Programmaontwerp en -Realisatie

Samenvatting door

Stijn Delbeke

2009-2010

Inhoud

Hoofdstuk 1. Kwaliteitseisen

1.1 Primaire eisen

Een **programma** moet

- Doen wat het moet doen
- Betrouwbaar zijn
- Gebruiksvriendelijk zijn
- Flexibel aangepast kunnen worden

Het **ontwikkelingsproces** moet

- Efficiënt zijn
- Irritatie van ontwikkelaar of klant zoveel mogelijk vermijden

1.2 Afgeleide eisen

- Software moet goed ingedeeld zijn (modulair) ↔ Spaghetticode
- Code moet makkelijk te begrijpen zijn
- Zoveel mogelijk hergebruik van code
- Alle delen testen alvorens te gebruiken

Hoofdstuk 2. Hergebruik

2.1 Voordelen en problemen

2.1.1 Voordelen

- Arbeidsbesparend
- Betere kwaliteit

2.1.2 Vormen van hergebruik

- Hergebruik van code binnen het project
- Hergebruik van code elders
- Hergebruik van ideeën

2.1.3 Problemen

- Soms te lang zoeken om herbruikbare elementen te vinden
- Element van buitenuit inpassen in eigen code:
    - Hoe wordt de programmamodule gebruikt?
    - Welke (globale) variabelen zijn relevant?

2.2 Bronnen

- Zeer onwaarschijnlijk dat geen enkele vorm van hergebruik mogelijk is
- Afweging maken:
    - Zelf schrijven; 3 belangrijke bronnen:
        - Repertoria met ideeën (boeken, internet, …), design patterns
        - Code gemaakt voor algemeen gebruik
        - Programma's die gelijkaardige functies uitvoeren
    - Andere code aanpassen

2.2.1 Overdrachtsgebieden

- Interface met de gebruiker
- Ideeën voor implementatie
    - Broncode of documentatie vereist

2.3 Hergebruik en eigen code

2.3.1 Standaardisatie van gegevens

- Standaardisatie naar vorm
  - Gebruik van een formaat dat algemeen gebruikt wordt.
    Vb.: XML
  - Voordeel: Toepassingen van buitenaf op de data loslaten
- Standaardisatie naar structuur
  - Opslaan van data die beantwoorden aan een bepaalde standaard
    Vb.:RNA-sequenties

Meestal komt een structuurstandaard bovenop een vormstandaard.

2.3.2 Modulaire indeling

- Houd methodes en operaties samenhangend
  - Een goede operatie voert één welomschreven actie uit
- Houd verschillende operaties consistent
  - Als 2 operaties gelijkaardige functies uitvoeren, moeten ze dat op een gelijkaardige manier doen
- Maak methodes algemeen

2.3.3 Robuustheid

- Bescherm de module tegen foute invoer
- Vermijd afhankelijkheden (van bv globale variabelen)
- Vermijd zogezegd redelijke veronderstellingen over invoer (cfr. MS Acces)

2.3.4 Documentatie

- Zelfdocumenterende code zonder bijkomende documentatie is onvoldoende
- Minimum:
  - Beschrijving van elke klasse
  - Duidelijke beschrijving van methodes:
    - Wat doet ze?
    - Precondities
    - Postcondities
- Niet alleen belangrijk voor hergebruik, maar ook:
  - Vergemakkelijkt het verbeteren van fouten
  - Verhoogt de flexibiliteit van het systeem

Hoofdstuk 3. Het ontwikkelingsproces

Het is altijd belangrijk dat men :

- Een idee heeft van wat men op welk moment moet doen
- Weet wat de beoogde resultaten zijn van de taak waarmee men bezig is
- Weet hoe men het moet doen

3.1 Indeling in fasen

Ontwikkeling is géén lineair proces!

3.1.1 Behoefteanalyse

- Functieanalyse
  - Beschrijving van de relevante processen, gebruikers, hulpmiddelen, doelstellingen, …
  - Resultaat hiervan geeft aan welke taken door wie worden uitgevoerd en wat de samenhang is van alle processen
- Definitiestudie
  - Welke taken die reeds uitgevoerd worden komen in aanmerking voor verbetering?
  - Welke mogelijkheden zijn er voor uitbreiding?

3.1.2 Ontwerp

In deze fase wordt vastgelegd wat het systeem allemaal zal moeten omvatten.

- Functioneel ontwerp
  - Beschrijft de taken waar het systeem bij te pas komt
  - Beschrijft de verantwoordelijkheden van de gebruikers
- Technisch ontwerp
  - Beschrijving van de nodige apparatuur
  - Beschrijving van de nodige pakketten
  - Beschrijving van de realisatiefase
  - Beschrijving van de invoering

3.1.3 Realisatie

- Installeren van hardware
- Installeren van softwarepakketten
- Programmeren
  - Testen
  - Documenteren

3.1.4 Invoering

- Conversie van gegevens van het oude systeem naar het nieuwe
- Informeren en/of opleiden van gebruikers
- **Formele acceptatie:** de klant neemt het systeem in gebruik en verklaart dat aan de vereisten voldaan is

3.1.5 Onderhoud en beheer

Vaak optredende problemen zijn:

- Fouten in de werking van het systeem
- Door gebruik van het systeem ontdekt men nog andere eisen of verwachtingen, die niet geïmplementeerd zijn
- Het systeem doet iets anders dan hetgeen de gebruikers voor ogen hadden

3.1.6 Kleine projecten

- Scheiding tussen verschillende fasen is zeer klein
- Het is belangrijk om GEEN stappen over te slaan!

