TDA550 – Objektorienterad programvaruutveckling IT, forts. kurs
Övning vecka 6

Pelle Evensen & Daniel Wetterbro

16 oktober 2012

Sammanfattning

Denna vecka ska vi titta på Trådar, inner-/anonyma klasser samt I/O.
Övningarna är graderade (något subjektivt) från lätt (*) till svår (***). Svårighetsgraden hos en övning har inte nödvändigtvis med lösningens storlek att göra.

1 Trådar

1.1 Runnable och Thread *

Skriv klart klasserna **ShoutTest**, **ShoutRunnable** och **ShoutThread** (fig. 1 – 3). Låt varje aktör vila en sekund mellan sina skrik.

Vilka fördelar respektive nackdelar finns det med **Thread** kontra **Runnable**?

```
public class ShoutTest {
    public static void main(String[] args) {
        Thread pelle = new ShoutThread("Pelle", "Daniel");
        Runnable daniel = new ShoutRunnable("Daniel", "Pelle");

        // Add your own code to make Pelle start shouting at
        // Daniel and vice versa.
    }
}
```

Figur 1: Klassen **ShoutTest**.

```
public class ShoutRunnable implements
        Runnable {
    private final String from;
    private final String to;

    public ShoutRunnable(String from, String to) {
        this.from = from;
        this.to = to;
    }

    private void shout() {
        System.out.println(from +
            " shouts: Done with " +
            " the exercises yet, " + to +
            "???");
    }

    // Add necessary methods here.
}
```

Figur 2: Klassen **ShoutRunnable**.

```
public class ShoutThread extends Thread {
    private final ShoutRunnable g;

    public ShoutThread(String from, String to) {
        g = new ShoutRunnable(from, to);
    }

    // Add necessary methods here.
}
```

Figur 3: Klassen **ShoutThread**.

1.2 Time of check/Time of use (TOCTOU) *

Ett ganska vanligt fel då man handskas med data som kan förändras av andra trådar eller processer är att man utgår från att data fortfarande har samma värde då man använder dem som när man senast läste av dem.

Detta fel är allmänt känt som ”Time of check/time of use”. Problemet kan alltså uppstå då det finns en möjlig tidsavvikelse mellan ”check” och ”use”.

Kan problemet uppstå för icke-muterbara klasser?

1.2.1 TOCTOU I ***

Med texten ovan som ledning, förklara hur vi kan bryta mot klassinvarianten i **Period** (fig. 4). Denna klass såg vi i vecka tre. De problem vi hade då löste vi med hjälp av defensiv in- och utkopiering. Klassen är efter denna åtgärd dock fortfarande inte trådsäker.

Hur kan vi lösa problemet (garantera att klassinvarianten alltid håller)?

```
/**
 * @invariant getStart() is before or at the same time as getEnd()
 */
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start the beginning of the period
     * @param end the end of the period; must not precede start
     * @pre start.compareTo(end) <= 0
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0) {
            throw new IllegalArgumentException(start + " after " + end);
        }
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
    }

    public Date getStart() {
        return new Date(start.getTime());
    }

    public Date getEnd() {
        return new Date(end.getTime());
    }
}
```

Figur 4: Klassen **Period**.

1.3 Deadlock ***

Betrakta klassen **Person**[1] (fig. 5). Här sitter Emil och Pelle och tänker. Det är dock inte helt lyckat att de kan störa varandra när som helst.

- Gör så att en person inte får bli störd medans den tänker. Det vill säga, om en annan tråd anropar `getAQuestionFrom` så får den andra tråden vänta tills den första tråden låtit personen tänka klart.
- Om man synkroniserar alla metoder, blir programmet trådsäkert? Antag t.ex. att man låter både `getAQuestion` och `getAnAnswer` vara `synchronized`.

```
public class Person extends Thread {
    private String name;
    private Person whomToAsk;

    public Person(String name) { this.name = name; }

    public void setWhomToAsk(Person whomToAsk) {
        this.whomToAsk = whomToAsk;
    }

    public String toString() { return name; }

    private void getAQuestionFrom(Person p) {
        try {
            System.out.println(this + " says: I got a question from " + p);
            // Think a while before answering...
            Thread.sleep(50);
            p.getAnAnswerFrom(this);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void getAnAnswerFrom(Person p) {
        System.out.println(this + " says: I got an answer from " + p);
    }

    public void run() { whomToAsk.getAQuestionFrom(this); }

    public static void main(String[] args) {
        Person p1 = new Person("Pelle");
        Person p2 = new Person("Emil");
        p2.setWhomToAsk(p1); p1.setWhomToAsk(p2);
        p1.start(); p2.start();
    }
}
```

