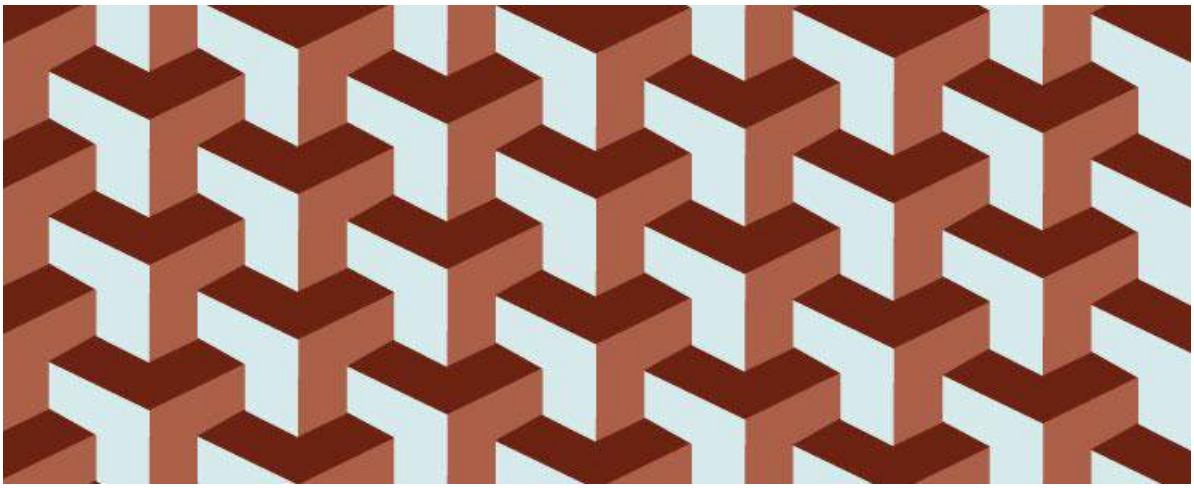
VDAB

samen sterk voor werk

Java

Design Patterns



Inhoudsopgave

1 INLEIDING

1.1 Doelstelling

Je leert werken met design patterns:
voorgedefinieerde oplossingen voor problemen die vaak voorkomen bij het schrijven van code.

1.2 Vereiste voorkennis

- Java programming fundamentals
- Lambda's

1.3 Nodige software

- Een Java IDE (NetBeans, Eclipse, …)

1.4 Algemeen

Bij het schrijven van code komen regelmatig soortgelijke problemen voor. Erich Gamma, Richard Helm, Ralph Johnson en John Vlissides hebben deze problemen beschreven in het boek Design patterns. Ze hebben elk probleem een naam gegeven en ook een mooie oplossing toegevoegd.

Het geheel van een probleem met zijn oplossing heet een design pattern (ontwerppatroon).

Ze hebben de patronen ingedeeld in drie categorieën

- Creational patterns
  lossen problemen op bij het *maken* van objecten.
- Structural patterns
  lossen problemen op bij het *samenstellen* van objecten.
- Behavioral patterns
  lossen problemen op bij het *samenwerken* van objecten.

De code in het boek is geschreven in de programmeertaal C++.
Andere mensen hebben de design patterns ook geschreven in andere programmeertalen.

In het begin gebruikte de Java code van design patterns enkel classes en interfaces.
Nu Java ook enums en lambda's bevat, worden ook deze taalelementen gebruikt
in design patterns, om deze nog korter uit te werken.

Je leert in deze cursus de meest gebruikte design patterns kennen.

1.5 Werkwijze

1. Je hebt een probleem in je code.
2. Je overloopt de design patterns en je zoekt het pattern dat overeenstemt met je probleem.
3. Je leest de voorbeeldoplossingscode van het pattern.
4. Je past deze code toe op je eigen probleem.

1.6 Principes

Bij design patterns worden enkele principes van goed programmeren gebruikt.
Je ziet hier onder deze principes.

1.6.1 Program to an interface, not an implementation

Het type van een variabele, van een parameter of het returntype van een method is beter een interface dan een class. Een variabele met als type een interface is *flexibeler* dan een variabele met als type een class. Een variabele met als type een interface kan verwijzen naar elk object van classes die de interface implementeren. Een variabele met als type een class kan enkel verwijzen naar een object van diezelfde class, of naar een derived class van die class.

1.6.2 Favor composition over inheritance

Composition (class A bevat een private variabele met als type een class B) is *flexibeler* dan inheritance (class B erft van class A). Inheritance kan niet wijzigen tijdens de uitvoering van je programma: je kan class B die erft van class A niet tijdens de uitvoering laten erven van een class C.

Als class A een private variabele bevat van het type B, kan deze variabele, als deze als type een interface heeft, eerst verwijzen naar een class C die de interface implementeert, en later verwijzen naar een class D die ook deze interface implementeert.

1.6.3 Loosely coupled design

Als een class A samenwerkt met een class B, probeer je deze classes zo te schrijven dat als je class A aanpast, je class B niet (of minimaal) moet aanpassen. Omgekeerd schrijf je class B zo dat als je class B aanpast, je class A niet (of minimaal) moet aanpassen.

1.6.4 Single responsibility

Een class heeft maar één reden tot verandering. Als een class twee verantwoordelijkheden heeft, zijn er twee mogelijke redenen om deze class te wijzigen. Als een class slechts één verantwoordelijkheid bevat, is er slechts één reden om deze class te wijzigen.
Slecht voorbeeld: een class stelt de werking van een scorebord van een spelletje voor én bepaalt hoe het scorebord wordt weggeschreven naar de harde schijf. Als de werking van het scorebord wijzigt moet je deze class wijzigen. Als je het scorebord niet meer naar een XML bestand wegschrijft, maar naar een database, moet je deze class ook wijzigen.
Verbeterd voorbeeld: één class stelt de werking van het scorebord voor. Je moet deze class slechts wijzigen als de werking van het scorebord wijzigt. Een andere class bepaalt hoe je het scorebord wegschrijft naar de harde schijf. Je moet deze class slechts wijzigen als je de manier van wegschrijven wijzigt. Je moet hierbij de eerste class (die de werking van het scorebord voorstelt) niet wijzigen. Elke class is klein en overzichtelijk.

1.6.5 A class should be open for extension and closed for modification

Als hetgeen van een class verwacht wordt uitbreidt, moet je deze class niet wijzigen, maar maak je een nieuwe afgeleide class van de oorspronkelijke class. Het is in deze afgeleide class dat je de uitbreiding schrijft. Gezien je de oorspronkelijke class niet moet wijzigen, kan je ook geen fouten introduceren in de oorspronkelijke class. Elke class blijft klein en overzichtelijk.

2 SINGLETON

2.1 Categorie

Creational design pattern

2.2 Probleem

Je gebruikt het singleton design pattern als van een class slechts één object bestaat in de werkelijkheid. Het singleton design patterns zorgt er voor

- dat er maar één object kan bestaan in je code (en niet meerdere per ongeluk).
- dat dit ene object overal in je code gemakkelijk aanspreekbaar is.

2.3 Voorbeeld

In de software van een auto heb je maar één object van het type motor.

2.4 Oplossing

Je beschrijft het object niet als een class, maar als een enum met één waarde.
Deze ene waarde stelt het object voor dat maar één keer voorkomt in de werkelijkheid.
Daarnaast kan de enum variabelen en methods bevatten die bij dit ene object horen.

```
package be.vdab;
public enum Motor {
  INSTANCE;                                                    ❶
  private boolean gestart;
  public void start() {
    gestart = true;
    System.out.println("gestart");
  }
  public void stop() {
    if (gestart) {
      gestart = false;
      System.out.println("gestopt");
    }
  }
}
```

(1) Deze enige waarde in de enum stelt het enige object voor.

Je gebruikt de motor als volgt:

```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    Motor.INSTANCE.start();
    stopDeAuto();
  }
  public static void stopDeAuto() {
    Motor.INSTANCE.stop();                                     ❶
  }
}
```

(1) Het object is gemakkelijk aanspreekbaar. Je moet het niet als parameter van de method `stopDeAuto` binnenkrijgen. Je moet het ook niet bijhouden in een private variabele.

Singleton: zie takenbundel

Singleteon 2: zie takenbundel

3 SIMPLE FACTORY

3.1 Categorie

Creational design pattern: een fabriek (factory) *maakt* objecten

3.2 Probleem

Je moet op meerdere plaatsen in je code kiezen welk object je maakt uit een reeks van objecten. Als je deze code herhaalt, moet je deze code ook op meerdere plaatsen aanpassen bij fouten en bij uitbreidingen.

