Inhoud

Eindtoets

Eindtoets

INTRODUCTIE

Deze eindtoets is bedoeld als voorbereiding op het tentamen van de cursus Objectgeoriënteerd programmeren in Java 2 en is te beschouwen als proeftentamen. Het is belangrijk dat u de eindtoets pas probeert te maken op het moment dat u denkt klaar te zijn met de tentamenvoorbereiding. Hebt u over dat laatste nog twijfels, bekijk dan nog eens de leerdoelen en bestudeer de samenvattingen in de leereenheden om te ontdekken welke onderdelen u nog onvoldoende beheerst.

Toegestane hulpmiddelen

Tijdens het tentamen mag het cursusmateriaal geraadpleegd worden. Dat cursusmateriaal hoeft niet 'schoon' te zijn, maar mag ook aantekeningen en dergelijke bevatten. De cursussite op studienet vermeldt welk ander materiaal u eventueel nog mag raadplegen.

Tentamenduur

Een tentamen duurt drie uur. We adviseren u dan ook de eindtoets binnen een aaneengesloten periode van drie uur te maken.

Samenstelling

Het aantal opgaven, de moeilijkheidsgraad en de verdeling over de leerstof komen overeen met het tentamen. Het tentamen bestaat uit vier vragen van gelijke zwaarte (25 punten per vraag). Het zwaartepunt van vraag 1 ligt in blok 1, van vraag 2 in blok 2, van vraag 3 in blok 3 en van vraag 4 in blok 4.

Terugkoppeling

De antwoorden op de opgaven staan in de terugkoppeling. We willen echter benadrukken dat u het meest leert als u eerst de opgaven maakt en pas daarna de antwoorden controleert.

Beoordeling

Het aantal punten dat u per opgave kunt behalen, staat bij die opgave vermeld. U kunt in totaal maximaal 100 punten halen. Voor een voldoende voor het tentamen moet u tenminste 55 punten behalen.

Studeeraanwijzingen
De studielast van deze eindtoets bedraagt circa 4 uur, inclusief het nakijken van de opgaven aan de hand van de terugkoppeling.

Opgaven

OPGAVE 1

Deze opgave gaat uit van de klassen Bag en Verzameling zoals beschreven in leereenheid 3 van de cursus. De interface van klasse Bag is gegeven op pagina 105 van deel 1. De klasse Verzameling is een subklasse van Bag; een specificatie van de methoden van Verzameling is getoond in figuur 1.

Method Summary

Verzameling	doorsnede(Verzameling v) Berekent de doorsnede van de verzameling met een andere verzameling.
Verzameling	vereniging(Verzameling v) Berekent de vereniging van de verzameling met een andere verzameling.
boolean	voegToe(int elem) Voegt een nieuw element toe aan de verzameling; heeft alleen effect als het element nog niet tot de verzameling behoort.

Methods inherited from class verzameling.Bag

afmeting, bevat, elementen, equals, isLeeg, maakLeeg, toString, verwijder

FIGUUR 1 Methoden van klasse Verzameling

Gegeven is het volgende codefragment.

```
Bag b = new Verzameling();
b.voegToe(1);
b.voegToe(1);
```

3 punten

a Wordt dit codefragment door de compiler geaccepteerd? Zo niet, is het voor de compiler acceptabel te maken door een cast toe te voegen? Licht uw antwoord kort toe; als er een cast nodig is, toon die dan.

3 punten

b Wat gebeurt er bij verwerking van het fragment (met cast, indien nodig)? Indien het fragment correct verwerkt wordt, hoeveel elementen bevat b dan na afloop? Licht uw antwoord toe en ga daarbij in op welke versie van voegToe wordt uitgevoerd.

Bekijk nu het volgende codefragment, waarbij gegeven is dat b een variabele is van (gedeclareerd) type Bag.

```
b.voegToe(1);
Verzameling v = b;
v.voegToe(1);
```

3 punten

c Wordt dit codefragment door de compiler geaccepteerd? Zo niet, is het voor de compiler acceptabel te maken door een cast toe te voegen? Licht uw antwoord kort toe; als er een cast nodig is, toon die dan.

3 punten

d Het codefragment toont geen toekenning aan b. Stel dat ook het actuele type van b op het moment van verwerking Bag is. Wat gebeurt er dan bij verwerking van het fragment (met cast, indien nodig)? Indien het fragment correct verwerkt wordt, hoeveel elementen bevat b dan na afloop?

De klassen Bag en Verzameling uit leereenheid 3 kunnen alleen gebruikt worden voor het opslaan van gehele getallen. We kunnen echter ook generieke versies van deze klassen maken, die gebruikt kunnen worden voor de opslag van willekeurige elementen (maar wel zo dat alle elementen in één bepaalde Bag of Verzameling allemaal van hetzelfde type zijn). Ter onderscheid van de oorspronkelijke klassen noemen we deze klassen GenBag en GenVerzameling.

5 punten

e De eerste regel(s) van de klassen Bag en Verzameling zien er uit als volgt:

```
public class Bag {
  // dit attribuut bevat de elementen in de bag
  private ArrayList<Integer> elementen = null;


public class Verzameling extends Bag {
```

Toon de overeenkomstige regels van de klassen GenBag en GenVerzameling.

8 punten

f Het verschil A – B van twee verzamelingen A en B is de verzameling van alle elementen van A die niet voorkomen in B. Toon een volledige definitie van een (nieuwe) methode verschil van klasse GenVerzameling die dit verschil als waarde teruggeeft.

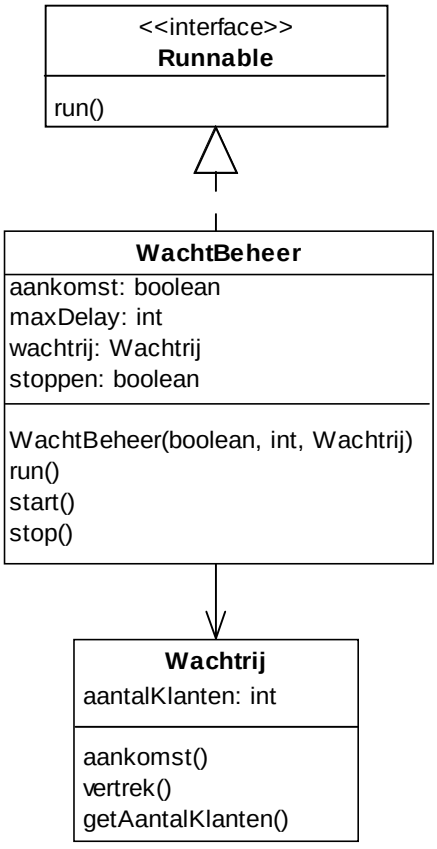
OPGAVE 2

Deze opgave gaat over de simulatie van een wachtrij, bijvoorbeeld voor een balie. Zo af en toe komt er een klant binnen. Is de balie vrij, dan wordt de klant meteen geholpen; zo niet, dan gaat deze wachten. Als een klant afgehandeld, is de volgende aan de beurt. De tijd tussen de aankomst van twee opeenvolgende klanten is variabel, evenals de tijd die het duurt om een klant te helpen.

Bij deze eenvoudige simulatie kan de gebruiker de *gemiddelde* tijd instellen tussen twee opeenvolgende aankomsten van een klant, evenals de *gemiddelde* tijd die het kost om een klant te helpen. De simulatie toont de groei van de wachtrij bij verschillende gemiddelden (zie figuur 2; elk vierkantje representeert een klant). De klant bij de balie wordt ook meegerekend.

FIGUUR 2 Er staan zes klanten in de rij

Figuur 3 toont het ontwerp van de domeinlaag van deze applicatie (met de GUI houden we ons verder niet bezig).



FIGUUR 3 Ontwerp voor wachtrij-applicatie

De klasse Wachtrij representeert de wachtrij. Alleen de lengte van de rij is van belang; deze is vastgelegd in het attribuut aantalKlanten. Elke keer dat de methode aankomst wordt aangeroepen, komt er een klant bij; elke keer dat de methode vertrek wordt aangeroepen, gaat er een klant af, tenzij het aantal klanten al 0 was.

Een instantie van de klasse WachtBeheer is verantwoordelijk voor het bepalen van de tijd die verstrijkt tussen twee opeenvolgende gebeurtenissen van hetzelfde type (dus tussen twee aankomsten of twee vertrekken). Er zijn twee instanties van deze klasse nodig, één voor de aankomst van klanten en één voor het vertrek. De constructor heeft drie parameters:
– een boolean die aangeeft of het gaat om aankomst (true) of vertrek (false)
– een int die de *gemiddelde* wachttijd weergeeft tussen twee van die gebeurtenissen
– de wachtrij.

In de methode start wordt een draad gecreëerd en gestart die met tussenpozen de methode aankomst respectievelijk vertrek aanroept op de wachtrij. De tijd dat de draad wacht voor de volgende aanroep wordt gedaan, wordt random bepaald, als een getal tussen 0 en maxDelay. Hierbij is maxDelay gelijk aan het dubbele van de gemiddelde wachttijd, die als parameter aan de constructor is meegegeven.

De methode stop zorgt dat de draad eindigt.

6 punten

a Toon een volledige implementatie van de klasse Wachtrij. U hoeft geen package- en importopdrachten te tonen, en ook geen javadoc. Zijn er methoden die u synchronized maakt? Licht dit toe.

9 punten

b Toon een volledige implementatie van de klasse WachtBeheer. Ook hier hoeft u geen package- en importopdrachten te tonen en geen javadoc.
Aanwijzingen
De (object)methode nextInt(int n) van klasse Random levert een random-waarde r met $0 \leq r < n$. De klasse heeft een parameterloze constructor

6 punten

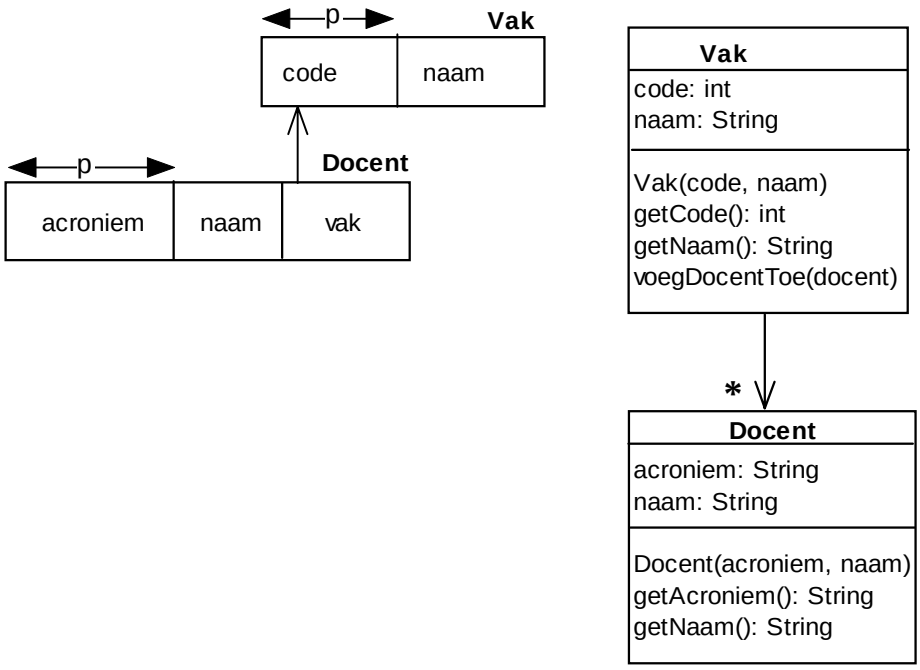
c De klasse WachtBeheer implementeert in dit ontwerp de interface Runnable. Zou WachtBeheer in plaats daarvan gebruik kunnen maken van een Timer?

4 punten

d Toon een codefragment dat de volledige simulatie van een wachtrij creëert en start.

OPGAVE 3

Gegeven is een zeer eenvoudige database van vakken en docenten, als getoond in figuur 4. Elke docent geeft één vak; een vak heeft een of meer docenten. Een vak heeft een unieke code (een geheel getal); een docent heeft een uniek acroniem (een lettercode zoals “BPO” of “HJP”, die gebaseerd is op de naam van de docent).
Figuur 4 toont ook het bijbehorende domeinmodel.



FIGUUR 4 Eenvoudige database

Deze opgave gaat over de datalaag van een applicatie die vakken en bijbehorende docenten kan inlezen.

3 punten

a De kolom vak in de tabel Docent heeft als waarde een geheel getal, overeenkomend met een vakcode. Waarom bevat de klasse Docent in het domeinmodel geen overeenkomstig attribuut vak van type int?

Een applicatie die werkt met deze database, heeft een datalaag bestaande uit één klasse VakMapper. De belangrijkste methode van deze klasse is de methode leesAlles, met de volgende specificatie:

```
/**
 * Deze methode leest alle vakken en bijbehorende docenten
 * uit de database. De methode gooit geen exception op;
 * als er bij het lezen iets mis gaat, wordt een
 * foutmelding afgedrukt en de applicatie beëindigd.
 * @return  een lijst van alle vakken met hun docenten
 */
public ArrayList<Vak> leesAlles()
```

4 punten

b Een of meer attributen van de klasse VakMapper zijn van het type PreparedStatement. Toon aan ieder attribuut van dit type een toekenning, waarbij u er van uit mag gaan dat attribuut con van type Connection de connectie met de database bevat.

7 punten

c Toon een implementatie van de methode leesAlles. Definieer hulpmethoden als u daar behoefte aan heeft. NB: Figuur 4 toont ook de methoden van de klassen Vak en Docent.

De programmeur van de applicatie bedenkt dat het eigenlijk handiger is om de vakkenlijst als één object op te slaan, en voegt daartoe (onder meer) aan de klasse VakMapper een methode toe als volgt:

```
/**
 * Deze methode serializeert een lijst met vakken en
 * bewaart de geserializeerde lijst in een bestand met
 * de naam "vakken.ser". Deze methode gooit geen
 * exception op, maar geeft een foutmelding en stopt
 * als het serializeren mislukt
 * @param vakkenlijst  de lijst met vakken
 */
public void serializeerAlles(ArrayList<Vak> vakkenlijst)
```

4 punten

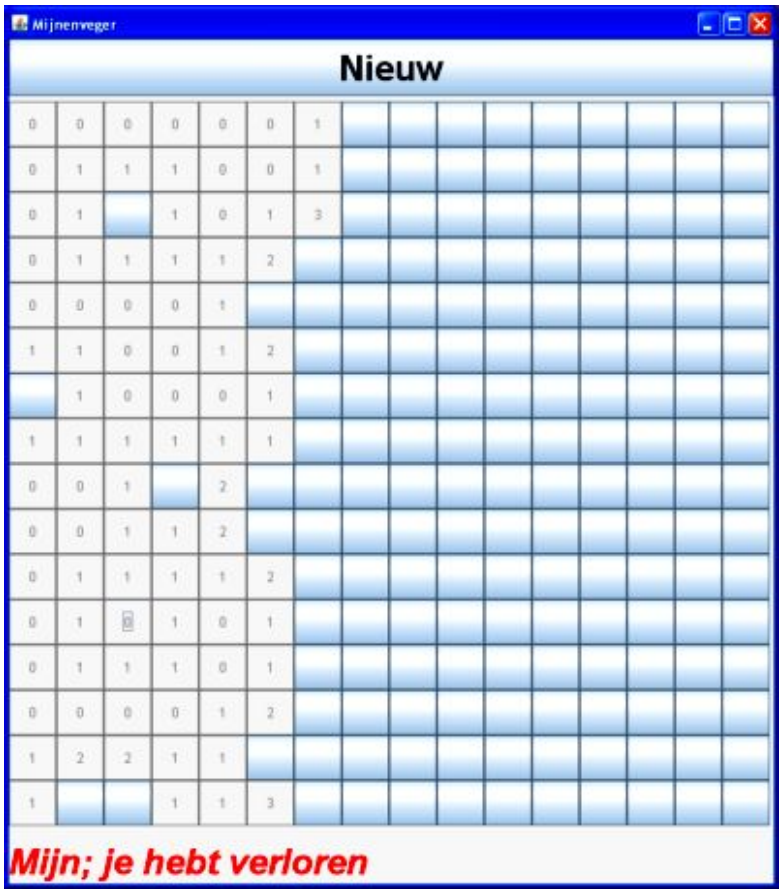
d Welke wijzigingen zijn nodig in de klassen uit het domeinmodel om deze uitbreiding mogelijk te maken? U hoeft geen code te tonen.

7 punten

e Toon een implementatie van de methode serializeerAlles.

OPGAVE 4

Figuur 5 toont een eigen implementatie van het spel Mijnenveger.



FIGUUR 5 Mijnenveger in Java

Het frame bevat een knop met opschrift Nieuw waarmee een nieuw spel gestart kan worden, een matrix van 16 bij 16 knoppen en een label die aan het eind van het spel aangeeft of de speler gewonnen of verloren heeft.

Het mijnenveld bevat in totaal 40 mijnen, verspreid over de 16 x 16 posities. De speler klikt steeds op een knop. Bevindt zich op die positie een mijn, dan is het spel uit: de speler heeft verloren. Bevindt zich op die positie geen mijn, dan wordt de knop disabled en krijgt als tekst het aantal mijnen (minimaal 0, maximaal 8) in de aangrenzende posities. De speler kan zo gaandeweg ontdekken waar de mijnen zitten.

De applicatie bevat vier eigen klassen, als volgt:
– De klasse Positie representeert één positie in het mijnenveld. De klasse beheert twee gegevens over deze positie: bevat deze een mijn, en is de positie al opengeklikt.
– De klasse Mijnenveld representeert het mijnenveld behoren bij één spel. Bij creatie van het mijnenveld worden de mijnen (random) verdeeld.

– De klasse MijnenvegerFrame representeert het hoofdvenster van de GUI (zie figuur 5).
– De klasse MijnenPanel representeert het panel met knoppen. Opengeklikte knoppen zijn disabled en hebben een opschrift (NB: ze verdwijnen dus niet; de cijfers staan gewoon op de disablede knoppen!).

De applicatie gebruikt voor het melden van winst of verlies aan het eind van het spel het Observerpatroon.

10 punten

a Toon van deze applicatie een globaal ontwerp in de vorm van een klassendiagram. Licht uw antwoord toe. U hoeft geen attributen en methoden van de klassen op te nemen.

5 punten

b Welke layoutmanagers worden gebruikt in deze applicatie? Motiveer uw antwoord.

De klasse MijnenPanel heeft de volgende globale structuur.

```
public class MijnenPanel extends JPanel {
  private JButton[][] knoppen = null;
  private Mijnenveld mijnenveld = null;

  /**
   * De constructor geeft het panel een GridLayout,
   * creëert de knoppen, voegt ze toe en geeft ze
   * een event handler
   * @param Mijnenveld  het mijnenveld
   */
  MijnenPanel(Mijnenveld mijnenveld) {
    ...
  }

  /**
   * Klasse voor het afhandelen van een klik op
   * een knop.
   */
  public class KlikLuisteraar extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      ...
    }
  }
}
```

10 punten

c Toon een implementatie van de constructor van MijnenPanel.
Aanwijzing
De klasse Mijnenveld beschikt over de volgende constanten:

```
public static final int RIJEN = 16;
public static final int KOLOMMEN = 16;
```

TERUGKOPPELING

Uitwerking van de opgaven

1 a De code is correct. Bij de toekenning van een waarde van type Verzameling aan een variabele van type Bag is geen cast vereist omdat Bag een superklasse is van Verzameling (een zogenaamde “upcast”, zie het schema op pagina 71).

b Het actuele type van b bepaalt welke versie van de methode voegToe wordt uitgevoerd. Dit actuele type is Verzameling en dus wordt die versie uitgevoerd. Na verwerking van het getoonde fragment bevat b daarom één element (de tweede 1 wordt niet toegevoegd).

c De compiler accepteert de toekenning aan v niet. Verzameling is een subklasse van Bag en dus is er sprake van een “downcast” (zie pagina 71). De code is voor de compiler acceptabel te maken door een cast toe te voegen:

```
b.voegToe(1);
Verzameling v = (Verzameling)b;
v.voegToe(1);
```

d Indien het actuele type van b Bag is, leidt verwerking van dit fragment tot een ClassCastException: de belofte aan de compiler dat b het actuele type Verzameling heeft, is immers niet nagekomen.

e Deze regels zien er bijvoorbeeld als volgt uit:

```
public class Bag<E> {
  // dit attribuut bevat de elementen in de bag
  private ArrayList<E> elementen = null;


public class Verzameling<E> extends Bag<E> {
```

f Een implementatie van de methode vereniging ziet er uit als volgt.

```
public GenVerzameling<E> verschil(GenVerzameling<E> v) {
  // maak een nieuw verzamelinginstantie
  GenVerzameling<E> verschil = new GenVerzameling<E>();
  // voeg elementen van "eigen" verzameling toe, mits
  // die niet ook in v voorkomen
  for (E elem: this.elementen()) {
    if (!v.bevat(elem)) {
      verschil.voegToe(elem);
    }
  }
  return verschil;
}
```

2 a De klasse Wachtrij ziet er bijvoorbeeld als volgt uit.

```
public class Wachtrij {
  private int aantalKlanten;

  public synchronized void aankomst() {
```

```
    aantalKlanten++;
  }

  public synchronized void vertrek() {
    if (aantalKlanten > 0) {
      aantalKlanten--;
    }
  }

  public int getAantalKLanten() {
    return aantalKlanten;
  }
}
```

De methode aankomst verhoogt het aantal klanten; de methode vertrek verlaagt het aantal klanten mits dit groter was dan 0. Omdat beide methoden hetzelfde attribuut wijzigen, is het veilig om ze synchronized te maken. In dit geval kan er weinig misgaan, maar dat zou anders zijn wanneer verschillende draden toegang zouden hebben tot de methode vertrek: dan kan het aantal klanten kleiner worden dan 0.
Als u de methoden niet synchronized heeft gemaakt en goed beargumenteert waarom dat in dit geval geen kwaad kan, is het ook goed.

b Een mogelijke uitwerking is de volgende.

```
public class WachtBeheer implements Runnable {
  private boolean aankomst = false;
  private int maxDelay = 0;
  private Wachtrij wachtrij = null;
  private boolean stoppen = false;

  public WachtBeheer(boolean aankomst,
                     int gemiddeldDelay,
                     Wachtrij wachtrij) {
    this.aankomst = aankomst;
    this.maxDelay = 2 * gemiddeldDelay;
    this.wachtrij = wachtrij;
  }

  public void run() {
    Random random = new Random();
    while (!stoppen) {
      int delay = random.nextInt(maxDelay);
      try {Thread.sleep(delay);}
      catch (InterruptedException e){}
      if (aankomst) {
        wachtrij.aankomst();
      }
      else {
        wachtrij.vertrek();
      }
    }
  }

  public void stop() {
    stoppen = true;
  }

  public void start() {
    Thread thread = new Thread(this);
    thread.start();
  }
```

```
}
```

De methode start creëert en start een nieuwe draad.
De methode stop geeft attribuut stoppen de waarde true, zodat de draad zal eindigen.
De lus in de methode run wordt verwerkt zolang het attribuut stoppen de waarde false heeft. In elke doorgang door de lus wordt een delay bepaald dat tussen 0 en maxDelay ligt.

c Bij gebruik van een Timer wordt een bepaalde methode uitgevoerd met vaste tussenpozen. Daar hebben we in dit geval niet veel aan; in dit geval moet de tijd tussen twee opeenvolgende aankomsten en vertrekken immers variëren. Het moet dus echt met een draad.

d Dit fragment ziet er bijvoorbeeld als volgt uit:

```
Wachtrij wachtrij = new Wachtrij();
WachtBeheer aankomstBeheer =
    new WachtBeheer(true, 1000, wachtrij);
aankomstBeheer.start();
WachtBeheer vertrekBeheer =
    new WachtBeheer(true, 1000, wachtrij);
vertrekBeheer.start();
```

3 a De kolom vak in de database bevat een verwijssleutel naar de tabel Vak. Verwijssleutels horen in deze vorm in een objectgeoriënteerd model niet thuis; als er een verwijzing nodig is van Docent naar Vak, dan gaat dat met een associatie (een attribuut van type Vak).

b We tonen twee mogelijkheden.
– Het versturen naar de database van een query is een dure operatie; als alles moet worden ingelezen kunnen we dat het best met zo min mogelijk queries doen. De meest efficiënte oplossing leest alles in één keer in door gebruik te maken van een JOIN:

```
pselectalles = con.prepareStatement(
   "SELECT * FROM Vak " +
   "JOIN Docent ON vak.code = Docent.vak");
```

De resultaattabel bij het gebruik van deze SQL-opdracht heeft de volgende vorm:

Vak.code	*Vak.naam*	*Docent.acroniem*	*Docent.naam*
231	Wiskunde I	JLO	Lodder
231	Wiskunde I	BPA	Pauw
311	Engels	LBA	Benvenuti
315	Spaans	AVE	Veen
315	Spaans	HSI	Sint

– Het is ook mogelijk om twee SQL-opdrachten te gebruiken, één voor de vakken en één voor de docenten bij een gegeven vak. Er zijn dan dus twee attributen van type PreparedStatement, bijvoorbeeld pselectvakken en pselectdocenten.
De toekenningen daaraan zien er als volgt uit:

```
pselectvakken = con.prepareStatement(
```

```
        "SELECT * FROM Vak");

pselectdocenten = con.prepareStatement(
        "SELECT * FROM Docent WHERE vak = ?");
```

Hoewel de eerste oplossing beter is, rekenen we de tweede ook goed (deze cursus is geen databasecursus).

c Ook nu tonen we twee uitwerkingen.

– Gebruik van het PreparedStatement-object pselectalles leidt tot code als volgt:

```
public ArrayList<Vak> leesAlles() {
  ArrayList<Vak> vakken = new ArrayList<Vak>();
  try {
    ResultSet res = pselectalles.executeQuery();
    int laatsteCode = -1;
    Vak vak = null;
    while (res.next()) {
      int code = res.getInt(1);
      if (code != laatsteCode) {
        String vaknaam = res.getString(2);
        vak = new Vak(code, vaknaam);
        vakken.add(vak);
        laatsteCode = code;
      }
      String acroniem = res.getString(3);
      String docentnaam = res.getString(4);
      vak.voegDocentToe(new Docent(acroniem, docentnaam));
    }
  }
  catch (SQLException e) {
    System.out.println("Het inlezen is mislukt");
    System.exit(0);
  }
  return vakken;
}
```

Algoritmisch is dit iets complexer dan de tweede mogelijkheid; we moeten bijhouden welk vak we al gezien hebben en pas een nieuw vak maken als er een nieuwe code opduikt.

– Bij gebruik van de twee PreparedStatement-objecten pselectvakken en pselectdocenten krijgen we bijvoorbeeld de volgende implementatie (we definieerden een hulpmethode leesDocenten die de docenten toevoegt aan een eerder ingelezen vak).

```
public ArrayList<Vak> leesAlles() {
  ArrayList<Vak> vakken = new ArrayList<Vak>();
  try {
    ResultSet res = pselectvakken.executeQuery();
    while (res.next()) {
      int code = res.getInt(1);
      String naam = res.getString(2);
      Vak vak = new Vak(code, naam);
      leesDocenten(vak);
      vakken.add(vak);
    }
  }
  catch (SQLException e) {
    System.out.println("Het inlezen is mislukt");
```

```
    System.exit(0);
  }
  return vakken;
}


private void leesDocenten(Vak vak) throws SQLException {
  pselectdocenten.setInt(1, vak.getCode());
  ResultSet res = pselectdocenten.executeQuery();
  while (res.next()) {
    String acroniem = res.getString(1);
    String naam = res.getString(2);
    vak.voegDocentToe(new Docent(acroniem, naam));
  }
}
```