3.2 Indeling naar acties

- De meeste acties zijn verbonden aan een van de vijf fasen, behalve:
    - Documenteren
    - Kwaliteitscontrole
- In grotere ontwikkelingsgroepen heeft iedereen zijn specialiteit
- Elk deelresultaat kan gecontroleerd worden op kwaliteit
    - Bevat de behoefteanalyse geen leemtes?
    - Is het ontwerp coherent?
- Gebruikersdocumentatie moet zo vroeg mogelijk gemaakt worden

Hoofdstuk 4. Modellering en softwareontwikkeling

Twee basisgedachten voor objectgericht ontwerp zijn:

- Een systeem kan het duidelijkst worden beschreven door het te beschouwen als een verzameling interagerende objecten
- Een gemakkelijk te begrijpen programma is zo ingedeeld dat het een afbeelding is van de realiteit

Takendiagram = formeel UML-diagram voor de behoefteanalyse

4.1 Indeling

4.1.1 Modules en koppeling

Module = abstract onderdeel van een systeem dat weinig koppeling heeft met de rest van het systeem

Interface (= grens) van een module bepaalt:

- Wat de buitenwereld van een module kan verwachten
- Hoe de inwendige structuur moet reageren op boodschappen van buitenaf

4.1.2 Inkapseling

De grenzen van een goede inkapseling zijn:

- Duidelijk omschreven
- Zo klein mogelijk: geen overbodige interacties

4.1.3 Niveaus en indeling

Elke module is een systeem op zich en kan worden opgedeeld in deelmodules.

Er ontstaan niveaus:

- Op elk niveau moeten modules zoveel mogelijk gelijkwaardig zijn
- Op geen enkel niveau mogen onoverzichtelijk veel modules zijn

Aggregatie: relatie tussen een module en zijn deelmodules

4.2 Nut van modulair programmeren

- Overzichtelijk
  - Programmeur moet alleen de interface en het inwendige maken
  - Gebruiker moet enkel de interface kennen
- Kan apart getest worden ➔ grotere betrouwbaarheid
- Grotere flexibiliteit
- Herbruikbaarheid

4.3 Nadelen van modulair programmeren

- Implementatie van een procedure heeft altijd neveneffecten
  ➔ Deze moeten tot een minimum herleid worden
- De meest efficiënte implementatie is afhankelijk van de omgeving waarin de module gebruikt wordt, en deze is niet altijd gekend
- Overhead

Hoofdstuk 5. Objecten en berichten

5.1 Waarschuwing

5.2 Wat is een object?

Object = een ding dat interageert met zijn omgeving maar ook een bestaan op zichzelf heeft

- Een object is het geheel van eigenschappen en interacties relevant voor het systeem
- **Toestand:** wordt beschreven door een aantal attributen
- **Identiteit:** wordt aangeduid met een naam
  - Een object kan meer dan 1 naam hebben
  - Naam kan veranderen
- **Interactie:** zenden en ontvangen van berichten
  - Voor elk soort berichten heeft een object een *operatie*: een procedure die aangeeft hoe het object reageert op dit bericht
  - Objecten kunnen op 2 manieren reageren op een bericht:
    - Zijn inwendige toestand veranderen
    - Afhankelijk van zijn toestand reageren met een bericht

5.3 Klassen

Klasse = beschrijving van gelijkaardige objecten

Drievoudig doel:

- Men moet niet elk object afzonderlijk beschrijven
  - Bespaart werk
- Het geheel wordt overzichtelijker
- Het is makkelijker om foutloos wijzigingen aan te brengen

5.4 Soorten modules

- **Objecten:**
  - Reactie op een bericht hangt af van de inwendige toestand
- **Functies en filters:**
  - Sturen als reactie op een bericht slechts één bericht terug
  - De inhoud van dat bericht hangt af van het ontvangen bericht en de waarden van de parameters
- **Coördinerende modules:**
  - Hebben geen eigen toestand
  - Als reactie op een bericht kunnen ze objecten aanspreken en aan de hand van hun reactie verschillende berichten terugsturen

Bij een objectgericht ontwerp is het bijna altijd verkeerd om modules te maken die geen objecten zijn!

Hoofdstuk 6. Het statische model

Drie modellen:

- **Statisch model**
  - = *klassenmodel*
  - Wat zijn de elementen van het systeem?
    - Klassendiagram
- **Dynamisch model**
  - Hoe gedragen elementen zich en hoe veranderen ze?
    - Takendiagram
    - Sequentiediagrammen
    - Samenwerkingsdiagrammen
    - Activiteitsdiagrammen
    - Statendiagrammen
- **Functioneel model**
  - Hoe worden gegevens omgevormd?
    - Gegevensstroomdiagrammen (≠ UML)

Het statisch en dynamisch model worden samen gemaakt.

6.1 Associaties en verbindingen

Verbinding = een relatie tussen twee objecten a en b waarbij a een bericht kan sturen naar b

Algemene regel:
Als b een bericht verstuurt naar a als antwoord op een (vragend) bericht van a, dan moet er een verbinding bestaan van a naar b, maar niet omgekeerd.

Associatie = er kan tussen elk object van klassen A en B een verbinding bestaan, maar dit is niet noodzakelijk.

6.2 Het UML-diagram

Het Klassendiagram in zijn eenvoudigste vorm:

- Rechthoeken verbonden met streepjes
  - Rechthoek = klasse
  - Streepje = associatie
- Streepjes worden van commentaar voorzien:
  - Werkwoordvorm
  - Multipliciteit
  - Rollen in de relatie

Er kunnen meerdere associaties bestaan tussen verschillende klassen.