3.3 Voorbeeld

Je moet op meerdere plaatsen in je code kiezen of je een Tekst object, een Rekenblad object of een Presentatie object maakt, gebaseerd op een bestandsextensie (.docx, .xlsx, .pptx).

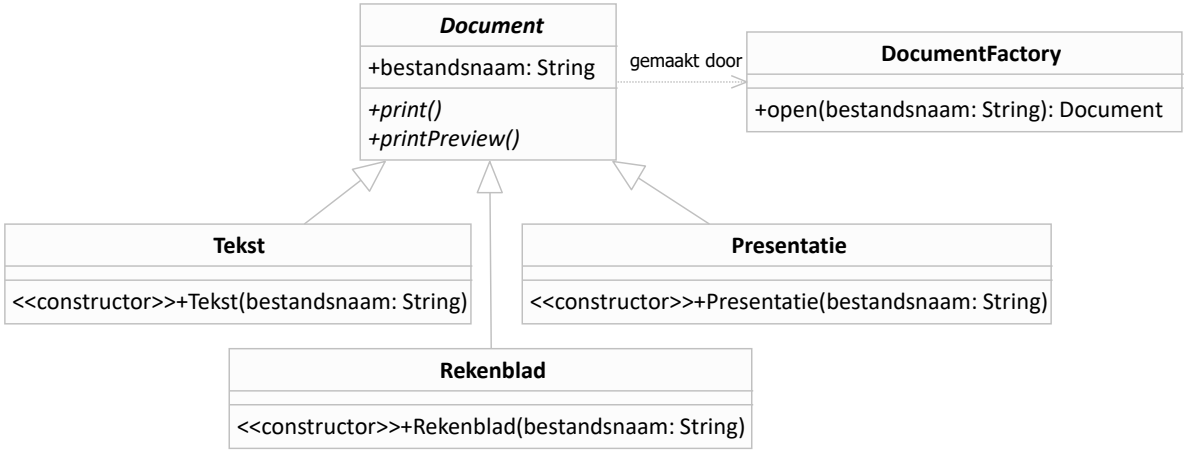
Het codefragment om de keuze te maken:

```
switch (extensie) {
  case "docx":
    return new Tekst(bestandsnaam);
  case "xlsx":
    return new Rekenblad(bestandsnaam);
  case "pptx":
    return new Presentatie(bestandsnaam);
  default:
    throw new IllegalArgumentException();
}
```

Als je dit code fragment meerdere keren in je applicatie opneemt, moet je dit code fragment meerdere keren aanpassen als er nog een bestandstype (bvb. database) bijkomt.

3.4 Oplossing

Je schrijft de code die de beslissing neemt niet meerdere keren in je programma, maar één keer in een class (de factory) die je van overal in je programma kan oproepen.



De method open geeft een `Tekstverwerker`, `Rekenblad` of `Presentatie` object terug, naargelang de extensie van de bestandsnaam. Bemerk dat het returntype van de method `Document` is (niet `Tekst`, `Presentatie` of `Rekenblad`) om deze flexibiliteit te bekomen.

Je ziet in het voorbeeld dat de class een duidelijke naam heeft: DocumentFactory, maw een factory voor documenten. Dit is een 'best practice': zo ziet een programmeur onmiddellijk welk design pattern is gebruikt en waarvoor deze class dient.

```
package be.vdab;
public class DocumentFactory {
  public Document open(String bestandsnaam) {
    String extensie = bestandsnaam.substring(bestandsnaam.length() - 4);
    switch (extensie) {
      case "docx":
        return new Tekst(bestandsnaam);
      case "xlsx":
        return new Rekenblad(bestandsnaam);
      case "pptx":
        return new Presentatie(bestandsnaam);
      default:
        throw new IllegalArgumentException();
    }
  }
}
```

```
package be.vdab;
public abstract class Document {
  private final String bestandsnaam;
  public Document(String bestandsnaam) {
    this.bestandsnaam = bestandsnaam;
  }
  public abstract void print();
  public abstract void printPreview();
}
```

```
package be.vdab;
public class Tekst extends Document {
  public Tekst(String bestandsnaam) {
    super(bestandsnaam);
  }
  @Override
  public void print() {
    System.out.println("een afdruk van een tekst");
  }
  @Override
  public void printPreview() {
    System.out.println("een afdrukvoorbeeld van een tekst");
  }
}
```

```
package be.vdab;
public class Rekenblad extends Document {
  public Rekenblad(String bestandsnaam) {
    super(bestandsnaam);
  }
  @Override
  public void print() {
    System.out.println("een afdruk van een rekenblad");
  }
  @Override
  public void printPreview() {
    System.out.println("een afdrukvoorbeeld van een rekenbload");
  }
}
```

```
package be.vdab;
public class Presentatie extends Document {
  public Presentatie(String bestandsnaam) {
    super(bestandsnaam);
```

```
  }
  @Override
  public void print() {
    System.out.println("een afdruk van een presentatie");
  }
  @Override
  public void printPreview() {
    System.out.println("een afdrukvoorbeeld van een presentatie");
  }
}
```

Je kan deze classes als volgt gebruiken:

```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    DocumentFactory factory=new DocumentFactory();
    Document document = factory.open("liedje.docx");
    document.printPreview();
    document.print();
  }
}
```

3.5 Factory als singleton

Je hebt maar één object van de factory nodig in je applicatie. Je kan daarom deze factory als een singleton schrijven. Dit is een voorbeeld van het combineren van design patterns. Dit combineren van design patterns wordt ook compound patterns genoemd.

```
package be.vdab;
public enum DocumentFactory {
  INSTANCE;
  public Document open(String bestandsnaam) {
    String extensie = bestandsnaam.substring(bestandsnaam.length() - 4);
    switch (extensie) {
      case "docx":
        return new Tekst(bestandsnaam);
      case "xlsx":
        return new Rekenblad(bestandsnaam);
      case "pptx":
        return new Presentatie(bestandsnaam);
      default:
        throw new IllegalArgumentException();
    }
  }
}
```

Je gebruikt de classes nu als volgt:

```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    Document document = DocumentFactory.INSTANCE.open("liedje.docx");
    document.printPreview();
    document.print();
  }
}
```

Simple factory: zie takenbundel

4 BUILDER

4.1 Categorie

Creational design pattern

4.2 Probleem

Je moet veel parameters meegeven om een object aan te maken. Voorbeeld:

```
Inwoner inwoner = new Inwoner("Olivier", "Gerard", 1, 3, true, false);
```

Deze constructor heeft veel parameters, wat de leesbaarheid niet bevordert. Er wordt aangeraden dat een constructor maximum vier parameters heeft. De betekenis van de parameterwaarden is ook niet duidelijk: is de voornaam van de inwoner Olivier of Gerard ? Wat is de betekenis van de waarde 1, de waarde 3, de waarde true en de waarde false ? Je weet dit slechts als je de source of de javadoc van deze constructor inziet.

4.3 Oplossing

Maak een extra class (`InwonerBuilder`) die helpt deze class (`Inwoner`) stap per stap op te bouwen:

```
InwonerBuilder builder = new InwonerBuilder();
Inwoner inwoner = builder.metVoornaam("Olivier")
                         .metFamilienaam("Gerard")
                         .metAantalKinderen(1)
                         .metAantalKerenVerhuisd(3)
                         .metGehuwd(true)
                         .metGescheiden(false)
                         .maakInwoner();
```

Bemerk dat je op iedere `met…` method een andere method kan oproepen (method chaining). Dit heet een fluent (vloeiende) interface.