Figur 5: Klassen **Person**.

[1] Lånad av Bror Bjerner från tentamen april 2009.

1.4 Bonusuppgift: Ofullständig klassinvariant & race conditions ***

Ett ”race condition” är en defekt i ett program vilket gör att programmet bara fungerar korrekt när saker sker i en viss (oväntat välbestämd) ordning.

Klassen **LightColor** (fig. 6, sid. 6) vill vi kunna använda för att garantera att vi bara kan få tag på ”ljusa” **Color**-objekt. Kriteriet för ”ljus” är att medelvärdet av intensitetsvärdena för röd, grön & blå är över 128.

Följande deluppgifter visar på varför denna specifikation är otillräcklig;

1. Det saknas en klassinvariant om man vill kunna garantera att `asColor` alltid returnerar ett giltigt **Color**-objekt. Fundera ut hur en lämplig komplettering av den befintliga invarianten skulle kunna se ut och lägg till eventuell kod i `checkInvariant`.

   Ledtråd: Fundera ut vad lämpligt *pre* för metoden `asColor` skulle kunna vara. Bör metoden under några omständigheter kasta exception?

2. Vad bör *pre* och *post* vara för metoden `setColor`? Använd följande notation: **x** är parametern `x` och **after.x** är instans-/klassvariabeln `x`:s tillstånd då metoden returnerar.

3. Visa varför man fortfarande inte är garanterad att `asColor` returnerar en ”ljus” färg, om än ett giltigt **Color**-objekt. Tänk på att flera trådar kan dela på samma **LightColor**-objekt. *Det finns alltså ett race condition i koden.*

4. Lägg till lämplig kod som gör att båda invarianterna håller samt garanterar att man alltid får tillbaka en ljus färg av `asColor`.

```
/**
 * @invariant r + g + b > THRESHOLD
 * @invariant Think of one more reasonable invariant...
 */
public class LightColor {
    public static final int THRESHOLD = 384;
    private int r, g, b;

    /**
     * @throws IllegalArgumentException if the colour specified by r, g, b is too dark.
     */
    public LightColor(int r, int g, int b) {
        setColor(r, g, b);
    }

    /**
     * @pre What should the precondition(s) be?
     * @post What should the postcondition(s) be?
     * @throws IllegalArgumentException if the colour specified by r, g, b is too dark.
     */
    public void setColor(int r, int g, int b) {
        if(checkInvariant(r, g, b)) {
            this.r = r;
            this.g = g;
            this.b = b;
        } else {
            throw new IllegalArgumentException(
                    "The sum of r, g, b must be greater than " + THRESHOLD);
        }
    }

    // See setColor(int, int, int)
    public void setColor(Color c) {
        setColor(c.getRed(), c.getGreen(), c.getBlue());
    }

    /**
     * Returns a Color-view of this object.
     * @pre What should the precondition(s) be?
     * @throws Can this method throw?
     */
    public Color asColor() {
        return new Color(r, g, b);
    }

    // Make sure that the the rgb triple will satisfy the invariant.
    private boolean checkInvariant(int r, int g, int b) {
        return r + g + b > THRESHOLD;
    }
}
```

Figur 6: Klassen **LightColor**.

2 Innerklasser och anonyma klasser

2.1 Det kan vara fint med få klasser. . . *

Vissa gränssnitt i Java är mycket små. I detta fall ska vi använda **Comparator** för att visa hur vi kan undvika att skapa en namngiven klass för varje beteende vi vill ha.