Hoofdstuk 7. Het dynamisch model

7.1 Het takendiagram

Het takendiagram geeft aan welke berichten het systeem uitwisselt met de buitenwereld. Het geeft dus aan welke taken het systeem moet vervullen en welke invoer het daarvoor nodig heeft.

7.1.1 Actoren

Actor = een module buiten het systeem die interageert met (delen van) het systeem.

Twee centrale vragen:

- Wie zijn de actoren? (niet noodzakelijk personen)
- Voor elke actor:
  - Wat zijn zijn behoeften (opdrachten aan het systeem)?
  - Voor welke opdrachten van andere actoren levert hij diensten aan het systeem?

Bij een taak kunnen verschillende actoren betrokken zijn.

Als verschillende taken een gemeenschappelijk deel hebben, kunnen deze opgedeeld worden in deeltaken.

7.2 Beschrijving van de taken

/

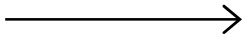
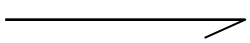
7.3 Sequentiediagrammen

Het sequentiediagram beschrijft de opeenvolging van interacties tussen de modules.

Opbouw:

- Verticale stippellijn
  - Geeft het bestaan van het object aan
- Dubbele verticale lijn
  - Object wordt actief gedurende de taak
- Pijl
  - Geeft een boodschap weer: deze wordt verklaard met een woord
  - Vier verschillende soorten:
    - Synchrone boodschap
      - Zender verwacht van ontvanger dat hij de boodschap verwerkt terwijl hij wacht

        
      - Ontvanger krijgt de controle tot hij deze teruggeeft

        
    - Doorgave
      - De zender heeft zijn werk gedaan, is niet meer actief en geeft de controle door
      - 
    - Asynchrone boodschap
      - De zender wacht niet op antwoord, maar behoudt de controle
      - De ontvanger verwerkt de boodschap terwijl de zender ondertussen met iets anders bezig is (parallellisme)
      - 

7.4 Statendiagrammen

De toestand van een object wordt bepaald door de waarde van zijn attributen.

Als de reacties van een object op een bericht duidelijk verschillend zijn naargelang deze toestand, dan worden de mogelijke toestanden verdeeld in **staten**.

Opbouw:

- Afgeronde rechthoek
  - Bepaalde staat van een object
- Startpunt
  - Beginstaat van een object
- Pijl
  - Overgang tussen verschillende staten
  - Als er een actie is bij de overgang:
    - Aanduiden na de gebeurtenis die de overgang veroorzaakt
    - Voorzien van een schuine streep (= teken van een operatie)
- Wachter
  - Een voorwaarde die tussen rechte haakjes wordt gezet
  - De overgang vindt slechts plaats als aan deze voorwaarde voldaan is

Gebeurtenissen die leiden tot een overgang:

- **Synchrone boodschap**
  - Wordt ontvangen door het object
  - Deze boodschap geeft de controle door ➔ object wordt actief
- **Verandering**
  - Een voorwaarde verandert van waar naar onwaar, of omgekeerd
- **Tijdsinterval**
  - De duur wordt gerekend vanaf het ogenblik waarop de staat bereikt wordt
  - Kan ook op een bepaald tijdstip ipv na een bepaalde tijd
- **Ontvangst van een signaal**
  - Alles wat niet bij het voorgaande past

7.4.1 Automaten

Automaat = een object met maar één operatie.

- Bij elke aankomst van een bericht is er een overgang naar een staat (eventueel dezelfde).
- Eventueel kan het object één bericht naar een ander object sturen.
- De werking van het object wordt volledig beschreven door de overgangen.

Hoofdstuk 8. Terug naar het klassenmodel

Klassenmodel	Computerprogramma
Minder technische informatie	Variabelen hebben een type
Geen onderscheid tussen public en private ➔attributen moeten zichtbaar zijn	Attributen zijn meestal verborgen
Conceptueel Nadruk op duidelijkheid	Gericht op implementatie Nadruk op efficiëntie: ✓ Werkt snel ✓ Makkelijk te onderhouden

8.1 Attributen en toestand

- De toestand van een object wordt aangeduid als een lijst van waarden van attributen
- De waarde van een attribuut is NOOIT een ander object
- De id-sleutel in een databank is GEEN attribuut in het klassenmodel.
  - Tenzij als sleutel een echt attribuut gebruikt wordt
- Staat van objecten
  - Wordt meestal gegeven door een aantal (logische) variabelen
  - Als een klasse verschillende staten heeft, moeten deze af te leiden zijn uit de waarden van attributen

8.2 Operaties en methodes

Operatie = een actie die door een object wordt uitgevoerd.

- Wordt ingeleid door een bericht dat ontvangen wordt
- Hoort bij het object dat het bericht ontvangt
- Sommige operaties zijn **polymorf**: toepasbaar op verschillende klassen
- Een *methode* is de implementatie van een operatie voor een klasse
- Kan resulteren in een aantal dingen:
  - Wijzigen van de toestand van het object
  - Sturen van berichten naar andere objecten
  - **Navraag:** Terugsturen van een bericht

Om te vermijden dat het diagram overladen en onduidelijk wordt:

- Navragen naar attributen worden niet apart vermeld als operaties
- Andere navragen worden beschouwd als *afgeleide attributen*

Samenvattend:

- Maximum drie vakjes in de klassenrechthoek:
  - Naam
  - Attributen
  - Operaties

8.3 Associaties

Vuistregel:
Vermeld op een diagram óf geen enkele pijl, óf alle pijlen, zodat men duidelijk kan zien waar er geen pijl aanwezig is.

Afgeleide associaties:

- Alleen vermelden als ze belangrijk zijn
- Hoeft theoretisch niet navigeerbaar te zijn

8.4 Veralgemening en overerving

8.4.1 Soorten overerving

Abstracte klasse = een klasse die geen objecten kan hebben, maar wel deelklassen. Ze is een onvolledige beschrijving van objecten die op verschillende manieren wordt aangevuld door de beschrijvingen van de deelklassen.

Concrete klasse = een klasse die wel elementen kan hebben.

Overerving = de deelklasse neemt alle eigenschappen van zijn ouder over

Uitbreiding = een deelklasse kan nieuwe eigenschappen hebben

Restrictie = de deelklasse legt voorwaarden op aan objecten van de bovenklasse
Als gevolg van restrictie kunnen operaties op twee manieren veranderen:

- Bepaalde operaties worden eenvoudiger
  - Efficiëntere implementatie
- Bepaalde operaties brengen een object buiten de deelklasse
  - Vb.: een cirkel in één richting uitrekken levert geen cirkel op
  - Oplossingen:
    - De operatie in kwestie wordt beperkt of onderdrukt door de deelklasse
      Er ontstaan twee soorten objecten:
      - Objecten in de deelklasse, waar we operaties niet op kunnen toepassen
      - Objecten in de bovenklasse die voldoen aan de voorwaarde, maar toch niet in de deelklasse zitten
    - Model met boven- en onderklassen laten vallen:
      - Een staat definiëren in de bovenklasse
      - Nadeel: restricties met extra operaties worden niet netjes geïmplementeerd

8.4.2 Methodes overschrijven

Mogelijke redenen:

- Omwille van de extensie
- Omwille van efficiëntie
- Omwille van restrictie

Vuistregels:

- Navragen worden gewoon overgeërfd
- Veranderingen van toestand worden overgeërfd bij gewone deelklassen, en bij restricties zolang de restrictievoorwaarden niet verbroken worden
- Veranderingen die restrictievoorwaarden verbreken wijzen op problemen
    - Eventueel vermijden door het invoeren van een speciale staat
- Operaties die overschreven worden moeten hetzelfde gedrag vertonen, en bij voorkeur dezelfde parameters hebben

Meervoudige overerving = een klasse kan verschillende bovenklassen hebben

8.4.3 Overerving omwille van opportuniteit

In de praktijk een van de belangrijkste vormen van overerving.

Voor dit type overerving mag van alle voorgaande regels afgeweken worden.

Komt voor als men niet om logische redenen maar om praktische redenen een bovenklasse met deelklassen invoert.
Eigenlijk wil men de bestaande klasse gewoon vervangen door een nieuwe; men wil echter die nieuwe klasse niet van de grond af opbouwen.

8.5 Functionele modellen

- Wordt verwaarloosd door UML
- Hoort noch bij het statische klassenmodel, noch bij het dynamische model

Inhoud:

- Geeft functies of gegevensstromen weer
- Beschrijft hoe de resultaten van een berekening worden afgeleid van bepaalde gegevens
- Bestaat uit verschillende gegevensstroomdiagrammen
- Laat zien *wat* er moet gebeuren, maar niet *door wie* of *waarom*
- Bevat activiteiten en de gegevensstroom ertussen
- Objecten en actoren worden alleen vermeld voor zover ze de oorsprong of het doel van de gegevens zijn, of als ze tussentijdse resultaten opslaan

Beslissingsruit = een ruit met een ingang en meerdere uitgangen. Bij elke uitgang staat een wachter: de voorwaarde geeft aan of deze wachter gebruikt wordt.

8.6 Bijkomende documentatie

Woordenboek = bevat een beschrijving van alle termen die in de diagrammen te vinden zijn

- Klassen
- Attributen
- Operaties
- Taken
- Acties
- …

Handleiding = legt communicatie tussen gebruiker en systeem vast

- Op papier
- Elektronisch

Hoofdstuk 9. Behoefteanalyse en modellering

Het doel is *niet* om vast te stellen wat de klant *denkt* dat hij nodig heeft,
maar *wel* om vast te stellen wat hij nodig *heeft*.

- Behoefteanalyse
  - Takendiagram
- Modellering
  - Beschrijving van de werking van het systeem en zijn omgeving
  - Hier wordt vaak ontdekt dat de behoefteanalyse (nog) niet compleet is

9.1 De behoefteanalyse

Doel: opmaken van een inventaris van de vereisten aan het systeem:

- Wie gaat het systeem gebruiken?
- Wat verwachten de gebruikers van het systeem?
- Hoe geraakt het systeem aan de informatie die het nodig heeft om aan deze verwahctingen te voldoen?

9.1.1 Het resultaat

In principe is het resultaat van de behoefteanalyse een takendiagram.

