©2018 Uno Holmer
holmer 'at' chalmers.se

TENTAMEN: Objektorienterad programutveckling, fk

Läs detta!

- *Uppgifterna är inte ordnade efter svårighetsgrad.*
- Börja varje hel uppgift på ett nytt blad. Skriv inte i tesen.
- Ordna bladen i uppgiftsordning.
- Skriv din tentamenskod på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Programkod skall skrivas i Java 5, eller senare version, och vara indenterad och renskriven.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklarationer, parameterlistor etc. får ej ändras.
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1

Denna metod bryter mot en viss designprincip, vilken? Refaktorera koden! Du får införa nya metoder.

```
public void func(int[] arr) {
    double mean = 0.0;
    int sum = 0;
    for ( int x : arr )
        sum += x;
    mean = sum/(double)arr.length;
    if ( mean < 10 ) {
        // Normalize low
        for ( int i = 0; i < arr.length; i++)
            arr[i] = arr[i] + 42;
    } else if ( mean > 20 ) {
        // Normalize high
        for ( int i = 0; i < arr.length - 1; i++)
            arr[i] = (arr[i] - arr[i+1])*2;
    }
}
```

(7 p)

Uppgift 2

Modifiera klassen så att den blir en *Singleton*-klass. Du kan utelämna sådant som inte är relaterat till *Singleton*-egenskapen.

```
public class MyClass {
    private Set<String> set;
    public MyClass() {
        set = new HashSet<>();
    }
    public void add(String s) {
        set.add(s);
    }
    public boolean contains(String s) {
        return set.contains(s);
    }
}
```

(6 p)

Uppgift 3

I ett program för utdelning av belöningar till anställda med olika anställningsår hittades nedanstående metod. Den bryter mot flera designprinciper. Nämn en sådan.

```
public static void giveBonus(ArrayList l) {
    for ( Object obj : l ) {
        if ( obj instanceof Academic ) {
            Academic a = (Academic)obj;
            if ( a.yearsEmployed() > 30 )
                a.take(new ExpensiveBonus());
            else if ( a.yearsEmployed() > 20 )
                a.take(new MediumBonus());
            else if ( a.yearsEmployed() > 10 )
                a.take(new CheapBonus());
            else
                a.take(new DefaultBonus());
        } else if ( obj instanceof OfficeClerk ) {
            OfficeClerk c = (OfficeClerk)obj;
            if ( c.yearsEmployed() > 30 )
                c.receive(new ExpensiveBonus());
            else if ( c.yearsEmployed() > 20 )
                c.receive(new MediumBonus());
            else if ( c.yearsEmployed() > 10 )
                c.receive(new CheapBonus());
            else
                c.receive(new DefaultBonus());
        } else {
            // possibly additional similar cases ...
        }
    }
}
```

Klasserna `Academic` och `OfficeClerk` deklareras

```
public class Academic {
    public void take(Object arg) { ... }
    public int yearsEmployed() { ... }
    // Other members omitted
  }
}
public class OfficeClerk {
    public void receive(Object arg) { ... }
    public int yearsEmployed() { ... }
    // Other members omitted
}
```

Bonusklassernas interna detaljer är oviktiga för uppgiften så de utelämnas här.

Refaktorera koden ovan genom att tillämpa designmönstren *Strategy* och *Factory*. Du får införa ytterligare lämpliga klasser och gränssnitt och relatera dessa till de givna klasserna.

(8 p)

Uppgift 4

Följande gränssnitt beskriver tärningar. Man kastar en tärning med `roll` och läser av värdet (antal ögon) med `getValue`:

```
public interface Die {
    void roll();
    int getValue();
}
```

Klassen `StandardDie` implementerar `Die` men vi går inte närmare in på dess detaljer. I en av kursens övningar diskuterades hur designmönstret *Decorator* kan användas för att definiera klassen `NoRepeatDie`. Ett objekt av denna klass gav kombinerat med en annan tärning en tärning som aldrig ger samma utfall två gånger i följd. I den här uppgiften skall vi på liknande sätt använda samma designmönster för att definiera klassen `HistoryDie`. Den fungerar som en vanlig tärning men den kan komma ihåg de senaste utfallen som kan efterfrågas med metoderna

```
public int available()
public int lookBack(int n) throws IndexOutOfBoundsException
```