De oorspronkelijke class:

```
package be.vdab;
public class Inwoner {
  private final String voornaam;
  private final String familienaam;
  private final int aantalKinderen;
  private final int aantalKeerVerhuisd;
  private final boolean gehuwd;
  private final boolean gescheiden;
  public Inwoner(String voornaam, String familienaam, int aantalKinderen,
    int aantalKeerVerhuisd, boolean gehuwd, boolean gescheiden) {
    this.voornaam = voornaam;
    this.familienaam = familienaam;
    this.aantalKinderen = aantalKinderen;
    this.aantalKeerVerhuisd = aantalKeerVerhuisd;
    this.gehuwd = gehuwd;
    this.gescheiden = gescheiden;
  }
  @Override
  public String toString() {
    return voornaam + ' ' + familienaam;
  }
}
```

De builder class

```
package be.vdab;
public class InwonerBuilder {
  private String voornaam;
  private String familienaam;
  private int aantalKinderen;
```

```
  private int aantalKerenVerhuisd;
  private boolean gehuwd;
  private boolean gescheiden;
  public InwonerBuilder metVoornaam(String voornaam) {                    ❶
    this.voornaam = voornaam;
    return this;
  }
  public InwonerBuilder metFamilienaam(String familienaam) {
    this.familienaam = familienaam;
    return this;
  }
  public InwonerBuilder metAantalKinderen(int aantalKinderen) {
    this.aantalKinderen = aantalKinderen;
    return this;
  }
  public InwonerBuilder metAantalKerenVerhuisd(int aantalKerenVerhuisd) {
    this.aantalKerenVerhuisd = aantalKerenVerhuisd;
    return this;
  }
  public InwonerBuilder metGehuwd(boolean gehuwd) {
    this.gehuwd = gehuwd;
    return this;
  }
  public InwonerBuilder metGescheiden(boolean gescheiden) {
    this.gescheiden = gescheiden;
    return this;
  }
  public Inwoner maakInwoner() {
    return new Inwoner(voornaam, familienaam, aantalKinderen,
      aantalKerenVerhuisd, gehuwd, gescheiden);
  }
}
```

(1) Iedere method van een builder geeft dezelfde builder terug. Dit laat de fluent interface toe: `.metVoornaam("Olivier").metFamilienaam("Gerard")`

De Main class

```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    InwonerBuilder builder = new InwonerBuilder();
    Inwoner inwoner = builder.metVoornaam("Olivier")
                             .metFamilienaam("Gerard")
                             .metAantalKinderen(1)
                             .metAantalKerenVerhuisd(3)
                             .metGehuwd(true)
                             .metGescheiden(false)
                             .maakInwoner();
    System.out.println(inwoner);
  }
}
```

Je kan in de main nog altijd de onleesbare `Inwoner` constructor oproepen in plaats van de `InwonerBuilder` te gebruiken. Je verhinder dit oproepen met volgende stappen:

1. Je kopieert in de source `InwonerBuilder` de regels `class…` tot het einde van de source op het klembord.
2. Je plakt de inhoud van het klembord voor de sluit accolade van de class `Inwoner`.
3. Je wijzigt `public class InwonerBuilder` naar `public static class InwonerBuilder`
4. Je maakt de constructor van `Inwoner` private, zodat enkel de nested class `InwonerBuilder` deze kan oproepen.

5. Je wijzigt in de class Main twee keer `InwonerBuilder` naar `Inwoner.InwonerBuilder`.
6. Je verwijdert de source `InwonerBuilder.java`

4.4 Het Builder pattern in de Java API

De Java API bevat een class StringBuilder, waarmee je stap per stap een String opbouwt:

```
String resultaat =
  new StringBuilder("Ali Baba en de ").append(40).append(" rovers").toString();
```

Builder: zie takenbundel

5 FACADE

5.1 Categorie

Structural Design Pattern

5.2 Probleem

De gebruiker van een verzameling classes moet een moeilijk algoritme schrijven om die classes aan te spreken. Dit patroon is zeker interessant als één ontwikkelaar de verzameling classes schreef en een andere ontwikkelaar deze moet gebruiken en er een moeilijk algoritme moet op schrijven.

5.3 Voorbeeld

Om een lening toe te kennen, moeten drie voorwaarden voldaan zijn:

- De persoon moet voldoende beroepsinkomsten hebben (€ 2500).
- Het saldo op de rekening van de persoon moet positief zijn.
- De persoon mag geen andere leningen lopende hebben.

Zonder façade is het de *gebruiker* van de classes `Loon`, `Rekening` en `Leningen` die bepaalt dat deze drie voorwaarden samen moeten voldaan worden. Het façade design pattern verplaatst dit algoritme naar de binnenkant van een extra façade class (`LeningVerstrekker`).
De gebruiker roept vanaf dan een eenvoudige method op van die façade class.

5.4 Zonder façade

```
package be.vdab;
import java.math.BigDecimal;
public class Beroepsinkomsten {
  private BigDecimal maandWedde;
  public Beroepsinkomsten(BigDecimal maandWedde) {
    this.maandWedde = maandWedde;
  }
  public BigDecimal getMaandWedde() {
    return maandWedde;
  }
}
```

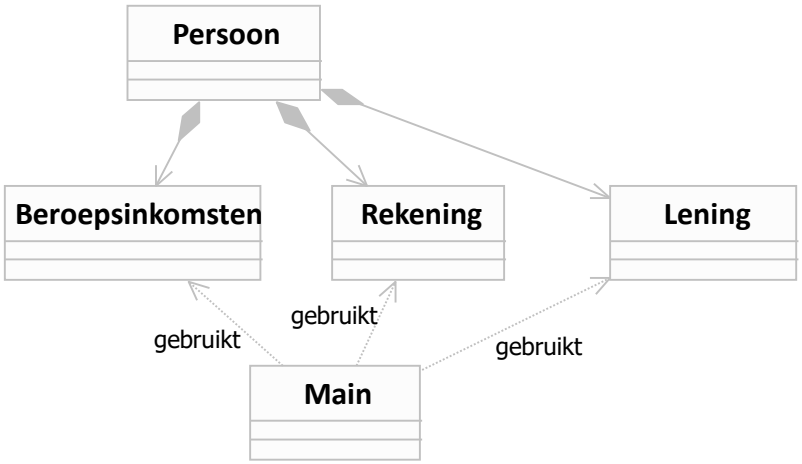
```
package be.vdab;
import java.math.BigDecimal;
public class Rekening {
  private BigDecimal saldo = BigDecimal.ZERO;
  public void storten(BigDecimal bedrag) {
    saldo = saldo.add(bedrag);
  }
  public BigDecimal getSaldo() {
    return saldo;
  }
}
```

```
package be.vdab;
import java.math.BigDecimal;
public class Lening {
  private BigDecimal bedrag;
  public Lening(BigDecimal bedrag) {
    this.bedrag = bedrag;
  }
  public BigDecimal getBedrag() {
    return bedrag;
  }
}
```

```
package be.vdab;
import java.util.ArrayList;
import java.util.List;
public class Persoon {
  private Beroepsinkomsten beroepsinkomsten;
  private Rekening rekening;
  private final List<Lening> leningen = new ArrayList<>();
  public Persoon(Beroepsinkomsten beroepsinkomsten, Rekening rekening) {
    this.beroepsinkomsten = beroepsinkomsten;
    this.rekening = rekening;
  }
  public void addLening(Lening lening) {
    leningen.add(lening);
  }
  public Beroepsinkomsten getBeroepsinkomsten() {
    return beroepsinkomsten;
  }
  public Rekening getRekening() {
    return rekening;
  }
  public List<Lening> getLeningen() {
    return leningen;
  }
}
```

```
package be.vdab;
import java.math.BigDecimal;
public class Main {
  public static void main(String[] args) {
    Persoon persoon = new Persoon(
      new Beroepsinkomsten(BigDecimal.valueOf(3_000)), new Rekening());
    if (persoon.getBeroepsinkomsten().getMaandWedde()                    ❶
        .compareTo(BigDecimal.valueOf(2_500)) >= 0
      && persoon.getRekening().getSaldo().compareTo(BigDecimal.ZERO) > 0
      && persoon.getLeningen().isEmpty()) {
      System.out.println("Lening goedgekeurd");
    } else {
      System.out.println("Lening afgekeurd");
    }
  }
}
```

(1) De Main, gebruiker van de classes, bevat het moeilijk algoritme dat bepaalt of een rekening toegekend wordt.



