TENTAMEN

Softwaresystemen

vakcode: 201300071
datum: 19 januari 2015
tijd: 8:45 - 11:45

UITWERKING

Algemeen

- Bij dit tentamen mag gebruik gemaakt worden van: de handleiding, de slides van de colleges en de boeken die voorgeschreven zijn als studiemateriaal voor de module (of, als je deze niet hebt, een afdruk/kopie van de betreffende pagina's).
  Je mag *geen* gebruik maken van: uitwerkingen van opgaven die op Blackboard gepubliceerd zijn (recommended exercises en oude tentamens) of eigen materiaal (afdrukken van je eigen practicumopgaven, aantekeningen in welke vorm dan ook).
- Het aantal punten voor het tentamen wordt meegenomen in de berekening van het eindcijfer van de module, op de manier zoals aangegeven in de handleiding.
- Dit tentamen bestaat uit 7 opgaven, waarvoor in het totaal 100 punten behaald kunnen worden. Het minimale aantal punten per opgave bedraagt 0 punten. Het eindcijfer van het tentamen wordt bepaald door de optelsom van de punten per opgave.
- Studenten die *alleen* de programmeerlijn volgen, of *alleen* het vak Programmeren 2 herkansen, hoeven Opgave 7 niet te maken. Zij kunnen maximaal 85 punten halen. Het eindcijfer voor het tentamen wordt als volgt berekend: (aantal behaalde punten * 10/85).
- Studenten die *alleen* de ontwerplijn volgen, hoeven alleen Opgave 7 te maken. Zij kunnen maximaal 15 punten halen. Het eindcijfer voor het tentamen wordt als volgt berekent: (aantal behaalde punten * 10/15).

Opgave 1 (15 punten)

In dit tentamen gaan we een aantal interfaces, klassen en methoden ontwikkelen die gebruikt kunnnen worden voor een database met koopwoningen.

a. *(5 punten)* Definieer een interface `Woning` met methoden om de volgende kenmerken van een woning op te vragen: prijs, aantal kamers, en aantal verdiepingen. Geef specificaties die het gewenste gedrag van de methoden beschrijven. N.B. er worden niet hele uitgebreide specificaties verwacht, maar wel meer dan alleen `ensures true;`

b. *(5 punten)* Een appartement is de benaming voor een woning van één verdieping (bijvoorbeeld in een flat). Implementeer een klasse `Appartement`. Denk hierbij ook aan de constructor.

c. *(5 punten)* Een studio of eenkamerappartement is een klein appartement bestaande uit één kamer, meestal bedoeld voor één persoon. Implementeer een klasse `Studio` als uitbreiding van `Appartement`. Denk hierbij ook aan de constructor.

Antwoord op opgave 1

a. *(5 punten)* Puntenverdeling: 2 voor code, 3 voor specificatie. Aftrek:

- Methodenaam ipv result in postconditie: -2
- Typefout in specificatie: -3
- Methodebodies of variabeledeclaraties in interface: -2

```
public interface Woning {
    /*@ ensures \result >= 0; */
    public int getPrijs();

    /*@ ensures \result >= 1; */
    public int getAantalKamers();

    /*@ ensures \result >= 1; */
    public int getAantalVerdiepingen();
}
```

b. *(5 punten)* Aftrek:

- `implements Woning` verkeerd/ontbreekt: -3
- Ongedefinieerde variabele voor een aantal verdiepingen: -3
- Wel waarde voor aantal verdiepingen, maar niet constant: -2
- 1 ipv constante gebruikt: -1
- Non-private variables: -2

```
public class Appartement implements Woning {
    private int prijs;
    private int kamers;
    public static final int AANTAL_VERDIEPINGEN = 1;

    public Appartement(int p, int k) {
        prijs = p;
        kamers = k;
    }
```

```
    public int getAantalVerdiepingen() {
        return AANTAL_VERDIEPINGEN;
    }

    public int getPrijs() {
        return prijs;
    }

    public int getAantalKamers() {
        return kamers;
    }
}
```

c. *(5 punten)* Aftrek:

- Geen `extends`: -3
- Overbodige `implements`: -2
- Variabelen uit superklasse opnieuw gedeclareerd: -3
- Verkeerde aanroep `super`: -1
- 1 ipv constante gebruikt: -1

```
public class Studio extends Appartement {

    public static final int AANTAL_KAMERS = 1;

    public Studio(int p) {
        super(p, AANTAL_KAMERS);
    }

    public int getAantalKamers() {
        return AANTAL_KAMERS;
    }
}
```

Opgave 2 (20 punten)

Hieronder staat een klasse `Aanbod` die een verzameling beschikbare koopwoningen bijhoudt.

```
public class Aanbod {
    //@ private invariant aanbod.size() == aantal;
    private Set<Woning> aanbod= new HashSet<Woning>();
    private int aantal;

    /** Retourneert alle woningen uit het aanbod. */
    /*@ pure */ public Set<Woning> getAanbod() {
        return aanbod;
    }

    /** Retourneert een map die gegeven een aantal kamers k,
      * alle huizen oplevert met k kamers. */
    public Map<Integer,Set<Woning>> groepeerOpAantalKamers() {
        // te implementeren
    }

    /** Retourneert de n huizen uit het aanbod met de laagste prijs, gesorteerd
      * van laag naar hoog. */
    public List<Woning> laagstePrijs(int n) {
        // te implementeren
    }

    public void nieuweWoning(Woning w) {
        aanbod.add(w);
        aantal = aantal + 1;
    }
```

a. *(10 punten)* Implementeer de methode `groepeerOpAantalKamers`. Deze methode levert een map op die gegeven een aantal kamers, alle woningen oplevert met dit aantal kamers. Dus gegeven een key `k`, zal `groepeerOpAantalKamers.get(k)` alle woningen met `k` kamers opleveren.

b. *(10 punten)* Implementeer de methode `laagstePrijs`. Deze levert de `n` goedkoopste huizen uit het aanbod, gesorteerd van laag naar hoog.

Antwoord op opgave 2

Bij beide onderdelen zijn o.a. de volgende criteria gebruikt voor aftrek:

- Instantiatie van interface: -2
- Gebruik van get operaties op set: -2
- Bij onderdeel a: inefficiente implementatie: -2
- Gebruik van modifiers voor lokale variabelen: -2
- Een enkele woning opleveren ipv een set: -3
- Geen test of element al voorkomt in de map (en daardoor overschrijven): -4

a. *(10 punten)*

```
public Map<Integer,Set<Woning>> groepeerOpAantalKamers() {
    Map<Integer,Set<Woning>> result = new HashMap<Integer,Set<Woning>>();
    for(Woning woning : aanbod) {
        int kamers = woning.getAantalKamers();
        if (result.containsKey(kamers)){
            result.get(kamers).add(woning);
        } else {
            Set<Woning> woningSet = new HashSet<Woning>();
            woningSet.add(woning);
            result.put(kamers, woningSet);
        }
    }

    return result;
}
```

b. *(10 punten)*

```
public List<Woning> laagstePrijs(int n) {
    // Resultaatlijst waar alle woningen gesorteerd op prijs in komen
    List<Woning> result = new ArrayList<Woning>();

    for(Woning originalWoning : aanbod){
        // Zodra de prijs van een woning kleiner is dan een prijs van een
        // reeds gesorteerde woning, zetten we hem op die plek.
        boolean found = false;
        // ipv break kan ook een extra boolean gebruikt worden
        for (int i = 0 ; i < result.size() && !found; i++){
            if(originalWoning.getPrijs() < result.get(i).getPrijs()) {
                result.add(i,originalWoning);
                found = true;
            }
        }
        // Als de prijs van een woning niet kleiner was dan een woning
        // reeds gesorteerd, dan moet hij achteraan de lijst.
        if(!found){
            result.add(originalWoning);
        }
    }
    // haal de top uit de gesorteerde lijst.
    return result.size() <= n ? result : result.subList(0, n);
}
```

Opgave 3 (15 punten)

a. *(5 punten)* Pas de implementatie van `nieuweWoning` aan, zodat deze een zelfgedefinieerde exceptie oplevert als de woning al in het aanbod voorkomt.

b. *(10 punten)* Voeg een methode aan `Aanbod` toe, *inclusief specificatie*

```
public Set<Woning> nieuweWoningen(Set<Woning> ws)
```

die probeert alle woningen in `ws` aan het aanbod toe te voegen, en alle woningen waarvoor dit niet lukt (omdat de woning al in het aanbod zit) teruglevert.

Antwoord op opgave 3

a. *(5 punten)*

```
public void nieuweWoning2(Woning w) throws WoningException {
    if (aanbod.add(w)) {
        aantal = aantal + 1;
    } else {
        throw new WoningException();
    }
}

class WoningException extends Exception {
    // Empty body suffices for the purpose at hand
}
```

b. *(10 punten, waarvan 5 voor de specificatie)*

```
/*@ requires ws != null;
    ensures (\forall Woning w;
             \result.contains(w); \old(getAanbod()).contains(w));
    ensures (\forall Woning w;
             \old(getAanbod()).contains(w); \result.contains(w));
    ensures getAanbod().containsAll(\old(getAanbod());
    ensures getAanbod().containsAll(ws);
    ensures (\forall Woning w;
             getAanbod().contains(w);
             ws.contains(w) || \old(getAanbod()).contains(w));
 */
public Set<Woning> nieuweWoningen(Set<Woning> ws) {
    Set<Woning> result = new HashSet<Woning>();
    for (Woning woning : ws) {
        try {
            nieuweWoning2(woning);
        } catch (WoningException e) {
            result.add(woning);
        }
    }
    return result;
}
```

De eerste twee `ensures` specificeren samen dat het resultaat *precies* bestaat uit de woningen in `ws` die al in het oude aanbod zaten; de voldende drie `ensures` specificeren samen dat het nieuwe aanbod *precies* bestaat uit de vereniging van het oude aanbod en `ws`. Als er `pure` verzamelingtheoretische operaties in de `Collection`-inderface zouden bestaan, zouden deze postcondities heel wat bondiger op te schrijven zijn!

Opgave 4 (10 punten)

De volgende interface bevat functionaliteit om een selectie uit een woningaanbod te maken.

```
public interface Wens {
    /** Retourneert alle aangeboden woningen die voldoen aan deze wens. */
    public Set<Woning> passend(Aanbod a);
}
```

Geef twee implementaties van dit interface:

a. *(5 punten)* Een klasse `PrijsWens`, met velden voor de onder- en bovengrens aan de prijs van de gewenste woningen. Een woning voldoet aan een `PrijsWens` als de prijs van die woning in het opgegeven bereik ligt.

b. *(5 punten)* Een klasse `CombinatieWens`, met een veld `Set<Wens> wensen` waarin deelwensen verzameld zijn; een woning voldoet aan de `CombinatieWens` als hij aan alle deelwensen voldoet.

Antwoord op opgave 4

a. *(5 punten)*

```
public class PrijsWens {
    private int onder, boven;

    public PrijsWens(int onder, int boven) {
        this.onder = onder;
        this.boven = boven;
    }

    public Set<Woning> passend(Aanbod a) {
        Set<Woning> result = new HashSet<Woning>();
        for (Woning w: a.getAanbod()) {
            if (w.getPrijs() >= onder && w.getPrijs() <= boven) {
                result.add(w);
            }
        }
        return result;
    }
}
```

b. *(5 punten)*

```
public class CombinatieWens {
    private Set<Wens> wensen = new HashSet<Wens>();

    public void nieuweWens(Wens w) {
        wensen.add(w);
    }

    public Set<Woning> passend(Aanbod a) {
        Set<Woning> result = new HashSet<Woning>(a.getAanbod());
        for (Wens w: wensen) {
            result.retainAll(w.passend(a));
        }
        return result;
    }
}
```

Opgave 5 (10 punten)

Verschillende makelaars en kopers kunnen tegelijkertijd, in parallel, het aanbod bekijken, door methoden in de klasse `Aanbod` aan te roepen. Makelaars kunnen ook de methode `nieuweWoning` aanroepen, om een woning aan het aanbod toe te voegen. *Ga in deze opgave uit van de oorspronkelijke implementatie van `nieuweWoning` zoals hierboven gegeven, en niet van je eigen versie van opgave 3.*

Met deze implementatie is het mogelijk dat 2 makelaars tegelijkertijd een woning proberen toe te voegen.

a. *(3 punten)* Leg uit wat er dan fout kan gaan

b. *(4 punten)* Illustreer dit aan de hand van een voorbeeldexecutie

c. *(3 punten)* Leg uit hoe dit opgelost kan worden

Antwoord op opgave 5

a. *(3 punten)* Als de methode tegelijk wordt uitgevoerd, is het mogelijk dat de teller (aantal) niet meer correct is.

b. *(4 punten)* Stel de lijst is leeg en er worden twee woningen tegelijk toegevoegd: 1. Eerst worden de woningen aan de lijst toegevoegd dat klopt; 2. Echter de regel aantal = aantal + 1 wordt nu bij beide makelaars tegelijk uitgevoegd. Bij beide wordt eerst de huidige inhoud van aantal opgehaald dit is in beide gevallen 0. Vervolgens wordt er bij makelaar A 1 bij opgeteld en is de inhoud van aantal gelijk aan 1. Vervolgens gaat makelaar B ook verder met het uitgelezen getal (0) en telt er 1 bij op en past aantal aan naar 1. De inhoud van aantal zou 2 moeten zijn, maar is 1.

c. *(3 punten)* De methode synchronized maken.

Opgave 6 (15 punten)

Stel je voor dat bovenstaande woningverkoop-site inderdaad geïmplementeerd is en waar gedurende de tijd vele additionele features aan toegevoegd zijn. Een van de toegevoegde features is accounts (voor makelaars en bezoekers) met een wachtwoord login. Oorspronkelijk werden de wachtwoorden simpelweg opgeslagen, maar later werd in plaats daarvan de cryptografische hash (bijvoorbeeld MD5 of SHA256) van het wachtwoord opgeslagen.

a. *(3 punten)* Stel de site staat enkel toe dat het wachtwoord precies uit 6 karakters bestaat en enkel cijfers, de letters a t/m d, de letters A t/m D en de karakters ";" en ":" gebruikt mogen worden. Hoeveel verschillende mogelijkheden voor wachtwoorden zijn er en ligt je antwoord toe?

b. *(5 punten)* Waarom is het slimmer om de hash van het wachtwoord op te slaan? Wat is het voordeel van een hash functie als scrypt ten opzichte van MD5 of SHA256 voor het opslaan van wachtwoorden?

Nu blijkt er een probleem te zijn met de site en jij wordt gevraagd het op te lossen. Je komt onderstaande broncode tegen. Blijkbaar heeft iemand bedacht om voor de zogenaamde flexibiliteit ergens in het login proces het wachtwoord te voorzien met een string dat beschrijft welke (cryptografische) hash-functie gebruikt moet worden. Dus een wachtwoord "fiets" in combinatie met de hash functie MD5 wordt doorgeven als "MD5:fiets".

```
private Map<String, String> passwordDB;

/**
 * Generates the hex-encoded hash of the password using a very flexible
 * scheme. The hash-function to use is embedded in the password: it is
 * separated by a colon. Example passwords: "MD5:abcd" or "SHA1:s3cr3t".
 * @throws NoSuchAlgorithmException
 */
public String getPWHash(String password) throws NoSuchAlgorithmException {
    String[] r = password.split(":");
    String prefix = r[0];
    String realPassword = r[1];
    MessageDigest md = MessageDigest.getInstance(prefix);
    md.update(realPassword.getBytes());
    byte[] digest = md.digest();
    return Hex.encodeHexString(digest);
}

public boolean login(String username, String password) {
    boolean result = true;
    if (passwordDB.containsKey(username)) {
        try {
            String passwordHash = getPWHash(password);
            if (!passwordDB.get(username).equals(passwordHash)) {
                result = false;
            }
        } catch (Exception e) {
            // Whatever, shouldn't happen, right?
        }
    } else {
        result = false;
    }
    return result;
}
```

De javadoc van de methode `String.split` vermeldt het volgende:

`public String[] split(String regex)`

Splits this string around matches of the given regular expression.

This method works as if by invoking the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The string "boo:and:foo", for example, yields the following results with these expressions:

Regex	Result
`:`	`{ "boo", "and", "foo" }`
`o`	`{ "b", "", ":and:f" }`

Parameters

regex - the delimiting regular expression

Returns

the array of strings computed by splitting this string around matches of the given regular expression

c. *(7 punten)* Stel een aanvaller heeft volledige controle over de inhoud van de variabelen `username` en `password` die meegegeven worden aan de `login`-methode. Er zijn (minstens) twee manieren voor deze aanvaller om de ervoor te zorgen (door manipulatie van de `username` en `password` variabelen) dat `login` true oplevert zonder dat de aanvaller echt een wachtwoord weet. Geef er minstens 1. Geef ook aan hoe je dit probleem (eenvoudig) zou kunnen verhelpen.

Antwoord op opgave 6

a. *(3 punten)* Er zijn 20^6 mogelijkheden.De toegestaande tekens zijn 0 t/m 9, a t/m d, A t/m D, ";" en ":". Dit zijn in totaal 20 karakters. Twintig karakters over 6 posities geeft: $20 * 20 * 20 * 20 * 20 * 20 = 10^6 = 64000000$ mogelijkheden.

b. *(5 punten)* Veel mensen hergebruiken hun wachtwoord op verschillende plekken. Als dan de database met wachtwoorden van een site op een of andere manier lekt, dan kunnen de wachtwoorden gebruikt worden om ook op andere websites en systemen in te loggen. Door een cryptografische hash-functie toe te passen wordt dat (tot op zekere hoogte) voorkomen. Een van de eigenschappen van een hash-functie is namelijk dat het erg moeilijk is om te inverteren, oftwel, het is moeilijk (bijna onmogelijk) om vanuit de uitkomst van een hash functie de originele input te reproduceren.
Hash-functies als MD5 en SHA256 zijn gemaakt voor snelheid. Een hash functie als scrypt is bewust zo gemaakt dat het veel rekenkracht en geheugen kost om een string te hashen. Hierdoor duurt het langer om een hash uit te rekenen en is het dus veel lastiger om een bruteforce aanval uit te voeren.

c. *(7 punten)* In de login methode wordt in de catch de boolean `result` niet op false gezet. Bij een exception geeft de login methode true terug. Er kan op (in ieder geval) twee manieren een exception worden gegooid. Er moet minstens 1 gegeven zijn.

- Indien je een ongeldig algoritme mee geeft (aap:wachtwoord) dan wordt er een `NoSuchAlgorithmException` gegooid.
- Indien je geen : meegeeft (wachtwoord), dan wordt er een `IndexOutOfBoundsException` gegooid.

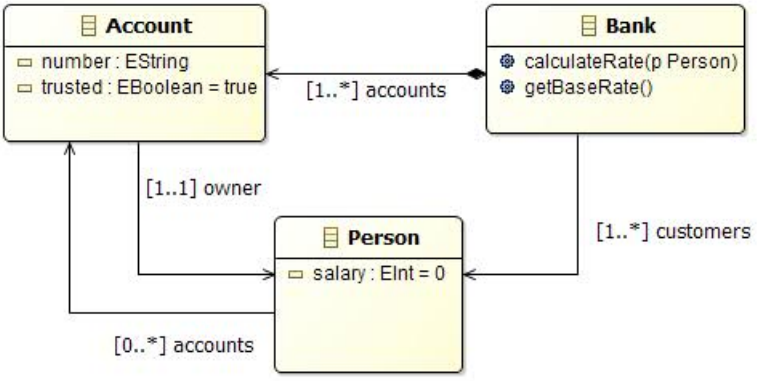
Een eenvoudige manier om dit te voorkomen is om de boolean `result` bij initialiseren op `false` te zetten.

Opgave 7 (15 punten)

Deze opgave hoeft alleen gemaakt te worden door studenten die de designlijn volgen, dus niet door studenten die alleen de programmeerlijn doen, of alleen Programmeren 2 herkansen.

In deze opgave wordt op twee plekken een OCL-constraint gevraagd. Deze hoeven niet syntactisch helemaal te kloppen, maar in grote lijnen moeten ze voldoen aan het formaat van OCL — zie §10.5 van Bennett et al.

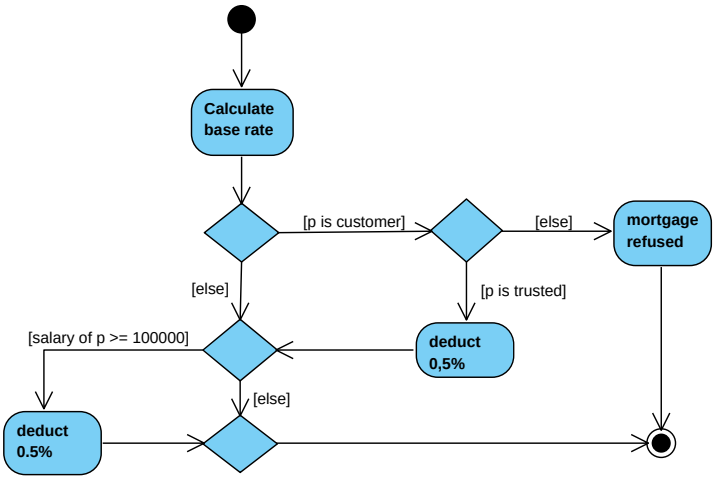
Om een hypotheek te verkrijgen laat een persoon zijn rentepercentage uitrekenen. De klassen die hierbij komen kijken zijn hieronder grafisch weergegeven.



a. (*3 punten*) Welke Java-representatie (m.a.w., welk type) zou je kiezen voor de vier associaties in het diagram?

b. (*3 punten*) Geef een model constraint voor het bovenstaande diagram in OCL.

Het volgende activity diagram specificeert de operatie `calculateRate(p:Person)` van de klasse `Bank`, waarin de rente van een hypotheek wordt berekend.



c. (*3 punten*) Geef een equivalente specificatie in de vorm van een decision table (zie Fig. 10.1 van Bennett et al.)

d. (*3 punten*) Geef een equivalente specificatie in de vorm van pseudo-code (zie §10.4 van Bennett et al.)

e. (*3 punten*) Geef een equivalente specificatie in de vorm van een OCL-constraint

Antwoord op opgave 7

a. (*3 punten*) Bijvoorbeeld:

- `Bank.accounts`: een `List<Account>` geordend op account number
- `Account.owner`: een `Person`-instantievariabele
- `Person.accounts`: een `Set<Account>`
- `Bank.customers`: geen (is afgeleid uit `Bank.accounts` en `Account.owner`)

b. (*3 punten*) Twee voorbeelden:

```
context Bank
inv: self.accounts -> forall(a | self.customers -> contains(a))
inv: self.customers -> forall(c | self.accounts -> exists(a | a.owner=c))

context Person
inv: self.accounts -> forall(a | a.owner = self)
```

c. (*3 punten*) Een equivalente decision table (N.B.: er zijn ook andere goede antwoorden):

Conditions and actions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Conditions					
Is person a customer?	N	N	N	Y	Y
Is person trusted?	–	–	N	Y	Y
Is salary at least 100000?	N	Y	–	N	Y
Actions					
Calculate base rate	X	X	X	X	X
Deduct 0,5%		X		X	
Deduct 1,0%					X
Refus mortgage			X		

d. (*3 punten*) Equivalente pseudo-code:

```
calculate base rate;
if p is customer
then if p is trusted
     then deduct 0,5%
     else refuse mortgage
     end if
end if
if salary of p >= 100000
then deduct 0,5%
end if
```

e. (*3 punten*) Een equivalente OCL constraint:

```
context Bank::calculateRate(p:Person)
pre:  not self.customers -> contains(p)
      or  self.accounts  -> exists(a | a.trusted and a.owner = p)
post: return getBaseRate()
             - (self.customers -> contains(p) ? 0,5 : 0)
             - (p.salary >= 100000 ? 0,5 : 0)
```