Black Box testen

De *programmatest* / Unit test en *integratietest* vallen in TMap onder het begrip White Box testen; hiermee wordt volgens TMap dus verstaan het testen met gebruikmaking van kennis van de interne opbouw van het systeem. Hierbij wordt er gecontroleerd of de meest elementaire delen of verzamelingen zijn gecodeerd conform de technische specificaties. Ook kan er beter getest worden op alle if-then statements. Hiervoor zijn o.a. decision, branch coverage testtechnieken voorhanden. Hieronder worden enkel technieken beschreven, maar er zijn veel nadere technieken beschikbaar, speciaal voor de white box testen.

Black Box testen betreft het testen van de "buitenkant" van het systeem; het testen of het programma voldoet aan de van de (afgesproken) functionaliteit. Dit houdt in dat het systeem wordt getest in de vorm zoals de (toekomstige) gebruikers deze te zien krijgen.

Wanneer White Box testsoorten verder worden vergeleken met Black Box testsoorten, zoals de systeemtest en de acceptatietest, zijn er een aantal belangrijke verschillen te constateren:

- Bij de unit tests / programmatests is vaak de ontdekker van de fouten (= tester) dezelfde persoon als de oplosser van de fouten (= ontwikkelaar). Dit betekent dat de communicatie over de fouten minimaal kan zijn;
- De insteek van het White Box testen is dat alle fouten zijn opgelost vóórdat de programmatuur overgedragen wordt. De rapportage van het White Box testen kan daarom beperkter zijn dan bij het Black Box testen. Veelal worden deze fouten ook niet geregistreerd, maar worden ze direct opgelost.
- Bij White Box testen zijn vaak ontwikkelaars aan het testen. De basishouding van een ontwikkelaar is aan te tonen dat het gerealiseerde product werkt, terwijl een tester het verschil tussen de vereiste en actuele kwaliteit van het product wil aantonen (en hiervoor actief op zoek gaat naar fouten). Dit verschil in 'mindset' betekent dat omvangrijk en/of diepgaand White Box testen haaks staat op de basishouding van de ontwikkelaar en daarmee veel weerstand oproept en/of

- resulteert in onzorgvuldig uitgevoerde tests;
- In tegenstelling tot de Black Box tests, maken de White Box tests veelal integraal onderdeel uit van het systeemontwikkelingsproces.
- Omdat White Box testen gebruik maakt van kennis van de interne werking van het systeem, worden andersoortige fouten gevonden dan bij Black Box tests. Van White Box tests kan (en mag) bijvoorbeeld verwacht worden dat elk statement in de programmacode een keer is doorlopen. Een dergelijke dekkinggraad is voor Black Box tests in de praktijk erg moeilijk te bereiken. Het is dus niet goed mogelijk om White Box tests te vervangen door Black Box tests.

2.3 Waarom is Unit test belangrijk?

De basis van elke test is: wat moet het programma c.q. de unit doen en/of wat gebeurt er als het programma iets niet kan uitvoeren. Komt er een nette foutmelding, of blijft het programma hangen op die unit, wordt het programma onverwacht afgesloten, et cetera?

De ontwikkelaar moet dus vooraf bedenken wat dit stuk code moet doen en wat moet er gebeuren als de code en/of een request niet uitgevoerd kan worden. Door duidelijke meldingen en ingebouwde testen kan de goede en foute werking van de unit / class aangetoond worden. Deze testen kunnen bij elke build uitgevoerd worden. Gaat er iets mis dan is makkelijker te traceren waar het mis gaat. Ook moet tijdens de bouw telkens de code (per unit / bean etc.) getest worden; dan kan men ook snel verifiëren of het goed gaat. Het is net als een wandeling in een vreemde stad; je kijkt op het straatnaambordje en een kaart of je wel de goede kant op gaat. Ook voor onderhoud is testen belangrijk; traceerbaarheid is belangrijk.

Delen van een systeem hoeven niet aangepast te zijn maar kunnen ineens niet meer samenwerken met aangepaste delen. Door tests in de code te bouwen komen foutmelding en de locatie waar deze 'fout' zich bevindt sneller naar boven. Het oplossen van fouten is dan geen dagrovende taak, maar wordt een sport. Hoe snel kunnen we systeemdelen weer operationeel krijgen; hoe snel kunnen we aanpassingen goed doorvoeren?

2.4 Basis van Unit test

De basis van de bouw van code moet worden: doordenken, doorvragen wat deze functionaliteit moet doen; wat is het doel van dit deel. De ontwikkelaar moet dus denken wat dit stuk code moet doen; niet alleen om het te bouwen maar ook om te controleren. En wat moet er gebeuren als de unit / code niet uitgevoerd kan worden. Want je doet als ontwikkelaar eigenlijk niet anders dan dat je een bevinding / bug aan het oplossen bent. Wat doet het programma, waarvoor is het bedoeld, waarom werkt het niet en hoe wordt de fout opgelost?

Stappen voor code schrijven met unit test:

- Bedenk een test voor de gevraagde functionaliteit (wat is de bedoeling van de code)
- Let ook op wat er moet gebeuren als het fout gaat
- Schrijf de test
- Schrijf de code
- Voer de test uit (ook voor de foutsituatie).

Daarnaast moet je als ontwikkelaar ook rekening houden met integratie van diverse units. Een unit kan op zich goed werken maar in een volgende fase moeten diverse units wel met elkaar samenwerken. De units en de daarbij behorende tests moeten daarom zo geschreven zijn, dat bij fouten/bevindingen snel op te sporen is waar de fout zich voordoet c.q. waar problemen zijn met de integratie van de verschillende units.

2.5 Testtechnieken

Voor het bouwen van test tijdens Unit tests zijn bepaalde technieken voorhanden. Zo kan men situaties bedenken voor elke if-then statement (branch coverage); beslissingen zoals Ja en Nee (decision coverage) etc. In de onderstaande tabel worden de meest gebruikte technieken voor zowel White Box als Black Box testen genoemd.

White Box testsoorten	Blackbox Testsoorten o.a.
Beslissingstabellentest	Semantische test
Branch Coverage	Syntactische test
Decision Coverage	Proces Cyclus test (voor gebruikers acceptatietest)
Branch decision coverage	Elementaire vergelijkingstest
Grenswaarden analyse	Gegevens cyclus test (CRUD-matrix)
Equivalentieklassen	

Voor White Box testen in het algemeen en Unit test in het bijzonder zijn twee technieken makkelijk toe te passen door de ontwikkelaars:

1. Equivalentieklasse

Een klasse van waarden die bij elkaar horen, bijvoorbeeld omdat ze tot eenzelfde beslissing leiden. Bijvoorbeeld valt een bepaald getal wel of niet binnen de grens, zoals korting die wordt verleend aan mensen onder de 18 en boven de 65 jaar. Dus waarden voor korting opvoeren, zoals een getal tussen de 17 of 66 en voor geen korting een getal tussen 18 en 65.

2. Grenswaardenanalyse

Testtechniek om numerieke waarden te testen met zo weinig mogelijk testgevallen, door slechts enkele waarden rondom het bereik van een numerieke waarde te kiezen. Deze

techniek wordt vaak in combinatie met equivalentieklassen gebruikt. Dus een uitbreiding op bovenstaand voorbeeld en dus ook testen op gelijk aan 18 en 65.

Beide technieken worden uitvoerig beschreven in de bijlage, zie daarvoor hoofdstuk 6.

3 Testen met JUnit

3.1 JUnit Framework

Om het werk voor de ontwikkelaar gemakkelijker te maken zijn er diverse hulpmiddelen om hem/haar de ondersteunen. JUnit [1] is voor de java ontwikkelaar het voornaamste en meest gebruikte framework. Het is gemaakt door Kent Beck en Erich Gamma en stamt af van Beck's sUnit, het unit test framework voor SmallTalk [6].

Met het JUnit framework is het mogelijk om voor individuele java methoden tests te schrijven en uit te voeren. Aan de hand van voorbeelden gaan we laten zien hoe JUnit toegepast kan worden en wat, uit ontwikkelaars oogpunt, voordelen zijn van het unit testen van code.

3.1.1 JUnit versies

Op dit moment zijn JUnit 3.8 en 4 de meest gebruikte versies. JUnit 4 biedt een aantal handige nieuwe functies maar Java 5 is nodig om unit 4 te kunnen gebruiken. Omdat nog niet iedere ontwikkelomgeving JUnit 4 ondersteunt en er daarnaast nog genoeg projecten zijn die nog met Java 1.4 werken, behandelen we hier vooral JUnit 3.8. Tests die geschreven zijn voor JUnit 3.8 werken ook onder JUnit 4 maar niet noodzakelijk andersom.

Op de DevX website over applicatieontwikkeling staat een kort maar duidelijk artikel waarin de nieuwe mogelijkheden van JUnit 4 beschreven worden [24].

3.1.2 Assertions

In een unit test voeren we ontwikkelde code uit en testen we of het resultaat van die aanroep aan bepaalde verwachtingen voldoet. Deze verwachtingen zijn op te stellen door middel van assertions.

Er zijn meerdere verschillende assertions. We kennen assertEquals, assertNotNull, assertNotSame, assertNull, assertSame, assertTrue en assertFalse methoden. De eerste vijf vergelijken twee verschillende objecten of primitieven met elkaar en de laatste twee controleren of iets true of false is.

3.2 Andere frameworks

Naast JUnit zijn er verschillende frameworks die extra mogelijkheden naast JUnit bieden voor specifieke doeleinden. Bijvoorbeeld DbUnit [13], dat het gemakkelijker maakt om binnen een JUnit test een database te gebruiken en te testen. Een aantal van deze aanvullingen zal in een later hoofdstuk worden beschreven.

Een alternatief voor JUnit is TestNG [14]. JUnit en TestNG hebben veel overeenkomsten, zeker sinds JUnit versie 4. TestNG is flexibeler voor grote testSuites, maar JUnit wordt meer gebruikt. Een vergelijking is te vinden op de website van IBM developerWorks [15].

3.3 De praktijk

Hieronder de simpele voorbeeldklasse Voorbeeld1, die maar één methode aanbiedt, namelijk het aangeven of een getal positief is. De methode isPositief geeft een boolean terug die aangeeft of int x een positieve waarde heeft of niet. De methode hebben we nog niet geïmplementeerd, dat doen we pas als we een test voor deze methode hebben geschreven. Deze aanpak wordt test driven development genoemd.

```
public class Voorbeeld1 {  
  
    public static boolean isPositief(int x){  
    }  
}
```

Test driven development (TDD) is een aanpak waarbij de ontwikkelaar eerst unit tests schrijft die de te ontwikkelen functionaliteit uittesten en vervolgens pas ontwikkelt. Hierbij is een ontwikkelaar klaar met ontwikkelen zodra alle tests slagen. De ontwikkelmethode eXtreme Programming (XP) [19] hanteert deze aanpak.

Een voordeel van deze aanpak is dat je als ontwikkelaar gedwongen wordt om eerst op een hoog niveau na te denken over je functionaliteit en daarna pas over de werkelijke implementatie. Daarnaast bouw je niet meer dan nodig is en zorg je automatisch voor een goede testdekking.

Aan de hand van dit eenvoudige voorbeeld laten we zien hoe je een unit test voor deze klasse schrijft. We beginnen met een nieuwe klasse die een extensie is van de klasse TestCase uit het JUnit framework. Over het algemeen geldt dat de testklasse dezelfde naam heeft als de te testen klasse, met Test eraan toegevoegd. Dit is niet noodzakelijk maar wel zo duidelijk. Wij maken dus de testklasse Voorbeeld1Test.