Metoden `available` returnerar antalet tillgängliga sparade värden (noll från början) och `lookBack` det önskade värdet i historiken. Undantaget skall kastas om det önskade värdet inte existerar. Hur klassen skall kunna användas framgår av följande exempel:

```
Die standardDie = new StandardDie();
HistoryDie d = new HistoryDie(standardDie,3);
for ( int i = 0; i < 10; i++ )
    d.roll();
for ( int k = 0; k < d.available(); k++ )
    System.out.print(d.lookBack(k));
```

Den andra parametern till konstruktorn i `HistoryDie` anger hur många värden som maximalt skall sparas. Om t.ex. utfallen i loopen blev 4, 2, 6, 6, 1, 5, 3, <u>2, 4, 3</u> så skall utskriften bli 3 4 2. Använd en lämplig samlingsklass för att spara värden.

(8 p)

Uppgift 5

I en applikation för grafik infördes bl.a. klasserna `Rectangle` och `Group`.

```
public class Rectangle {
    private int width,height;
    public Rectangle(int width,int height) {
        setSize(width,height);
    }
    public int getWidth() {  return width; }
    public int getHeight() { return height; }
    public int getArea() { return width*height; }
    public void setSize(int width,int height) {
        this.width = width;
        this.height = height;
    }
}
```

```
public class Group {
    private List<Rectangle> allRectangles;
    private int totalArea;
    public Group() {
        allRectangles = new ArrayList<>();
        totalArea = 0;
        checkInvariant();
    }
    private void checkInvariant() {
        assert (getArea() == totalArea) : msg(totalArea,getArea());
    }
    private String msg(double expected,double found) {
        return "Area mismatch, expected: " + expected + ", actual: " + found;
    }
    public void add(Rectangle r) {
        checkInvariant();
        allRectangles.add(r);
        totalArea += r.getArea();
        checkInvariant();
    }
    public void resize(int scaleFactor) {
        checkInvariant();
        for ( Rectangle r : allRectangles ) {
            totalArea -= r.getArea();
            r.setSize(r.getWidth()*scaleFactor,
                      r.getHeight()*scaleFactor);
            totalArea += r.getArea();
        }
        checkInvariant();
    }
    public int getArea() {
        int sum = 0;
        for ( Rectangle r : allRectangles )
            sum += r.getArea();
        assert sum == totalArea : msg(totalArea,sum);
        return sum;
    }
}
```

När satserna

```
Group g = new Group();
Rectangle r = new Rectangle(10,20);
g.add(r);
System.out.println("Total area of g: " + g.getArea());
```

exekverades blev utskriften

```
Total area of g: 200
```

När satsen

```
System.out.println("Total area of g: " + g.getArea());
```

litet senare åter igen exekverades inträffade exekveringsfelet: *forts. nästa sida ->*

```
java.lang.AssertionError: Area mismatch, expected: 200.0, actual: 800.0
      at Group.getArea(Group.java:40)
      at Main.main(Main.java:19)
```

trots att vare sig `add` eller `resize` anropats för `g` efter den förra utskriften. Hur kan denna situation ha uppstått? Föreslå förändringar i `Rectangle` och `Group` så att felet elimineras. *Tips:* Ett nyckelord i sammanhanget är muterbarhet.

(8 p)

Uppgift 6

a)

Studera följande klasser:

```java
public interface Int {
    /**
     * Precondition: Pre
     * Postcondition: Post
     */
    void service();
}
```

```java
public class Main {
    public static void main(String[] arg) {
        client(new Impl1());
        client(new Impl2());
        client(new Impl3());
        client(new Impl4());
    }
    public static void client(Int impl) { ... }
}
```

```java
public class Impl1 implements Int {
    /**
     * Precondition: Pre1
     * Postcondition: Post1
     */
    public void service() {}
}
```

```java
public class Impl2 implements Int {
    /**
     * Precondition: Pre2
     * Postcondition: Post2
     */
    public void service() {}
}
```

```java
public class Impl3 implements Int {
    /**
     * Precondition: Pre3
     * Postcondition: Post3
     */
    public void service() {}
}
```

```java
public class Impl4 implements Int {
    /**
     * Precondition: Pre4
     * Postcondition: Post4
     */
    public void service() {}
}
```

För för- och eftervillkoren gäller följande förhållanden:

- Pre1 är starkare än Pre, Post1 är svagare än Post.
- Pre2 är svagare än Pre, Post2 är svagare än Post.
- Pre3 är starkare än Pre, Post3 är starkare än Post.
- Pre4 är svagare än Pre, Post4 är starkare än Post.