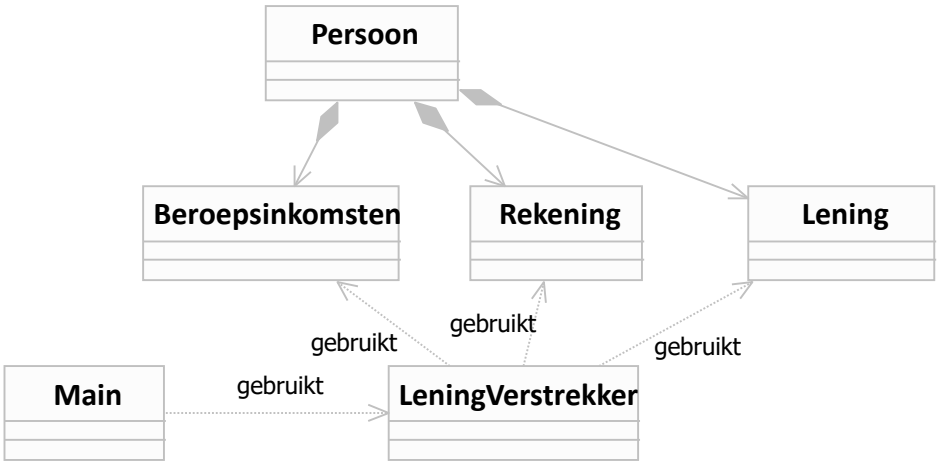
5.5 Oplossing met façade

```
package be.vdab;
import java.math.BigDecimal;
public class LeningVerstrekker {                                    ❶
  public boolean isLeningGoedgekeurd(Persoon persoon) {
  return persoon.getBeroepsinkomsten().getMaandWedde()
      .compareTo(BigDecimal.valueOf(2_500)) >= 0
    && persoon.getRekening().getSaldo().compareTo(BigDecimal.ZERO) > 0
    && persoon.getLeningen().isEmpty();
  }
}
```

(1) Deze class speelt de rol van façade en heeft het moeilijk algoritme in zich.

De class Main, gebruiker van de classes, wordt nu eenvoudig:

```
package be.vdab;
import java.math.BigDecimal;
public class Main {
  public static void main(String[] args) {
    Persoon persoon = new Persoon(
      new Beroepsinkomsten(BigDecimal.valueOf(3_000)), new Rekening());
    LeningVerstrekker verstrekker=new LeningVerstrekker();
    if (verstrekker.isLeningGoedgekeurd(persoon)) {
      System.out.println("Lening goedgekeurd");
    } else {
      System.out.println("Lening afgekeurd");
    }
  }
}
```



Facade: zie takenbundel

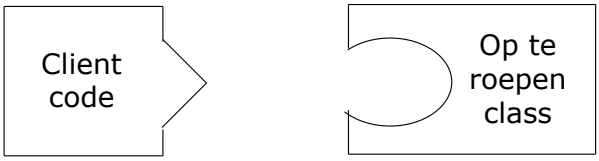
6 ADAPTER

6.1 Categorie

Structural design pattern

6.2 Probleem

Je wil een class aanspreken, maar die heeft niet de methods die jij verwacht aan te spreken.



6.3 Voorbeeld

Je gebruikte vroeger een spellingscontrole library. De aan te spreken class was als volgt:

OudeSpellingsControle
-taal: String -tekst: String -fouten: String[*]
+setTaal(taal: String) +setTekst(tekst: String) +controleerSpelling() +getAantalFouten(): int +getFout(index: int): String

```
package be.vdab;
public class OudeSpellingsControle {
  private String taal;
  private String tekst;
  private String[] fouten;
  public void setTaal(String taal) {
    this.taal = taal;
  }
  public void setTekst(String tekst) {
    this.tekst = tekst;
  }
  public void controleerSpelling() {
    fouten = new String[] {"onmidelijk", "paralelogram"};
  }
  public int getAantalFouten() {
    return fouten.length;
  }
  public String getFout(int index) {
    return fouten[index];
  }
}
```

Je gebruikte deze library als volgt:

```
OudeSpellingsControle controle = new OudeSpellingsControle();
controle.setTaal("nl");
controle.setTekst("Ik kom onmidelijk met een paralelogram");
controle.controleerSpelling();
int aantalFouten = controle.getAantalFouten();
for (int index = 0; index != aantalFouten; index++) {
  System.out.println(controle.getFout(index));
}
```

De firma die deze library onderhoudt gaat echter failliet.
Vanaf nu moet je een andere spellingscontrole library gebruiken.
De aan te spreken class heeft andere methods dan de oude library

NieuweSpellingsControle
-taal: String -tekst: String
<<constructor>>+NieuweSpellingsControle(taal: String, tekst: String) +controleer():String[*]()

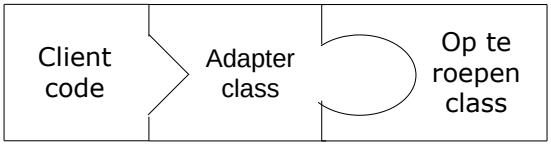
```
package be.vdab;
public class NieuweSpellingsControle {
  private final String taal;
  private final String tekst;
  public NieuweSpellingsControle(String taal, String tekst) {
    this.taal = taal;
    this.tekst = tekst;
  }
  public String[] controleer() {
    return new String[] {"onmidelijk","paralelogram"}; // dummy voorbeeld
  }
}
```

Zonder het adapter design pattern moet je overal in je code, waar je de oude library opriep, meerdere regels aanpassen om de nieuwe library op te roepen.

```
NieuweSpellingsControle controle =
  new NieuweSpellingsControle("nl", "Ik kom onmidelijk met een paralelogram");
Arrays.stream(controle.controleer()).forEach(fout -> System.out.println(fout));
```

6.4 Oplossing

Je maakt een adapter class. Deze heeft dezelfde method declaraties als de oude library.
De class vertaalt de oproepen van deze methods naar oproepen naar de nieuwe library.



OudeSpellingsControle
-taal: String -tekst: String -fouten: String[*]
+setTaal(taal: String) +setTekst(tekst: String) +controleerSpelling() +getAantalFouten(): int +getFout(index: int): String

SpellingscontroleAdapter
-taal: String -tekst: String -fouten: String[*]
+setTaal(taal: String) +setTekst(tekst: String) +controleerSpelling() +getAantalFouten(): int +getFout(index: int): String

gebruikt

NieuweSpellingsControle
-taal: String -tekst: String
<<constructor>>+NieuweSpellingsControle(taal: String, tekst: String) +controleer():String[*]()

```
package be.vdab;
public class SpellingsControleAdapter {
  private String taal;
  private String tekst;
  private String[] fouten;
  public void setTaal(String taal) {
    this.taal = taal;
  }
  public void setTekst(String tekst) {
    this.tekst = tekst;
  }
  public void controleerSpelling() {
    NieuweSpellingsControle controle = new NieuweSpellingsControle(taal, tekst);
    fouten = controle.controleer();
  }
  public int getAantalFouten() {
    return fouten.length;
  }
  public String getFout(int index) {
    return fouten[index];
  }
}
```

Nu moet je op meerdere plaatsen maar één regel aanpassen:

```
SpellingsControleAdapter controle = new SpellingsControleAdapter();
controle.setTaal("nl");
controle.setTekst("Ik kom onmidelijk met een paralelogram");
controle.controleerSpelling();
int aantalFouten = controle.getAantalFouten();
for (int index = 0; index != aantalFouten; index++) {
  System.out.println(controle.getFout(index));
}
```

Opmerking: adapters bestaan ook in de werkelijkheid.
Voorbeeld: de pinnen van een elektrisch apparaat stemmen niet overeen met de gaten in een stopcontact. Je steekt tussen het elektrisch apparaat en het stopcontact een adapter om dit probleem op te lossen:



Adapter: zie takenbundel

7 COMPOSITE

7.1 Categorie

Structural design pattern

7.2 Probleem

Sommige data heeft een boomstructuur. Hoe kan je op dezelfde manier de elementen benaderen die nog vertakkingen hebben als de elementen die geen vertakkingen hebben ?

