Videregående programmering i Java

Dag 10 - Flertrådet programmering

Fremlæggelse af programmering/status for projekter
Dokumentation med javadoc
Flertrådede designmønstre: Arbejdstråd, Producent-Konsument, Trådpulje
Evt.: Kommunikation mellem processer og tråde i et GUI-program
Evt.: Synkronisering af processer og tråde, semaforer [TJT] om tråde
Evt.: Monitorer, Java synkronisering, Deadlocks

Sidste gang - spørgsmål?

Dag 9 - Andre designmønstre

Andre designmønstre: Uforanderlig, Fluevægt, Lagdelt Initialisering, Komposit/Rekursiv Komposition, Kommando/Ændring
Fremlæggelse af programmering/status for projekter
Evt.: Brug af Properties-filer (og præferencer i JDK 1.4)
Evt: Kommunikation med udklipsholder og Drag'n'drop

Læsning: VP 18, How to Use Drag and Drop and Data Transfer.

Krav til projektet

Hvordan går det med projektet?

Javadoc

```
/**
 * Eksempel på en kommenteret klasse.
 */
public class EnKommenteretKlasse
{
    /**
     * Et eksempel på en metode. Metoden tjener
     * til at vise hvordan javadoc virker.
     *
     * @param enStreng strengen
     * @param etTal tallet
     *
     * @return strengen og tallet sat sammen
     */
    public String enMetode(String enStreng,
                           int etTal)
    {
       return enStreng+etTal;
    }
}
```

javadoc EnKommenteretKlasse.java

De vigtigste klasser i jeres projekt skal være dokumenteret med Javadoc!

Class EnKommenteretKlasse

```
java.lang.Object
  |
  +--EnKommenteretKlasse
```

public class **EnKommenteretKlasse**
extends java.lang.Object

Eksempel på en kommenteret klasse.

Constructor Summary

EnKommenteretKlasse()

Method Summary

java.lang.String	**enMetode**(java.lang.String enStreng, int etTal) Et eksempel på en metode.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

EnKommenteretKlasse

```
public EnKommenteretKlasse()
```

Method Detail

enMetode

```
public java.lang.String enMetode(java.lang.String enStreng,
                                 int etTal)
```

Et eksempel på en metode. Metoden tjener til at vise hvordan javadoc virker.

Parameters:

enStreng - strengen
etTal - tallet

Returns:

strengen og tallet sat sammen

Afprøvning (test)

- Vigtigt hvis man ønsker programmer af høj kvalitet
- Formål: At finde (logiske) fejl
  - Psykologisk modstrid
- Jo tidligere fejl findes, desto lettere (og dermed billigere) er det at finde dem.
- Afprøvning bør foregå på alle stadier af udviklingsprocessen.

- Unit testing:
  - Hver enhed (klasse) afprøves uafhængigt før enhederne integreres i hele systemet.
- Integrations- og systemtest:
  - Afprøve delene og hele systemet samlet.
- Accepttest/slutbrugertest:
  - Sikrer at sytemet opfører sig som forventet af brugeren.

Afprøvningsstrategier

- Blackbox testing
  - Afprøvning af delens grænseflade til resten af systemet.
- Whitebox testing
  - Afprøvning sker i forhold til at man ved hvordan delens er implementeret (den indre logik og kodestruktur).

Blackbox Testing

- Ækvivalensinddeling:
  - Mængden af det mulige input inddeles i ækvivalensklasser.
  - Grænseværdier bør afprøves.
  - Mængden af det mulige input bør indeholde både gyldigt og ugyldige input.
- Positiv afprøvning:
  - Testtilfælde som forventes at gå godt.
- Negativ afprøvning:
  - Testtilfælde som forventes at gå galt.

- JUnit
  - Et system til Unit testing (Blackbox-afprøvning af klasser).
- Regressionstest
  - Tidligere test som er gået godt bør gentages hver gang der er foretaget ændringer i klassen.