Fullfölj implementationen av klassen **SortingFactory** (fig. 7) med hjälp av anonyma klasser. Varför är skalimplementationerna för `numberOfCharactersComparator` och `stringPrefixComparator` olika?

```
public class SortingFactory {
    private static final Comparator<String> numberOfCharactersComparatorInstance =
        new Comparator<String>() {
        // Your implementation here.
        };

    /**
     * Returns a comparator for strings, ordering in ascending order of
     * string length.
     */
    public static Comparator<String> numberOfCharactersComparator() {
        return numberOfCharactersComparatorInstance;
    }

    /**
     * Returns a comparator for strings, ordering in ascending order
     * but ignores every character past prefixLength.
     *
     * With prefixLength set to 2, "Pelle" and "Peter" should compare
     * as equal but with prefixLength set to 3, "Pelle" would come before "Peter".
     *
     * @param prefixLength The maximum number of characters to use
     *                     when comparing.
     */
    public static Comparator<String> stringPrefixComparator(final int prefixLength) {
        // Create and return a new instance of an anonymous class here.
    }
}
```

Figur 7: Klassen **SortingFactory**.

2.2 Statiska och icke-statiska Innerklasser **

En innerklass (i detta fall egentligen nästlad statisk/icke-statisk klass) kan vara lämplig att använda då vi vill samla ihop data och få ett konsistent gränssnitt utan att för den skulle tvingas generalisera/exponera utåt.

Med en privat innerklass kan vi vid ett senare tillfälle byta ut implementationen utan att påverka någon annan klass än den omgivande klassen.

Fullfölj implementationen av klassen **GameKeeper** (fig. 8). Ska klassen **Game** vara en *statisk* innerklass eller inte?

Item 22 i [Blo08] har ett mycket bra och ingående resonemang om de fyra olika typerna av nästlade klasser som finns i Java.

```
public class GameKeeper {
    private static Map<String, Game> games = new HashMap<String, Game>();

    public static void registerGame(String name, GameModel model,
                    GameView view) {
        games.put(name, new Game(model, view));
    }

    public static GameView getGameView(String name) {
        return games.get(name).view;
    }

    public static GameView getGameModel(String name) {
        return games.get(name).model;
    }

    // Implement the inner class Game here.
}
```

Figur 8: Klassen **GameKeeper**.

3 I/O

3.1 Heltalsförflyttning *

I den här uppgiften ska heltal (int) läsas från tangentbordet och lagras i en [konkret implementation av] **List<T>**. Därefter ska alla heltal i listan skrivas till dels en binär fil och dels en textfil.

I den binära filen kan talen lagras efter varandra eftersom en int alltid upptar fyra byte. I textfilen avskiljs varje tal genom att ligga på en egen rad. De båda filnamnen ska ges som argument till programmet på kommandoraden. Programmet kan t.ex. köras genom att skriva:

```
java IntegerTransfer intsbin ints.txt
```

Fullfölj implementationen av klassen **IntegerTransfer** (fig. 9) genom att ersätta alla förekomster av "..." med din egen lösning.

```
public class IntegerTransfer {
    private static final int ERROR_ARGS = 1;
    private static final int ERROR_FILES = 2;

    public static void readIntegers(...) throws Exception {
        ...
    }

    public static void writeIntegers(...) throws Exception {
        ...
    }

    public static void main(String[] args) {
    // Check number of arguments
    if (args.length < 2) {
        System.out.println("Too few arguments"
                + "\nUsage IntegerTransfer <binary output file> "
                + "\n<text output file>");
        System.exit(ERROR_ARGS);
    }

    // Create storage
    ...

    // Read and write numbers
        try {
            readIntegers(numStorage);
            writeIntegers(numStorage, args[0], args[1]);
        } catch (Exception e) {
            System.err.println(e.getCause().getMessage());
            System.exit(ERROR_FILES);
        }
    }
}
```

Figur 9: Klassen **IntegerTransfer**.

Referenser

[Blo08] Joshua Bloch. *Effective Java.* Addison-Wesley, 2nd edition, 2008.