Een harde schijf is een voorbeeld van zo'n boomstructuur. De harde schijf bestaat uit een root directory. Deze bestaat op zijn beurt uit directories en bestanden. Ieder van die directories bestaat weer uit directories en bestanden. Het probleem is: hoe kan je deze twee soorten (directories en bestanden) benaderen om er eenzelfde handeling (bvb. verwijderen) op uit te voeren?
Als je een directory verwijdert, moet je alle bestanden in die directory verwijderen, maar ook alle subdirectory's. Om zo'n subdirectory te verwijderen, moet je ook daar alle subdirectory's en bestanden verwijderen, …

7.3 Voorbeeld

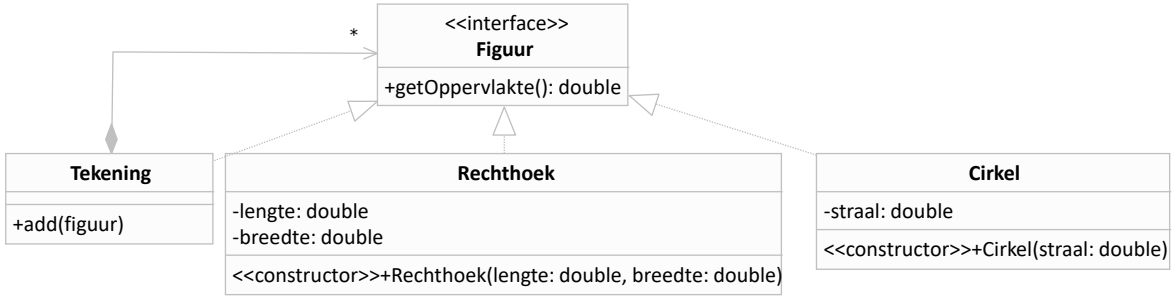
Een tekening in een cad-cam programma bestaat uit figuren (rechthoeken, cirkels, …).
Maar één van de figuren kan op zijn beurt weer een tekening zijn (tekening in tekening).
Deze tweede tekening bestaat terug uit figuren en eventuele tekeningen.

Om de totale oppervlakte te weten van alle figuren in de tekening, moet je de som maken van de figuren in die tekening plus de som van alle figuren in tekeningen van die tekening …

Het composite design pattern laat dit op een elegante manier toe.

7.4 Oplossing

Je benoemt het onderdeel dat vertakkingen heeft (tekening), en je maakt er een class van.
Je benoemt de onderdelen die geen vertakkingen hebben (rechthoek, cirkel) en je maakt er classes van. Al deze classes implementeren een gemeenschappelijke interface class (figuur).
Je voegt aan deze interface method declaraties toe voor alle methods die je vindt in alle subclasses.



Bemerk dat een tekening (via composition) terug uit figuren bestaat.

```
package be.vdab;
public interface Figuur {
  double getOppervlakte();
}
```

```
package be.vdab;
public class Rechthoek implements Figuur {
  private final double lengte;
  private final double breedte;
  public Rechthoek(double lengte, double breedte) {
    this.lengte = lengte;
    this.breedte = breedte;
  }
```

```
  @Override
  public double getOppervlakte() {
    return lengte * breedte;
  }
}
```

```
package be.vdab;
public class Cirkel implements Figuur {
  private final double straal;
  public Cirkel(double straal) {
    this.straal = straal;
  }
  @Override
  public double getOppervlakte() {
    return straal * straal * Math.PI;
  }
}
```

```
package be.vdab;
import java.util.ArrayList;
import java.util.List;
public class Tekening implements Figuur {
  private final List<Figuur> figuren = new ArrayList<>();
  public void add(Figuur figuur) {
    figuren.add(figuur);
  }
  @Override
  public double getOppervlakte() {
    double oppervlakte = 0;
    for (Figuur figuur : figuren) {                    ❶
      oppervlakte += figuur.getOppervlakte();
    }
    return oppervlakte;
  }
}
```

(1) Je overloopt alle figuren in de huidige figuur. Je maakt daarbij geen onderscheid tussen tekeningen (figuren die op zich weer figuren bevatten), rechthoeken of cirkels.

Je gebruikt deze classes als volgt:

```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    Tekening tekening = new Tekening();
    tekening.add(new Rechthoek(2, 1));
    tekening.add(new Cirkel(3));
    Tekening subTekening = new Tekening();
    subTekening.add(new Rechthoek(3, 2));
    subTekening.add(new Cirkel(4));
    tekening.add(subTekening);
    System.out.println(tekening.getOppervlakte());
  }
}
```

Composite: zie takenbundel

8 DECORATOR

8.1 Categorie

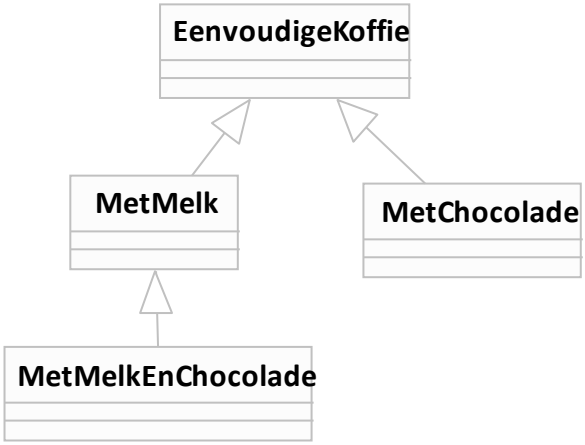
Structural design pattern

8.2 Probleem

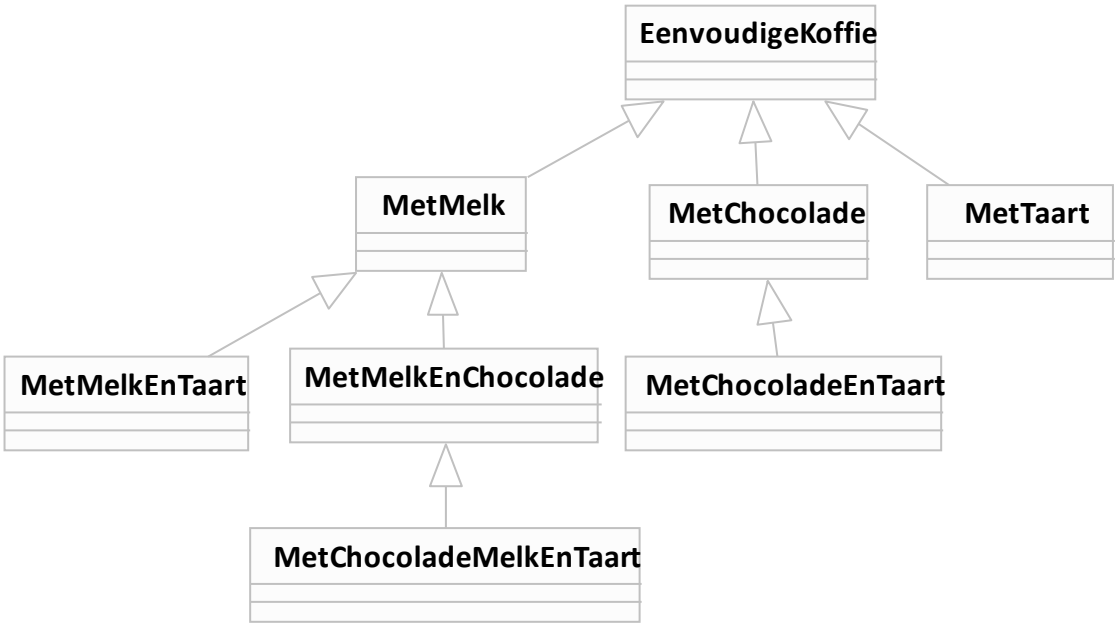
Hoe kan je, op een dynamische manier, verantwoordelijkheden toevoegen aan een class ?

Voorbeeld: Een eenvoudige koffie heeft een prijs en een bereidingswijze. Je wil aan deze koffie eventueel melk toevoegen. Dit wijzigt de prijs en de bereidingswijze. Je kan aan een koffie ook chocolade toevoegen. Ook dit wijzigt de prijs en de bereidingswijze. Je kan aan een koffie ook én melk én chocolade toevoegen. Ook dit wijzigt de prijs en de bereidingswijze.

