De kern van modelleren

Samenvatting door

Stijn Delbeke

2009-2010

Inhoud

MVC en het UML-model

MVC = **M**odel-**V**iew-**C**ontroller

- Er moeten een aantal klassen worden gemaakt die niet in het model staan
  - Technische klassen
- Controller en view mogen weggelaten worden uit het sequentiediagram omdat ze gemeenschappelijk zijn aan alle sequentiediagrammen

Hoofdstuk 1. Testen en debuggen

Debuggen bestaat uit vier fasen:

1. Een fout wordt *ontdekt*
2. De fout wordt *gelokaliseerd*
3. De fout wordt *geïdentificeerd*
4. De fout wordt *opgelost*

1.1 Testen

Bij kleine systemen met slechts enkele klassen gebeurt het testen op twee niveaus:

- Laagste niveau: *unit testing*
  - Individuele klassen
  - Testen zijn gericht op één of op enkele methodes
- Integratietests
  - Gaan na of de individuele klassen goed samenwerken

Tests worden meerdere keren uitgevoerd. Redenen:

- Bij het debuggen van software is het mogelijk dat men bij een poging om een fout te verhelpen er andere introduceert
- Bij flexibele technieken, zoals Extreme Programming, wordt ervan uitgegaan dat geschreven code steeds kan veranderd worden.
  - Er worden tests geschreven vóór de code zelf.
  - Bij het programmeren worden deze tests voortdurend herhaald

Testsuites:

- Testproces zo vlot mogelijk laten verlopen
  - Volledig automatiseren
- Mag niet te veel uitvoer produceren als er iets fout gaat
- *Fixture*
  - Omgeving om te testen
  - Binnen een fixture kunnen één of meerdere tests worden uitgevoerd
  - Daarna wordt de fixture afgebroken
- *TestCase*
  - Biedt de mogelijkheid om een fixture met een of merdere tests te definiëren
- *TestSuite*
  - Kan meerdere Tests omvatten
  - Zowel TestCases als andere suites
- *Assert*
  - Bepaalt de uitslag van de test
  - Foutmelding als niet aan de voorwaarde voldaan is

1.2 Fouten lokaliseren

Fouten kunnen optreden in de gegevensdefinities, maar houden niet altijd verband met uitvoerbare code.

1. Zoek in welke procedure de fout optreedt
2. Als de procedure zinvol kan opgedeeld worden met tussentoestanden, zoekt men het stuk code waar de fout haar oorsprong heeft = **sectie**

Mogelijkheden als een fout optreedt:

- We hebben een gegeven met een verkeerde waarde, en we zoeken de plaats in de code die verantwoordelijk is voor deze verkeerde waarde
  - Lokalisatieprobleem
  - Welke code verandert de waarde?
  - Methodes:
    - Door middel van een debugger
    - Code invoegen die extra data uitschrijft
- We hebben een gegeven met een verkeerde waarde, en we weten door welke sectie deze verkeerde waarde veroorzaakt wordt
  - Fout identificeren
  - Oorzaken:
    - Fout ligt in de code zelf
    - Code is correct, maar werkt met de verkeerde gegevens

1.3 Identificeren van een fout

We hebben een sectie code en we weten dat het resultaat verkeerd is. We moeten dus rekening houden met het volgende:

- De postcondities van de code
  - Zijn niet voldaan
  - Wat is de overtreding van de postcondities?
- De precondities van de code
  - Mogelijke oorzaak van de fout als deze niet voldaan zijn
  - Wat is er juist verkeerd gegaan?
- De logica van de code
  - De precondities zijn voldaan, maar de postcondities niet
    - Er scheelt iets met de code zelf
  - Code stap voor stap overlopen, dit kan op twee manieren:
    - We laten de code lopen. Tussenresultaten kunnen we controleren door met een debugger te werken of door tussengevoegde uitschrijfoperaties
    - We werken manueel: de code wordt niet uitgevoerd, maar we controleren wat er gebeurt door uit het hoofd, of met pen en papier, de werking van de code na te bootsen

