OO Ontwerp I

1 Objecten

1.1 Inleiding

Software objecten zijn een weerspiegeling van 'real world' objecten. Net zoals deze 'real world' objecten hebben software objecten ook:

- Eigenschappen
- Een eigen gedrag
- Kunnen ze onderscheiden worden

Software toepassingen zijn ook gelijkaardig aan 'real life':

- Complexe interactie tussen veranderende software objecten
    - Een object kan ook een andere object vragen om iets te doen.

Via abstractie ontstaan soorten of type objecten, deze soorten hebben dezelfde eigenschappen en hetzelfde gedrag. → Klasse

Een klasse bevat dus de omschrijving van eigenschappen en gedrag van soortgelijke objecten.

Een object is bijgevolg een instantie van een klasse.

1.2 Objecten

In de OO-wereld is alles een object, een object wordt beschreven door een klasse, een object is een instantie van een klasse.

Elk object heeft een toestand, een gedrag en een identiteit.

1.2.1 Toestand

De toestand omvat:

- Alle eigenschappen van het object
- De huidige waarden van alle eigenschappen van het object

De toestand van een object is omschreven in de klasse. Een eigenschap wordt namelijk voorgesteld door een attribuut.

Een attribuut bestaat uit een naam en type.

1.2.2 Gedrag

Het gedrag omvat:

- De diensten die het object aanbiedt
- Wat het object kan doen

Objecten sturen boodschappen naar andere objecten. Zo'n object doet beroep op diensten van het 2e object.

Objecten ontvangen boodschappen van andere objecten. Zo'n object doet iets op vraag van het 2e object of het object voert iets als reactie uit op de ontvangen boodschap.

Het gedrag van een object is in de klasse omschreven. Het gedrag wordt namelijk voorgesteld door methodes.

Een methode is een beschrijving van een duidelijk afgebakende taak.

Het gedrag wordt gecodeerd en niet getoond in de klasse.

Een methode declaratie heeft:

- Een naam
- Een eventuele parameterlijst
- Een eventueel returntype
- De code die de reactie beschrijft

We onderscheiden vier soorten operaties:

- Operaties om nieuwe objecten aan te maken → Constructor
- Operaties om informatie te vragen over een attribuut van het object → Getter
- Operaties om een bepaald attribuut van het object te wijzigen → Setter
- Operaties om een object een bepaalde actie te laten uitvoeren → Acties

Overloading: zelfde naam voor methode maar verschillende parameterlijst

1.2.3 Identiteit

Ieder object heeft een unieke identiteit.

- identiteit maakt het mogelijk een object te onderscheiden van elk ander object, onafgezien van zijn toestand
- twee objecten, met identiek dezelfde eigenschappen blijven twee unieke, verschillende objecten

1.2.4 Relaties

Een applicatie bestaat uit objecten die samenwerken.

Interactie is enkel mogelijk wanneer de objecten elkaar kennen. Dit wordt gerealiseerd door relaties tussen objecten te leggen.

1.3 UML-klassendiagram

Klassenaam
omschrijving toestand (opsomming attributen)
omschrijving gedrag (opsomming methodes)

Klassenaam:	gecentreerd bovenaan.	
Toestand:	1. visibiliteit 2. naam	3. type 4. Standaardwaarde
Gedrag:	1. visibiliteit 2. naam	3. parameters 4. Datatype
Contructor:	1. visibiliteit 2. naam	3. parameters

1.3.1 Inkapseling

- Laat flexibel software ontwerp toe
- Behoedt objecten, en dus het systeem, om in ongeldige toestanden verzeild te raken

Inkapseling wordt gerealiseerd via:

- Visibiliteit: private/public
    - Dit moeten we zelf instellen
- Gescheiden signatuur en implementatie van operaties
    - Dit is altijd zo

inkapseling via een klasse:

- Maak een binnenkant die andere objecten niet kunnen zien
    - Gebruik private attributen
    - Gebruik eventueel enkele private methodes
        - Ongeacht visibiliteit is de implementatie van methodes steeds verborgen
- Maak een buitenkant die andere objecten wel kunnen zien
    - Dit noemen we de interface van de klasse
    - Gebruik publieke methodes
    - Voorzie alles wat met objecten moet kunnen gedaan worden…
    - Merk op dat de implementatie van de methodes niet zichtbaar is voor de buitenwereld

2 OO ontwerp en GRASP

2.1 Sequentiediagram

Een sequentiediagram (SD) toont hoe de objecten samenwerken om een bepaalde systeemoperatie te realiseren.