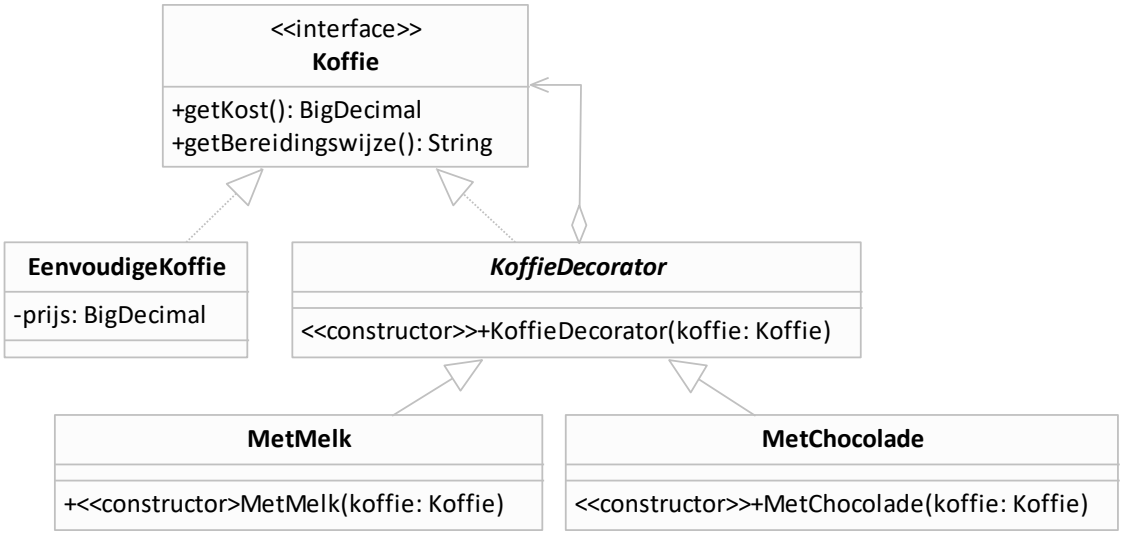
Als je dit probeert op te lossen met inheritance, krijg je veel classes:



Zeker als er daarna nog een taart kan toegevoegd worden, zou je nog meer classes krijgen:



8.3 Oplossing



Je vertrekt vanaf een EenvoudigeKoffie: `new EenvoudigeKoffie()`. Als de klant geen melk en geen chocolade wenst, heb je met dit object genoeg om de kost en de bereidingswijze te bepalen.

Als de klant melk wenst, "decoreer" je de EenvoudigeKoffie met MetMelk:
`new MetMelk(new EenvoudigeKoffie());`.
De kost wordt de kost van de koffie plus de kost van de melk.

Als de klant chocolade wenst, "decoreer" je de `EenvoudigeKoffie` met `MetChocolade`:
`new MetChocolade(new EenvoudigeKoffie());`.
De kost wordt de kost van de koffie plus de kost van de chocolade.

Als de klant melk én chocolade wenst, "decoreer" je de `EenvoudigeKoffie` met `MetMelk` én met `MetChocolade` : `new MetChocolade(new MetMelk(new EenvoudigeKoffie()));`.
De kost wordt de kost van de koffie plus de kost van de melk plus de kost van de chocolade.

Hetgeen `MetMelk` en `MetChocolade` gemeen hebben, plaats je in een gemeenschappelijke base class KoffieDecorator

Alle "decorators" (`MetMelk` en `MetChocolade`) en `EenvoudigeKoffie` implementeren de interface `Koffie`: je kan van alle "decorators" én van `EenvoudigeKoffie` de kost berekenen en de bereidingswijze bepalen.

Er is ook een aggregatie tussen `KoffieDecorator` en `Koffie`:
elke "decorator" onthoudt in zich de koffie die hij "decoreert".

- Dit is een `EenvoudigeKoffie` bij `new MetMelk(new EenvoudigeKoffie());`
- Dit is een `MetMelk` bij `new MetChocolade(new MetMelk(new EenvoudigeKoffie()));`.

```
package be.vdab;
import java.math.BigDecimal;
public interface Koffie {
  BigDecimal getKost();
  String getBereidingswijze();
}
```

```
package be.vdab;
import java.math.BigDecimal;
public class EenvoudigeKoffie implements Koffie {
  @Override
  public BigDecimal getKost() {
    return BigDecimal.valueOf(3);
  }
```

```
  @Override
  public String getBereidingswijze() {
    return "maal de koffiebonen, laat kokend water over het poeder lopen";
  }
}
```

```
package be.vdab;
public abstract class KoffieDecorator implements Koffie {
  protected final Koffie gedecoreerdeKoffie;
  public KoffieDecorator(Koffie koffie) {
    this.gedecoreerdeKoffie = koffie;
  }
}
```

```
package be.vdab;
import java.math.BigDecimal;
public class MetMelk extends KoffieDecorator {
  public MetMelk(Koffie gedecoreerdeKoffie) {                    ❶
    super(gedecoreerdeKoffie);                                   ❷
  }
  @Override
  public BigDecimal getKost() {
    return super.gedecoreerdeKoffie.getKost().add(BigDecimal.ONE);   ❸
  }
  @Override
  public String getBereidingswijze() {
    return super.gedecoreerdeKoffie.getBereidingswijze()         ❹
      + ", warm de melk, voeg de melk toe";
  }
}
```

```
package be.vdab;
import java.math.BigDecimal;
public class MetChocolade extends KoffieDecorator {
  public MetChocolade(Koffie gedecoreerdeKoffie) {               ❺
    super(gedecoreerdeKoffie);
  }
  @Override
  public BigDecimal getKost() {
    return super.gedecoreerdeKoffie.getKost().add(BigDecimal.valueOf(2));   ❻
  }
  @Override
  public String getBereidingswijze() {
    return super.gedecoreerdeKoffie.getBereidingswijze()         ❼
      + ", schilfer de chocolade, voeg de schilfers toe";
  }
}
```

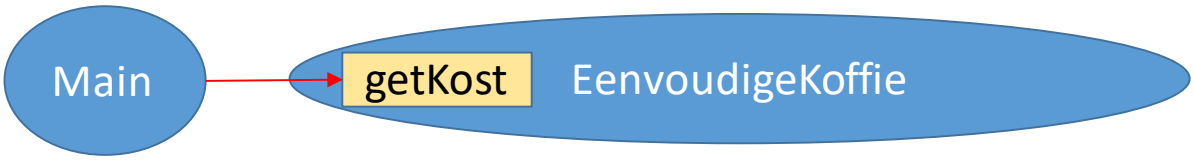
```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    Koffie eenvoudig = new EenvoudigeKoffie();
    System.out.println(eenvoudig.getKost());
    System.out.println(eenvoudig.getBereidingswijze());
    System.out.println();
    Koffie metMelk = new MetMelk(new EenvoudigeKoffie());
    System.out.println(metMelk.getKost());
    System.out.println(metMelk.getBereidingswijze());
    System.out.println();
    Koffie metChocolade = new MetChocolade(new EenvoudigeKoffie());
    System.out.println(metChocolade.getKost());
    System.out.println(metChocolade.getBereidingswijze());
```

```
    System.out.println();
    Koffie metMelkEnChocolade
      = new MetChocolade(new MetMelk(new EenvoudigeKoffie()));
    System.out.println(metMelkEnChocolade.getKost());
    System.out.println(metMelkEnChocolade.getBereidingswijze());
  }
}
```

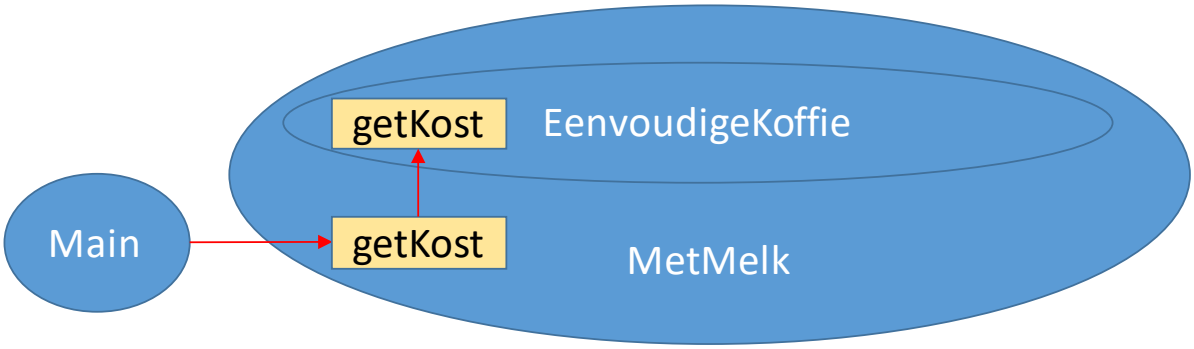
(1) `MetMelk` decoreert een `EenvoudigeKoffie` die (onder de gedaante van `Koffie`) als constructor parameter binnenkomt.
(2) Je geeft deze parameter door aan de base class constructor. Deze onthoudt de parameter in een protected variabele.
(3) De kost van `MetMelk` is de kost van de koffie die hij decoreert plus de kost van de melk zelf: één euro.
(4) De bereidingswize van `MetMelk` is de bereidingswijze van de koffie die hij decoreert plus de bereidingswijze van de melk zelf.
(5) `MetChocolade` decoreert een `EenvoudigeKoffie` of een `MetMelk` die (onder de gedaante van `Koffie`) als constructor parameter binnenkomt.
(6) De kost van `MetChocolade` is de kost van de koffie die hij decoreert plus de kost van de chocolade zelf: twee euro.
(7) De bereidingswize van `MetChocolade` is de bereidingswijze van de koffie die hij decoreert plus de bereidingswijze van de chocolade zelf.