Om goed te begrijpen wat de functies en eigenschappen zijn van elke variabele, moeten volgende zaken goed in het oog gehouden worden:

- De functie van de variabele
  - Zou duidelijk moeten zijn aan de hand van zijn naam
  - Eventueel aan de hand van commentaar
- Eventuele beperkingen op de toegestane waarden
- Als de variabele een object is moet er op volgende eigenschappen gelet worden:
  - Waarde van attributen
  - Eventueel de lengte van een tabel

1.3.1 ADVB(S)-classificatie

A. Algoritme

- ✓ **Eén-ernaast**
  - De code berekent een getal dat één hoger of lager is dan de gewenste waarde
- ✓ **Logica**
  - Er zit een logische fout in het algoritme
  - Vaak is dit een gevolg van onduidelijke beschrijving van pre- of postcondities
- ✓ **Validatie**
  - Er wordt niet juist gecontroleerd of variabelen een correcte waarde hebben
- ✓ **Efficiëntie**
  - De code loopt veel te traaf of heeft te veel geheugen nodig
  - Dikwijls te wijten aan een ontwerpfout

D. Data

- ✓ **Index**
  - o De index van een tabel neemt verkeerde waarden aan
- ✓ **Grens**
  - o Er zijn fouten bij de speciale behandeling aan het begin of einde van een reeks gegevens
- ✓ **Voorstelling**
  - o Een bug veroorzaakt door de manier waarop data worden voorgesteld in het geheugen
- ✓ **Geheugen**
  - o Het programma maakt fouten bij het geheugenbeheer
    - Vb.: vergeten van `delete`

V. Vergeten

- ✓ **Init**
  - o Een variabele krijgt geen geldige beginwaarde
- ✓ **Ontbreekt**
  - o Een noodzakelijke opdracht ontbreekt
- ✓ **Locatie**
  - o Een opdracht staat op de verkeerde plaats

B. Blunder

- ✓ **Variabele**
  - o Er wordt een verkeerde variabele gebruikt
- ✓ **Uitdrukking**
  - o De berekening van een uitdrukking bevat een fout
- ✓ **Taal**
  - o Een bug specifiek aan de syntax van de programmeertaal

S. Symmetrie

- ✓ Sommige opdrachten hebben een tegenpool, waardoor een koppel ontstaat:
  - o Openen/sluiten van bestanden
  - o C++: `new/delete`
- ✓ De tweede helft van dit koppel wordt vaak vergeten

Hoofdstuk 2. Refactoring

2.1 De geschiedenis van Netscape

2.2 Toepasbaarheid van herwerken

Eén van de basisprincipes van herwerken is om twee processen te scheiden:

- Men maakt eerst de bestaande code klaar om wijzigingen aan te brengen
- Daarna wijzigt men de functionaliteit

Herwerken = reorganisatie van de code *zonder* dat de functionaliteit wijzigt

De resulterende code dient po exact dezelfde manier te werken als de oorspronkelijke. De bedoeling hiervan is tweevoudig:

- De code zo **duidelijk** en begrijpbaar mogelijk maken
- Software **flexibel** maken

Situaties waarbij het aangewezen is van te herwerken:

- Bij het gebruik van *incrementele* ontwikkelingsmethodes
- Bij het *hergebruiken* van code
    - Om de code te leren kennen
- Bij het ontwikkelingsproces
    - Met oog op testen en debuggen

Het is uit den boze om de fundamentele structuur van de software te hertekenen: wie veel wijzigingen tegelijkertijd doorvoert loopt een groot risico op verandering van functionaliteit.