We bekijken eerst de onderdelen van een SD:

- Deelnemers (hier objecten) en hun levenslijn
- Boodschappen
    - Tussen de deelnemers
    - Found message: boodschap die van buiten het systeem verstuurd is.
    - Creatie en vernietigen van een object
    - Objecten die hun eigen methodes oproepen
    - Boodschap = methodesignatuur
- Frames
    - Herhaling (loop)
        - Loop(boolean)
        - Loop(n,m)
        - Loop(for each object in collection)
    - Selectie (alt)
    - Optioneel (opt)
    - Altijd een guard plaatsen
    - Geneste frames
        - Frames die zich binnen een andere frame bevinden
- Relateren van interactie diagrammen

2.2 Design Diagram Class

Het DCD is een UML klassendiagram waarop softwareklassen worden gemodelleerd.

Uitgedrukt door een UML klassendiagram.

2.2.1 Associaties

Associaties op een DCD geven aan wie wie kent…
= navigeerbaarheid

Associaties op een DCD worden attributen in een softwareklasse.

Een associatie heeft:

- Visibiliteit
- Rolnaam
- Multipliciteit
- Navigeerbaarheid

Via associaties op een DCD kunnen we aangeven wie wie kent…
Navigeerbaarheid wordt hier belangrijk…
Rolnaam en multipliciteit wordt enkel aan de navigeerbare kant van de associatie vermeld.

2.3 Responsibility-Driven Design

- Software bestaat uit objecten die samenwerken
- Elk object speelt een rol in het geheel
- Elk object neemt een deel van de verantwoordelijkheden op zich (naar gelang zijn rol)

Tijdens het ontwerpproces ligt de focus op het identificeren van verantwoordelijkheden en het toewijzen van verantwoordelijkheden aan de gepaste klassen/objecten.

2 soorten verantwoordelijkheden:

- Doing verantwoordelijkheden van een object:
    - het object doet zelf iets, zoals het berekenen van een waarde of het creëren van een ander object
    - acties in andere objecten initiëren
    - beheren en coördineren van activiteiten in andere objecten
- Knowing verantwoordelijkheden van een object:
    - weet van eigen private data
    - weet van gerelateerde objecten
    - weet van dingen die het kan berekenen of afleiden

2.4 General Responsibility Assignment Software Patterns

Een patroon heeft

- een naam
- een omschrijving van het probleem
- een omschrijving van de oplossing

GRASP patronen beschrijven de fundamentele principes voor het toekennen van verantwoordelijkheden aan objecten.

Verantwoordelijkheden worden gerealiseerd via methodes.

GRASP omvat 3 patronen.

2.4.1 Controller

Probleem:

Welk is het eerste object, achter de User Interface, dat verantwoordelijk is voor het ontvangen en coördineren van een systeemoperatie?

Oplossing:

Ken de verantwoordelijkheid toe aan een klasse die het volgende representeert:

- het volledige systeem, of een belangrijk onderdeel ervan, een 'root object' (~facade controller)
- een use case scenario waarin de systeemoperatie voorkomt (~use case, of session controller)

Onthoud:

- Controller verleent toegang tot de domeinobjecten.
- Ze ontvangt en coördineert systeemoperaties.
    - Ontvangt
        - De user interface roept diensten op van de DomeinController, deze is het enige aanspreekpunt van de domeinlaag die de business logica bevat)
    - Coördineert
        - Kan een tussentoestand bijhouden, waardoor het de volgorde van afhandelen van use case-stappen kan verzekeren. Dit kan zijn binnen één en dezelfde use case of over verschillende use cases heen.

Good practice:

- Opsplitsing 'knowing' en 'doing' verantwoordelijkheden
    - een methode implementeert ofwel een 'doing' verantwoordelijkheid, ofwel een 'knowing' verantwoordelijkheid, maar niet beide!
    - indien een systeemoperatie beide soorten verantwoordelijkheden bevat zullen we die implementeren adhv twee methodes…

2.4.2 Expert

Probleem:

Wat is het algemeen principe dat we kunnen hanteren voor het toekennen van verantwoordelijkheden aan objecten?

Oplossing:

Ken de verantwoordelijkheid toe aan de Information Expert - dit is de klasse die alle informatie, nodig om de verantwoordelijkheid te realiseren, heeft.