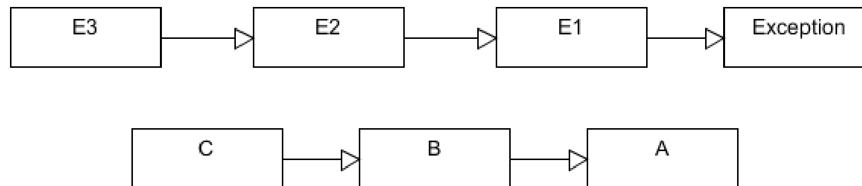
Vilket av metodanropen i `main` respekterar Liskovs substitutionsprincip? Motivera svaret!

(3 p)

Forts. nästa sida ->

b)

Vilka metoddeklarationer bland a-h nedan är typkorrekta om de placeras i klassen `Impl`?




```
public interface Int {
    B func(B arg) throws E2,E3;
}
```

```
public class Impl implements Int {
    public typ func(typ arg) throws typer { ... }
}
```

```
a) public B func(B arg) throws E2,E3 { ... }
b) public C func(B arg) throws E2,E3 { ... }
c) protected B func(B arg) throws E2,E3 { ... }
d) public B func(C arg) throws E2,E3 { ... }
e) public C func(B arg) throws E2 { ... }
f) public C func(B arg) { ... }
g) public C func(B arg) throws E1,E2 { ... }
h) public A func(B arg) throws E3 { ... }
```

(4 p)

Uppgift 7

Studera följande klasser:

```
public class A {
    public void f() { System.out.println("A.f"); }
    public static void g() { System.out.println("A.g"); }
    public void h(Object x) { System.out.println("A.h(Object) " + x); }
    public void h(String x) { System.out.println("A.h(String) " + x); }
    public void i(Object x) { System.out.println("A.i(Object) " + x); }
}
```

```
public class B extends A {
    public void f() { System.out.println("B.f"); }
    public static void g() { System.out.println("B.g"); }
    public void h(Object x) { System.out.println("B.h(Object) " + x); }
    public void i(String x) { System.out.println("B.i(String)" + x); }
}
```

```
public class C extends A {
    private int n;
    public C(int n) { this.n = n; }
    public boolean equals(C obj) {
        return obj != null && obj.n == n;
    }
}
```

```
// Vad skrivs ut?
A p = new A();
p.f();
p.g();
p = new B();
p.f();
p.g();
p.h("apa");
p.h(123);
p.i("bepa");
p = new C(42);
System.out.println(p.equals(new C(42)));
```

Vad skrivs ut när satserna i rutan ovan exekveras? Ange vilka principer som bestämmer vilken metod som exekveras för respektive metodanrop!

(8 p)

Uppgift 8

Binärfiler är oftast omöjliga att läsa med vanliga texthanterande program. Denna uppgift går därför ut på att skriva ett program som läser heltal av typen `long` från en binärfil och skriver dessa i textform, ett per rad, i en nyskapad textfil. Om indatafilen t.ex. heter `longnumbers` så skall utdatafilen ges namnet `longnumbers.txt`. Programmet bör ha den övergripande strukturen:

```
public class FileTranslator {
    public static void main(String[] arg) {
        ...
    }
    // ytterligare privata metoder ...
}
```

Namnet på indatafilen skall ges som argument till `main` när programmet körs, ex:

```
> java FileTranslator longnumbers
```

Eventuella undantag som orsakas av filhanteringen skall hanteras i `main`.
Tips: Använd en `DataInputStream` vid läsningen från filen.