Grafishe voorstellingen:

Je kan de kost vragen van een EenvoudigeKoffie, zonder decorator:

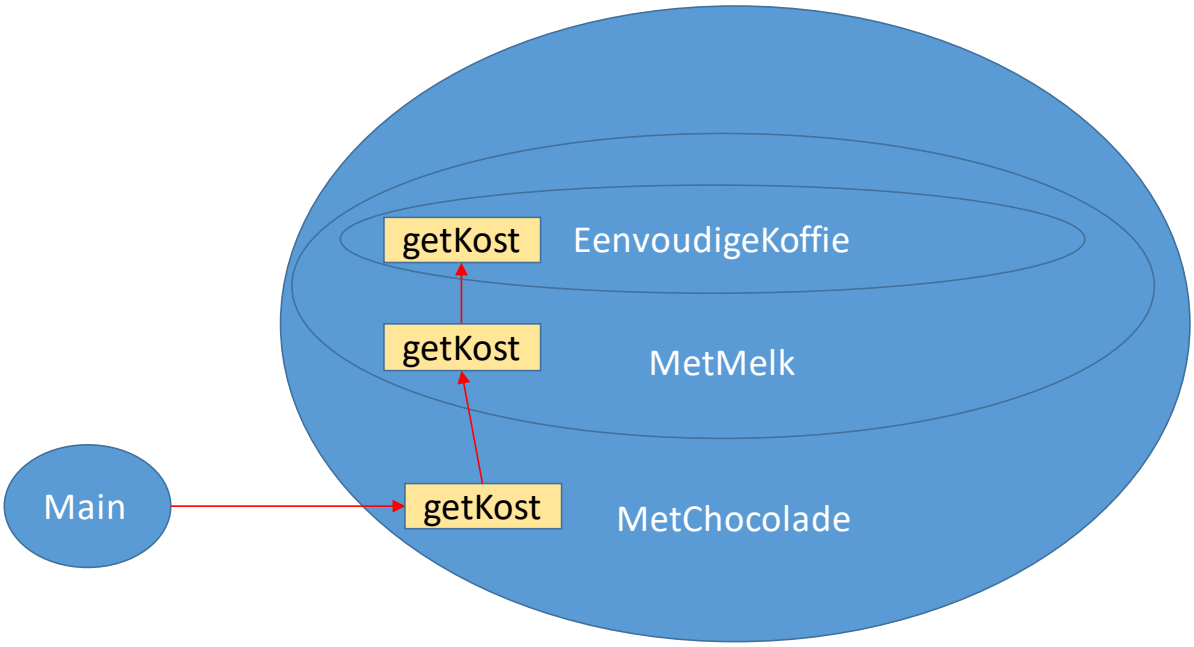


Je kan ook de kost vragen van een `EenvoudigeKoffie`, gedecoreerd met een `MetMelk`:



De `getKost` method van `MetMelk` roept dan de `getKost` method van `EenvoudigeKoffie` op.

Je kan ook de kost opvragen van een `EenvoudigeKoffie`, gedecoreerd met een `MetMelk` én met een `MetChocolade`:



De `getKost` method van `MetChocolade` roept dan de `getKost` method van `MetMelk` op.
Deze roept op zijn beurt de `getKost` van EenvoudigeKoffie op.

De decoratie is nu hard gecodeerd in de Main.
Je kan de decoratie ook dynamisch doen, bijvoorbeeld op basis van invoer van de gebruiker:

```
package be.vdab;
import java.util.Scanner;
public class Main {
  public static void main(String[] args) {
    Koffie koffie = new EenvoudigeKoffie();
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Melk (j/n):");
      if ("j".equals(scanner.next())) {
        koffie = new MetMelk(koffie);
      }
      System.out.print("Chocolade (j/n):");
      if ("j".equals(scanner.next())) {
        koffie = new MetChocolade(koffie);
      }
      System.out.println(koffie.getKost());
      System.out.println(koffie.getBereidingswijze());
    }
  }
}
```

8.4 Het decorator pattern in de Java API

Het decorator pattern wordt gebruikt bij bestandsverwerking in de Java API. Voorbeeld.

Je maakt in de root van de harde schijf een directory `data`.
Je maakt daarin volgend tekstbestand met als bestandsnaam `liedje.txt`.

```
Jantje zag eens pruimen hangen,
O! als eieren zo groot.
't Scheen, dat Jantje wou gaan plukken,
Schoon zijn vader 't hem verbood.
```

Je leest dit bestand regel per regel, voorzien van lijnnummers:

```
package be.vdab;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
public class Main {
  public static void main(String[] args) {
    try (LineNumberReader reader = new LineNumberReader(                ❶
      new BufferedReader(                                               ❷
        new FileReader("/data/liedje.txt"))) {
      String line = reader.readLine();
      while (line != null) {
        System.out.print(reader.getLineNumber());
        System.out.print(':');
        System.out.println(line);
        line = reader.readLine();
      }
    } catch (IOException ex) {
      ex.printStackTrace(System.err);
    }
  }
}
```

(1) Een `LineNumberReader` decoreert een `BufferedReader`. De `LineNumberReader` voegt daarbij de functionaliteit toe bij te houden de hoeveelste lijn gelezen wordt.

(2) Een `BufferedReader` decoreert op zijn beurt een `FileReader`. De `BufferedReader` voegt daarbij de functionaliteit toe de leessnelheid te verhogen door in grote blokken te lezen.

Decorator: zie takenbundel

9 OBSERVER

9.1 Categorie

Behavioral design pattern

9.2 Probleem

Een ding of persoon uit de werkelijkheid heeft

- Eigenschappen
- Handelingen die het ding of de persoon kan uitvoeren
- Gebeurtenissen die in het ding of de persoon optreden.

Je drukt eigenschappen uit als private variabelen in een class.
Je drukt handelingen uit als methods in een class.
Java heeft geen basisingrediënt om gebeurtenissen in een class uit te drukken.

9.3 Voorbeeld

Een koerswijziging is een gebeurtenis die optreedt in een aandeel. Wanneer een koerswijziging gebeurt, willen de aandeelhouders die dit aandeel in bezit hebben, hiervan op de hoogte gebracht worden. Je wil in de class Aandeel niet hard coderen welke deze aandeelhouders zijn.

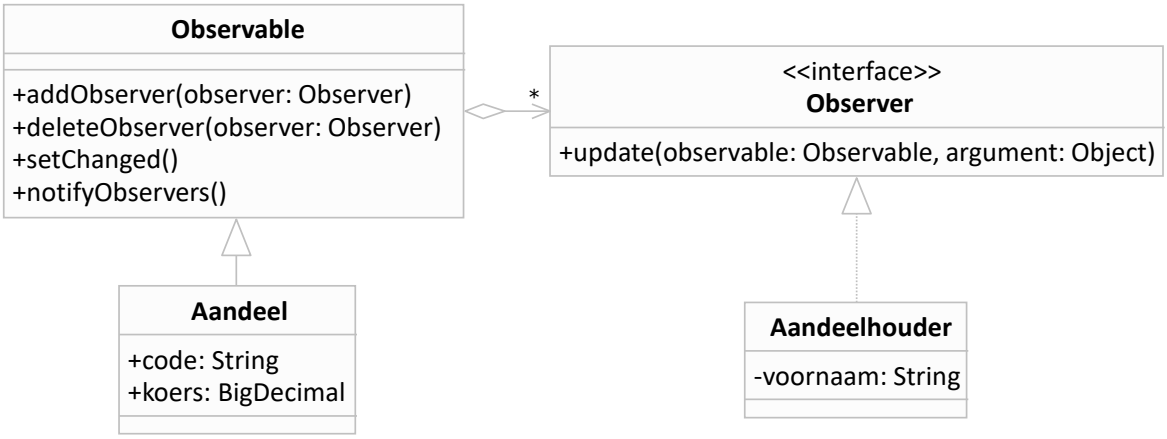
9.4 Oplossing

Het observer design pattern lost dit probleem op. In een object kunnen gebeurtenissen optreden. Andere objecten reageren als zich een gebeurtenis voordoet in het object.