Good practice:

- bepaal heel duidelijk de verantwoordelijkheid
    - kijk eerst op het DCD of daar geen klassen zijn die in aanmerking komen…
    - in tweede instantie kan het DomeinModel een inspiratiebron zijn voor een geschikte klasse…
    - mogelijks maak je gewoon een nieuwe klasse, die een duidelijke rol en verantwoordelijkheid krijgt

2.4.3 Creator

Probleem:

Wat is er verantwoordelijk voor het aanmaken van nieuwe instanties van een klasse?

Oplossing:

Ken de verantwoordelijkheid om instanties van klasse A te creëren toe aan klasse B, wanneer 1 of meer van de volgende gelden:

- B bevat instanties van A (associatie: bevat, heeft)
- B is een aggregatie of een compositie van A
- B gebruikt A intensief
- B bevat de data om A te initialiseren

Good practice:

- kijk eerst op het DCD of daar geen klassen zijn die in aanmerking komen…
- in tweede instantie kan het DomeinModel een inspiratiebron zijn voor een geschikte klasse…
- mogelijks maak je gewoon een nieuwe klasse, die een duidelijke rol en verantwoordelijkheid krijgt

Vaak komt bij creator pattern een repository naar boven als de oplossing.

UI	Domeincontroller	Domeinlaag
Eenvoudige datatypes		objecten

4 Het drielagenmodel

4.1 Applicatie

Goede applicatie?

- Voldoet aan de verwachtingen
    - Verificatie door ontwikkelaar
    - Validatie door klant
- 100% bugvrij kan nooit gegarandeerd worden, maar kans op bugs zo klein mogelijk maken

Bij een goed object georiënteerd ontwerp krijgt elke klasse een eigen duidelijk afgebakende verantwoordelijkheid.

op een hoger niveau kunnen we ook groepen van klassen maken die een duidelijk afgebakende verantwoordelijkheid hebben.
→ software architectuur

Scheid de klassen met verantwoordelijkheden voor:

- opslag van gegevens
- verwerking van gegevens
- presentatie van gegevens

Domeinlaag:

- Het domein bevat de gegevens die in een applicatie een wezenlijke rol spelen.
- Het domein bevat de logica van de applicatie: d.i. de onderlinge verbanden tussen de gegevens, bewerkingen en verder alles wat met de gegevens gedaan moet worden, maar niet de in- en uitvoer van de gegevens.

Presentatielaag

- Alle in- en uitvoer behoort tot de presentatielaag

4.2 Architectuur: drielagenmodel

- Een laag is een groepering van klassen die verantwoordelijk zijn voor een belangrijk onderdeel van de software
- Elke laag heeft eigen verantwoordelijkheden
- De meeste meerlagenarchitecturen kennen minstens 3 lagen

Presentatielaag	Domeinlaag	Persistentielaag
Alles wat met invoer en uitvoer te maken heeft	Alles wat met de businesslogica te maken heeft	Alles wat met de opslag van gegevens te maken heeft

Iedere laag steunt op de onderliggende laag: een laag maakt gebruik van de diensten aangeboden door de onderliggende laag.

Door gebruik te maken van packages kunnen we onze klassen groeperen in overeenstemming met de lagen in onze software architectuur.

In UML stellen we dit voor via een package diagram.

De StartUp-klasse, deze bevat de main-methode, we maken een domeincontroller, en geven deze door aan een initieel GUI object…

4.2.1 De presentatielaag

- De logica in een visuele representatie (een scherm genaamd) beperkt zich tot:
    - Representatie gegevens
        - Een scherm presenteert de gegevens aan de gebruiker
    - Wijzigen gegevens
        - Een scherm ondersteunt de gebruiker bij het invoeren en wijzigen van gegevens, zowel met visuele hulpmiddelen, als met formattering en validaties
    - Doorgeven acties
        - Een scherm fungeert als doorgeefluik voor acties die de gebruiker initieert op het scherm, deze acties worden aan de domeinlaag doorgegeven.
- Een scherm is passief
    - Zelfs het ophalen van de te representeren gegevens is de verantwoordelijkheid van het domein...

Net zoals bij de ontwikkeling van de domeinlaag gegronde ontwerpprincipes worden gebruikt, hoort dit ook zo te gebeuren voor de presentatielaag.

