OBJECT-GEORIËNTEERD PROGRAMMEREN: EEN PERSOONLIJK VERHAAL

OVERZICHT

Deze presentatie vertelt het verhaal van mijn zoektoch naar wat object-georiënteerd programmeren (OOP) is:

1. Introductie - Mijn persoonlijke achtergrond en motivatie
2. Context - Als we code schrijven .....
3. Voorbeelden - Praktische voorbeelden van OOP-concepten
4. Conclusie - Wat we hebben geleerd?
5. Vragen - Open discussion over OOP

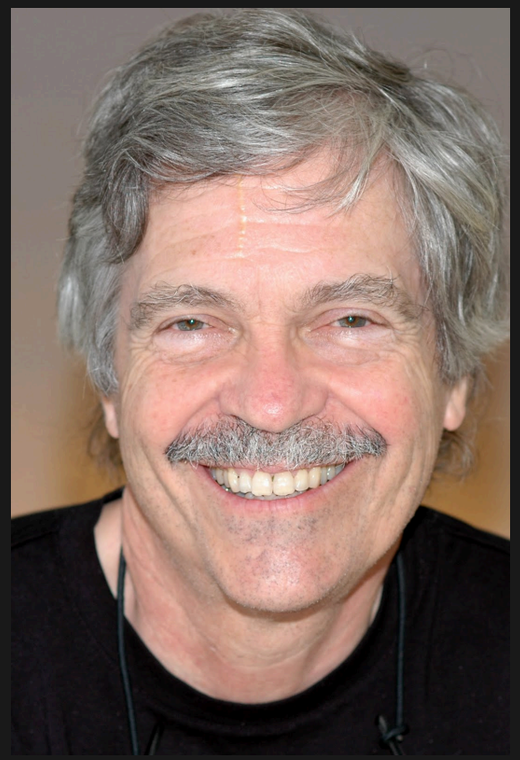
1.1 INTRODUCTIE

- Geschiedenis gestudeerd
- Programmeren om overal te kunnen werken

ACHTERGROND

- 15 jaar: kennismaking met GwBasic op DOS 5.1
- Vanaf 2000 experimenteren met :
  - C++, Delphi, Python, VBScript, Smalltalk
- 2006: junior ontwikkelaar
  - => .Net
  - => MCPD (2008)

1.2 ALAN KAY OOPSLA 1997



- Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind.
- (https://en.wikiquote.org/wiki/Alan_Kay)

Hoe kan dan dat talen als C# en Java OO worden genoemd? Beide lijken heel erg op C++, maar dan met een GC?

1.3 UITDAGING

Voor mij was de uitdaging in 2006 om .Net en de opmerkingen van Alan Kay te verenigen.

N.b. Wie veel leest over Alan Kay dat hij zeer kritisch over de stand van zaken van de het maken van software.

1.4 VOORBEELDEN VAN WAT MIJ BEZIG HIELD

ENUMS

- Smalltalk heeft geen enums
- Als iets geen object is kan nooit een 'message' zijn.
- Maar wat is een 'message' eigenlijk?

PROPERTIES

- Properties: Smalltalk heeft alleen methods
- Hoe kun je 'data hiding' doen met properties?

1.5 ANDERE VRAGEN/OBSERVATIES

- Wat is immutability als ik properties heb? Wat is immutablity? Mag ik dan een lijst nooit meer aanpassen ReadOnlyList?
- Ik zag veel koppelingen tussen code en data, precies zoals dat bij procedureel programmeren gebeurt, dus wat is dan eigenlijk het verschil?
- Hoe zorg ik dat afhankelijkheden minimaal zijn en ook goed beheersbaar blijven?
- Alan Kay heeft code omschreven als *executable math*. Dat geldt inderdaad voor Lisp, maar hoe is dat in gangbare OOP als je zoiets als een *early return* kunt doen of als je de hele tijd data loopt rond te pompen? Gangbare objecten zien er niet uit als wiskundige functies.

1.6 CENTRALE VRAAG

Hoe verenig ik de opmerkingen van de bedenker van de term OOP met de dagelijkse praktijk?