JUnit i praksis

- De fleste udviklingsværktøjer har JUnit indbygget
- JBuilder
  - demo
- BlueJ
  - Kan interaktivt ”optage” afprøvningseksempler
  - demo

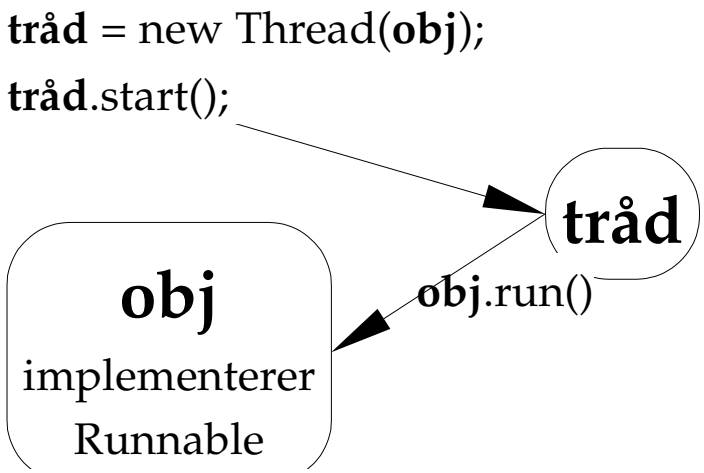
Flertrådet programmering - resumé

```
public class BenytSnakkesagligePersoner
{
    public static void main(String[] arg)
    {
       SnakkesagligPerson p = new SnakkesagligPerson();

       // Opret ny tråd, klar til at udføre p.run()
       Thread t = new Thread(p);

       t.start(); // Nu starter personen med at snakke

       // Det kan også skrives meget kompakt
       new Thread(new SnakkesagligPerson()).start();
    }
}
```



- Samtidighedsproblemet
  - Hvis 2 tråde kommunikerer
    - Fejl, der er svære at finde
  - Løsninger
    - Semaforer
      - Synchronized (obj)
        - Dyr i CPU-tid!!
      - obj.wait() og obj.notify()
    - Flertrådede designmønstre
    - Med JDK1.5: java.util.concurrent

```
public class SnakkesagligPerson implements Runnable
{
    public void run()
    {
       for (int i=0; i<5; i++)
       {
          System.out.println("Bla bla bla "+i);
          try { Thread.sleep(ventetid); }
          catch (InterruptedException e) { }
       }
    }
}
```

Designmønstret Producent-Konsument

Problem: En separat tråd skal udføre noget (f.eks. nogle opgaver). Opgaverne kommer løbende (fra andre tråde) og skal udføres i rækkefølge.

Løsning: Lad de andre tråde *producere* opgaver, som lægges i en liste. Den separate tråd *konsumerer* disse opgaver (fjerner dem fra listen og udfører dem)

- Trådene kommunikerer gennem listen
  - her skal bruges en semafor for at løse samtidighedsproblemet
- Konsument-tråden skal vente på opgaver
  - obj.wait() og obj.notify()

Designmønstret Trådpulje

Problem: Mange opgave skal udføres. Det er for resursekrævende at oprette en ny tråd per opgave.

Løsning: Lad en Objektpulje varetage og genbruge trådene. Lad klienter få opgaverne udført ved at sætte dem i kø i trådpuljen.