(8 p)

Lösningsförslag till tentamen *P r e l i m i n ä r*

Kurs	**Objektorienterad programutveckling, fk**
Tentamensdatum	**2018-04-04**
Program	**DAI2**
Läsår	**2017/2018, lp 2**
Examinator	**Uno Holmer**

Uppgift 1 (7 p)

Metoden i tesen bryter mot designprincipen *separation of concern*. Den löser fyra olika uppgifter: Beräknar medelvärdet av fältelementen, väljer väg beroende på vad medelvärdet blev, samt utför två olika beräkningar (normaliseringar) av fälten. Detta kan delas upp i metoderna:

```
public void funcRefactored(int[] arr) {
    normalize(arr,computeMean(arr));
}

private double computeMean(int[] arr) {
    int sum = 0;
    for ( int x : arr )
        sum += x;
    return sum/(double)arr.length;
}

private void normalize(int[] arr,double mean) {
    if ( mean < 10 )
        normalizeLow(arr,mean);
    else if ( mean > 20 )
        normalizeHigh(arr,mean);
}

private void normalizeLow(int[] arr,double mean) {
    for ( int i = 0; i < arr.length; i++)
        arr[i] = arr[i] + 42;
}

private void normalizeHigh(int[] arr,double mean) {
    for ( int i = 0; i < arr.length - 1; i++)
        arr[i] = (arr[i] + arr[i+1])*2;
}
```

Uppgift 2 (6 p)

```
public class MySingletonClass {
    private Set<String> set;
    private static MySingletonClass instance = null;
    private MySingletonClass() {
        set = new HashSet<>();
    }
    public synchronized static MySingletonClass getInstance() {
        if ( instance == null )
            instance = new MySingletonClass();
        return instance;
    }
    public void add(String s) {
        set.add(s);
    }
    public boolean contains(String s) {
```

```
        return set.contains(s);
    }
}
```

Uppgift 3 (8 p)

Designen bryter mot designprinciperna *DIP* och *LSP*. Fallanalysen över olika personalkategorier är oflexibel och typosäker. Om metoderna take och receive unifieras kan vi införa ett övergripande gränssnitt för anställda. Även bonusklasserna bör ha ett övergripande gränssnitt.

```
public interface Employee {
    void receive(Bonus x);
    public int yearsEmployed();
}
public class Academic implements Employee {
    public void receive(Bonus x) { ... }
    public int yearsEmployed() { ... }
}
public class OfficeClerk implements Employee {
    public void receive(Bonus x) { ... }
    public int yearsEmployed() { ... }
}

public interface Bonus {}
public class CheapBonus implements Bonus { ... }
public class DefaultBonus implements Bonus { ... }
public class MediumBonus implements Bonus { ... }
public class ExpensiveBonus implements Bonus { ... }
```

Koden som skapar bonusobjekten med utgångspunkt i anställningsår är starkt beroende av specifika typnamn. Detta placeras lämpligen i en objektfabrik.

```
public class BonusFactory {
    public static Bonus createBonus(int years) {
        if ( years > 30 )
            return new ExpensiveBonus();
        else if ( years > 20 )
            return new MediumBonus();
        else if ( years > 10 )
            return new CheapBonus();
        else
            return new DefaultBonus();
    }
}
```

Metoden `giveBonus` kan nu skrivas om på ett typsäkert sätt genom att tillämpa polymorfism:

```
public static void giveBonus(ArrayList<Employee> l) {
    for ( Employee e : l )
        e.receive(BonusFactory.createBonus(e.yearsEmployed()));
}
```

Uppgift 4 (8 p)

```
public abstract class AbstractDieDecorator implements Die {
    private Die die;
    public AbstractDieDecorator(Die die) {
        this.die = die;
    }
    @Override
    public int getValue() {
        return die.getValue();
    }
    @Override
    public void roll() {
        die.roll();
    }
}

public class HistoryDie extends AbstractDieDecorator {
    private LinkedList<Integer> history;
    private final int capacity;

    public HistoryDie(Die die,int capacity) {
        super(die);
        this.capacity = capacity;
        history = new LinkedList<>();
    }
    @Override
    public void roll() {
        super.roll();
        if ( capacity > 0 ) {
            if ( history.size() == capacity )
                history.removeLast();
            history.addFirst(getValue());
        }
    }
    public int available() {
        return history.size();
    }
    public int lookBack(int n) throws IndexOutOfBoundsException {
        if ( n < 0 || n >= available() )
            throw new IndexOutOfBoundsException("HistoryDie: index out of
range");
        return history.get(n);
    }
}
```

Uppgift 5 (8 p)

I gruppklassens `add`-metod adderas *muterbara* rektangelobjekt till gruppens interna lista. Inget hindrar att sådana objekt delas av flera gruppobjekt. När `resize` anropas för ett gruppobjekt kan därför ett annat muteras implicit genom att ett delat rektangelobjekt muteras av `resize`. Gruppklassens klassinvariant håller alltså inte.

```
Group g = new Group();
Rectangle r = new Rectangle(10,20);
g.add(r);
System.out.println("Total area of g: " + g.getArea());
Group g2 = new Group();
g2.add(r);  // r delas nu av både g och g2
g2.resize(2); // Invarianten bryts
System.out.println("Total area of g: " + g.getArea());
```

En lösning på problemet är att göra rektangelklassen kopierbar och låta gruppklassen addera kopior av rektangelobjekt.

```
public class Rectangle implements Cloneable {
    ...
    public Rectangle clone() {
        try {
            return (Rectangle)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
public class Group {
    ...
    public void add(Rectangle r) {
        checkInvariant();
        allRectangles.add(r.clone());
        totalArea += r.getArea();
        checkInvariant();
    }
    ...
}
```

Uppgift 6 (3+4 p)

a)
Metoden `client` har en parameter av typen `Int` och kan anropas med vilket implementerande objekt som helst förutsatt att dess `service`-metod har minst lika stark specifikation som i `Int`. Den enda klassen som uppfyller detta är `Impl4`. I `Impl1` är specifikationen svagare och i de två övriga ojämförbara.

b)
I en klass som implementerar `Int` måste `func` i subklassen:

- vara minst lika synlig som i `Int`
- ha en returtyp som är en subtyp till `B` (kovarians)
- ha samma parametertyp
- kasta kompatibla undantag: inga undantag, `E2`, `E3` eller `E2` och `E3`

Detta uppfylls av fallen a,b,e, och f.

Uppgift 7 (8 p)

Variabeln `p` har *statisk* typ `A`. Den *dynamiska* typen går från `A` till `B` och slutligen till `C`. Variabelns statiska typ bestämmer vilken *klassmetod* som anropas, dess dynamiska typ vilken *instansmetod* som anropas. Metoden `h` *överlagras* i `A`. Vid överlagring matchas anropet mot den metod vars parameterprofil överensstämmer bäst med typerna i anropet. Därefter anropas den valda metoden eller en överskuggning av den. Utskrifterna blir:

```
A p = new A();                // Utskrift
p.f();                        // A.f      trivialt
p.g();                        // A.g      trivialt
p = new B();
p.f();                        // B.f      f är en instansmetod omdef. i B
p.g();                        // A.g      g är en klassmetod
p.h("apa");                   // A.h(String) apa      1)
p.h(123);                     // B.h(Object) 123      2)
p.i("bepa");                  // A.i(Object)bepa      3)
p = new C(42);
print(p.equals(new C(42)));   // false                4)
```

1) `h` överlagras i `A`. Anropet matchar `A.h(String)` vilken anropas.
2) Anropet matchar `A.h(Object)` som överskuggas av `B.h(Object)` vilken anropas.
3) Anropet matchar `A.i(Object)` vilken anropas eftersom den inte överskuggas i B.
4) Anropet matchar `Object.equals(Object)` vilken anropas. Den överskuggas inte i `C`.
Anropet `p.equals(new C(42))` returnerar förstås `false` eftersom `Object.equals` baseras på
referenslikhet.

Uppgift 8 (8 p)

```
public static void main(String[] arg) {
    try {
        copyBinaryToText(arg[0]);
    }
    catch ( FileNotFoundException e ) {
        System.out.println("Cannot open " + e.getMessage());
    }
    catch ( IOException e ) {
        e.printStackTrace();
    }
}

private static void copyBinaryToText(String inFileName)
throws IOException
{
    DataInputStream in = openBinaryInFile(inFileName);
    PrintWriter out = createTextOutFile(inFileName + ".txt");
    copyFile(in,out);
    in.close();
    out.close();
}

private static DataInputStream openBinaryInFile(String fileName)
throws IOException
{
    return new DataInputStream(new FileInputStream(fileName));
}

private static PrintWriter createTextOutFile(String fileName)
```

```
throws IOException
{
    return new PrintWriter(new FileWriter(fileName));
}

private static void copyFile(DataInputStream in,PrintWriter out)
throws IOException
{
    while ( in.available() > 0 ) {
        long value = in.readLong();
        out.println("" + value);
    }
}
```