We hebben een lijst van alle taken en hun beschrijving.
Deze beschrijving moet volgende elementen bevatten:

- Naam
- Actoren
- Benodigde informatie

Er moet ook een verdere beschrijving van elke taak gemaakt worden waarin communicatie tussen systeem en actoren verder worden uitgewerkt.
Twee belangrijke redenen waarom de verdere beschrijving vaak bij de behoefteanalyse wordt gerekend:

- Soms helpt het uitwerken van de interface om de taken beter te definiëren
- De interface wordt vaak mee door de klant bepaald
  - Nauwe interactie tussen klant en ontwerper is typisch voor de behoefteanalyse

Bij het opstellen van deze beschrijving wordt gebruik gemaakt van verschillende technieken:

- **Scenario =** opeenvolging van de handelingen van de gebruiker en de reactie daarop van het systeem
  - Verschillende alternatieve wegen worden in verschillende scenario's behandeld
  - Foutafhandeling wordt niet vermeld in het scenario
  - Hulp en handleiding worden ook niet in het scenario vermeld

- **Prototype =** een versie van het te bouwen systeem die eruit ziet als het uiteindelijke systeem, mar intern eenvoudiger is opgebouwd
    - De bedoeling is om aan te tonen dat de behoefteanalyse gelukt is
    - Vooral belangrijk in de gevallen waarbij er uitgebreide interactie tussen gebruiker en software is
        - Zeer goed om problemen met de gebruikersinterface op te sporen
        - Minder goed in het opsporen van vergeten taken
    - Zeer veel werk ➔ enkel essentiële elementen worden opgenomen
- **Dummy** = geprogrammeerd scenario, werkt niet met echte gegevens

9.1.2 Gebruikersinformatie

- Gedeelte van de informatie wordt verkregen door goed ontworpen interface
- Bijna altijd noodzaak aan bijkomende informatie
    - Beschrijving waarvoor de software dient
    - Beschrijving hoe het programma gebruikt moet worden

Documentatie pas achteraf schrijven = slechte aanpak:

- Uitstel leidt tot afstel
- Grondige kennis van het systeem leidt tot slechte documentatie
- Als je de handleiding tijdens (of vlak na) de behoefteanalyse schrijft:
    - Een handleiding bevat een beschrijving van alle taken van het systeem, en dit kan dus zeer goed het enige resultaat zijn van de behoefteanalyse

9.1.3 Hergebruik bij de behoefteanalyse

- Hier wordt de basis gelegd voor alle hergebruik van bronnen van buitenaf
- Veel mogelijkheden voor hergebruik van ideeën in deze fase
- Kijken naar andere programma's
    - Goede eigenschappen overnemen
    - Slechte eigenschappen vermijden

9.2 Modellering

Eerste stap in programmaontwerp: model maken van de realiteit; dit model vormt de ruggengraat van het programmaontwerp.

WEL	NIET
Algemene probleembeschrijving	
Bestaande objecten	Gegevensstructuren
Wat moet er gebeuren	Hoe moet het gebeuren
Context van de toepassing	Implementatiealgoritmes
Hoeveel middelen beschikbaar zijn	Welke middelen nodig zijn

Het volledige model bevat twee luiken:

- Formele model bestaande uit **UML-diagrammen**
- Bijkomende beschrijvingen, verzameld in een **woordenboek**

De normale weg op tot een goed model te komen:

1) Maak een ruwe schets van het klassendiagram
    a. Belangrijk: alle juiste klassen identificeren
    b. Duidelijke attributen en associaties
2) Maak taak voor taak een sequentiediagram
    a. Controleer altijd het klassendiagram op ontbrekende elementen en vul dit aan
    b. Ga na of aan het functioneel model beschrijvingen moeten worden toegevoegd
3) Maak, indien nodig, statendiagrammen op

9.3 Het statisch model

Bij het opstellen van het klassendiagram houdt men min of meer deze volgorde aan:

1. Identificeer objecten en klassen
2. Zoek de associaties
3. Zoek de attributen van objecten en verbindingen
4. Zoek naar mogelijke overerving
5. Controleer

Houd bij elke stap zowel het klassendiagram als het woordenboek bij en besteed nog niet te veel tijd aan visuele opmaak van het diagram.

Objecten en klassen

- Begin met een lijst **substantieven** uit de beschrijvingen
- Verwijder woorden die naar hetzelfde object of dezelfde klasse verwijzen
    - Houd alleen het duidelijkste woord over
    - Sommige substantieven verwijzen naar een rol die een object speelt in een verbinding
        - Kan nuttig zijn om associaties op te sporen
        - Zet deze woorden in een aparte lijst met mogelijke associaties
- Sommige woorden duiden attributen of associaties aan
    - Houd deze ook apart voor later
- Sommige objecten zijn onbelangrijk
- Keuze maken of je iets als bericht of object zal beschouwen

Associaties

- Worden vaak aangeduid door **werkwoorden**
- Ga opnieuw van veel te veel naar genoeg
- Sommige werkwoorden geven meer een actie aan dan een associatie
- Zoek de multipliciteit
    - Multipliciteit aan de ruitkant van een aggregatie is altijd
        - 1
        - 0..1

Attributen

- Bekijk eerst alle klassen en associaties om attributen te vinden
- Werp al een eerste blik op het takendiagram
    - Waar zit de informatie om taken uit te voeren?

Overerving

- Belangrijk: hier pas aan beginnen nadat alle klassen en attributen gekend zijn
- Is overerving nuttig?

Controle

- Algemene controle van het klassenmodel
  - We willen een *duidelijk* en *logisch samenhangend* model
- Signalen dat er iets mis loopt:
  - Een klasse heeft een dubbele rol
    - Opsplitsen?
  - Een attribuut of operatie hoort niet duidelijk bij de een of andere klasse
    - Klasse ontbreekt?
  - Een klasse heeft geen attributen
    - Is het wel een klasse?
  - Een operatie heeft informatie nodig uit een andere klasse, maar er is geen associatie
    - Staat de operatie verkeerd?
    - Ontbreekt de nodige associatie?
  - Een associatie wordt door geen enkele operatie gebruikt
    - Associatie overbodig?
  - Een attribuut wordt niet gebruikt door de operaties van zijn klasse
    - Is het attribuut wel nodig?
    - Staat het attribuut op de juiste plaats?

9.4 Het dynamische model

Het dynamische model laat het tijdsafhankelijke gedrag van het systeem en zijn objecten zien.

De basis voor het dynamisch model is het takendiagram.
In een informele probleembeschrijving kunnen taken op verschillende manieren beschreven worden.
De voornaamste zijn:

- Vereisten voor het systeem
- Beschrijving hoe gegevens het systeem binnenkomen
- Beschrijving van een taak als een hele reeks samenhangende gebeurtenissen
- Het begin van een taak, door de actie van een actor

Opnieuw: van veel te veel naar genoeg

Voor de detailbeschrijving van de taken gaat men terug naar het klassenmodel:

- Zijn alle gegevens die nodig zijn aanwezig en bereikbaar?
- Zijn alle operaties die nodig zijn om een taak uit te voeren aanwezig en bereikbaar?

De detailbeschrijving kan verschillende vormen aannemen, naargelang de vorm van de taak:

- Een simpele navraag of ingave van gegevens
  - Klassendiagram: kan de plaats van de gegevens bereikt worden?
  - Woordenboek: te volgen pad
  - Eventueel sequentiediagram

- Een enkelvoudige taak
  - Opdracht van gebruiker ➔ respons van het systeem
  - Verdere formele beschrijving is overbodig
  - Informele beschrijving kan nuttig zijn
- Complexe taken
  - Meer beschrijving nodig in de vorm van
    - Functionele diagrammen
    - Sequentiediagrammen

Sequentiediagrammen worden meestal opgesteld aan de hand van scenario's. Indien deze scenario's niet zijn opgemaakt bij de behoefteanalyse, dient dit nu te gebeuren.

Na het opmaken van sequentiediagrammen dient men de nodige statendiagrammen te maken.
Er zijn bijna zeker verschillende staten als:

- Bepaalde operaties niet altijd kunnen uitgevoerd worden
- Een object soms wel, en soms geen berichten als antwoord op een ontvangen bericht stuurt, en dit op basis van zijn toestand

In het algemeen wijst elke voorwaardelijke splitsing in een sequentiediagram op verschillende staten.

In het woordenboek worden de essentiële kenmerken van elke staat vermeld.

En verder:

- Een bericht moet een duidelijke afzender en ontvanger hebben
- Alle staten van een object worden beschreven door een toestand
  - Er moet een attribuut zijn dat aangeeft in welke staat een object zich bevindt

9.5 Het functionele model

Statisch en dynamisch model voldoen niet als er

- berekeningen moeten gebeuren
- een ingewikkelde gegevensstroom is

Meestal is de relatie taak-informatie zeer eenvoudig en is er geen functionele analyse nodig.
Bij meer ingewikkelde gevallen gaan we systematisch te werk:

1. Maak een lijst van informatie bij in- en uitgang van de taak
2. Maak een gegevensstroomdiagram
3. Beschrijf hoe de uitvoer functioneel afhangt van de invoergegevens

9.5.1 Vergelijkende controle

- Onmogelijk aan te tonen dat het opgestelde model goed is
- WEL mogelijk om te bewijzen dat er fouten in zitten

Controleer als dynamische en statische modellen goed op elkaar afgestemd zijn:

- Een object kan alleen een bericht naar een ander object sturen als er een verbinding mee bestaat, die navigeerbaar moet zijn in de juiste richting
- Een object kan enkel een bericht ontvangen als het daarvoor een operatie heeft
- Elk attribuut van een object moet een waarde hebben
- Voor elke associatie moet een object weten met welke andere objecten het verbonden is

Hoofdstuk 10. Ontwerp

9 stappen:

1. Bepalen van hulpbronnen
2. Kijken naar hergebruik
3. Ontwerp van systeeminterface
4. Ontwerp van foutafhandeling
5. Ontwerp van beveiliging
6. Ontwerp van de testen
7. Aanvullen van het model met technische klassen en private functies
8. Dataontwerp
9. Procedureontwerp

Bij kleine systemen kunnen sommige van deze stappen zeer eenvoudig zijn, en bovendien kunnen sommige stappen opgenomen worden in de vorige fase (de behoefteanalyse) of de volgende fase (de realisatie).

10.1 Bepalen van hulpbronnen

Omgeving kiezen waarin het systeem wordt opgebouwd en waarin het moet werken

- Hardware
- Software
- Programmeertaal

10.2 Kijken naar hergebruik

- Bespaart werk
- Leidt tot betere kwaliteit van software
- Hergebruik van
  - Code
  - Ideeën
- Er bestaan bijna zeker al gelijkaardige toepassingen

10.3 Ontwerp van de systeeminterface

Kan eventueel al in een vorige fase gebeurd zijn, door het maken van een prototype.

Standaardisatie:

- Gebruikersinterface moet logisch zijn
- Gebruiker moet intuïtief kunnen handelen

10.4 Ontwerp van de foutafhandeling

Gestandaardiseerde foutafhandeling:

- Vergemakkelijkt programmeren en testen
- De code controleert als aan alle vereisten voldaan is
  - Variabelen hebben een waarde
  - Hulpbronnen zijn aanwezig
  - …
- Bij gebruik van `throw-catch`:
  - Waar wordt de `exception` opgevangen?
  - Hoe wordt de `exception` opgevangen?

10.5 Ontwerp van de beveiliging

Twee aspecten:

- Beveiliging tegen interferentie door onbevoegden
  - Vooral bij internettoepassingen
  - Hoe voorkomen?
  - Niet alleen beschermen tegen kwaadwillige aanvallen, maar ook tegen onwillekeurige foutieve handelingen van gebruikers
    vb. per ongeluk wissen van bestanden
- Mogelijkheid tot falen van hard- en software
  - Systeem buiten gebruik
  - Gegevens kunnen verloren gaan
    - Regelmatige backups!

Bij grote systemen

- Definiëren van deelsystemen
  - = een geheel van klassen met een vrij nauwe samenhang en duidelijk omschreven functies
- Systeem ontplooien
  - Eng.: *to deploy*
  - = het toewijzen van verschillende delen van het systeem aan verschillende toestellen of logische delen van de hardware

10.6 Ontwerpen van de testen

- Iedereen maakt fouten
- Mate van testen is afhankelijk van de kost van fouten
  - Grotere individuele kost per fout ➔ meer testen
- Testen door gebruiken is inefficiënt
- Nadeel van testen:
  - Nadat een fout is opgetreden is ze nog niet gelokaliseerd
  - Opsporen van de foutoorzaak kan veel tijd in beslag nemen
- Aparte componenten testen
  - Methodes van elke klasse
  - Elke klasse in haar geheel

10.7 Aanvullen van het model met technische klassen en private functies

- Toevoegen van klassen en/of objecten
  - Meestal containers
  - Klasse ≠ module
    - Een klasse is een beschrijving van een groep modules, de objecten
- Tabellen
  - Geen instantiatie van `class`
  - Kan wel een object vormen
- Programmeerprincipes die leiden tot het gebruik van `private`:
  - Schrijf nooit code twee keer
    - Als twee operaties gemeenschappelijke code delen, moet deze ondergebracht worden in een aparte operatie
  - Deel code logisch in
    - Als een operatie uit verschillende, duidelijk te onderscheiden, delen bestaat, moet deze opgesplitst worden

10.8 Dataontwerp

Grote systemen:

- Afgescheiden van de implementatie

Kleine systemen:

- Resultaat van het ontwerp wordt direct in code uitgedrukt

10.9 Procedureontwerp

Zie Dataontwerp.

Hoofdstuk 11. Realisatie

Realisatiefase: omzetten van het model in code

Alleen het statische model wordt expliciet gecodeerd. Het dynamische en functionele model worden niet apart geïmplementeerd: het zijn hulpmiddelen bij het uitwerken van de operaties/methodes.

Uitwerking van het statische model:

- Dataontwerp: implementatie van de gegevens
- Procedureontwerp: implementatie van de operaties

11.1 Dataontwerp

11.1.1 Attributen

Gegevens worden ingedeeld in drie soorten:

- **Omgevingsgegevens** die direct beschikbaar zijn
  - Worden opgeslagen in het geheugen of op een of ander medium
  - Vb.: een printer kan aan- of uitgeschakeld zijn
- **Niet-permanente gegevens**
  - Geldigheidsduur: niet langer dan de tijd dat een bepaald programma loopt
  - Kan worden opgeborgen in programmavariabelen
- **Permanente gegevens**
  - Zijn ook geldig als er geen programma draait
  - Worden opgeborgen in bestanden
  - Een object of klasse kan over verschillende bestanden verspreid zijn

11.1.2 Afgeleide attributen en statenvariabelen

Objectgerichte talen:

- Private (data hiding)
- Public

Uniformiteitsregel:

Alle attributen moeten privaat zijn en mogen enkel door middel van operaties publiek gemaakt worden.

Statenvariabele: een afgeleid attribuut dat aangeeft in welke staat men zich bevindt

11.1.3 Permanente gegevens en programma's

Permanente gegevens moeten op een of andere manier gelinkt worden aan variabelen in het programma om er gebruik van te kunnen maken.

Door de variabele te initialiseren wordt ze verbonden met een object. Dit is zeer duidelijk als de gegevens van de objecten in een bestand staan: door de gegevens in te lezen wordt de variabele verbonden met het object, dat deel is van het bestand.

11.1.4 Associaties

- Gaat het om permanente of niet-permanente gegevens?
- Levensduur kan nog korter zijn dan e loop van een operatie
- Multipliciteit
    - Associatie langs beide zijden beperkt: wijzers
    - Associatie onbeperkt: gelinkte lijst van wijzers

De gelijkenis tussen het klassendiagram en het ER-diagram (**Entity-Relation)** is vrij los. Het is meestal vrij gemakkelijk om een ER-diagram uit het klassenmodel af te leiden, en objectgericht ontwerp kan dus makkelijk gebruikt worden voor het ontwerp van een databank.

Duurzame verbinding: bestaat langer dan de uitvoering van een operatie.

11.2 Procedureontwerp

Bij het dynamisch ontwerp worden operaties omgezet in procedures. Uiteindelijk komt een operatie overeen met de verwerking van een binnengekomen bericht.

Indien een operatie voorkomt bij verschillende klassen, dan heeft elke klasse een zogenaamde methode voor de operatie.

- **Precondities**
    - Geven aan welke voorwaarden vervuld moeten zijn opdat de operatie haar taak zou kunnen uitvoeren
    - *"Wat mag de operatie van de wereld verwachten"*
- **Postcondities**
    - Geven aan wat er allemaal gebeurd moet zijn als de operatie gedaan is
    - "*Wat kan de wereld van de operatie verwachten"*

Beschrijving van een procedure (ook bij kleine modellen!):

- Definiëren van de omgeving
  - Wat zijn de relevante gegevens waarvan de procedure gebruik kan maken?
    - Het ontvangen bericht (bv parameters van de functieoproep)
    - De toestand van het object waar de operatie bij hoort
  - Beschrijven van foutafhandeling
    - In een robuust client-servermodel vertrouwt de server nooit een client
  - Beschrijving van de operaties van andere objecten die de procedure mag gebruiken
  - Beperkingen op de waarden van attributen moeten worden ondergebracht in een aparte staat
- Definiëren van de mogelijkheden
  - Aangeven welke extra bronnen kunnen gebruikt worden
- Definiëren van het resultaat
  - Wat moet er veranderd zijn?
    - Bij objectgericht ontwerp kan dit enkel de toestand zijn!
  - Eventueel versturen van berichten
- Beschrijven van algoritmen
  - Geven aan hoe van men van de precondities naar de postcondities kan gaan
  - Functioneel model kan als leidraad gebruikt worden

De interface tussen een methode/lidfunctie en de rest van het systeem wordt beschreven door:

- Hoofding van de lidfunctie
- Gegevensvelden van een object

De interface moet zo klein mogelijk zijn:

- Geef geen overbodige parameters mee aan een lidfunctie
- Maak geen attributen die niet de toestand weergeven

De overgang van pre- naar postcondities is niet altijd triviaal, de operatie moet dan verder worden opgesplitst. Hiervoor zijn twee mogelijkheden:

- Een samenhangend deel van de te volgen procedure onderbrengen in een private hulpoperatie
- De verschillende delen in lijn samenhouden
  - Geen aparte operaties voor delen code
  - Operatie wordt ingedeeld in verschillende delen die na elkaar geprogrammeerd worden

11.3 Niet-objectgerichte talen

Enkele basisregels om een objectgericht ontwerp om te zetten in een niet-objectgerichte taal:

- Gegevens moeten gegroepeerd worden volgens objecten
  - Soms bestaan hiervoor technieken (Vb.: `struct` in C)
  - Als geen technieken voorhanden zijn: begrijpelijke naamgeving
- Operaties dienen gelinkt te worden aan objecten, hiervoor bestaan twee technieken:
  - Wel groeperingstechniek aanwezig:
    - Men zal steeds het object als eerste parameter meegeven, met de naam *self* (≈ *this* in C++/Java)
  - Geen groeperingstechniek:
    - Klassenaam als prefix voor de operatienaam gebruiken

In dergelijke talen is echter geen *overerving* of *data hiding* mogelijk.

11.4 Programmeerstijl

Voordelen van goed leesbare code:

- Vermijden van bepaalde fouten
- Zorgt voor duidelijkheid
- Makkelijk te wijzigen en te hergebruiken

Programmeren heeft te maken met duidelijkheid. Drie belangrijke aspecten hiervan zijn:

- Goede code heeft een duidelijke vormgeving
- Goede code heeft een eenvoudige logische structuur
- Goede code is gedocumenteerd
  - Gebruik duidelijke namen
  - Commentaar waar nodig

11.4.1 Vormgeving

- Indentatie
- Gebruik van witruimte
  - Spaties
  - Nieuwe lijnen
  - Blanco lijnen

11.4.2 Logische structuur

- Vermijd zo veel mogelijk verrassingen
  - Gebruik geen te ingewikkelde constructies
- Voorwaardelijke uitvoering
  - `If-else`
  - Meestal ok
- Meervoudige splitsing
  - `Switch-case`
  - Mag alleen gebruik worden als er een variabele is die duidelijk aangeeft welke tak van de meervoudige splitsing moet worden uitgevoerd
- Lus met teller
  - `For`
  - Geef duidelijk aan wanneer de lus stopt
  - Geen sprongen uit de lus maken! (cfr. `break`)
  - Tellerwaarde niet willekeurig veranderen
- Lus zonder teller
  - `While`
  - Opletten dat de lus niet te veel of te weinig uitgevoerd wordt

Er is één uitzondering wat betreft de opeenvolging van opdrachten: `throw`.

11.4.3 Documentatie

Naamgeving

- Leesbaar en zinvol
- Opletten met nietszeggende lange namen (Vb.: *geheelgetal*)
- Korte variabelen zijn toegelaten in drie gevallen:
  - Lopende index van een for-lus
  - Parameter van een (wiskundige) functie
  - Als het programma gebaseerd is op een tekst of op wiskundige formules waar korte namen gebruikt worden
- Constanten:
  - Hoofdletters

Commentaar

- Alles wat niet direct duidelijk is
- Twee soorten commentaar:
  - Detailcommentaar
    - Kleine dingen toelichten
  - Algemeen commentaar
    - Geeft van een stuk code aan wat ze doet

11.5 IDE

IDE = Integrated **D**evelopment **E**nvironment

- Specifieke ondersteuning voor het editeren van code in een bepaalde taal
- Mogelijkheid om vlot hulp te vinden over beschikbare functies, operaties en klassen
- Koppeling met de compiler
    - Van foutmeldingen direct naar de juiste plaats in de code springen
    - Aanmaken van projecten
        - = structuur die aangeeft welke afhankelijkheden er zijn tussen de codebestanden
- Koppeling met de debugger