4.2.2 De domeinlaag

De domeinlaag is het hart van de applicatie. Hierin wordt er besloten wat er moet gebeuren met de invoer van de gebruiker, worden validaties, berekeningen en bedrijfslogica uitgevoerd, en vindt het transport plaats van de gegevens uit de database naar de presentatielaag en omgekeerd.

De domeinlaag bevat:

- "echte" domeinklassen (zie DCD)
- De domeincontroller (aanspreekpunt voor de presentatielaag)
    - De domeincontroller bevat een publieke methode voor elke systeemoperatie
- Eventuele usecase-controllers
    - Bij grotere applicaties kan de domeincontroller te groot worden
    - Om beheersbaarheid te kunnen blijven garanderen kunnen de operaties nog ‘logisch’ opgedeeld worden per use case.
    - Voor elke use case heb je dan een UseCaseController
        - De domeincontroller delegeert zijn werk dan als het ware naar deze UseCaseControllers die op hun beurt andere klassen in het domein aanspreken.

- Controller klassen
    - Voeren use cases van de applicatie uit
- Domeinklassen
    - Realisatie van concepten uit het domein van de applicatie
        - Individuele instanties
        - Lijst van instanties
    - Realisatie van de applicatiespecifieke bedrijfslogica
        - Validaties, berekeningen, …
- Vanuit deze laag kan communicatie plaatsvinden met de persistentielaag
    - Opvragen van gegevens en persisteren van wijzigingen
    - Deze communicatie gebeurt vanuit de "repositories"
        - Andere klassen uit de domeinlaag kunnen communiceren met deze repositories, maar niet rechtstreeks met de persistentielaag

4.2.3 De persistentielaag

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

"Object relational gap":

- SQL datatypes verschillen van Java datatypes
- Associaties
    - Associaties tussen objecten zijn navigeerbaar, in het RM heb je steeds bidirectionele verbanden via PK-FK
    - n:n associaties tussen objecten zijn mogelijk maar n:n verbanden bestaan niet in het RM
- In het RM kent men geen
    - Encapsulatie
    - Overerving
    - Polymorfisme

Om deze kloof te dichten zal er een Object Relational Mapping moeten gebeuren.

ORM:

- Vertaalslag van OO-visie naar RM-visie
- We kunnen ervoor opteren om dit zelf te implementeren
    - Dit is een heel complexe taak…
    - Haalbaar voor kleinschalige projecten
- We kunnen gebruik maken van een ORM tool
    - Hogere productiviteit
    - Betere onderhoudbaarheid

    - Betere performantie
    - DB onafhankelijkheid

De PersistentieController is het enig aanspreekpunt voor de bovenliggende domeinlaag.

Deze klasse PersistentieController:

- is verantwoordelijk voor het aanmaken van connecties naar de database.
- delegeert het werk naar de zogenaamde ‘Mapper klassen’.

Een data klasse (ook wel mapper klasse) heeft volgende verantwoordelijkheden:

- Uitvoeren: formuleert queries of stored procedures voor het
    - C(reate): opslaan van nieuwe gegevens in de database
    - R(ead): ophalen van gegevens uit de database
    - U(pdate): opslaan van gewijzigde gegevens in de database
    - D(elete): verwijderen van 1 of meerdere records uit database
- En laat deze door de database uitvoeren
    - Transformeren: van de typen uit de applicatie naar de typen uit de database en vice versa
    - Maak 1 mapper per tabelhiërarchie (vb klant met verschillende leveringsadressen)

5 Testen

5.1 Waarom

Unit testing vereenvoudigt het leven van een ontwikkelaar.

In UP is unit testen “a way of life” : je schrijft testen VOOR je code schrijft en je schrijft een test telkens je een bug tegenkomt.

Doel:

- Minder fouten
- Minder debuggen
- Betere documentatie
- Betere code: leesbaarheid, aanpasbaarheid, uitbreidbaarheid

5.2 Hoe testen

debugger: menselijke tussenkomst nodig, traag, saai

print statements: menselijke tussenkomst nodig, traag, vervuilde code, "Scroll Blindness"

→ Gebruik unit testen: automatisch, snel, geen code vervuiling en het is leuk, het geeft een goed gevoel.

5.3 Wat is een unit test

Is een methode die test of 1 bepaalde methode doet wat ze moet doen gegeven 1 bepaald concreet geval.

Unit test is de enige test die gedaan wordt door de ontwikkelaars zelf. De andere testen worden door een speciaal testteam gemaakt.

