LÖSNINGSFÖRSLAG - 100406

Uppgift 1.

a) Objekten som tillhör en icke-muterbar klass är oförändliga, dvs de behåller under hela sin livstid det tillstånd som de fick när de skapades. Icke-muterbara objekt har många fördelar:

- De är lätta att förstå.
- De är alltid trådsäkra.
- Det är ofarligt att lämna ut referenser till dem. Det gör inget om en massa olika objekt råkar använda ett och samma immutable object.
- Deras interna data kan återanvändas. Om ett nytt immutable object ska skapas där vissa fält inte skiljer sig från de i ett befintligt objekt av samma klass kan alla oförändrade fält referera till exakt samma instanser som i det befintliga objektet.
- Det är lätt att använda immutable objects som tillstånd i andra objekt.
- Det är ofarligt att skicka dem till andra processer i distribuerade system.

b)

A a = **new** A();	Skriver ut:	A.A()
B b = **new** B();	Skriver ut:	A.A()
		B.B()
A ab = **new** B();	Skriver ut:	A.A()
		B.B()
a.f(ab);	Skriver ut:	A.f(A)
a.g(b);	Skriver ut:	A.g(A)
b.f(ab);	Skriver ut:	B.f(A)
b.g(a);	Skriver ut:	A.g(A)
b.h(ab);	Skriver ut:	B.h(A)
ab.f(a);	Skriver ut:	B.f(A)
ab.h(a);	Ger kompileringsfel, metoden h finns inte klassen A	
ab.f(ab);	Skriver ut:	B.f(A)

Uppgift 2.

a)

i) ABD

ii) ACD

b)

Interfacet har två ansvarsområden och strider mot S*ingle Responsibility Principle* .

```
public interface Contact {
    public void dial(String pno);
    public void hangup();
    public boolean isConnected();
}
public interface Transmit {
    public void send(char[] c),
    public void recive(char[] c);
 }
```

c)

```
import java.util.*;
public class MyDS<T> {
    private List<T> data;
    public MyDS() {
        data = new ArrayList<T>();
    }

    public T first() {
        return  data.get(0);
    }

    public void append(T val) {
        data.add(val);
    }
    // fler metoder som vi inte bryr oss om i denna uppgift
}
```

Uppgift 3.

a) Om man skulle införa ytterligare en klass måste man ändra i metoden doSwitch.

b) Inför ett gränssnitt som klasserna A, B och C implementerar. Alternativt kan man införa en abstrakt klass som klasserna A, B och C utökar. Gränssnitt är att föredra eftersom klasser endast kan ärva från en klass, men kan implementera ett godtyckligt antal interface.

// Inför ett interfac

```
public interface I {
    public void doIt();
}
public class A implements I {
    public void doIt() {
        System.out.println("This is A");
    }
}
public class B implements I {
    public void doIt() {
        System.out.println("This is B");
    }
}
public class C implements I {
    public void doIt() {
        System.out.println("This is C");
    }
}
public static void doSwitch(I obj) {
    obj.doIt();
}
```

// Inför en abstrakt klass

```
public abstract class Abst {
     public abstract void doIt();
}

public class A extends Abst {
    public void doIt() {
        System.out.println("This is A");
    }
}
public class B extends Abst  {
    public void doIt() {
        System.out.println("This is B");
    }
}
public class C extends Abst  {
    public void doIt() {
        System.out.println("This is C");
    }
}
public static void doSwitch(Abst obj) {
    obj.doIt();
}
```

Uppgift 4.

a) Ett problem Kalle kan få är att meddelanden försvinner eftersom set() skulle kunna anropas två gånger i följd av samma tråd. Ett annat problem som kan uppstå är att den läsande tråden stjäl all tillgänglig CPU-tid (busy-wait) vilket kan göra att andra trådar, t.ex. den som anropar set() aldrig får köra.

b)

```
public class Buffer {
   private String message;
   public synchronized void set(String m) {
      while (m != null) {
         try {
            wait();
         } catch(InterruptedException e) {}
      }
      message = m;
      notifyAll();
   }
   public synchronized String get() {
      while (message == null) {
         try {
            wait();
         } catch(InterruptedException e) {}
      }
      String m = message;
      message = null;
      notifyAll();
      return m;
   }
}
```

Uppgift 5.