Herwerken steunt zeer dikwijls op het testen van de herschreven code

- Gebruik van een automatische testomgeving is volstrekte noodzaak
- Oorzaak van bugs kan snel worden opgespoord

<u>Voordelen</u> van herwerkingstools voor refactoring:

- Goedkoper
- Veel sneller
- Herwerkingsacties worden verweven in het proces van maken of wijzigen van software
    - Men moet de code die men niet moet aanpassen niet eens bekijken

<u>Nadelen</u>:

- Minder goed gestructureerde code

2.3 Uitwerking

De volgende stappen worden ondernomen bij herschikking:

1. Het overlopen van de code en het ontdekken van problemen
    - Als de code niet als natuurlijk ervaren wordt door derden moet herschikking overwogen worden
    - Verbetering van leesbaarheid en flexibiliteit
2. Het bepalen van de juiste actie of acties om te herschikken
3. Het uitvoeren van de acties en het testen van het resultaat

Voor het ontdekken van probleemcode zijn er een aantal herkenningspunten:

- Drie grote niveaus waarop kan gewerkt worden
    - Binnen een enkele klasse
    - Samenwerking tussen twee klassen
        - Apart
        - Binnen een hiërarchie van overerving
    - Tussen verschillende hiërarchieën van klassen
- Twee criteria die gebruikt worden:
    - Begrijpelijkheid van de code
    - Flexibiliteit van de code
        - **Eenheid van plaats**: als om een functie te veranderen code moet worden aangepast op verschillende plaatsen, is dit te veel werk
        - **Standaardisatie**: de verandering moet gebeuren op een manier die geen inventiviteit van de programmeur vraagt

2.4 Problemen binnen een klasse

2.4.1 Onduidelijke naamgeving

Zie hoger.

2.4.2 Onjuiste indeling private operaties

Private hulpoperaties

- Worden vaak vergeten bij het voor de eerste keer programmeren
- Pas later heeft de programmeur het gevoel dat hij “deze code al eens gezien heeft”
- Bij het vinden van duplicaatcode wordt de *extraheer methode* gebruikt en worden verschillende kopieën van de code allemaal vervangen door een oproep naar de nieuwe operatie

2.4.3 Overmaat aan conditionele uitdrukkingen

Conditionele uitdrukkingen zorgen ervoor dat één stuk code verschillende zaken doet, en met moet de vraag stellen of deze verschillende zaken niet moeten gescheiden worden.

Bij het bestaan van aparte staten en/of deelklassen kunnen we volgende problemen tegenkomen:

- De conditie hangt niet af van de toestand van het object zelf, maar van de toestand van een ander object, die eventueel staten of deelklassen bevat.
- De conditie hangt af van de toestand van het object, maar de toestand kan niet zo veranderen dat een ander alternatief gekozen wordt.
- De conditie hangt af van de toestand van het object, maar een object kan soms het ene alternatief en soms een ander kiezen. In dit geval zijn er staten.

2.4.4 Onechte gegevensvelden/deelobjecten

Gegevensvelden moeten de toestand van het object aanduiden. Een gegevensveld dat alleen zin heeft als een of andere operatie bezig is, dient binnen die operatie te blijven.

Eventueel kunnen meerdere operaties gebundeld worden in een publieke operatie of klasse. Dergelijke klasse noemt men een **methodeklasse**. Strikt genomen is het geen klasse, want vaak is de enige publieke operatie de constructor: deze voert de operatie uit, en daarna kan het object verdwijnen.

2.4.5 Te grote klasse

Het is bijna steeds mogelijk om klassen met te veel gegevensvelden en/of operaties op te splitsen.

2.4.6 Overdaad aan parameters

Er is bijna steeds iets mis met een operatie die te veel parameters heeft!

- De grens van een module moet zo klein mogelijk zijn
- Eventueel referentie naar een object doorgeven in plaats van alle parameters

2.4.7 Intelligente attributen

Operaties zijn toegespitst op een gegevensveld waardoor het gegevensveld extra functionaliteit heeft.

2.5 Problemen tussen twee klassen

2.5.1 Duplicaatcode

Het is mogelijk dat het dupliceren van code niet eenvoudig te vermijden is.

Moeten wijzigingen bij beide duplicaten veranderd worden?