Het object waarin zich gebeurtenissen voordoen heet *Observable*.

Het object (of objecten) dat reageert op die gebeurtenissen heet *Observer.*

Het design pattern is zeer flexibel: het observable bevat de code van de gebeurtenissen, maar bevat niet het type van de observers of het aantal observers.
Dit laat toe observers toe te voegen zonder de observable te wijzigen
(Dit volgens het principe *a class should be closed for modification)*.



Om dit design pattern uit te werken gebruik je classes en interfaces uit de Java API.

- Je erft je class die de rol speelt van observable (`Aandeel`) van `Observable`.
- Je class die de rol speelt van observer (`Aandeelhouder`) implementeert `Observer`.

De class Observable bevat een verzameling Observers. De class Aandeel erft deze verzameling. Deze verzameling bevat de aandeelhouders die het aandeel in bezit hebben. Je kan een observer aan deze verzameling toevoegen met de method `addObserver`. Je kan een observer uit deze verzameling verwijderen met de method `deleteObserver`. Als een gebeurtenis optreedt, roep je de method `setChanged` op, gevolgd door de method `notifyObservers`. Deze method overloopt de observers in de verzameling en roept op elke observer de method `update` op, om de observer van de gebeurtenis op de hoogte te brengen. Deze method bevat twee parameters.

De eerste parameter is de Observable waarin de gebeurtenis zich voordoet.
Het gebruik van de tweede parameter valt buiten het bereik van deze cursus.

```
package be.vdab;
import java.math.BigDecimal;
import java.util.Observable;
public class Aandeel extends Observable {
  private final String code;
  private BigDecimal koers;
  public Aandeel(String code) {
    this.code = code;
  }
  public void setKoers(BigDecimal nieuweKoers) {
    this.koers = nieuweKoers;
    setChanged();
    notifyObservers();
  }
  public String getCode() {
    return code;
  }
  public BigDecimal getKoers() {
    return koers;
  }
}
```

```
package be.vdab;
import java.util.Observable;
import java.util.Observer;
public class Aandeelhouder implements Observer {
  private final String voornaam;
  public Aandeelhouder(String voornaam) {
    this.voornaam = voornaam;
  }
  @Override
  public void update(Observable observable, Object argument) {
    if (!(observable instanceof Aandeel)) {
      throw new IllegalArgumentException();
    }
    Aandeel aandeel = (Aandeel) observable;
    System.out.println(
      voornaam + " reageert op de nieuwe koers van " +
      aandeel.getCode() + ':' + aandeel.getKoers());
  }
}
```

Je gebruikt deze classes als volgt:

```
package be.vdab;
import java.math.BigDecimal;
public class Main {
  public static void main(String[] args) {
    Aandeel aandeel = new Aandeel("ORCL");
    Aandeelhouder larry = new Aandeelhouder("Larry");
    Aandeelhouder james = new Aandeelhouder("james");
    aandeel.addObserver(larry);
    aandeel.addObserver(james);
    aandeel.setKoers(BigDecimal.valueOf(39));
    System.out.println();
    aandeel.deleteObserver(james);
    aandeel.setKoers(BigDecimal.valueOf(40));
  }
}
```

Observer: zie takenbundel

10 STRATEGY

10.1 Categorie

Behavioral design pattern

10.2 Probleem

Je hebt voor een bepaald algoritme meerdere implementaties waaruit je moet kiezen.
Je wil deze implementaties niet in de class zelf uitwerken, maar in de *gebruiker* van de class (bvb. de class Main). Op die manier moet je de class niet wijzigen als er nog een algoritme bijkomt (Je volgt hiermee het principe *A class should be closed for modification*).

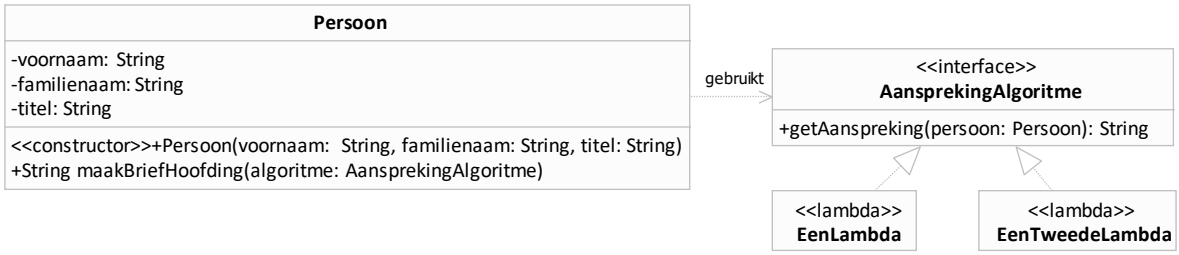
10.3 Voorbeeld

Een persoon heeft een voornaam, een familienaam en een titel. Er bestaan meerdere algoritmes om een briefhoofding voor de persoon te maken. Een eerste algoritme geeft een formele aanspreking (`Geachte heer Smits`). Een tweede algoritme geeft een informele aanspreking (`Dag Jean`). Je wil deze algoritmes niet hard coderen in de class Persoon. Zo kan je later nog algoritmes toevoegen.

10.4 Oplossing

Je beschrijft het algoritme in een functional interface.

Je geeft een lambda, die dit algoritme implementeert, mee als parameter van de method in de class Persoon die dit algoritme gebruikt.




```
package be.vdab;
@FunctionalInterface
public interface AansprekingAlgoritme {
  String getAanspreking(Persoon persoon);
}
```

```
package be.vdab;
public class Persoon {
  private String voornaam;
  private String familienaam;
  private String titel;
  public Persoon(String voornaam, String familienaam, String titel) {
    this.voornaam = voornaam;
    this.familienaam = familienaam;
    this.titel = titel;
  }
  public String getVoornaam() {
    return voornaam;
  }
  public String getFamilienaam() {
    return familienaam;
  }
  public String getTitel() {
    return titel;
  }
```

```
  public String maakBriefHoofding(AansprekingAlgoritme algoritme) {
    StringBuilder builder = new StringBuilder();
    builder.append("Brussel,");
    LocalDate vandaag = LocalDate.now();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("d/M/yyyy");
    builder.append(formatter.format(vandaag));
    builder.append("\n");
    builder.append(algoritme.getAanspreking(this));
    builder.append("\n");
    return builder.toString();
  }
}
```

Je gebruikt deze classes als volgt:

```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    Persoon[] personen=new Persoon[]  {
      new Persoon("Jean", "Smits", "heer"),
      new Persoon("Jeanine", "Desmet", "mevrouw")};
    // briefhoofdingen met informele aansprekingen:
    for (Persoon persoon : personen) {
      System.out.println(persoon.maakBriefHoofding((pers) ->
       "Dag " + pers.getVoornaam()));
    }
    // briefhoofdingen met formele aansprekingen:
    for (Persoon persoon : personen) {
      System.out.println(persoon.maakBriefHoofding((pers) ->
        "Geachte " + pers.getTitel() + ' '+ pers.getFamilienaam()));
    }
  }
}
```

In dit voorbeeld bevatten de algoritmes slechts één lijn code. Als de algoritmes meerdere lijnen code bevatten, kan je ze uitschrijven als static methods:

```
package be.vdab;
public class Main {
  public static void main(String[] args) {
    Persoon[] personen = new Persoon[] { new Persoon("Jean", "Smits", "heer"),
      new Persoon("Jeanine", "Desmet", "mevrouw") };
    for (Persoon persoon : personen) {
      System.out.println(persoon.maakBriefHoofding(
        (pers) -> informeel(pers.getVoornaam())));
    }
    for (Persoon persoon : personen) {
      System.out.println(persoon.maakBriefHoofding(
        (pers) -> formeel(pers.getFamilienaam(), pers.getTitel())));
    }
  }
  private static String informeel(String voornaam) {
    return "Dag " + voornaam;
  }
  private static String formeel(String familienaam, String titel) {
    return "Geachte " + titel + ' ' + familienaam;
  }
}
```

Strategy: zie takenbundel

11 COLOFON

Domeinexpertisemanager: Jean Smits

Moduleverantwoordelijke: Hans Desmet

Medewerkers: Hans Desmet

Versie: 20/8/2018

Nummer dotatielijst: