Övning vecka 3.

Denna vecka ska vi beröra designprinciper och refaktorering. Huvudtemat är dock *design by contract* och *Liskov Substitution Principle*. Vi kommer även att se på ett par av Java's mörka skrymslen.

Uppgift 1 Design principer och refaktorering

En fastighetsskötare har bl.a. ansvar för att temperaturen i hyresgästernas lägenheter är på en behaglig nivå. Därför har denne valt att förlita sig på ett automatiserat system. Metoden nedan är ett utdrag ur systemet och kontrollerar temperaturen lägenhet för lägenhet och gör, om nödvändigt, temperaturförändringar:

```
public void changeTemp() {
   for (Apartment a : apartments) {
      if (!a.tempIsPerfect()) {
         int change = a.tempDifference();
         a.setTemp(change);
      }
   }
}//changeTemp
```

Metoden strider dock mot en grundläggande design princip. Vilken? Åtgärda koden på så sätt att denna designprincip följs!

Uppgift 2 Design by contract

a) Skriv specifikationen för nedanstående metod

```
/**
   @pre    . . .
   @post  . . .
   @returns  . . .
**/
public static double mean (int[] a, int n) {
   double sum = 0;
   for (int i = 0; i < n; i++)
      sum = sum + a[i];
   return sum / n;

}//mean
```

b) Betrakta följande specifikation för metoden getMostFrequent:

```
// returns: the character that appears most frequently in str
// throws: NullPointerException if str == null
public char getMostFrequent(String str) { . . . }
```

Denna specifikation är inte väldefinierad för alla möjliga indatavärden. Ge (minst två) exempel på en indatasträng för vilken beteendet av metoden inte är väldefinierat.

Uppgift 3 Design by contract

Som ett arbetsprov i rekryteringsprocessen hos Quality Software AB har en programmerare fått i uppgift att skriva en klass för en farthållare som i vissa metoder använder sig av kontrakt (”programming by contract”). Resultatet ser du nedan:

```
/**
* CruiseControl for use in motorized vehicles
* @invariant mTargetSpeed <= mLegalSpeed && mTargetSpeed >= 0
*/
public class CruiseControl  {
   private boolean mActive;
   private float mTargetSpeed;
   private float mLegalSpeed;

   /**
   * Creates a new inactive CruiseControl
   */
   public CruiseControl()  {
       mTargetSpeed = 0;
       mLegalSpeed = GPS.getCurrentLegalSpeed();
       mActive = false;
       assert isLegal(mTargetSpeed);
       assert invariant();
   }

   /**
   * Method for checking the invariant
   */
   public boolean invariant() {
       return mTargetSpeed <= mLegalSpeed && mTargetSpeed >= 0;
   }

   /**
   * Activates the Cruise control
   * @pre speed >= 0
   * @post mTargetSpeed >=0
   */
   public void activate(float speed) {
       assert invariant();
       assert speed >= 0;
       mActive = true;
       mTargetSpeed = speed;
       assert invariant();
   }

   //Checks if speed is below or equal to the current maximum legal speed
   private boolean isLegal(float speed) {
       assert invariant();
       return speed <= mLegalSpeed;
   }

   /**
   * Changes the preferred speed which the cruise control tries to achieve
   * @param speed the new preferred speed
   * @pre speed >= 0 && mActive
   * @post isLegal == true
   */
   public void setTargetSpeed(float speed) {
       assert invariant();
       if (isLegal(speed) )
           mTargetSpeed = speed;
       else
           speed = mLegalSpeed;
       assert isLegal(speed);
       assert invariant();
   }
}//CruiseControl
```

Programmeraren har slarvat och kommer förmodligen inte att få någon anställning.

a) Varför är kontraktet för metoden setTargetSpeed inte giltigt? Förslå en ändring av klassen som rättar till detta, och motivera dina ändringar.

b) Håller klassinvarianten (för den del av klassen du kan se här)? Motivera ditt svar.

Uppgift 4 Specifikationer

Betrakta nedanstående tre specifikationer för metoden

double log(**double** x)

som returnerar den naturliga logaritmen för argumentet x.

I) @requires x > 0
@return y such that |e^y - x| <= 0.1

II) @return y such that |e^y - x| <= 0.001
@throws IllegalArgumentException if x <= 0

III) @requires x > 0
@return y such that |e^y - x| <= 0.001

IV) @return y such that |e^y - x| <= 0.001 if x > 0 and Double.NEGATIVE_INFINITY if x <= 0

För vart och ett av nedanstående par av specifikationer, ange vilken specifikation som är starkast. Om detta inte går att avgöra svara att de är lika starka.

a) I och II

b) I och III

c) I och IV

d) II och III

e) II och IV

f) III och IV

Uppgift 5 Liskov Substitution Principle

Liskovs substitutionsprincip (LSP) säger att alla objekt av en subtyp U ska kunna fungera som substitut för objekt av vilken som helst av U:s supertyper.

a) Är CTH en korrekt subtyp till klassen University i enlighet med typsystemet i Java? Är CTH en *äkta* subtyp till klassen University i enligt med *Liskov Substitution Principle*? Ge en kort motivering till ditt svar!

```
public class University {
   // effects: s.salary' > 40000
   public void graduate(Student s) { . . . }
}//University

public class CTH extends University {
   // effects: s.salary' > 40000 and s.hasTechReviewSubscription' = true
   public void graduate(Student s) { . . . }
}//CTH
```

b) Är Oxford en korrekt subtyp till klassen University i enlighet med typsystemet i Java? Är Oxford en *äkta* subtyp till klassen University i enligt med *Liskov Substitution Principle*? Ge en kort motivering till ditt svar!

```
public class University {
   // requires: s.bankBalance > 100000
   public void payTuition(Student s) { . . . }
}//University

public class Oxford extends University {
   // requires: s.bankBalance > 200000
   public void payTuition(Student s) { . . . }
}//Oxford
```

c) Är GIH en korrekt subtyp till klassen University i enlighet med typsystemet i Java? Är GIH en *äkta* subtyp till klassen University i enligt med *Liskov Substitution Principle*? Ge en kort motivering till ditt svar!

```
public class University {
   public void graduate(Student s) { . . . }
}//University

public class GIH extends University {
   //throws  CannotSwimCheckedException if student can't swim
   public void graduate(Student s) throws CannotSwimCheckedException { . . . }
}//GIH
```

d) Är CTH en korrekt subtyp till klassen Univeristy i enlighet med typsystemet i Java? Är CTH en sann subtyp till klassen University i enligt med *Liskov Substitution Principle*? Ge en kort motivering till dina svar!

```
public class University {
   // returns: number between 0 and 4.0
   public float gpa(Student s) { . . .}
}//University

public class CTH extends University {
   // returns: number between 0 and 5.0
   public float gpa(Student s) { . . . }
}//CTH
```

e) Är KTH en korrekt subtyp till klassen Univeristy i enlighet med typsystemet i Java? Är CTH en sann subtyp till klassen University i enligt med *Liskov Substitution Principle*? Ge en kort motivering till dina svar! Det gäller att Geek är en sann subtyp till Student.

```
public class University {
   public Student studentBodyPresident() { . . . }
}//University

public class KTH extends University {
   public Geek studentBodyPresident() { . . . }
}//KTH
```

Uppgift 6

Typreglerna i Java för fält (arrays) säger att fälttyper är *kovarianta* med respekt på deras elementtyper. Med andra ord: om S är en subtyp till T, så är S[] en subtyp till T[]. Till exempel gäller att Integer[] och Double[] är en subtyper till Number[], eftersom Integer och Double är en subtyper till Number.

Visa med ett exempel hur denna typregel skapar ett osäkert typsystem, dvs. ge ett exempel som är korrekt enligt Javas subtypsregel för fält, men som ger upphov till ett exekveringsfel.

Uppgift 7 Samma kontra lika

Betrakta nedanstående klass:

```
public class References {
    public static void main(String[ ] args) {
        stringRefs();
        integerRefs();
    }

    private static void print(String exp, boolean b) {
        System.out.println(exp + " -> " + b);
    }

    private static void printIdEq(Object a, Object b, String aName, String bName) {
        System.out.println(aName + " == " + bName + " -> " + (a == b) + "\t"
                           + aName + ".equals(" + bName + ") -> " + a.equals(b));
    }

    private static void stringRefs() {
        String s3 = new String("False");
        String s4 = new String("False");
        printIdEq(s3, s4, "s3", "s4");
        String s5 = "True";
        String s6 = "Tr" + "ue";
        printIdEq(s5, s6, "s5", "s6");
        String s7 = "False";
        String sx = "F";
        String s8 = sx + "alse";
        printIdEq(s7, s8, "s7", "s8");
    }

    private static void integerRefs() {
        Integer i = new Integer(2);
        Integer j = new Integer(2);
        print("i >= j", i >= j);
        print("i == j", i == j);
        print("i == 2", i == 2);
    }
}//References
```

När main-metoden exekveras erhålls följande utskrifter:

```
s3 == s4 -> false       s3.equals(s4) -> true
s5 == s6 -> true        s5.equals(s6) -> true
s7 == s8 -> false       s7.equals(s8) -> true
i >= j -> true
i == j -> false
i == 2 -> true
```

Troligen är inte detta vad man förväntar sig! För att förstå varför utskriften blir som den blir måste man söka förklaringen i hur Java har implementerat klassen String och omslagsklassen Integer. Slå upp detta i *The Java Language Specification, Java SE 7 Edition* (delkapitel 3.10.5 respektive 5.1.1 – 5.1.8) och förklara sedan resultatet i ovanstående utskrift.