Template-mönstret:

```
public abstract class ArithmeticExpr implements Expr {
    private Expr expr1, expr2;
    protected ArithmeticExpr(Expr expr1, Expr expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    protected abstract String getOpString();
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("(").append(expr1);
        buffer.append(getOpString());
        buffer.append(expr2).append(")");
        return buffer.toString();
    }
    // omissions
}
public class Add extends ArithmeticExpr {
    public Add(Expr expr1, Expr expr2) {
        super(expr1, expr2);
    }
    protected String getOpString() {
        return "+";
    }
    // omissions
}
public class Mul extends ArithmeticExpr {
    public Mul(Expr expr1, Expr expr2) {
        super(expr1, expr2);
    }
    protected String getOpString() {
        return "*";
    }
    // omissions
}
```

Strategimönstret:

```
public interface OperationStrategy {
    public String oper();
}
public class Add implements OperationStrategy {
    public String oper() {
        return "+";
    }
}
public class Mul implements OperationStrategy {
    public String oper() {
        return "-";
    }
}
public class ArithmeticExpr implements Expr {
    private Expr expr1, expr2;
    private OperationStrategy strategy;
    protected ArithmeticExpr(OperationStrategy strategy, Expr expr1, Expr expr2) {
        this.strategy = strategy;
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("(").append(expr1);
        buffer.append(strategy.oper());
        buffer.append(expr2).append(")");
        return buffer.toString();
    }
    // omissions
}
```

Uppgift 6.

```
import java.util.*;
public class Synonyms {
    private Map<String, ArrayList<String>> ordbank = new HashMap<String, ArrayList<String>>();
    public void add(String word1, String word2) {
        addSynonyms(word1, word2);
        addSynonyms(word2, word1);
    }//add

    private void addSynonyms(String word, String synonym) {
        ArrayList<String> synonymer = ordbank.get(word);
        if (synonymer == null) {
            synonymer = new ArrayList<String>();
            synonymer.add(synonym);
            ordbank.put(word,synonymer);
        }
        else if (!synonymer.contains(synonym)){
            synonymer.add(synonym);
        }
    }//addSynonymers

    public void remove(String word1, String word2) {
        removeSynonyms(word1, word2);
        removeSynonyms(word2, word1);
    }//remove

    private void removeSynonyms(String word, String synonym) {
        ArrayList<String> synonymer = ordbank.get(word);
        if (synonymer != null)
            synonymer.remove(synonym);
    }//removeSynomymers

    public List<String> getSynonyms(String word) {
        return ordbank.get(word);
    }//getSynonyms

    public String toString() {
        return ordbank.size()+" ord med synonymer";
    }//toString
}//Synonyms
```

Uppgift 7.

```
import java.util.*;
public class SortSynonyms implements Comparator<String> {
    public int compare(String s1, String s2) {
        int len1 = s1.length();
        int len2 = s2.length();
        if  (len1 != len2)
            return len1 - len2;
        else
        return s1.compareTo(s2);
    } //compare
}//SortSynonyms

public static void printSynonyms(String word, List<String> synonyms) {
    if (synonyms == null || synonyms.size() == 0)
        System.out.println("Ordet " + word + " finns ej");
    else {
        SortSynonyms sort = new SortSynonyms();
        Collections.sort(synonyms, sort);
        String res = "Synonymer till " + word + ": ";
        for (int i = 0; i < synonyms.size(); i++) {
            res = res + synonyms.get(i);
            if (i < synonyms.size()-1)
                res = res + ", ";
        }
        System.out.println(res);
    }
}//printSynonyms
```