- De twee klassen hebben een gemeenschappelijke voorouder
  - Code kan gemeenschappelijk ondergebracht worden bij deze voorouder
- Er bestaat nog geen bovenklasse, maar het is zinvol een te creëren
  - Dit zal waarschijnlijk een abstracte klasse zijn
- De twee klassen zijn niet nauw verwant
  - Gemeenschappelijke code onderbrengen bij een derde klasse

2.5.2 Veel gebruik van gegevensvelden van de andere klasse

- Soms is het nodig twee klassen samen te voegen. Dit is bijna zeker het geval als er een 1-1 associatie tussen de twee klassen bestaat.
- Soms is het nodig ettelijke methodes en/of gegevensvelden van de ene naar de andere klasse te verhuizen, zodat twee minder gekoppelde klassen ontstaan.

2.5.3 Onvolledige klassen

Als er geen andere methodes zijn dan get- en setoperaties, dan is er een andere klasse, of zijn er meerdere, die zorgen voor de operaties.

Oplossingen:

- Code uit de manipulerende klasse(n) verhuizen naar de probleemklasse
- Probleemklasse afschaffen

2.5.4 Onafhankelijke klassen met gelijkaardige functionaliteit

Onafhankelijk: niet nauw verwant door overerving

- Nagaan of ze niet op een of andere manier samenhoren
  - Let op de methodes die hetzelfde doen
- Proberen in een hiërarchie te krijgen
  - De ene wordt bovenklasse, de andere deelklasse
  - Er wordt een nieuwe bovenklasse voorzien en beide worden deelklassen

2.5.5 Problemen met delegatie

Een object kan een bericht delegeren naar een ander object, we spreken van een *delegerend* object en een *uitvoerend* object.

Opletten met klassen die alleen maar delegeren en zelf niets doen!

Uizondering: *façadepatroon*

- Façcade = de grens tussen het eigenlijke systeem en een deelsysteem
- Verbetert de modulariteit van het systeem

Wat te doen met zuiver delegerende objecten?

- Klasse heeft geen meerwaarde
  - Overwegen om deze klasse te schrappen
- Klasse heeft wel een meerwaarde
  - Deelklasse maken van de uitvoerende klasse
  - De objecten van de originele klassen *verpakken* in een **wrapper class**
    - Elk object met aggregatie bevat een object van de oorspronkelijke klasse
    - Gebruik:
      - Als het gedrag van de oorspronkelijke klasse enigszins gewijzigd moet worden
      - Als we niet alle publieke operaties van de uitvoerende klasse publiek willen maken
    - Alleen geschikt op klassenniveau, niet op objectniveau

2.5.6 Het hagelverwijderingssymptoom

Als men een bepaalde aanpassing doet, en met moet op veel verschillende plaatsen veranderingen doorvoeren, dan is dit het teken dat het bepaalde aspect dat men moet veranderen verspreid is over de code. Men gaat dan proberen methodes of datavelden te verplaatsen, en eventueel een klasse laten opslorpen door een andere om dit fenomeen te bedwingen.

2.5.7 Divergente verandering

Wanneer men een bepaalde klasse of operatie moet aanpassen bij een aantal niet-verwante veranderingen van de software, dan is het duidelijk dat dit item verschillende taken vervult. Waarschijnlijk is het dan beter om het item op te splitsen in kleinere eenheden.

Verhogen van flexibiliteit:

- Gebruik van een aantal technieken
- Soms overbodig, omdat ze flexibiliteit leveren op onnodige plaatsen
    - Toenemend aantal klassen zorgt voor een ingewikkeldere structuur
    - Beter om de structuur te vereenvoudigen, ook al wordt die daardoor meer rigide

2.6 Problemen tussen hiërarchieën

Actie: ***ontwarren van hiërarchieën***

Symptoom: ***parallelle hiërarchieën***

Oplossing:

- Voor elke factor een hiërarchie maken
- Vb.: p30

Hoofdstuk 3. Dwarsaspecten

3.1 Een voorbeeld: foutafhandeling

Twee soorten fouten kunnen optreden:

- Onhandelbare fouten
  - Opvangen met excepties
  - Leiden tot het afsluiten van de applicatie met een melding die het soort fout aangeeft
- Handelbare fouten
  - Opvangen met excepties
  - Voor elke fout wordt een apart ontwerp gemaakt waarin moet gedefinieerd worden:
    - Wat de naam is van de aparte klasse van exceptie gereserveerd voor de fout
    - In welke omstandigheden zo'n fout optreedt
    - Waar de fout kan opgevangen worden

Klassieke benadering: dit beleid vastleggen vóór de code geschreven wordt.
Nadelen:

- De programmeur moet herhaaldelijk dezelfde code schrijven
- De programmeur moet dit beleid steeds in het achterhoofd houden
  - <-> modulaire indeling: slechts met één ding tegelijk bezig
  - Kans op fouten en weglatingen vergroot
- Er zijn problemen met hergebruik
- Als men het beleid wil wijzigen dient men alle code te herzien

Dwarsaspecten: aspecten die voor problemen kunnen zorgen:

- Klassenindeling
  - Indeling van geaggregeerde structuren, die problemen oplevert bij het doorlopen
- Synchronisatie
  - Bij het delen van resources moet telkens gecontroleerd worden als deze wel vrij zijn
- Gedistribueerde locatie
  - **Configuratie =** bij een multiprocessorensysteem worden de objecten verdeeld over de verschillende processoren
  - De ontwikkelaar moet soms zelf aandacht besteden aan de configuratie omdat die van het OS niet voldoet
- Real-Time systemen
  - Bepaalde berekeningen moeten op tijd klaar zijn
- Herstel van hardwarefouten
  - Back-up- en terugzetprocedures
- Ontwikkeling versus gebruik
  - Software moet zich anders gedragen bij ontwikkeling dan bij dagelijks gebruik
    - Foutafhandeling
    - Logging

De behandeling van het dwarsaspect wordt apart gedefinieerd, en dan op een of andere manier verspreid naar verschillende plaatsen in de code.

Metaobjecten = controleren de acties van gewone objecten en passen deze aan

Het uitgangspunt voor het doorlopen van klassenstructuren is de code waarin alleen toepassingslogica verwerkt zit, met daarnaast de definitie van de dwarsaspecten. Er zijn verschillende mogelijkheden om deze twee te combineren:

- De code horend bij de dwarsaspecten wordt *verweven* met de code van de toepassingslogica
  - Gebeurt voor of tijdens het compileren
- Tijdens het lopen van het programma
  - De werking van het programma met toepassingslogica wordt gemonitord door een metaprogramma dat de werking van het toepassingsprogramma kan wijzigen en onderbreken
  - Wordt niet veel gebruikt

3.2 Metaobjecten

OpenC++ = systeem waarbij metaobjecten gebruikt worden om code aan te passen bij compilatie

Programmeur maakt twee programma's:

- Basisprogramma
- Metaprogramma
  - Beschrijft hoe het basisprogramma moet aangepast worden

Gebruik:

1. Het metaprogramma wordt door een preprocessor verwerkt, waarbij de uitbreidingen worden omgezet naar standaard C++
2. Het metaprogramma wordt gecompileerd, waarbij een parser wordt ingelinkt die een C++ programma kan analyseren
3. Het basisprogramma wordt door de parser verdeeld in kleine eenheden. Tevens bepaalt de parser de structuur van het programma
4. Op basis van deze gegevens zet het metaprogramma het basisprogramma om naar standaard C++, die gecompileerd wordt door een gewone C++ compiler

Data die door de parser ter beschikking gesteld worden van het metaprogramma:

- Tekst van de code
  - Opgesplitst in tokens
  - Boomstructuur
- Gegevens over de structuur van de code
  - Lijst van klassen
  - Per klasse:
    - Bovenklassen
    - Onderklassen
    - Gegevensvelden
    - Operaties
    - Naam van de metaklasse
  - Omgeving
    - Zichtbare variabelen en hun type