Sinds 2005 is er veel gebeurt op het internet, dus het kan zijn dat de vraag nu wat achterhaald overkomt.

2.1 CONTEXT/DISCLAIMER - DEEL 1

- Er is de afgelopen jaren veel geschreven over OOP op het internet.
- Veel daarvan was niet beschikbaar toen ik dit aan het leren was.
- Het is allemaal heel abstract en ik wilde concreet leren hoe OOP te doen.
- Wat is OOP nou eigenlijk volgens Alan Kay?
  - https://www.purl.org/stefan_ram/pub/doc_kay_oop_en

2.2 CONTEXT/DISCLAIMER - DEEL 1

- Deze links kende ik toen niet
  - https://www.reddit.com/r/programming/comments/brtzgs/alan_kay_did_not_invent_objects/?rdt=59028
  - https://blog.metaobject.com/2019/11/what-alan-kay-got-wrong-about-objects.html
  - https://news.ycombinator.com/item?id=21559526
  - https://www.reddit.com/r/programming/comments/duq83c/what_alan_kay_got_wrong_about_(
  - https://dev.to/ericnormand/programming-paradigms-and-the-procedural-paradox

2.3 WERKOMGEVING: CONTEXT/DISCLAIMER - DEEL 2

- Mijn collega's waren allemaal procedureel getraind.
- Van 2013-2020 werkte ik met een eigen framework van mijn werkgever dat was ontworpen als database direct naar HTML-tabel. Niet erg OOP en sowieso lastig om business-logica toe te voegen
- Geen goede voorbeelden
- Geen collega's die me konden helpen
- Het internet hielp niet voldoende
  - Microsoft tutorial
  - Java tutorial
- Het blog van Yegor Bugayenko werd uitgelegd waarom bepaalde code niet OOP was. Het gaf argumenten én voorbeelden hoe het dan wel moest.

2.4 ZIJSTAP

Waarom is de vraag naar wat is OOP interessant? Inmiddels is de term veel besproken op het internet en eigenlijk verworden tot een rode vlag voor een stier.

Daarom eerst even een korte zijstap: een reflectie over code

2.5 AANNAMES IN CODE

- Inhoud van de code: regels code
- Context van de code: kosten/bedrijfstructuur

2.6 INHOUD VAN DE CODE

- functionaliteit, doet het wat het moet doen
- onderhoudbaarheid, kunnen we makkelijk fouten herstellen
- uitbreidbaarheid, hoe makkelijk kun je nieuwe wensen toevoegen

-> accidental complexity vs essential complexity

2.7 CONTEXT VAN DE CODE

- opstartkosten, wat kost het om zo snel als mogelijk punt te realiseren
- afschrijving, hoelang mag de code meegaan voor het opnieuw gebouwd kan worden

2.8 ONDERHOUDBAARHEID VAN DE CODE

Alles van wat ik zojuist noemde heeft zijn weerslag in de onderhoudbaarheid van code. Uiteindelijk is code bedoeld voor mensen om te lezen en om soms uitgevoerd te worden door apparaten.

=> Aan de slag

2.9 DISCLAIMER

- OOP is een keuze.
- Ik ga niet zeggen wat je wel of niet moet doen.
- De voorbeelden zijn wat triviaal, maar lijken me vanwege de eenvoud heel leerzaam.
- Maar bij echt OOP zijn er wel *do*'s en *don't*-s. Die ga ik behandelen.
- Je mag het oneens zijn met mijn *do*'s en *don't*-s.

Misschien denk je: "Waarom vond je OOP zo moeilijk? en is het je allemaal duidelijk vanaf het begin. Ik wil aangeven dat het mij niet duidelijk was en dat het internet als geheel vol zit met slechte voorbeelden. Dat maakt het tot een valide gespreksonderwerp en als jij het allemaal meteen hebt begrepen dan is dat dus goed voor jou en petje af, dat je in een keer door de rommel heen kon kijken. Dat geldt voor een heleboel andere mensen niet.

3.1 VOORBEELDEN

Aan de hand van de volgende onderwerpen zal ik proberen OOP duidelijk te maken

- Statics/Public/Private constants
- Message passing / composability
- DTO

3.2 VOORBEELD CONSTANTS

```
class Records{
  void write(Writer out) {
    for (Record rec: this.all) {
      out.write(rec.toString());
      out.write(Constant.EOL);
    }
  }
}

class Rows {
  void print(PrintStream pnt) {
    for (Row row : this.all) {
      pnt.printf(
        "{ %s }" , row, Constant.EOL
      );
    }
  }
}
```

```
// Harde koppeling met een constant, die niet aan te passen is.
// ==> Daarom op een andere manier
```

3.2 GEBRUIK EEN OBJECT

```
class EOLString {
  private final String origin;

  EOLString(String src) {
    this.origin = src;
  }

  @Override
  String toString() {
    return String.format("%s\r\n", origin);
  }
}
```

3.3 OBJECTEN HEBBEN EEN INTERFACE

```
class Records {
  void write(Writer out) {
     for (Record rec: this.all) {
      out.write(new EOLString(rec.toString()));
    }
  }
}

class Rows {
  void print(PrintStream pnt) {
    for (Row row : this.all) {
      pnt.printf(
        new EOLString(
          String.format("{ %s }" , row)
        )
      );
    }
  }
}
```

3.4 WIL JE ANDER GEDRAG MAKEN

```
class EOLString {
  private final String origin;

  EOLString(String src) {
    this.origin = src;
  }

  String toString() {
    if (/* this is Windows */ ) {
      throw new Exception(
        "We're on Windows, can't use EOL, sorry"
      );
    }
    return String.format("%s\r\n", origin)
  }
}
```

3.5 MESSAGE PASSING / COMPOSABILITY

- In OOP gaat het over het samenstellen van objecten uit andere kleinere objecten.
- In zuivere OOP sturen objecten alleen berichten naar elkaar.

=> een dergelijk bericht is zelf ook een object.

3.6.1 VOORBEELD 1: PROCEDUREEL VS COMPOSITIE

```
float rate;
if (client.age() > 96 ){
    rate = 2.5
} else {
    rate = 3.0
}
```

3.6.2 VOORBEELD 1: PROCEDUREEL VS COMPOSITIE

```
float rate;
if (client.age() > 96 ){
    rate = 2.5
} else {
    rate = 3.0
}
```

IETS BETER?

```
float rate = new If(
    client.age() > 65 ,
    2.5, 3.0
)

rate.print();
```

3.6.3 VOORBEELD 1: PROCEDUREEL VS COMPOSITIE

KRITISCHE OPMERKINGEN

```
float rate = new If(
    client.age() > 65 ,
    2.5, 3.0
)
```

`rate` is een object, maar in de constructor gebruiken we een vergelijking waar bij we iets over een ander object vragen. Is jouw leeftijd hoger dan een een grenswaarde?

3.6.4 VOORBEELD 1: PROCEDUREEL VS COMPOSITIE

IS DIT DE OPLOSSING?

```
float rate = new If(
    new Greater(client.age(), 65),
    2.5, 3.0
)
```

- Nu is het procedurele weg, maar het object staat nog steeds open.
- `Greater` is niet zo'n duidelijk object

Dat kan beter.

3.6.5 VOORBEELD 1: PROCEDUREEL VS COMPOSITIE

LAATSTE POGING

```
float rate = new If(
    new GreaterThen(
        new AgeOf(client),
        65
    ),
    2.5, 3.0
)
```

- Hier hebben we alleen berichten. We vragen dingen aan objecten.
- Alle objecten zijn immutable.
- We kunnen aan de code opbouw zien wat de bedoeling is.
- Als ik het antwoord wil hebben dan doen we `rate.print()`

3.7.1 VOORBEELD 2: ITERABLE - PROCEDUREEL

```
public Iterable<File> find(String mask) {
    if (mask == null){
        //find all files
    } else {
        // find files by mask
    }
}

// hier vraag je iets óver het object en niet áán het object
```

```
public Iterable<File> find(Mask mask) {
    if (mask.empty()){
        //find all files
    } else {
        // find files by mask
    }
}

// hier vraag je iets áán het object
```

3.7.2 ITERABLE - OOP

```
public Iterable<File> find (Mask mask){
    Collection<File> files = new LinkedList<>();
    for (File file : /* all files*/) {
        if (mask.matches(file)){
            files.add(file)
        }
    }
    return files;
}

// Hier is mask een echt object dat berichten ontvangt

interface Mask {
  boolean matches(File files)
}

// voorkom NULL
class AnyFile implements Mask {
  @Override
  boolean matches(File file){
    return true;
  }
}
```

3.8.1 MESSAGE PASSING / COMPOSABILITY

- Om terug te komen op statics en constants:
- Daarmee kun je geen grotere objecten vormen omdat ze geen constructor hebben.

3.8.2 MESSAGE PASSING / COMPOSABILITY

- In OOP 'vragen' objecten dingen aan elkaar, maar gaan nooit OVER andere objecten.
- Dus nooit vragen ben jij van type X?
- Dat zou vanuit het code pad al duidelijk moeten zijn. Iedere vraag naar een type is een ontwerpfout.

3.8.3 MESSAGE PASSING / COMPOSABILITY

- Geen noodzaak voor *null.*
- Gebruik een default object dat voldoet aan een interface.

N.b. bedenk nogmaals *code as executable math*. *Null* is een *leaky abstraction*. Het betekent er staat niets op de geheugenplek waar je naar verwijst. Wiskundige functies kennen dat niet. Een geheugenplek is een implementatie-detail.

3.9.1 DTO

- Data Transfer Object
- Gegevens transporteren van het ene systeem naar een ander systeem

=> ORM

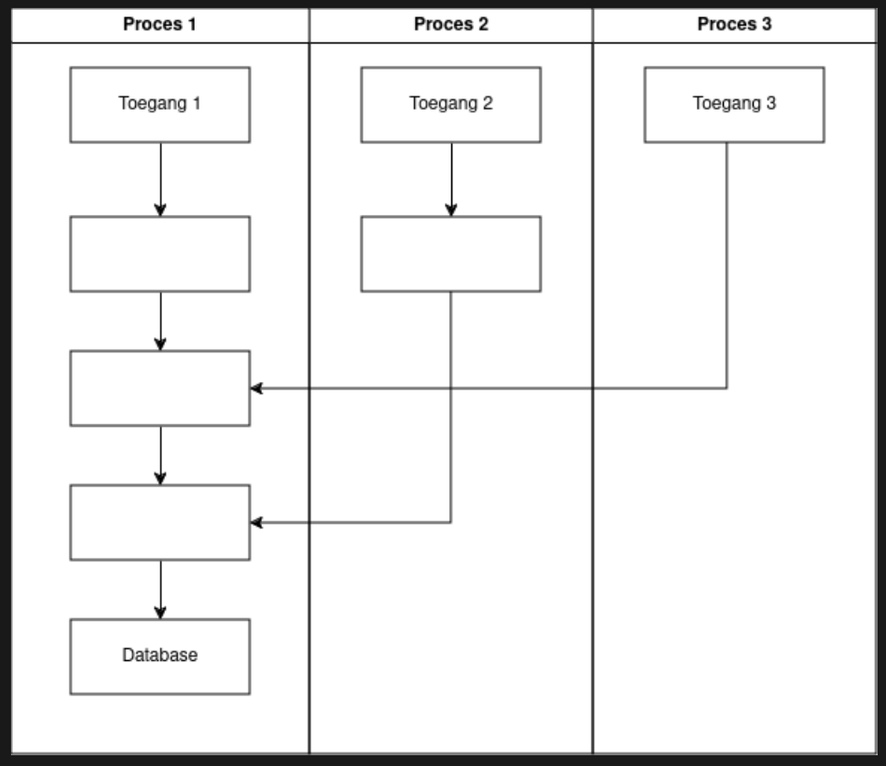
3.9.2 KLASSIEK VOORBEELD: EEN BOEK DTO

```
Book book = api.loadBookById(123);
database.saveNewBook(book);

Book loadBookById(int id) {
  JsonObject json = /* Load it from RESTful API */
  Book book = new Book();
  book.setISBN(json.getString("isbn"));
  book.setTitle(json.getString("title"));
  book.setAuthor(json.getString("author"));
  return book;
}

void saveNewBook(Book book) {
  Statement stmt = connection.prepareStatement(
    "INSERT INTO book VALUES (?, ?, ?)"
  );
  stmt.setString(1, book.getISBN());
  stmt.setString(2, book.getTitle());
  stmt.setString(3, book.getAuthor());
  stmt.execute();
}

// Deze code nodigt uit tot complexiteit
```

3.9.3 DTO

- Gebruik van Properties: huh? Hoe is dat OOP?
- Dat is het niet, want in OOP doen we aan *data hiding*. Properties zetten een object juist wijd open. (ja, het is iets genuanceerder)
- Door het open zetten van een object ben je er nooit zeker van dat alle data ook daadwerkelijk aanwezig is.

3.9.4 COMPLEXITY



Meerdere controlepunten

Vereenvoudigen door een speciale controle class te maken

Dat is niets anders dan het probleem verplaatsten

3.9.5 DTO: EEN UITWERKING

- Stel dat de lagen er geen twee, maar 5 zijn. Dan moeten er dus 4 keer gegevens worden overgezet.
- Stel dat er 3 manieren zijn om uit te komen bij `saveNewBook(book)`
- Op allerlei plekken in de code moeten er dan controles worden toegevoegd om data-integriteit te waarborgen.

3.9.6 DTO

DTO is procedureel en we gingen OOP gebruiken.

Heel veel code is ***procedureel programmeren met classes***.

Ik heb een bak met data en die moet door een procedure heen.

=> Oplossing is immutable classes

3.9.7.1 OOP-VOORBEELD: EEN BOEK

Voorbeeld van: https://www.yegor256.com/2016/07/06/data-transfer-object.html

```
Book book = api.bookById(123);
book.save(database);

// Het boek wordt gevraagd zich weg te schrijven op het gegeven object. Dat object is een
interface.

Book bookById(int id) {
  return new JsonBook(
    /* RESTful API access point */
  ); // directe inkapseling
}

class Book {
constructor(/* whatever */ )

void save(Database db) {
  JsonObject json = /* Load it from RESTful API */
  db.createBook(
    json.getString("isbn"),
    json.getString("title"),
    json.getString("author")
  );
}

// of als je wat convenience wrappers wil hebben kan de laatste method ook zo

void save(Database db) {
  db.create()
    .withISBN(json.getString("isbn"))
    .withTitle(json.getString("title"))
    .withAuthor(json.getString("author"))
    .deploy();
  }
}
```

3.9.7.2 OOP-VOORBEELD: EEN BOEK

Dit is dan de database-class:

```
class Database (){
    Connection connection;
    String ISBN;


    void Database(Connection connexion){
        this.connection = connexion
    }

    void create() {
        return this;
    }

    void withISBN(String ISBN){
        this.ISBN = ISBN;
        return this;
    }

    void deploy() {
        // write to database
    }
}
```

4.1 CONCLUSIE

WEET IK NU WAT OOP/MESSAGE PASSING IS?

Er zijn heel veel meningen over wat Alan Kay bedoelde, bijv. deze van Yegor

- https://www.yegor256.com/2017/12/12/alan-kay-was-wrong.html

```
"Kay's OO is a _metaparadigm_, which is why it was superseded by
implementation-oriented OO. One is on use every day, one is a
relatively vague idea about building systems rather than their
components. This is made obvious in the notion that the internet is at
least analogous to Kay's OO. The amount of infrastructure and
complexity of a single object in that infrastructure is no longer
comprehensible to a single person, and that's a real issue if it is to
be the central paradigm of programming."
```

- https://www.reddit.com/r/programming/comments/bqu8td/a_simple_explanation_of_what_alan_k
- https://hillelwayne.com/post/alan-kay/
- https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f

Alan Kay vlak na zijn presentatie in '97 op OOPSLA

- http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

"... the realization that *assignments are a metalevel change from functions, and therefore should not be dealt with at the same level* -- this was one of the motivations to encapsulate these kinds of state changes, and not let them be done willy nilly."

=> Hier lezen we weer dat andere citaat van Alan Kay dat ik eerder aanhaalde: *executed math*.

=> Dit is waarom NULL niet zo moeten bestaan. (Wat overigens wel bestaat in Smalltalk)

Alan Kay in 2003:

"My math background made me realize that each object could have *several algebras associated with it*, and there could be families of these, and that these would be very very useful....."

4.2 CONCLUSIE

Mijn zoektocht naar wat de uitspraak van Alan Kay dat OOP zeker geen C++ was voor mij door alle voorbeelden op het internet moeilijk om te beantwoorden. Tot ik in 2018 het blog van Yegor Bugayenko tegenkwam dat daadwerk de voorbeelden bevat die ik nodig had om OOP te begrijpen.

De algemene uitspraken van de vorige pagina hielpen mij niet. Wat betekent dat in de praktijk? Zoals gezegd staat het internet vol met verkeerde voorbeelden.

Ik heb heel veel geleerd van de voorbeelden van Yegor Bugayenko, zowel op het internet als in zijn boeken, die ik kan aanraden. Die hielpen mij OOP beter te begrijpen. Er was echter een ander gesprek dat OOP/message passing voor mij het meest duidelijk maakt.

OOP GAAT OVER HET BESCHRIJVEN VAN *GEDRAG*, NIET HET UITTYPEN ERVAN.

Objecten definiëren gedrag en dat sturen we rond. Dat is het verschil tussen declaratief en imperatief.

4.3.1 LAATSTE VOORBEELD: IMPERATIEF

Dit voorbeeld is imperatieve code: we beschrijven wat er moet gebeuren.

```
import java.util.Scanner
public class Main {
  public static void manin(String... args) {
    int n = (int) (Math.random() * 100.0d);
    int t= 0;
    while (true) {
      if (++t > 5) {
        System.out.println(
          "You lost, sorry. It was: " + n
        );
        break;
      }
      System.out.print(
        "Guess a number in 0 .. 99 reange: "
      );
      int x = new Scanner(System.in).nextInt();
      if (x < n) {
        System.out.println("Too small.");
      } else if (x > n) {
        System.out.println("Too big.");
      } else {
        System.out.println("Bingo!");
        break;
      }
    }
  }
  System.out.println(
    "Tanks for playing with me!"
  )
}
```

4.3.2 LAATSTE VOORBEELD: DECLARATIEF - OOP

Dit is hetzelfde algoritme, maar dan compositioneel. Het zijn objecten die berichten sturen. Ik vertel niet aan de computer welke bits / bytes moeten worden verplaatst, maar ik vertel de lezer van het programma wat ik wil bereiken.

```
public class Main {
  public static void main(String... args){
    Secret secret = new Secret();
    new WinOrLooseMessage(
      new Attempts(
        new VerboseDiff(
          new Diff(
            secret,
            new Guess()
          )
        ), 5
        ),
        secret
      ).speak();
  }
}
```

De rest van de code is te vinden in het boek Vol.2 p. 118 - 127

4.3 CODEERAANBEVELINGEN

- Schrijf fragile code
- Gebruik geen constants / globale variabelen
- Geen *null*
- Gebruik interfaces
- Immutable classes
- Geen code in de constructor
- Maak classes klein, zodat ze composable zijn

Allerlaatste opmerking:

Ik vind dat degene die een term bedenkt de enige autoriteit is over wat de term betekent. Ook als deze term vaag blijkt te zijn door nieuwe ontwikkelingen, die nog niet goed waren uitgekristaliseerd op het moment van bedenken van de term.

VRAGEN?

Hoofdstukken

1. Introductie - Mijn persoonlijke achtergrond en motivatie
2. Context - Als we code schrijven .....
3. Voorbeelden - Praktische voorbeelden van OOP-concepten
4. Conclusie - Wat we hebben geleerd?
5. Vragen - Open discussion over OOP