5.4 The 3A pattern

De code in een testmethode bestaat uit 3 delen: de 3A-regel

- Arrange: Initialiseer de nodige variabelen
- Act: Voer test effectief uit, roept de te testen methode op
- Assert: Vergelijkt het bekomen resultaat met het verwachte resultaat

Testen kan je uitvoeren:

- Een test kan slagen → de geteste methode is voor dit concreet geval correct geïmplementeerd.
- Een test kan falen → de geteste methode is voor dit concreet geval NIET correct geïmplementeerd.

Unit testen kunnen ook nagaan of een methode een exception werpt

5.5 Wanneer unit testen aanmaken

Na ontwerp en voor het schrijven van de code van een klasse.

- De ontwikkelaar moet dus vooraf bedenken wat dit stuk code moet doen en wat moet er gebeuren als de code niet (volledig) uitgevoerd kan worden.
- Door duidelijke meldingen en ingebouwde testen kan de goede en foute werking van de unit/class aangetoond worden.

5.6 Wat testen en hoe

Maak een package testen

- Voor elke klasse in domein maken we een testklasse aan
- De testklasse bevat 1 of meerdere test methodes voor elke niet triviale (business) methode binnen de te testen klasse
    - Meestal worden geen testen aangemaakt voor getters
    - Soms wel voor setters (afhankelijk van de functionaliteit die de setter bevat). Meestal worden setters ook door andere methodes getest
- Wat
    - Getallen
    - Verzamelingen
    - Verschillende paden
        - If-then-else
        - Do while, while, for, for each
- Hoe? We kijken naar de input-output en bedenken concrete gevallen
    - Normaal geval(len)
    - Speciale gevallen: lege string, null, grenswaarden
- Elk concreet geval wordt een test methode met een betekenisvolle naam. De naam zegt ons iets over het concrete geval.

Een testklasse kan ook een @Before bevatten.
= Wat ingesteld wordt voordat de test gedraaid wordt. Hier moet de repository een aantal spelers bevatten.

5.7 Voordelen

- Door eerst de test te schrijven, kijk je naar een klasse als een gebruiker van de klass
- Als je, op basis van de UML design van een klasse, moeilijk een test kan schrijven, is deze design verkeerd
- Je krijgt dus snel feedback op UML design van de klasse_
- Kleine blokjes functionaliteit zijn al testbaar
- Tests beschrijven de functionaliteit, vorm van documentatie
- Tests zijn geautomatiseerd (of kunnen geautomatiseerd worden in builds)
- Gestage ontwikkeling : kleine stukjes functionaliteit bijbouwen en je bent zeker dat het bestaande nog steeds werkt.
- Code wordt compacter
- Testen vereist discipline en creativiteit

5.8 Nadelen

- Weerstand bij de ontwikkelaars, mindset kost tijd
- Het schrijven van unit testen kost tijd

- Tijd is geld
- Onder tijdsdruk worden de testen als wisselgeld gezien (kan later)

Het resultaat:

- Merendeel van de ontwikkelaars schrijven testen achteraf

6 Pijler van OO

De 4 pijlers van OO:

1. Inkapseling
2. Abstractie
3. Overerving
4. Polymorfisme

6.1 Abstractie

6.1.1 Abstracte klassen

Geen instantiatie mogelijk.

De naam van de abstracte klasse en de abstracte methoden staan cursief in het UML-klassendiagram.

Een abstracte klasse dient alleen ter overerving. Het is dus altijd de superklasse van één of meerdere subklassen.

Een abstracte klasse kan abstracte en concrete operaties bevatten.

Een abstracte operatie heeft GEEN implementatie in de klasse waarin de operatie gespecificeerd wordt.

Een klasse die erft van een abstracte klasse met een abstracte methode, moet:

- ofwel een implementatie hebben voor die methode
- ofwel zelf een abstracte klasse worden met diezelfde abstracte methode. Deze abstracte klasse is dus terug een superklasse met één of meerdere subklassen.

Van een abstracte klasse kan je geen instantie maken, maar in de constructor van de subklassen kan je wel de constructor van de superklasse aanspreken via super().

6.1.2 Interface

Er is één groot nadeel! Een klasse kan slechts van 1 andere klasse erven.
→ Interfaces kunnen ook zorgen voor polymorf gedrag EN 1 klasse kan meerdere interfaces implementeren.