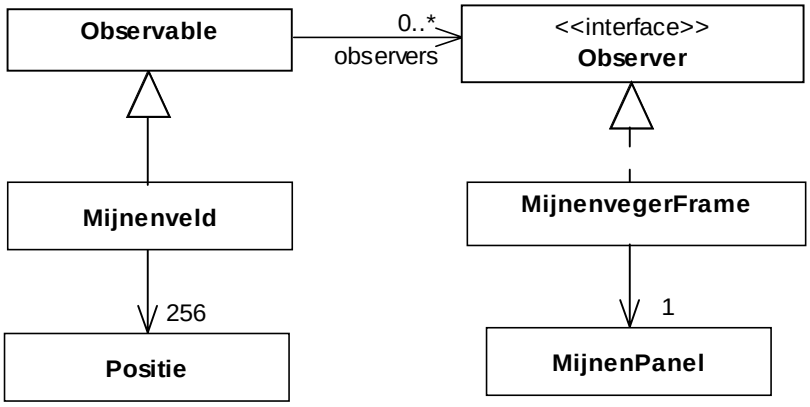
Er zijn allerlei varianten mogelijk. De hulpmethode kan zelf eventuele SQLExceptions afvangen. De hulpmethode kan ook alleen een vakcode als parameter krijgen en een lijst docenten als waarde teruggeven. Dat heeft als voordeel dat we een methode vermijden die zijn parameter wijzigt (al is dat voor een private methode wel accpetabel), maar als nadeel dat het minder efficient is, omdat Vak alleen een methode heeft om één docent toe te voegen; we moeten die lijst dan dus weer langs.

d De klassen Vak en Docent moeten beide de interface Serializable implementeren.

e Een mogelijke implementatie van deze methode is de volgende:

```
public void serializeerAlles(ArrayList<Vak> vakkenlijst) {
  ObjectOutputStream schrijver = null;
  try {
    schrijver = new ObjectOutputStream(
                  new FileOutputStream(
                    new File("vakken.ser")));
    schrijver.writeObject(vakkenlijst);
  }
  catch (IOException e) {
    System.out.println("Serializeren mislukt");
    System.exit(0);
  }
  finally {
    if (schrijver != null) {
      try {schrijver.close();}
      catch (IOException ioe) {}
    }
  }
}
```

4 a Figuur 6 toont een globaal klassendiagram.



FIGUUR 6 Globaal ontwerp voor MijnenvegerApplicatie

Het mijnenveld heeft 256 posities (* is ook goed); het MijnenVegerFrame heeft 1 MijnenPanel (de 1 mag ook worden weggelaten). Het Observer-patroon wordt gebruikt om in de klasse MijnenvegerFrame winst en verlies te melden. De klasse Mijnenveld beschikt over deze informatie en moet dus Observable zijn; de klasse MijnenvegerFrame beheert het label en is dus Observer.
Een alternatief is om het label zelf Observer te maken, maar dat vereist een extra klasse ObserverLabel (of zo) die niet is genoemd. Het is niet fout als u die in het ontwerp heeft opgenomen en tot Observer heeft gemaakt.

b Er worden twee layoutmanagers gebruikt:
– De contentPane van de klasse MijnenvegerFrame gebruikt een BorderLayout. De knop wordt in het noorden geplaatst, het mijnenPanel in het centrum en de label die de uitslag meldt, in het zuiden.
– De klasse MijnenPanel gebruikt een GridLayout om de knoppen te plaatsen.

c Een mogelijke uitwerking is de volgende.

```
MijnenPanel(Mijnenveld mijnenveld) {
  this.mijnenveld = mijnenveld;

  // zet de layout manager
  setLayout(new GridLayout(
             Mijnenveld.RIJEN, Mijnenveld.KOLOMMEN));

  // creëer de array van knoppen
  knoppen =
    new JButton[Mijnenveld.RIJEN][Mijnenveld.KOLOMMEN];

  // creëer een event handler voor alle knoppen
  KlikLuisteraar klikLuisteraar =
    new KlikLuisteraar();

  // Creëer nu de knoppen zelf.
  // Elke knop wordt aan het panel toegevoegd en
  // krijgt klikluisteraar als event handler
  for (int i=0; i < Mijnenveld.RIJEN; i++) {
```

```
    for (int j=0; j < Mijnenveld.KOLOMMEN; j++) {
      knoppen[i][j] = new JButton();
      add(knoppen[i][j]);
      knoppen[i][j].addMouseListener(klikLuisteraar());
    }
  }
}
```