In deze testklasse voegen we twee tests toe. Een test voor het geval dat het getal x wél een positieve waarde heeft en een test waarbij dat niet het geval is. Beide tests zijn van belang bij het testen van de methode. De waarden die we in de tests gebruiken horen namelijk ieder

tot een andere equivalentieklasse (zie 2.5) en zouden ieder tot een ander resultaat moeten leiden.

Als we voor deze klasse een test met JUnit schrijven dan kan deze er zo uit komen te zien.

```
public class Voorbeeld1Test extends TestCase {

    public void testIsPositiefWelPositief(){
        int x = 6;
        boolean out = Voorbeeld1.isPositief(x);
        assertTrue("Uitkomst zou true moeten zijn want 6 is positief",
            out);
    }

    public void testIsPositiefNietPositief(){
        int x = -5;
        boolean out = Voorbeeld1.isPositief(x);
        assertFalse("Uitkomst zou false moeten zijn want -5 is niet
            positief", out);
    }
}
```

De testklasse Voorbeeld1Test is een subklasse van TestCase. Hiermee erft hij alle eigenschappen van een testcase en kunnen de veronderstellingen (assertions) gedaan worden. Deze zijn te zien in het bovenstaande voorbeeld. Bij de eerste test verwachten we dat boolean out de waarde true heeft - *assertTrue*. Bij de tweede test verwachten we dat boolean out de waarde false heeft - *assertFalse*.

Bij het uitvoeren van de unit test wordt gekeken of deze veronderstellingen kloppen. Is dit niet het geval dan wordt de testmethode afgebroken en wordt er een failure gemeld.

Voor iedere type assertion zijn er verschillende aanroepen: assertion-methoden zonder en met een String als eerste argument. In het voorbeeld worden de laatste gebruikt. Op het moment dat een assertion faalt wordt deze String als omschrijving gebruikt voor het foutbericht en gemeld aan degene die de test uitvoert. Het is handig om tijdens het uitvoeren van een test meteen verklarende foutberichten te zien, daarom is het altijd aan te raden om assertions van zo'n bericht te voorzien.

JUnit kan alleen testklassen uitvoeren die een extensie zijn van de TestCase klasse. JUnit herkent de tests binnen de klasse aan het woord test. Bij het uitvoeren van de JUnit testcase worden alle methoden in de klasse die met het woord "test" beginnen herkend en uitgevoerd door het framework. In JUnit 4 in combinatie met Java 5 is het ook mogelijk een testcase aan te geven met de annotation `@test` [7]. In dat geval hoeven de tests niet met het woord 'test' te beginnen.

Kies in elk geval een beschrijvende naam zodat in een oogopslag duidelijk is wat er getest zou moeten worden. Bijvoorbeeld de naam van de methode die je wilt testen, bij meerdere tests van de methode gevolgd door een korte beschrijving. In het bovenstaande

voorbeeld hebben we gekozen hebben voor de methodenamen `testIsPositiefWelPositief` en `testIsPositiefNietPositief`.

Na het schrijven van de test maken we implementatie van de methode af.

```
public class Voorbeeld1 {  
  
    public static boolean isPositief(int x) {  
        return x>=0;  
    }  
}
```

3.4 De ontwikkelomgeving

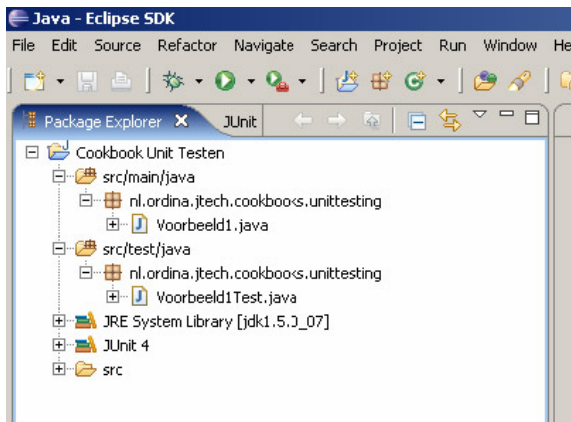
Een ontwikkelomgeving zoals Eclipse [2], NetBeans[25], IntelliJ IDEA[29] maakt het leven van ontwikkelaars een stuk gemakkelijker. Niet alleen door functies als syntax highlighting en code completion, maar ook het automatisch compileren en het snel starten van code. Ook unit tests moeten gemakkelijk en snel uit te voeren zijn binnen je ontwikkelomgeving. De resultaten van de uitgevoerde tests wil je vervolgens meteen op het scherm zien.

In de volgende hoofdstukken laten we zien hoe je met Eclipse (of op een Eclipse gebaseerde omgeving als MyEclipse[26] of IBM RSD[27]) je code kan unit testen. Eclipse is een van de meest gebruikte Integrated Development Environments (IDE).

3.4.1 Eclipse

Als we in de Eclipse ontwikkelomgeving willen werken met JUnit dan moeten we er allereerst voor zorgen dat we in ons project beschikken over het JUnit framework. Als dit niet het geval is kun je het framework toevoegen met (in Eclipse 3.2): rechtermuisknop op project -> Properties -> Add Library...

Ons testproject ziet er in Eclipse als volgt uit.



Hier vallende mappen `src/main/java` en `src/test/java` op. De source van het programma zelf bevindt zich in `src/main` en de unit tests in `src/test`. Voor deze indeling is gekozen om te zorgen dat code gescheiden blijft van de tests en om makkelijker, bijvoorbeeld met een automatische build-tool zoals Maven [3], een jar-build met alleen code te kunnen maken. Een soortgelijke indeling komt vaker voor; bovenstaande indeling is de default voor Maven 2 [8] en de Ordina Smart-Java Ontwikkelstraat [11].

Het uitvoeren van unit tests kan in Eclipse als volgt: rechtermuisknop op testklasse -> Run As -> JUnit Test

3.5 Voordelen van unit tests bij refactoring

In veel gevallen blijkt een stuk code niet goed genoeg. Er kunnen fouten in worden geconstateerd, of een requirement kan veranderen. Zo ook in bovenstaand voorbeeld. Stel dat we ervan uit gingen dat alle $x \geq 0$ positief waren maar we inmiddels te horen hebben gekregen dat nul niet positief én niet negatief is.

Een manier om een fout in de code aan te pakken is het eerst schrijven van een unit test die deze fout naar voren brengt en dan de code aan te passen. Dit is ook een vorm van test driven development. Zo kun je precies zien wanneer de bug is opgelost. Daarnaast zorgt het ervoor dat op het moment dat iemand per ongeluk de bug herintroduceert (dat zou niet de eerste keer zijn) hij bij het uitvoeren van de test meteen op de hoogte wordt gesteld.

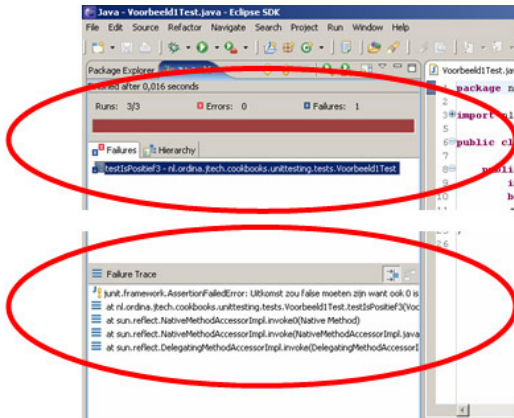
Het verbeteren van de code is in dit voorbeeld slechts een triviale wijziging maar toch gaan we deze aanpak toepassen. We schrijven een derde test. Op basis van grenswaardenanalyse (zie 2.5) hadden we al eerder kunnen besluiten om deze test te schrijven.

Samen met het testen voor de verschillende equivalentieklassen is het testen voor grenswaarden belangrijk om zekerheid te krijgen over het goed functioneren van een (deel)systeem.

```
public void testIsPositiefNul() {  
    int x = 0;  
    boolean out = Voorbeeld1.isPositief(x);  
    assertFalse("Uitkomst zou false moeten zijn want ook 0 is niet  
                positief", out);  
}
```

Als we deze uitvoeren zien we inderdaad dat de code niet goed werkt. Een grote rode balk in Eclipse vertelt ons dat er iets mis is. **Runs: 3/3 Errors: 0 Failures: 1**. Drie van drie tests zijn uitgevoerd waarvan één is gefaald bij een assertion. Als er in plaats van een falende assertion een exception was opgetreden dan hadden we dat onder het kopje Errors gezien.

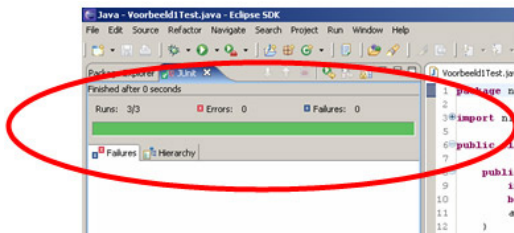
Daaronder zien we de Failure Trace waarin ook de probleemomschrijving die we zelf hebben geschreven wordt vermeld.



Nu passen we de programmacode aan tot:

```
public static boolean isPositief(int x) {  
    return x>0;  
}
```

En we draaien de test opnieuw...



We zien een groene balk. Dat betekent dat de tests geslaagd zijn. Drie testen zijn uitgevoerd en er zijn geen errors of failures opgetreden. De Failure Trace blijft dus leeg.

We weten nu dat het probleem is opgelost. En ook belangrijk: doordat we zien dat alle drie de tests slagen, weten we ook dat we niet per ongeluk iets anders hebben omgegooid toen we de code aanpasten. Dit geeft je als ontwikkelaar meer zekerheid en komt de kwaliteit van de code ten goede.

Hierbij komen we uit op een van de grootste voordelen van het hebben van unit tests. Het herhaaldelijk uitvoeren van alle tests geeft het inzicht of bestaande code nog steeds werkt. En als dit niet het geval is vertelt het testframework waar de problemen zich voordoen. Dit moet zowel een ontwikkelaar als een projectleider een veilig idee geven!

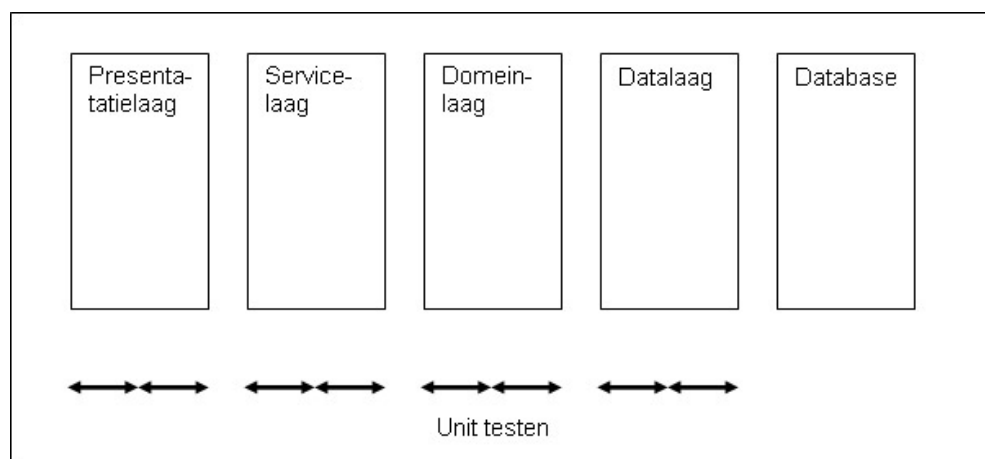
4 Een multi-tier Java EE applicatie testen

We hebben inmiddels kennis gemaakt met het schrijven en uitvoeren van unit tests. Maar de meeste applicaties zitten een stuk ingewikkelder in elkaar dan het genoemde voorbeeld. Een Java EE [12] applicatie bestaat normaal gesproken uit meerdere lagen en is afhankelijk van externe systemen zoals een database.

Dit hoofdstuk beschrijft hoe een JEE applicatie te testen is met JUnit en andere frameworks die daarbij kunnen helpen. Daarnaast geeft het uitleg over de afwegingen die gemaakt moeten worden bij het ontwerpen van de tests. De verschillende lagen van een applicatie, zoals beschreven in Systeemontwikkeling met J2EE [9], komen een voor een aan bod. Hiervoor gebruiken we een simpel issue tracking systeem als voorbeeld.

4.1 Opbouw

De applicatie bestaat uit vier lagen, de presentatielaag, de servicelaag, de domeinlaag en de datalaag. Deze delen van de software kunnen worden uitgevoerd op een JEE applicatieserver. Daarnaast wordt gebruik gemaakt van een database.



4.2 Afzonderlijke units testen

In principe wil je ieder min of meer op zichzelf staand programmaonderdeel (unit of work) testen met een unit test. Dit levert voor iedere laag een aantal unit tests op.

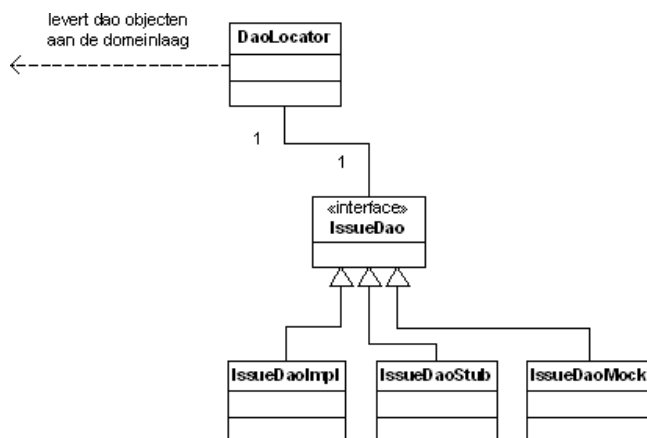
Daarnaast moet ook de integratie tussen de lagen getest worden. Hiervoor schrijf je integratie unit tests. Integratie unit tests zijn niets anders dan unit tests waarbij het aangeroepen programmaonderdeel gebruik maakt van andere onderdelen. Bij enkele unit tests wil je dit juist niet. Er zijn twee manieren om dit aan te pakken. Stubben en Mocken, hieronder verder beschreven.

Soms zijn applicatielagen of onderdelen daarvan zo sterk met elkaar verbonden dat de code moet worden aangepast voordat je met stubs of mocks kunt testen. Het is sowieso niet goed om zo'n verbondenheid tussen programmaonderdelen te hebben. Daarom is het niet erg als je de code aanpast om een betere scheiding te krijgen. In zo'n geval verbeter je namelijk de structuur van de code.

Een manier om programmaonderdelen beter van elkaar te scheiden is door abstractieniveau toe te voegen door het factory design pattern [21] of dependency injection design pattern [20] toe te passen.

In ons voorbeeld van het issue tracking systeem gebruiken we de onderstaande structuur met een locator en data access objects (dao). Deze zorgt voor een scheiding tussen domeinlaag en data laag zodat unit testen van onderdelen van de domeinlaag mogelijk is.

We kunnen de DaoLocator, die dao objecten aan de domeinlaag aanlevert, vertellen dat hij een specifiek soort IssueDao moet gebruiken. Doordat gebruik wordt gemaakt van de IssueDao interface zal de domeinlaag geen kennis hebben van de specifieke IssueDao implementatie en kunnen stub en mock implementaties worden ingezet. In onze unit tests zullen dat de IssueDaoStub en de IssueDaoMock zijn.



4.2.1 Stubs

Stubs zijn stukken code die gebruikt worden om delen van een applicatie af te schermen. Dit kan nodig zijn als een deel nog niet geïmplementeerd is of om een unit test op een deelapplicatie toe te passen. Een stub bevat geen logica en retourneert een vooraf ingestelde waarde.

In ons voorbeeld van het issue tracking systeem: Vanuit de domeinlaag willen we een lijst met verschillende toestanden van een issue (bv. new, in progress, enzovoorts) ophalen uit de database. Dit doen we door de data laag aan te spreken. Als deze functionaliteit nog niet voorhanden is kun je een stub schrijven die eenvoudigweg wat data retourneert waarmee je in de domeinlaag verder kan werken. Zo kunnen unit tests voor de domeinlaag getest worden zonder afhankelijk te zijn van data laag functionaliteit.

```
public class IssueDaoStub implements IssueDao{
    public String[] getIssueTypes(){
        return new String[]{"new", "in progress",
                           "rejected", "testing", "closed"};
    }
    ...
}
```

4.2.2 Mocks

Mock objecten zijn 'namaak'objecten die delen van functionaliteit van het na te bootsen object vervangen. Ook bieden mock objecten de mogelijkheid bij te houden hoe het mock object tijdens een test wordt aangeroepen. Dit zorgt ervoor dat vanuit unit testen of **vanuit het mock object zelf** aanroepen door de te testen code naar de mock kunnen worden gecontroleerd.

Met behulp van mock objecten wordt het mogelijk het gedrag van een ander object te testen op een manier die niet mogelijk is vanuit de unit test zelf. Doordat je de contoles in het mock object inbouwt kun je deze ook gemakkelijk hergebruiken in meerdere unit tests. Dit kan handig zijn, maar zorgt er ook voor dat je tests niet meer alleen in je testklasse beschreven staan.

In ons voorbeeld van het issue tracking systeem: In een unit test voor de domeinlaag willen we controleren of de domeinlaag een aanroep heeft gedaan om een issue op te slaan. Om dit te weten te komen kunnen we de IssueDao mocken. Dit mock object houdt een teller bij voor iedere keer dat de saveIssue methode wordt aangeroepen en genereert een fout als dat te vaak is gebeurt.

```
public class IssueDaoMock implements IssueDao {
    private int numberOfCalls;
    private int maxNumberOfCalls;

    public void saveIssue(Issue issue) {
        numberOfCalls++;
        if (numberOfCalls > maxNumberOfCalls) {
            throw new AssertionError("saveIssue called too often");
        }
    }
    ...
}
```

In de unit test zorgen we ervoor dat de gemoekte IssueDao in de DaoLocator zit zodat het business object uit de domeinlaag (de IssueBO) daarvan gebruikt maakt. Doordat dit business object is aangeroepen doen we een assertion op de numberOfCalls van het mock object.

```
public void testValidateAndSaveIssue() {
    DaoLocator daoLocator = DaoLocator.getInstance();
    Issue issue = new Issue();
    IssueDaoMock mock = new IssueDaoMock();
    daoLocator.setIssueDao(mock);

    IssueBO.validateAndSaveIssue(issue);
}
```

Er zijn verschillende frameworks die het werken met mock objecten vereenvoudigen. Deze nemen het schrijven van mocks uit handen door ze dynamisch te generen. Een voorbeeld hiervan is EasyMock [4].

4.2.3 EasyMock

EasyMock stelt je als ontwikkelaar in staat met weinig moeite mock objecten te gebruiken. Met de MockControl klasse genereert EasyMock runtime deze objecten. Je hoeft zelf geen mockimplementatie te schrijven. In onderstaand voorbeeld voor het issue trackingsysteem is de IssueDaoMock klasse dus niet meer nodig.

In de unit test kun je vervolgens 'expectations' aan een mock object toevoegen. Dit houdt in dat EasyMock controleert of bepaalde aanroepen gedaan worden. Ook return waarden geeft je indien nodig door aan het mock object.

De stappen:

- Eerst roepen we de saveIssue methode aan op het mock object. Hiermee geven we aan dat we één aanroep van de saveIssue methode verwachten.
- Vervolgens voeren we control.replay() uit. Hiermee wordt het mock object op 'scherp' gezet en is het klaar voor gebruik.
- Vervolgens roepen we de te testen code aan: IssueBO.validateAndSaveIssue(issue)
- En doen we een control.verify() die controleert of de verwachte aanroep daadwerkelijk heeft plaatsgevonden.

```
public void testValidateAndSaveWithEasyMock() {
    DaoLocator daoLocator = DaoLocator.getInstance();
    Issue issue = new Issue();
    MockControl control = MockControl.createControl(IssueDao.class);
    IssueDao mock = (IssueDao) control.getMock();
    mock.saveIssue(issue); //verwachte aanroep
    control.replay();
    daoLocator.setIssueDao(mock);

    IssueBO.validateAndSaveIssue(issue);

    control.verify();
}
```

Bovenstaande manier van werken levert veel tijdswinst ten opzichte van het met de hand schrijven van mock objecten. Daarnaast zorgt het ervoor dat een test in een enkele testklasse geschreven wordt, in plaats van deels in de testklasse en deels in het mockobject. Dit maakt de tests makkelijker te overzien. Ook is deze methode minder gevoelig voor fouten na refactoring.

Het wordt aangeraden om een framework te gebruiken voor de creatie van mock objecten. Hiermee wordt snelheidswinst geboekt en kan de ontwikkelaar zich gefocust bezighouden met het schrijven van de werkelijke tests.

4.3 Exceptions

In iedere applicatie krijg je op een gegeven ogenblik te maken met de mogelijkheid dat er exceptions optreden. Belangrijk voor de stabiliteit en betrouwbaarheid van een applicatie is hoe je met deze exceptions omgaat. Vandaar dat we ook unit tests schrijven voor de situaties waarin iets mis gaat.

Ook hier kunnen we mock objecten gebruiken. We passen ons mock object aan zodat deze ook exceptions kan genereren. (Hiervan geen codevoorbeeld. In principe kan dit gewoon een `throw new xxxException()` zijn) Vervolgens schrijven we een test die het foutgedrag controleert. Deze test bevestigt dat er altijd een `BusinessException` optreedt als er iets in de Dao misgaat. Zo wordt er geconroleerd of de foutafhandeling in het business object goed wordt uitgevoerd (en er bijvoorbeeld niet een verkeerde waarde of een null object geretourneerd wordt).

```
public void testValidateAndSaveIssueWithPersistencyException() {
    DaoLocator daoLocator = DaoLocator.getInstance();
    Issue issue = new Issue();
    IssueDaoMock mock = new IssueDaoMock();
    mock.setFailWithPersistencyException();
    daoLocator.setIssueDao(mock);

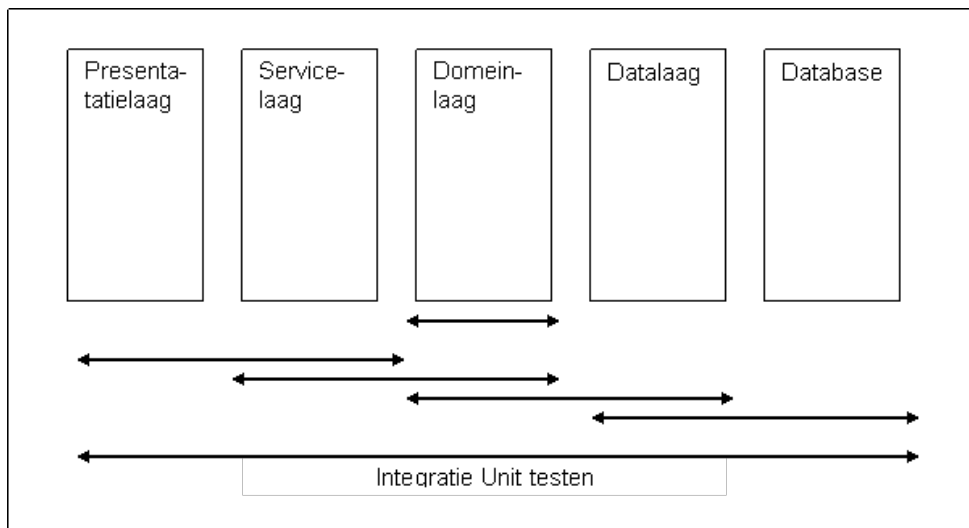
    try{
        IssueBO.validateAndSaveIssue(issue);
        fail("Er is geen exception opgetreden");
    } catch (BusinessException e) {
        assertTrue(true);
    }
}
```

De fail methode die aangeroepen wordt, is een speciale assertion methode van een JUnit TestCase die altijd faalt. Deze gebruik je dus op plaatsen in de test waar je verwacht niet te komen. Als de test er dan toch komt volgt er een foutmelding.

De `assertTrue(true)` aanroep is in principe niet nodig, maar je kunt deze gebruiken om te laten zien dat dit (het optreden van een Exception) inderdaad de bedoeling is van deze test.

4.4 Integratietests

Integratie unit tests zijn zoals al eerder genoemd tests die meerdere units tegelijk omvatten. In deze tests gaat het vooral om de samenwerking tussen de verschillende units, we hebben immers al unit tests die de werking van de afzonderlijke units controleren.



In de bovenstaande afbeelding is te zien hoe integratie unit tests de verschillende lagen en hun verbinding kunnen testen. Er zijn integratie unit tests die binnen een enkele laag verschillende klassen test, er zijn tests die betrekking hebben op twee of meer lagen en er zijn tests te schrijven die het hele systeem testen. Die laatste vorm integratie unit tests wordt ook wel systeemtest of functional unit test genoemd [10].

De database kan in integratie unit tests worden betrokken. Met integratie unit testen wil je controleren of de applicatie als geheel goed werkt. Het testen van de integratie van de data laag met de database draagt hieraan bij. Om dezelfde reden is het ook nuttig om integratie unit tests op een applicatieserver uit te voeren. Het geeft de zekerheid dat een applicatie goed werkt op de applicatieserver.

Tests waarbij een applicatieserver of database betrokken zijn kosten echter veel meer tijd om te schrijven en ook om (eventueel automatisch) uit te voeren. Daarom wordt er vaak niet voor gekozen, maar worden mock technieken gebruikt.

Een soortgelijke afweging kan ook gemaakt worden bij het kiezen tussen meerdere sets integratie unit tests over twee lagen of een enkele set functionele unit tests. De laatste zal ook de integratie tussen alle lagen testen, maar als deze test faalt is het minder duidelijk waar het probleem nu precies zit.

Het is aan te raden om een combinatie van lokaal uit te voeren en op een applicatieserver uit te voeren unit tests te schrijven. Zo kan de ontwikkelaar snel zijn code door de verzameling lokale tests laten controleren, en is ook de werking op de applicatieserver te verifiëren.

De functionele unit tests op de applicatieserver uitvoeren en de overige integratie unit tests lokaal uitvoeren is een goede verdeling. (Mochten er dan problemen optreden in de functionele unit tests die niet duidelijk te vinden zijn, dan kunnen er alsnog meer integratie unit tests die op de applicatieserver werken, geschreven worden.)

4.5 Databases unit testen

Bijna alle applicaties die men ontwikkelt maken gebruik van een database. Het is erg belangrijk dat bijvoorbeeld het wegschrijven of aanpassen van gegevens in de database goed gaat. Daarom willen we ook hier unit tests voor schrijven.

Hier zijn twee manieren van aanpak mogelijk. Als eerste kun je een echte database gebruiken om daar (integratie) tests op los te laten. Als tweede kun je de database mocken en specifiek de toegangslaag naar de database unit testen. Voor beide is iets te zeggen. Het liefst wil je dat je tests de werkelijkheid zo nauwkeurig mogelijk benaderen, maar ook moeten de tests snel uitgevoerd worden zodat je ze tijdens het ontwikkelen gemakkelijk tussendoor op je code kan proberen. Ook is het inrichten van een eigen testdatabase voor iedere ontwikkelaar misschien geen optie en het delen van een ontwikkeldatabase met meerdere testers kan tot botsingen leiden.

4.5.1 DbUnit

Om het uitvoeren van unit tests op een echte database gemakkelijker te maken kan DbUnit [13] gebruikt worden. DbUnit is een uitbreiding op JUnit die, voor het uitvoeren van een unit test, de database in een bepaalde toestand brengt. Op deze manier hoeft je niet iedere keer zelf te zorgen dat de database voordat je de test start met de juiste data gevuld is, en wordt voorkomen dat de data van een vorige testrun de test in de weg zit.

DbUnit biedt de mogelijkheid om gegevens uit een database te exporteren naar een XML bestand en andersom te importeren. Ook kan DbUnit na het uitvoeren van code met speciale assertions controleren of de data in de database overeenkomt met de verwachte data.

De meest eenvoudige manier om de database te laten vullen is door de unit test klasse te laten erven van DBTestCase. In dat geval moet de methode `getDataSet` geïmplementeerd worden. Deze methode levert de data die in de database moet worden geschreven, hier gebruiken we een XML file die de databasevulling beschrijft. Daarnaast moet je aangeven welke database er gebruikt wordt.

Een voorbeeld van een DBUnit testcase en een XML file voor het issue tracking systeem:

```
public class TestIssueDao extends DBTestCase{
    protected IDataset getDataSet() throws Exception {
        return new FlatXmlDataSet(new
            FileInputStream("issuedataset.xml"));
    }
    public void testSaveIssue() {
        //setup
        System.setProperty( PropertiesBasedJdbcDatabaseTester.
            DBUNIT_DRIVER_CLASS, "com.mysql.jdbc.Driver" );
        System.setProperty( PropertiesBasedJdbcDatabaseTester.
            DBUNIT_CONNECTION_URL, "jdbc:mysql://localhost/test" );
        System.setProperty( PropertiesBasedJdbcDatabaseTester.
            DBUNIT_USERNAME, "test" );
        System.setProperty( PropertiesBasedJdbcDatabaseTester.
            DBUNIT_PASSWORD, "test" );
        //test
        ...
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <table name="issue">
    <column>OBJECTID</column>
    <column>DATETIME</column>
    <column>USERID</column>
    <column>STATUS</column>
    <column>REMARKS</column>
    <row>
      <value>1</value>
      <value><![CDATA[2007-01-01 00:00:00.0]]></value>
      <value>testuser</value>
      <value>new</value>
      <value>This is a new issue</value>
    </row>
  </table>
</dataset>
```

4.5.2 MockRunner

De tweede mogelijkheid tot het testen van databaseintegratie is het mocken van de database. MockRunner [5] is hier een handig framework voor. MockRunner biedt de mogelijkheid een MockConnection aan te maken en deze te gebruiken in de test. Hiervoor moet de testcase wel de BasicJDBCTestCaseAdapter extenden.

Hieronder het voorbeeld:

```
private void prepareResultSet()
{
    MockConnection connection =
        getJDBCMockObjectFactory().getMockConnection();
    StatementResultSetHandler statementHandler =
        connection.getStatementResultSetHandler();
    MockResultSet result = statementHandler.createResultSet();
    result.addRow(new String[] { "TEST" });
    statementHandler.prepareGlobalResultSet(result);
}
```

Je kunt MockRunner ook gebruiken in combinatie met MockEJB. Zo is de initialContext te vervangen met een MockContext en daar vervolgens onder een jndi naam een MockDataSource aan te verbinden. Op deze manier communiceert de applicatie niet met de echte database maar met een MockDataSource. De MockDataSource kan geïnstrueerd worden om bij verschillende queries verschillende resultaten te leveren. Zie voor een voorbeeld met MockEJB hoofdstuk 4.5.2.

4.6 EJB's unit testen

Naast het testen van de database krijg je in een JEE applicatie ook te maken met het testen van EJB's. Ook hiervoor geldt dat je ze kunt testen op de applicatieserver, maar dat het makkelijk is om ze lokaal te unit testen. Er moet dus een afweging gemaakt worden of/welke tests op een applicatieserver uitgevoerd moeten worden. Voor het testen binnen de container van een applicatieserver gebruiken we Jakarta Cactus[23]. Voor het testen buiten de container is MockEJB [18] geschikt.

4.6.1 In container testen met Cactus

Met Cactus kun je testcases als servlet op de applicatieserver draaien. De code draait dan in de JVM van de applicatieserver en zo kun je dus ook EJB's waarvoor alleen een local interface beschikbaar is aanroepen.

Testklassen die als servlet gebruikt worden laat je ServletTestCase extenden. Deze servlet deploy je net als andere servlets in de webcontainer van de applicatieserver. Een nadeel van deze methode is dat je testcode op de applicatieserver zult moeten deployen. Als je wilt mocken of stubben moet je ook zorgen dat mock/stub objecten of de libraries van een framework als EasyMock op de server staan.

Om bestaande testcases opnieuw te gebruiken voor in container testen kun je de ServletTestSuite klasse gebruiken waaraan je TestCases of TestSuites toevoegt.

Nu een voorbeeld test. De test wordt lokaal gestart door het uitvoeren van de testcase zoals een gewone testcase. Wel moet de cactus.properties ingesteld staan zodat de testcase weet hoe hij een verbinding met de server kan maken.

Op de server wordt in dit voorbeeld de methode issueServiceGetIssue uitgevoerd.

```
public class TestIssueService extends ServletTestCase
{
    public void testServlet()
    {
        TestIssueService servlet = new TestIssueService();
        String issueId = "10004";
        Issue issue = servlet.issueServiceGetIssue(issueId);
        assertEquals(..., issue);
    }
    public Issue issueServiceGetIssue(String issueId) {
        ...
    }
}
```

Met behulp van de ServletTestCase kun je ook via een webinterface unit tests starten. Andere mogelijkheden zijn via een Ant script of via Maven de tests aanroepen. Op de website van Cactus is een uitgebreide howto te vinden.

4.6.2 MockEJB buiten de container

MockEjb biedt ons de mogelijkheid de container te mocken. Om buiten een applicatieserver om tests uit te voeren op EJB code kunnen we met behulp van MockEJB de InitialContext vervangen door een context binnen een MockContainer. Deze container bootst het gedrag van een echte container na. Op deze MockContainer deployen we te testen beans en/of te mocken beans.

De te testen code (bijvoorbeeld een serviceLocator) zal aan de hand van de jndi naam een bean opvragen en zo de in de mockcontainer geplaatste beans gebruiken. Hier een voorbeeld van de test voor de IssueService:

```
public void testAddIssue() throws Exception {
    MockContextFactory.setAsInitial();
    Context context = new InitialContext();
    MockContainer mockContainer = new MockContainer( context );
    SessionBeanDescriptor sampleServiceDescriptor =
        new SessionBeanDescriptor( "ejb/nl/ordina/jtech/cookbook" +
            "unittesting/service/issueservice",
            IssueServiceHome.class,
            IssueService.class,
            new IssueServiceBean() );
    mockContainer.deploy( sampleServiceDescriptor );

    ...
}
```

5 Testen in de ontwikkelstraat

5.1 Apache Maven2

Maven [3] is een Java-projectmanagementtool met als doel het verkrijgen van een diepgaand inzicht in status, kwaliteit en voortgang van een project. Maven biedt overkoepelende functionaliteit om tal van documentatie, compilatie, packaging en deployment plugins te combineren [9].

Voor het verkrijgen van dit inzicht biedt Maven onder andere de mogelijkheid om unit tests (automatisch) uit te voeren en overzichtelijke rapporten van de resultaten te genereren.

5.1.1 Conventies

Maven werkt volgens het “conventies boven configuratie” concept. Dit betekent in het kort dat als je afwijkt van de conventies die door Maven2 worden aanbevolen, je zult moeten configureren om het geheel werkend te maken. Advies is dus je zoveel mogelijk aan de conventies te houden.

Met betrekking tot tests zijn er de volgende conventies :

- `src/test/java` bevatten alle testcases en classes die noodzakelijk zijn voor het runnen van de testcases.
- `src/test/resources` bevat alle resources die noodzakelijk zijn voor het runnen van de tests, zoals bijvoorbeeld `.xml` files.

Deze scheiding betekent overigens ook meteen dat testclasses en resources niet op het classpath beschikbaar zijn voor de hoofd applicatie.

5.1.2 Gebruikstips

Onderstaande aanwijzingen voor het gebruik van Maven2 hebben betrekking op het uitvoeren van unit tests. Hierbij wordt enige kennis van Maven2 verondersteld. Voor een meer gedetailleerde uitleg over Maven is het (gratis) boek *Better Builds With Maven*[28] aan te raden. Uiteraard biedt de Maven website [3] ook veel informatie.

- Bij het bouwen van een project (zgn. artifact) via bijvoorbeeld het commando **`mvn clean install`**, worden alle unit tests uitgevoerd. Als er in het project tests zijn die falen, dan krijg je een “**build failed**” melding en wordt de artifact niet gebouwd.
 - Als je direct de tests wilt draaien draai je **`mvn clean test`**, waardoor Maven’s testfase wordt gestart. Aan de testfase zit standaard de Maven Surefire plugin gekoppeld, die het daadwerkelijke testen verzorgt.
- Aangezien het iedere keer draaien van alle tests nogal tijdrovend kan zijn, biedt Maven

ook de mogelijkheid om een individuele test te starten d.m.v. **mvn -Dtest=Testclass surefire:test**.

- Er zijn diverse aanpassingen te maken die het gedrag van de uit te voeren tests aanpassen:
 - **-Dmaven.test.failure.ignore=true**
Wordt gebruikt voor het negeren van test failures. Een van de belangrijkste redenen voor het negeren is als je bijvoorbeeld een website wilt maken met een testresultaten rapportage. Als je deze optie niet gebruikt in het geval van test failures, wordt de site nooit gebouwd.
 - **-Dmaven.test.skip=true**
Deze optie wordt gebruikt om het runnen van de tests volledig over te slaan. Dit kan je gebruiken in situaties waarbij je bijvoorbeeld zeer vaak een jar wilt bouwen en de bouwtijd daardoor te beperken.
 - ****/*Test*.java, **/*Test.java, **/*TestCase.java, includes en excludes**
Bovenstaand pattern wordt gebruikt om de testclasses te identificeren. Met een includes blok kan je andere classes specificeren die als classes aangemerkt moeten worden. De excludes is handig te gebruiken voor tests die tijdelijk niet meer functioneel zijn.
Al deze opties en configuraties zijn overigens ook in de pom.xml te verwerken.
- Mocht je dependencies willen definiëren die alleen maar gebruikt worden voor het runnen van je tests, kan je in de dependency de scope aanpassen naar test, zodat deze bij het deployen van je artifact niet als dependency wordt meegenomen naar bijvoorbeeld een productieserver.

5.1.3 Hergebruik test classes

Het is vaak handig om generieke test classes te hergebruiken voor andere projecten of modules. Dit kan heel simpel gedaan worden d.m.v. mvn test:jar, wat een artifact oplevert met de packaging test-jar. Een dependency op zo'n artifact kan heel simpel als volgt worden gezet :

```
<dependency>
  <groupId>nl.ordina.smart</groupId>
  <artifactId>smartutils</artifactId>
  <scope>test</scope>
  <type>test-jar</type>
</dependency>
```

5.2 Ordina Smart-Java ontwikkelstraat


De Smart-Java ontwikkelstraat [11], door Ordina ontwikkeld en een onderdeel van de Ordina Software Factory, leunt sterk op Maven2. De ontwikkelstraat heeft mogelijkheden om verschillende inzichtelijke rapporten te genereren, waaronder slagingspercentage unit testen en code coverage rapporten.

De ontwikkelstraat levert deze functionaliteit door gebruik te maken van code coverage tools (zie 5.3).

Hieronder een voorbeeld van een project dashboard zoals die door de Smart-Java ontwikkelstraat wordt gegenereerd. In het voorbeeld de voortgang van het project te volgen aan de hand van de gerealiseerde functiepunten. Ook is te zien dat de laatste tijd alle unit tests slagen en dat de code coverage op een acceptabel niveau is.

Metrieken

- Voortgang
- Dashboard**
- Project documentatie
- Project informatie
- Project rapporten
- Project
- Submodules
- Documentatie
- Project admin portal
- Maven project repo
- Ordina
- Software Factory
- RUP op maat
- J-Technologies
- Ordina

Built by 

Dashboard

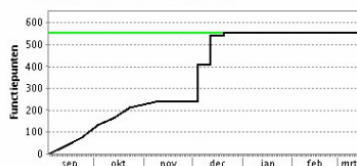
Startdatum: 1 aug 2006
 Eindopleving: 1 jan 2007
 Projectleider: Jan Jaap Zijlstra
 Release: 1.0

Project metriek	Totaal	Target
Gerealiseerde functionaliteit (in FP)	553	553
Openstaande bevindingen	38	

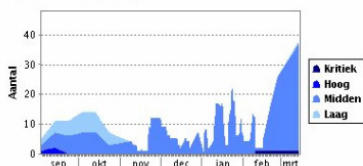
Source code metriek	Totaal	Target	OK
Lines Of Code (LOC) totaal	24145		
LOC per functiepunt gemiddeld	44		
Unit test failures (van totaal aantal)	0 (van 453)	0	✓
Coverage	59%	50%	✓
Code standaard overtredingen (per KLOC)	16	30	✓
Foutgevoelige code (aantal meldingen)	0	0	✓

Project Metrieken

Voortgang (Gerealiseerde functiepunten)

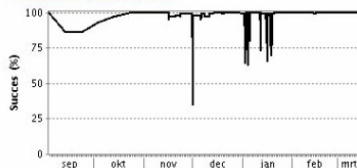


Openstaande bevindingen

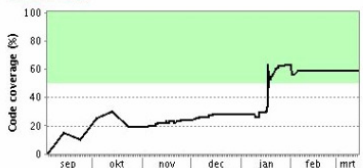


Source code metrieken

Slagingspercentage unit testen



Code coverage



6 Code coverage

Er zijn verscheidene producten op de markt die door middel van het uitvoeren van je unit tests kunnen berekenen welke code er getest wordt en welke code er nog niet door de unit tests geraakt wordt. Deze rapportage kan in een Maven projectportaal beschikbaar worden gesteld, maar is ook buiten een ontwikkelstraat te gebruiken.

Coverage rapportages kunnen waardevolle informatie geven. Zo kun je snel het bereik van je tests zien. Het is verstandig om een coverage van op zijn minst 60% na te streven.

Clover [16] is een veel gebruikt test coverage tool. Een gratis en open source alternatief voor Clover is Cobertura [22]. Beide tools geven je uitgebreide rapporten die op regelnummer niveau aan kunnen geven wat wel en niet getest wordt. Dit is waardevolle informatie die je kan helpen de tests te verbeteren.

Ook laten ze per project, per package en per klasse aan welk percentage regels en branches met de tests bereikt is. Hieronder een voorbeeldrapport van Cobertura:

Coverage Report - All Packages

Package /	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	46	70% 1308/1856	79% 213/270	2.458
net.sourceforge.cobertura.ant	10	48% 192/314	60% 26/43	1.871
net.sourceforge.cobertura.check	3	0% 0/150	0% 0/27	2.429
net.sourceforge.cobertura.coveragedata	10	N/A	N/A	2.239
net.sourceforge.cobertura.instrument	6	82% 143/174	91% 43/45	2.538
net.sourceforge.cobertura.merge	1	86% 30/35	100% 8/8	5.5
net.sourceforge.cobertura.reporting	3	86% 119/134	100% 25/25	2.882
net.sourceforge.cobertura.reporting.html	4	88% 446/508	96% 69/72	4.308
net.sourceforge.cobertura.reporting.html.files	1	87% 39/45	100% 4/4	4.5
net.sourceforge.cobertura.reporting.xml	1	100% 122/122	100% 8/8	1.421
net.sourceforge.cobertura.util	8	63% 156/246	84% 32/38	3.067
someotherpackage	1	83% 5/6	N/A N/A	1.2

Report generated by [Cobertura](#) 1.8 on 5/9/06 4:23 PM.

Aan de hand van zulke rapporten kun je uitstekend je uitstekend ontdekken welke delen van de code minder goed getest worden en kun je zien of je tests verbeterd moeten worden. Op een hoger niveau is in te schatten of een bepaald percentage van coverage gehaald wordt.

Met behulp van code coverage rapporten is de kwaliteit van unit tests te verbeteren. Er is snel te zien of uitzonderlijke gevallen en foutsituaties getest worden. Als dit niet het geval is, is het belangrijk hier extra tests te schrijven. Dit zijn juist de gevallen die je met (White Box) unit tests kunt bereiken en die voor een (Black Box) tester veel lastiger te testen zijn.

Hoewel code coverage een zeer nuttig hulpmiddel is bij het goed unit testen, zegt het coverage percentage ook niet alles over de kwaliteit van de tests. Tests die wel veel code uitvoeren, maar weinig controleren zullen een hoge coverage geven maar zijn niet altijd toereikend.

6.1 Plugins

Voor Clover zijn voor alle grote IDE's (Eclipse, NetBeans, IntelliJ, JBuilder, JDeveloper) plugins beschikbaar. Ook leveren Clover en Cobertura Ant scripts om de coverage rapporten te genereren.

7 Bijlagen

7.1 Principe van afleiden testgevallen

Er zijn verschillende principes waarvan de testspecificatietechnieken gebruik maken om de testgevallen af te leiden. Voor unittesten kunnen o.a. de volgende technieken worden gebruikt:

- Equivalentieklasse
- Grenswaardenanalyse (verdieping van equivalentieklasse)
- Verwerkingslogica / beslissingstabellentest, met diverse soorten diepgang
 - decision coverage
 - condition coverage
 - multiple condition coverage
 - decision/condition coverage condition

Hieronder worden de Equivalentieklasse en grenswaardenanalyse kort toegelicht. Daarnaast wordt hieronder uitgebreid stil gestaan bij de beslissingstabellen technieken. Wat is het en hoe kan het gebruikt worden. Beslissingstabellentechnieken gaat vaak om het testen van diverse If Then statements en de diepgang daarin.

7.2 Equivalentieklassen

Bij dit principe van het afleiden van testgevallen wordt onderzocht welke klassen van mogelijke invoerwaarden tot een zelfde soort verwerking leiden. Deze klassen worden equivalentieklassen genoemd (Engelse term: equivalence partitioning). Een andere term waaronder het afleiden met equivalentieklassen bekend staat is domain testing, hier wordt onderscheid gemaakt tussen geldige en ongeldige equivalentieklassen. Invoerwaarden uit de ongeldige equivalentieklasse leiden tot foutmeldingen. Invoerwaarden uit een geldige equivalentieklasse worden correct verwerkt. Door testgevallen te baseren op deze equivalentieklassen in plaats van op elke mogelijke invoerwaarde, blijft het aantal testgevallen beperkt, terwijl toch een goede dekking verkregen wordt.

Ter verduidelijking het volgende voorbeeld, waarbij de gegeven 'leeftijd' bij invoer aan de volgende controle wordt onderworpen:

$18 < \text{leeftijd} \leq 65$,

Voor leeftijd zijn nu 3 equivalentieklassen te onderscheiden, namelijk:

- a) leeftijd ≤ 18
- b) leeftijd heeft een waarde in de verzameling 19 t/m 65
- c) leeftijd > 65

Voor leeftijd te kiezen testgevallen zijn dan bijvoorbeeld 10 (ongeldig), 35 (geldig) en 70 (ongeldig).

Overigens moet opgemerkt worden dat testspecificatietechnieken die testgevallen afleiden op basis van de verwerkingslogica vaak impliciet ook equivalentieklassen hanteren.

Equivalentieklassering (als techniek volgens ISEB)	
Karakteristiek	WB en BB, formeel
Testsoort	Alle testsoorten, vooral integratie- en systeemtests
Testbasis	Alle soorten testbasis
Principe van afleiden	Klassen elk leidend tot andere verwerking.
Coverage	Condition coverage, uitbreidbaar naar decision coverage wanneer ook op de output equivalentieklassering wordt toegepast.
Variëren diepgang	<ul style="list-style-type: none"> • Meerdere waarden meetesten
Meting van dekkingsgraad	De dekkingsgraad kan worden berekend als: $\text{Dekkingsgraad} = \frac{\text{Aantal geraakte eq. klassen}}{\text{Totaal aantal eq. klassen}}$
Toepassing	Alle typen systemen
Kwaliteitseigenschap	Functionaliteit, connectiviteit

7.3 Grenswaardenanalyse

Een belangrijke aanscherping van het bovenstaande principe is grens' waardeanalyse (Engelse term: 'boundary value analysis'). De waarden die op en om de grenzen van een equivalentieklasse liggen noemen we de grenswaarde. Dit zijn waarden waar in de praktijk veel fouten gevonden worden. Bij het bepalen van de testgevallen worden waarden gekozen, rondom deze grenzen, dus elke grens wordt getest met minimaal twee testgevallen. Eén testgeval waarbij de invoerwaarde gelijk is aan de grens' en één daar net overheen. Toepassing van grenswaardenanalyse leidt tot meer testgevallen, maar vergroot de fout vindkans van de test ten opzichte van een willekeurige keuze uit de equivalentieklasse.

Voorbeeld: $18 < \text{leeftijd} \leq 65$

Voor leeftijd te kiezen grenswaarden zijn dan 18 (ongeldig), 19 (geldig), 65 (geldig) en 66 (ongeldig).

Mogelijke testwaarden zijn ook:

1. 10 (ongeldig)
2. 30 (geldig)
3. 70 (ongeldig)

Grenswaardenanalyse omvat overigens niet alleen de grenswaarden aan de invoerkant, maar ook aan de uitvoerkant. Stel dat een offertepagina maximaal 10 regels mag bevatten. Dan wordt dit getest door een offerte af te drukken met 10 (alle regels op één pagina) en 11 regels (elfde regel op de tweede pagina).

Een mogelijke verzwarende grenswaardenanalyse is nog dat bij een grens drie in plaats van twee waarden moeten worden gekozen. De extra waarde ligt dan net vóór de uiterste toegestane grens.

Voorbeeld: leeftijd ≤ 18

Voor leeftijd te kiezen grenswaarden zijn dan 17 (geldig), 18 (geldig) en 19 (ongeldig). Als deze vergelijking foutief 'leeftijd = 18' is geprogrammeerd, wordt dit met twee grenswaarden niet gevonden, maar wel met de extra waarde 17.

Grenswaarde-analyse (als techniek volgens ISEB)	
Karakteristiek	WB en BB, formeel
Testsoort	Alle testsoorten, vooral unittests
Testbasis	Alle soorten testbasis
Principe van afleiden	Klassen elk leidend tot andere verwerking met speciale aandacht op grenswaarden van elke van de klassen.
Coverage	Condition coverage
Variëren diepgang	Aantal te kiezen grenswaarden
Meting van dekkingsgraad	De dekkingsgraad kan worden berekend als: <div style="text-align: right; margin-right: 50px;"> Aantal unieke grenswaarden <hr style="width: 200px; margin: 0;"/> Totaal aantal grenswaarden </div> uitgevoerd Dekkingsgraad =
Toepassing	Alle typen systemen vooral testen van componenten met numerieke equivalentieklassen.
Kwaliteitseigenschap	Functionaliteit

7.4 Operationeel gebruik

Testgevallen kunnen ook worden afgeleid op basis van het verwachte gebruik van het systeem in de praktijk. De testgevallen simuleren als het ware de verschillende situaties zoals die in productie (zullen) voorkomen. Dit betekent bijvoorbeeld dat voor functies die in de praktijk heel veel gebruikt worden, evenredig veel testgevallen gespecificeerd worden, ongeacht de complexiteit of het belang van de functie. Veelal resulteert het testen op basis van operationeel gebruik in een groot aantal testgevallen die alle tot dezelfde equivalentieklasse behoren en derhalve geen grote kans hebben om nieuwe fouten te vinden. Operationeel gebruik wordt vaak toegepast bij het uitvoeren van performance tests.

7.5 Beslissingstabellentest / verwerkingslogica

7.5.1 Algemeen

De beslissingstabellentest is een formele testspecificatietechniek. Dit betekent dat de testgevallen volgens vaste regels uit de testbasis worden afgeleid. De beslissingstabellentest is afhankelijk van twee parameters. De ene parameter (testmaat) geeft aan in hoeverre afhankelijkheden tussen verschillende 'beslispunten' worden getest, de andere parameter (detailleringsmaat) geeft aan in hoeverre afhankelijkheden binnen 'beslispunten' worden getest, d.w.z. wat is de diepgang van het testen.. Afhankelijk van de keuze van deze parameters levert deze techniek veel of weinig testgevallen en een hogere of lagere dekingsgraad op. Een sterk punt van de beslissingstabellentechniek is dat met behulp van de parameters de test kan worden afgestemd op de teststrategie. Zodra de parameters zijn bepaald, ligt de werkwijze voor de beslissingstabellentest eenduidig vast. De beslissingstabellentest richt zich op de volledigheid en juistheid van de verwerking en daarmee op het kwaliteitsattribuut functionaliteit. Deze techniek is in principe ontwikkeld voor het white box testen, maar kan tevens worden toegepast bij een black box test.

7.5.2 Werkwijze

De stappen die ondernomen moeten worden om tot een beslissingstabellentest te komen zijn:

- Definieren logische testkolommen;
- Specificeren logische testgevallen;
- Specificeren fysieke testgevallen;
- Vaststellen initiële gegevensverzameling;
- Opstellen testscript.

7.5.3 Definieren logische testkolommen

Tijdens deze stap worden van ieder proces een of meer beslissingstabellen vervaardigd. Een beslissingstabel wordt gebruikt om de logische verwerking van een proces eenduidig weer te geven en vormt voor de tester de schakel tussen de testbasis en de logische testgevallen.

Het doel van het vervaardigen van de beslissingstabellen is drieledig:

- het expliciet vastleggen van de logische verwerking van de processen, zodat er tijdens volgende fasen geen onduidelijkheden optreden;
- het vaststellen van eventuele onvolkomenheden in de testbasis (bijvoorbeeld onvolledig beschreven situaties);
- het verkrijgen van een uitgangssituatie voor de logische testgevallen.

Het vervaardigen van de logische testkolommen valt in een drietal onderdelen uiteen:

- Destilleren 'triggers';
- Bepalen determinanten;
- Vervaardigen beslissingstabellen.

Ter verduidelijking wordt een voorbeeld gebruikt waarbij sprake is van een proces (taak) dat zich bezighoudt met de berekening van het uit te betalen salaris. Het proces wordt opgestart als onderdeel van de maandverwerking.

De persoon voor wie salaris wordt berekend, wordt geïdentificeerd door middel van een nummer, het salarisprogramma controleert of het nummer een geldig nummer is, hetgeen betekent dat de persoon in dienst is. Indien er sprake is van een niet geldig nummer genereert het systeem een foutmelding. (1)

De salarisberekening is als volgt:

ALS gewerkt = J EN aantal jaren > 10 (2)

DAN salaris := 10.000

ANDERS salaris := 5.000

ALS (getrouwd = J OF samenwonen = J) EN land = "Nederland" (3)

DAN salaris := salaris + 5.000

ANDERS ALS leeftijd > 18 EN leeftijd < 60 (4)

DAN salaris := salaris + 4.000

ANDERS salaris := salaris + 3.000

ALS gewerkt = J EN afdeling = "productie" EN leeftijd > 18 (5)

DAN salaris := salaris + 1.000

7.5.3.1 Destilleren 'triggers'

In deze eerste deelstap moeten de triggers (de activiteiten die het proces op gang brengen) worden opgespoord. Een proces kan worden 'getriggered' door een gebruiker, door een extern systeem of door een interne functie. Elk proces heeft minimaal één 'trigger', maar kan er meerdere hebben. In het voorbeeld is sprake van één 'trigger': de maandverwerking.

7.5.3.2 Bepalen determinanten

Tijdens deze tweede deelstap worden de determinanten bepaald. Dit zijn de kenmerken die van invloed zijn op de verwerking van het proces. Hierbij dient zowel aandacht te worden gegeven aan de kenmerken van de trigger als aan de kenmerken van de verwerking.

Een determinant kan zowel negatief als positief zijn. Een determinant is negatief als het proces altijd een foutmelding genereert zodra de determinant waar is. Een determinant is positief als die niet negatief is. Tijdens deze deelstap worden de determinanten opgespoord en wordt vastgesteld of deze negatief of positief zijn.

In het voorbeeld salarisberekening is sprake van een vijftal determinanten:

- | | | |
|-----------|---|--------------------------------|
| 0. | Maandverwerking | (trigger) |
| 1. | Numerum persoon ontbreekt | (negatieve determinant) |
| 2. | Indicatie gewerkt/aantal dienstjaren | (positieve determinant) |
| 3. | Burgerlijke staat/Land | (positieve determinant) |
| 4. | Leeftijd | (positieve determinant) |
| 5. | Gewerkt/Leeftijd/Afdeling | (positieve determinant) |

7.5.3.3 Vervaardigen beslissingstabellen

Identificatie tabel				
Logische testkolom	1	2	..	n
Trigger	1	1	..	1
Determinant 1	0	1	..	1
Determinant 2
Determinant 3
Determinant 4
Resultaat 1	x	nm
Resultaat 2

Tijdens de derde deelstap worden beslissingstabellen gemaakt op basis van de determinanten. De algemene vorm van een beslissingstabel is als volgt:

- Boven de dubbele streep bevatten alle cellen de waarden '0', '1' of ' '. De waarde '1' betekent dat de determinant waar is, de waarde '0' betekent dat de determinant onwaar is en de waarde ' ' tenslotte betekent dat de waarde van de determinant niet van belang is. De 'trigger' die van toepassing is, heeft altijd de waarde '1', de determinanten kunnen de waarde '0', '1' of ' ' hebben.
- Onder de dubbele streep bevatten alle cellen een 'X', een 'nm' of ze zijn leeg. Als er een 'X' staat treedt het betreffende resultaat op in die situatie, als een cel leeg is treedt het betreffende resultaat niet op in die situatie. Als er een 'nm' staat, is de betreffende situatie niet mogelijk, bijvoorbeeld omdat bepaalde waarden van determinanten elkaar uitsluiten.
- Iedere kolom van de beslissingstabel vormt een logische testkolom. Het deel boven de dubbele streep vormt de 'trigger' en de situatiebeschrijving en het deel onder de streep het gevolg c.q. de resultaten. In een beslissingstabel staat dus voor de verschillende situaties beschreven wat het gespecificeerde resultaat is als het proces een beslissing moet nemen op basis van een 'trigger'. De keuze van het aantal situaties dat wordt beschreven, is bepaald door de teststrategie.

•

7.5.3.4 Combinaties van determinanten

Voor de keuze van het aantal situaties worden de volgende mogelijkheden onderscheiden:

- alle combinaties van determinanten, dus alle mogelijke situaties worden getest (meerdere als – 2^n statements);
- het minimum aantal situaties wordt getest, waarbij iedere determinant in ieder geval één keer 0 en één keer 1 is;
- het aantal situaties dat wordt getest ligt tussen de hiervoor genoemde uiterste waarden.

De bovenstaande wijze van combineren wordt slechts toegepast bij positieve determinanten. De negatieve determinanten worden niet in combinatie met elkaar getest. Eén voor één dienen de negatieve determinanten de waarde waar aan te nemen. Tijdens de overige tests zijn deze determinanten dan onwaar.

Het is vrij eenvoudig na te gaan dat alle mogelijke situaties van de determinanten in de tabel staan vermeld. Indien dit het geval is betekent dit een 100% dekkinggraad op logisch specificatieniveau, met als nadeel het grote aantal logische testkolommen en dus testgevallen. De inspanning die moet worden verricht om de test uit te voeren is derhalve groot.

Voor processen met een lager belang, moet een andere werkwijze worden gevolgd, op basis waarvan het aantal logische testkolommen beperkt blijft en tegelijkertijd toch een redelijke dekkinggraad wordt bereikt. Om hiervoor een oplossing te bieden, wordt de parameter testmaat geïntroduceerd.

Een beslissingstabel heeft de testmaat 'n' als de tabel voor iedere groep van 'n' determinanten alle mogelijke combinaties bevat. Met andere woorden, als een beslissingstabel testmaat 'n' heeft, zijn er voor iedere groep van 'n' determinanten 2^n situaties in de beslissingstabel.

Het is mogelijk om het aantal logische testkolommen te berekenen per testmaat bij gegeven aantallen positieve determinanten en daaruit vervolgens de bereikte dekkinggraad. Hiermee is het dus mogelijk om bij een gegeven aantal determinanten een testmaat te kiezen die enerzijds het best past bij de gewenste dekkinggraad op basis van de teststrategie en anderzijds de begrote testinspanning niet overstijgt. Hierna zijn de resultaten van dergelijke berekeningen weergegeven in een tweetal tabellen voor processen met maximaal 10 positieve determinanten.

In de eerste tabel staat het aantal logische testkolommen per testmaat bij gegeven aantal positieve determinanten:

Testmaat:	1	2	3	4	5	6	7	8	9	10
1 determinant	2	2	2	2	2	2	2	2	2	2
2 determinanten	2	4	4	4	4	4	4	4	4	4
3 determinanten	2	4	8	8	8	8	8	8	8	8
4 determinanten	2	5	8	16	16	16	16	16	16	16
5 determinanten	2	6	10	16	32	32	32	32	32	32
6 determinanten	2	7	12	22	32	64	64	64	64	64
7 determinanten	2	8	14	29	49	64	128	128	128	128

8 determinanten	2	9	16	37	72	107	128	256	256	256
9 determinanten	2	10	18	46	102	172	228	256	512	512
10 determinanten	2	11	20	56	140	266	392	476	512	1024

Hierboven staat dan ook een overzicht van het aantal testgevallen bij uitvoering van alle varianten in meerdere als-dan niveaus.

In de volgende tabel staat de dekkinggraad (in percentage) per testmaat bij een gegeven aantal positieve determinanten:

Testmaat:	1	2	3	4	5	6	7	8	9	10
1 determinant	100	100	100	100	100	100	100	100	100	100
2 determinanten	50	100	100	100	100	100	100	100	100	100
3 determinanten	25	50	100	100	100	100	100	100	100	100
4 determinanten	13	31	50	100	100	100	100	100	100	100
5 determinanten	6	19	31	50	100	100	100	100	100	100
6 determinanten	3	11	19	34	50	100	100	100	100	100
7 determinanten	2	6	11	23	38	50	100	100	100	100
8 determinanten	1	4	6	14	28	42	50	100	100	100
9 determinanten	0,5	2	4	9	20	34	45	50	100	100
10 determinanten	0,2	1	2	5	14	26	38	46	50	100

Met behulp van deze tabellen kan de juiste keuze ten aanzien van de testmaat worden gemaakt, waarna de beslissingstabellen kunnen worden vervaardigd.

7.5.4 Specificeren logische testgevallen

In een beslissingstabel staan logische testkolommen. Iedere logische testkolom geeft aan welke resultaten er moeten optreden indien de determinanten de waarden hebben zoals gespecificeerd in de kolom. Tijdens de test moet worden geverifieerd of die resultaten ook daadwerkelijk optreden in de betreffende situatie.

Een logische testkolom lijkt derhalve veel op een logisch testgeval. De beslissingstabellentest techniek stelt als eis aan de logische testgevallen dat deze eenduidig zijn gedefinieerd, hetgeen bij de logische testkolommen niet het geval is. Dit komt omdat determinanten uit de logische testkolommen samengesteld kunnen zijn. Als een determinant bestaat uit meerdere enkelvoudige determinanten (bijvoorbeeld $X > 0$ EN $Y = 3$) kan de samengestelde determinant op meerdere manieren waar of onwaar zijn. De situatie ligt derhalve niet eenduidig vast.

7.5.4.1 Detailleringsmaat

Zoals de testmaat aangeeft in hoeverre combinaties tussen verschillende determinanten worden getest, geeft de detailleringsmaat aan in hoeverre combinaties van enkelvoudige determinanten binnen een samengestelde determinant worden getest.

Voor het nader detailleren van een samengestelde determinant bestaan verschillende detailleringsmaten. Ook hierbij geldt dat indien gekozen wordt voor een hoge mate van detaillering (hoge detailleringsmaat) dit leidt tot een relatief groot aantal testgevallen en dus een relatief hoge testinspanning. Het belang van het betreffende proces is ook hier weer doorslaggevend voor de detailleringsmaat die wordt gekozen.

De volgende detailleringsmaten worden onderkend:

- multiple condition coverage
- decision coverage
- condition coverage
- decision/condition coverage
- condition/determination coverage.

De betekenis en werking van de verschillende detailleringsmaten wordt toegelicht aan de hand van een voorbeeld:

'V is een samengestelde determinant die bestaat uit de enkelvoudige determinanten A, B en C ($V = A$ EN $(B$ OF $C)$)'. De samengestelde determinant V heeft de onderstaande waarheidstabel:

7.5.4.1.1 Multiple condition coverage

Nummer	A	B	C	V
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	0
5	1	1	0	1
6	1	1	1	1
7	1	0	0	0
8	1	0	1	1

Bovenstaande tabel geeft de eerste detailleringsmaat weer, de '**multiple condition coverage**'. Bij de 'multiple condition coverage' worden alle mogelijke combinaties binnen een samengestelde determinant uitgetest.

7.5.4.1.2 Decision coverage / branch coverage

De tweede detailleringsmaat is de '**decision coverage**' ook wel **branch coverage genoemd**. Deze detailleringsmaat test de juistheid van de samengestelde determinant door de samengestelde determinant als geheel één keer waar en één keer onwaar te laten zijn. Dit kan bijvoorbeeld de volgende waarheidslabel opleveren:

Nummer	A	B	C	V
1	0	1	1	0
2	1	1	1	1

De detailleringsmaat 'decision coverage' levert altijd slechts twee testgevallen op. De dekkingsgraad bij het toepassen van deze detailleringsmaat is uiteraard laag. Als de bovenstaande testgevallen zouden worden toegepast en er in plaats van $V = A \text{ EN } (B \text{ OF } C)$, $V = A \text{ EN } B$ zou zijn geïmplementeerd, wordt deze fout niet gevonden.

7.5.4.1.3 Condition coverage

De derde detailleringsmaat is de '**condition coverage**'. Deze detailleringsmaat test de juistheid van de samengestelde determinant door iedere enkelvoudige determinant één keer waar en één keer onwaar te laten zijn. In het voorbeeld kan dit de volgende tabel opleveren:

Nummer	A	B	C	V
1	0	1	1	0
2	1	0	0	0

Een belangrijk verschil met de vorige uitwerking is, dat het bij de 'condition coverage' mogelijk is voor de verschillende uitwerkingen dezelfde waarde van de samengestelde determinant te krijgen. Ook deze detailleringsmaat levert slechts twee testgevallen met eveneens een lage dekkinggraad. Indien de bovenstaande testgevallen zouden worden toegepast en er in plaats van $V = A \text{ EN } (B \text{ OF } C)$, $V = A \text{ EN } B \text{ EN } C$ zou zijn geïmplementeerd, wordt deze fout niet gevonden.

7.5.4.1.4 Decision/condition coverage

De vierde detailleringsmaat is de 'decision/condition coverage'. Deze detailleringsmaat test de juistheid van de samengestelde determinant door de samengestelde determinant één keer waar en één keer onwaar te laten zijn, onder de voorwaarde dat iedere enkelvoudige determinant eveneens één keer waar en één keer onwaar is. In het voorbeeld zou dit de volgende testgevallen kunnen opleveren:

Nummer	A	B	C	V
1	1	1	1	1
2	0	0	0	0

Evenals beide voorgaande detailleringsmaten levert ook deze detailleringsmaat slechts twee testgevallen en is de dekkinggraad wederom erg laag. Indien de bovenstaande testgevallen zouden worden toegepast en er in plaats van $V = A \text{ EN } (B \text{ OF } C)$, $V = (A \text{ OF } B) \text{ EN } C$ zou zijn geïmplementeerd, wordt deze fout niet gevonden.

7.5.4.1.5 Condition/determination coverage

De vijfde detailleringsmaat tenslotte is de '**condition/determination coverage**'. Deze detailleringsmaat test de juistheid van de samengestelde determinant door iedere enkelvoudige determinant twee keer de waarde van de samengestelde determinant te laten bepalen. Eén keer bepaalt de enkelvoudige determinant dat de samengestelde determinant waar is en één keer bepaalt de enkelvoudige determinant dat de samengestelde determinant onwaar is. In het voorbeeld kan dit de volgende testgevallen opleveren:

Nummer	A	B	C	V
1 (A is bepalend)	0	0	0	0
2 (B en C zijn bepalend)	0	0	1	0
3 (A en C zijn bepalend)	0	0	0	0
4 (A en B zijn bepalend)	0	0	1	0

Deze detailleringsmaat levert hoogstens ' $(3/2 * n) + 1$ ' testgevallen op, waarbij n het aantal enkelvoudige determinanten is. De dekkinggraad van deze detailleringsmaat is relatief hoog met een relatief beperkt aantal testgevallen. Het is zelfs moeilijk een voorbeeld te geven van een onjuiste implementatie van enkelvoudige determinanten in de samengestelde determinant

die door de voorgaande testgevallen niet wordt gevonden.

Als alle samengestelde determinanten volgens de geselecteerde detailleringsmaat zijn beschreven, zijn er twee mogelijkheden om de logische testkolommen in logische testgevallen te transformeren, namelijk **splitsen en onderbrengen**.

Bij het splitsen wordt iedere logische testkolom in meerdere logische testgevallen gesplitst. Het voordeel hiervan is dat bij een eventuele testbevinding de oorzaak snel kan worden achterhaald. Nadelig is echter dat het aantal logische testgevallen toeneemt. Bij het onderbrengen worden de verschillende testgevallen die door het detailleren van de samengestelde determinanten ontstaan, ondergebracht binnen de logische testkolommen. Het voordeel hiervan is dat dit een 1:1 transformatie oplevert (één logische testkolom wordt getransformeerd in één logisch testgeval), hetgeen weinig inspanning kost. Het nadeel hiervan is echter dat bij een eventuele testbevinding niet altijd direct duidelijk is waar de oorzaak ligt. In het algemeen wordt bij het transformeren van logische testkolommen naar logische testgevallen gekozen voor het onderbrengen. Alleen in uitzonderlijke gevallen (een zeer groot belang c.q. risico) verdient het splitsen de voorkeur. Als detailleringsmaat wordt meestal de maximale detailleringsmaat gekozen die nog kan worden ondergebracht in de bestaande logische testkolommen. Hierdoor blijft het aantal logische testgevallen, en dus tevens de testinspanning, beperkt.

De laatste stap die met betrekking tot het voorbeeld 'salarisberekening' is uitgewerkt is het transformeren van de logische testkolommen naar logische testgevallen. Het onderbrengen van de eerdere beschreven logische testkolommen in logische testgevallen heeft het volgende resultaat:

Salarisberekening					
Logisch testgeval	1	2	3	4	5
Trigger	1	1	1	1	1
Det. 1 nummer persoon	1	0	0	0	0
Det. 2 gewerkt = J	1	1	1	1	1
Det. 2 aantal jaren > 10	0	1	1	0	1
Det. 3 getrouwd = J	0	0	1	1	1
Det. 3 samenwonen = J	1	1	0	0	0
Det. 3 land = 'Nederland'	0	0	1	1	1
Det. 4 leeftijd > 18	1	0	-	-	-
Det. 4 leeftijd < 60	1	1	-	-	-
Det. 5 gewerkt = J	1	1	1	1	1
Det. 5 afdeling = 'Productie'	1	0	0	0	1
Det. 5 leeftijd > 18	1	0	1	1	1
Foutmelding	X				
Salaris		10.000	13.000	15.000	16.000

Bij het onderbrengen van de logische testgevallen ontstaat bij determinant 5 bij testgeval 2 een probleem bij het rechtstreeks overnemen van de bijbehorende 'waarde' uit de decision coverage tabel. Er geldt namelijk hier ook een afhankelijkheid met de waarde van leeftijd uit determinant 4. Hieruit blijkt reeds dat het vertalen van logische testkolommen naar logische testgevallen en later fysieke testgevallen de nodige accuratesse behoeft en niet slechts een kwestie is van rechtstreeks 'overnemen' uit de reeds aanwezige gegevens.

7.5.5 Specificeren fysieke testgevallen

De volgende stap is de vertaling van de logische testgevallen naar fysieke testgevallen. Tijdens deze stap wordt ieder logisch testgeval vertaald naar een gedetailleerde beschrijving waarin wordt aangegeven hoe er tijdens de testuitvoering precies moet worden gehandeld teneinde het logisch testgeval af te dekken.

Deze stap resulteert in een beschrijving van de fysieke testgevallen. De beschrijving van een fysiek testgeval kan er als volgt uitzien:

Nummer

Ieder fysiek testgeval is zodanig genummerd, dat precies kan worden vastgesteld van welk logisch testgeval het is afgeleid. Dit gebeurt door hetzelfde nummer aan te houden maar de letters LTG (logisch testgeval) te vervangen door FTG (fysiek testgeval). De nummering kan plaatsvinden op basis van de plaats van het testgeval in de beslissingstabel en/of kan een referentie zijn naar de testbasis waarvan het is afgeleid.

Determinanten

Het testgeval wordt bepaald door de waarden van de determinanten van het betreffende testgeval. Deze waarden moeten gedetailleerd en vooral concreet worden beschreven. Soms echter wordt de waarde van een determinant bepaald door een actie of de 'trigger'. In dit geval dient een gedetailleerde beschrijving plaats te vinden van de uit te voeren actie.

Resultaat

Het verwachte resultaat wordt bepaald en indien van toepassing berekend. Het verwachte resultaat moet zodanig zijn beschreven dat het controleerbaar is. Dit betekent dat beschreven moet worden wat er gecontroleerd moet worden, maar ook hoe het gecontroleerd moet worden.

Vaststellen initiële gegevensverzameling

7.5.6 Initiële database

Om de fysieke testgevallen te kunnen uitvoeren zijn bepaalde gegevens nodig. Van een aantal van deze gegevens is dit reeds expliciet aangegeven door de fysieke invullingen van de determinanten. Vaak echter is het nodig om bepaalde stamgegevens te definiëren naast de gegevens die primair voor de test noodzakelijk zijn. Vanwege beheer en hergebruik argumenten is het verstandig om deze gegevens vooraf reeds eenmalig te definiëren

en klaar te zetten (in plaats van tijdens de test te creëren). Tijdens deze stap wordt de zogenaamde initiële gegevensverzameling beschreven. Het wordt aanbevolen deze gegevens suggestieve namen te geven, zodat kan worden afgeleid bij welk testgeval ze horen. Als er 'tijdfafhankelijke' gegevens zijn, moet dit expliciet worden vermeld, zodat daar bij hertests later rekening mee kan worden gehouden.

7.5.7 Opstellen testscript

In het testscript moet worden vastgelegd in welke volgorde de fysieke testgevallen moeten worden uitgevoerd. Hierbij moet worden gestreefd naar optimale efficiëntie. Dit betekent dat er binnen de ruimte die daarvoor beschikbaar is, moet worden gekozen voor die volgorde die de minste omzettingen tot gevolg heeft (bijvoorbeeld in en uitloggen onder verschillende user id's of terugzetten van uitgangsdatabases) aangezien dit vaak tijdrovende activiteiten zijn. Algemene regels zijn hiervoor niet te geven.

7.5.8 Uitvoering en beoordeling

De uitvoering van de test gebeurt aan de hand van de testscripts. Men verkrijgt een goed inzicht in de voortgang van de test door bij te houden hoeveel testgevallen er in totaal gedefinieerd zijn, hoeveel er getest zijn en hoeveel er daarvan fout bevonden zijn. De testresultaten kunnen met behulp van het testscript worden vastgelegd. In het kader hiervan wordt een tweetal kolommen opgenomen in het testscript: 'ok' en 'opm'. In de kolom 'ok' wordt bijvoorbeeld met ja (geen fout bevonden) of nee (wel een fout gevonden) het testresultaat aangegeven, terwijl onder de kolom 'opm' het nummer van het bijbehorende probleemrapport kan worden vermeld. Een overall beoordeling met betrekking tot het uitgevoerde testscript kan eveneens worden opgenomen in het testscript.

Binnen Unit test kan men de testen uitvoeren en met de goede uitkomsten controleren; een goede test is ook het tonen van de juiste foutboodschap. Maar voor het draaien van deze unittest moeten wel de goede (en foute waarden) in de database opgenomen worden. Ook de testen die falen kunnen makkelijk in unit test ondervangen worden. Niet goed afgemaakte statements of foutieve waarden worden zo zichtbaar gemaakt.

8 Referenties

- [1] JUnit, <http://www.junit.org/>
- [2] Eclipse, <http://www.eclipse.org/>
- [3] Apache Maven (<http://maven.apache.org>)
- [4] EasyMock, <http://www.easymock.org/>
- [5] MockRunner, <http://mockrunner.sourceforge.net/>
- [6] sUnit, <http://sunit.sourceforge.net/>
- [7] An early look at JUnit 4,
<http://www-128.ibm.com/developerworks/java/library/j-junit4.html>
- [8] An introduction to Maven 2,
<http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html>
- [9] Systeemontwikkeling met J2EE, R. Lightmans et al., 2005, Sdu uitgevers bv, Den Haag
- [10] JUnit in Action, V. Massol, 2004, Manning Publications Co., Greenwich
- [11] Ordina Software Factory, <http://www.ordinasoftwarefactory.nl/>
- [12] Java Platform, Enterprise Edition, <http://java.sun.com/javaee/>
- [13] DbUnit, <http://dbunit.sourceforge.net/>
- [14] TestNG, <http://testng.org/>
- [15] In pursuit of code quality: JUnit 4 vs. TestNG, IBM developerWorks,
<http://www-128.ibm.com/developerworks/java/library/j-cq08296/>
- [16] Clover, <http://www.cenqua.com/clover/>
- [17] MockEJB, <http://www.mockejb.org/>
- [18] Testen volgens TMap, M. Pol, R. Teunissen, E. v. Veenendaal, 2002, Uitgeverij Tutein Noltemius, 's-Hertogenbosch
- [19] Extreme Programming, <http://www.extremeprogramming.org/>
- [20] Inversion of Control Containers and the Dependency Injection pattern
<http://www.martinfowler.com/articles/injection.html>
- [21] Design Patterns: Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, Reading, MA, 1995.
- [22] Cobertura, <http://cobertura.sourceforge.net/>
- [23] Apache Jakarta Cactus, <http://jakarta.apache.org/cactus/>
- [24] Get Acquainted with the New Advanced Features of JUnit 4,
<http://www.devx.com/Java/Article/31983>
- [25] NetBeans, <http://www.netbeans.org/>
- [26] MyEclipse, <http://www.myeclipseide.com/>
- [27] IBM Rational Systems Developer,
<http://www-128.ibm.com/developerworks/rational/products/rsd/>
- [28] Better Builds With Maven, <http://library.mergere.com>
- [29] IntelliJ IDEA, <http://www.jetbrains.com/idea/>

Meer nuttige informatie is te vinden op onder andere
<http://junit.sourceforge.net/doc/faq/faq.htm>