Voor elke klasse in de basiscode is er één metaobject. Dit heeft een dubbele functie:

- Opslagplaats voor de info over de klasse
- Omzetten van alle code die met die klasse te maken heeft

Stukken code in de parseboom:

- Klassenhoofding
- Definitie van een lidfunctie
- Oproep van een lidfunctie of een gegevensveld in code die tot de eigen of tot een andere klasse behoort

Twee strategieën voor het verweven van code:

- **Black box**
  - Basiscodefragmenten in functiedefinities worden als ontoegankelijk beschouwd
  - Adviescode kan alleen worden ingevoegd aan het begin en het einde van de uitvoering van publieke methodes
  - Vb.: samengestelde filters
- **White box**
  - Adviescode kan ook intern binnen functiedefinities worden toegevoegd
  - Vb.: aspectgericht programmeren

3.3 Aspectgeöriënteerd programmeren

AspectJ en AspectC++ zijn beide pakketten voor een aspectgericht programmeren waarbij de aspectcode toegevoegd wordt aan de basiscode voor compilatie.

Basiseenheid van de aspect code is een *aspect*.
Een aspect bestaat uit verschillende delen:

- Pointcut
  - Beschrijving van alle relevante *join points* (=de punten in de originele code waar code moet toegevoegd worden)
    Mogelijke join points:
    - Oproep van een lidfunctie
    - Uitvoering van een lidfunctie
    - Wijziging of lezen van publieke variabelen
- Advies
  - Code die moet ingevoerd worden bij elke join point van een point cut

Slices = voegt code toe op een bepaald punt in de basiscode

3.4 Samengestelde filters

Deze manier van werken kan worden beschouwd als een verdere beperking van het metaobjectmodel.

Bij het filtermodel wordt elk object voorzien van twee extra soorten eigenschappen:

- **Condities**
  - Zuivere query's
  - Er wordt meestal aangenomen dat ze een logische waarde teruggeven, maar dit is niet noodzakelijk
- **Filters**
  - Invoerfilters
  - Uitvoerfilters
  - Kan een boodschap accepteren of verwerpen
  - Meest voorkomende acties:
    - Doorsturen
      - Bericht wordt doorgestuurd naar zijn bestemmeling
    - Vervangen
      - Bericht wordt gewijzigd en dan doorgestuurd naar de volgende filter
    - Foutopvangen
      - Genereert een exceptie
    - Wachten
      - Bericht wordt bijgehouden en periodiek worden de voorwaarden gecontroleerd
      - Als de voorwaarde veranderd is, wordt het object doorgestuurd naar de volgende filter

- Meta
  - Bericht wordt *gereïficeerd*: er wordt een object gemaakt waarvan de attributen de eigenschappen van het bericht weergeven
  - Dit object wordt doorgestuurd naar een verwerkend object
  - Daarna gaat het naar de volgende filter
- Acties kunnen gecombineerd worden
- Bij een goed gebruik van de filtermethodiek wordt meestal alleen de staat van het object opgevraagd

3.5 Doorlopen van structuren

Wet van Demeter:

- Een methode van een bepaald object *self* mag alleen methodes aanroepen van vier groepen objecten:
  1. *Self* zelf
  2. Objecten doorgegeven met de parameters van de methode
  3. Objecten gecreëerd binnen de methode
  4. Objecten geassocieerd aan *self*, door een gewone associatie of aggregatie

Twee verweven aspecten:

- Waar zitten de objecten die de dienst moeten leveren?
- Wat moet er met die objecten gebeuren?

Twee manieren om aspecten te scheiden zodat het doorloopaspect gescheiden wordt van het andere:

- Gebruik van *iteratoren*
- Gebruik van *visitors* en doorloopfuncties
  - Doorloopfunctie: brengt een object van buiten (=*visitor*) bij elk object in de structuur