Een interface is een speciale abstracte klasse die ENKEL abstracte methodes bevat. Een interface is dus een contract.

Een klasse kan een interface-klasse implementeren: de klasse sluit dan een contract af en belooft alle gedrag in de interface beschreven op te nemen.

Een interface kan niet geïnstantieerd worden.

Gebruik:

- Polymorfisme toepassen
- Via interfaces kunnen we het gebrek aan ‘meervoudige overerving’ gedeeltelijk oplossen.

Wanneer:

- Interfaces dienen om gemeenschappelijk gedrag tussen ongerelateerde klassen te definiëren
- Een interface is enkel en alleen een voorschrift van een bepaald gedrag
- Een klasse kan meerdere interfaces implementeren, maar kan maar van 1 klasse erven

6.2 Overerving

Klassen hebben twee soorten afstammelingen:

- Instanties (of objecten) van een klasse
- Subklassen

Klassen kunnen

- Concreet zijn, wat betekent dat er instanties (objecten) van die klasse kunnen bestaan.
- Abstract zijn, wat betekent dat ze geen instanties kunnen hebben.
- Een Interface klasse zijn, deze bestaan enkel uit operaties. Ze beschrijven een contract (gedrag) zonder verdere specificaties.

Om gedupliceerde code te vermijden passen we generalisatie toe: we maken een aparte klasse die de gemeenschappelijke kenmerken (attributen, methoden, associaties) bevat. Deze klasse heet een superklasse.

Bij overerving start je met een superklasse en maak je 1 of meerdere specifieke subklassen aan.
= specialisatie.

In een subklasse wordt het gedrag van de superklasse verder uitgebreid en/of gespecialiseerd.

Overerven is een ‘is-een-relatie’.

Een klasse kan maar van één andere klasse erven.

Een subklasse erft van zijn superklasse:

- Toestand
- Associaties
- Gedrag
    - Een constructor wordt nooit geërfd, maar deze kan wel vanuit een constructor in een subklasse worden aangeroepen m.b.v. super (uitsluitend als eerste opdracht)

Subklassen hebben vanuit code geen toegang tot toestand en gedrag die private zijn in de superklasse

- De visibiliteit van toestand en gedrag:
    - - (private) onzichtbaar voor de ganse buitenwereld
    - + (public) zichtbaar voor de ganse buitenwereld
    - # (protected) zichtbaar voor de subklassen
    - ~ zichtbaar voor alle klassen in de package

- Subklassen hebben wel toegang tot public en protected operaties van superklasse. Hiervoor maakt subklasse gebruik van keyword super
- Subklassen erven wel de toestand (attributen) van de superklasse, maar ze kunnen enkel de toestand raadplegen/wijzigen via de getters en setters (public en protected), daar attributen visibiliteit private hebben

Uitbreiden gebeurt door nieuwe attributen en/of associates en/of methodes toe te voegen.

Specialiseren gebeurt door de implementatie van een methode in een subklasse aan te passen ("*overriding*"genaamd).

Waarom subklasse hiërarchieën gebruiken:

- Laat toe om beschrijving van complexe klassen stapgewijs op te bouwen
- Onderhoudbaarheid : aanpassingen zijn gelokaliseerd op het niveau waar ze gespecifieerd zijn
- Nieuwe subklassen kunnen eenvoudig worden toegevoegd zonder de andere klassen te beïnvloeden.

6.3 Polymorfisme

Polymorfisme = veelvormigheid

Kracht van polymorfisme = dynamische binding

Static binding (at compile time)
Als de methode niet bestaat in de klasse van je referentie dan krijg je een compilatiefout!

Dynamische binding (at runtime)
Bij aanroep van een methode wordt de methode gekozen die in de klasse van het werkelijke object zit. Als deze niet bestaat wordt ketting van superklassen afgelopen tot er een superklasse gevonden wordt die de methode bevat.

6.3.1 Waarom

Je kan je code vereenvoudigen:

- Associatie naar superklasse kan ook subklassen bevatten
- Je kan alle subklassen van een superklasse als de superklasse behandelen
- Polymorfe parameters en returntypes

Je kan je code makkelijk uitbreidbaar maken:

Met polymorfisme dient de implementatie niet te worden aangepast bij toevoegen van nieuwe subklassen.

- Bovendien erft de subklasse alle toestand en gedrag van de superklasse en dien je
- enkel de uitbreidingen en/of specialisaties toe te voegen aan de subklasse