- Mange forskellige implementationer i java.util.concurrent
  - F.eks. ThreadPoolExecutor
    - http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ThreadPoolE
  - Kun i JDK 1.5

- Eksempel på implementation på de følgende sider
  - Afsnit 16.7.2: Genbrug af tråde

Designmønstret Trådpulje

```
import java.util.*;

public class Traadpulje
{
    private List ledige = new ArrayList(); // Listen over ledige arbejdstråde

    /** Læg en opgave i kø til en arbejdstråd */
    public synchronized void startOpgave(Runnable opgave)
    {
      Arbejder a;

      if (ledige.isEmpty())
      {
        a = new Arbejder(); // ingen arbejdstråde ledige, en ny oprettes
        a.setDaemon(true);  // tillad systemet at lukke ned selvom tråden er aktiv
        System.out.println("Ny arbejdstråd oprettet.");
        a.start();
      } else synchronized(ledige) {
        a = (Arbejder) ledige.remove(ledige.size()-1);// tag arbejdstråd fra liste
      }

      synchronized(a)
      {
        if (a.opgave != null) throw new InternalError("Tråden kører allerede");
        a.opgave = opgave;
        a.notify();          // væk arbejdstråden der venter i wait()
      }
    }
```

Designmønstret Trådpulje

```
    /**
     * Arbejds-tråden.
     * Den er stærkt bundet til puljen så den er lagt som en privat indre klasse.
     */
    private class Arbejder extends Thread
    {
      private Runnable opgave = null;

      public final synchronized void run()
      {
        while (true) try
        {
          if (opgave != null)
          {
            System.out.println(this+" virker nu på "+opgave);
            opgave.run();      // udfør opgaven
            opgave = null;     // ... og glem den (!)
            synchronized(ledige) {ledige.add(this);} // læg tråd tilbage i listen
          }
          System.out.println(this+" venter på opgave.");
          this.wait();         // vent på at blive vækket med notify()
        } catch (Exception e) {
          System.err.println(this+": Fejl opstod i opgave "+opgave);
          e.printStackTrace();
        }
      } // slut på run()
    } // slut på den indre klasse Arbejder
}
```

JDK 1.5: java.util.concurrent

- Pakken java.util.concurrent.atomic
  - AtomicBoolean, AtomicInteger, AtomicReference, ...
  - Udelelige operationer (så man slipper for synchronized), a la
    - int getAndSet(int newValue)
    - int addAndGet(int delta)
    - int getAndDecrement()
    - boolean compareAndSet(int expect, int update)
- Pakken java.util.concurrent.locks
  - ReentrantLock
    - Reentrant (samme tråd kan låse flere gange)
  - ReentrantReadWriteLock
    - Læse- og skrivelås (flere kan godt læse samtidig)
    - Nedgradere fra skrivelås til læselås muligt
- Pakken java.util.concurrent
  - ConcurrentHashMap, CopyOnWriteArrayList, ...
  - Køer til variationer af Producent-Konsument-designmønstret
    - ArrayBlockingQueue, DelayQueue, PriorityBlockingQueue, SynchronousQueue
  - Trådpuljer
    - ThreadPoolExecutor, ScheduledThreadPoolExecutor

Flertrådede grafiske programmer

- Swing og AWT er **ikke** trådsikre!!!
  - Forestil dig at de var: Så skulle hver eneste set-metode være synkroniseret!!
- => hurtigere, mindre kode, nemmere at fejlfinde etc
  - Grafiske brugergrænseflader er sjældent flertrådede
- => *kun* GUI-tråden (Event-Dispatching Thread) må bruges!
  - Kun denne tråd må kalde metoder i og ændre i grafiske komponenter!
    - Dog må revalidate() og repaint() kaldes fra enhver tråd
  - Ofte, *men ikke altid*, går det godt alligevel selvom en anden tråd kalder

Flertrådede grafiske programmer

- Hvordan få et foretaget fra GUI-tråden?
  - Man kan lave et Runnable-objekt og få det udføret fra GUI-tråd
    - SwingUtilities.invokeLater(Runnable r)
    - SwingUtilities.invokeAndWait(Runnable r)
  - Større opgaver/beregninger
    - Brug SwingWorker-klassen
      - Sværere at forstå
      - Ikke del af standardbiblioteket
  - Gentagne hændelser
    - javax.swing.Timer får GUI-tråden til at kalde actionPerformed() på objekt