

Tentamen i Objektorienterad modellering och design

Tentamen består av 5 uppgifter om totalt 25 poäng. För godkänt betyg kommer att krävas högst 13 poäng. Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. Onödigt komplicerade lösningar kan ge poängavdrag. UML-diagram skall ritas i enlighet med UML-häftet. Man får förutsätta att det finns standardkonstruerare, som sätter attributens värden, i alla klasser. Sådana behöver ej redovisas. Om det är flera klasser som är helt analoga så räcker det att visa koden för en av dem.

Hjälpmedel: Martin: Agile Software Development
Andersson: UML-syntax
Föreläsningsbilderna F01-06.pdf
Holm: Java snabbreferens
Java snabbreferens och UML-häftet finns att låna hos skrivningsvakten.

1 I en modell av en liten dator finns följande klasser.

```
public class Add implements Instruction {
    private int address1, address2, address3;

    public void execute(Word [] memory) {
        int int1 = memory[address1].getValue();
        int int2 = memory[address2].getValue();
        memory[address3].setValue(int1 + int2);
    }
}

public class Word {
    private int value;
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
    public String toString() {
        return String.valueOf(value);
    }
}
```

Modellen bryter både mot integritetsprincipen och lokalitetsprincipen.

Modifera klassen Word så att den inte avslöjar vilken representation som används och implementerar operationer på rätt plats. Anpassa Add-klassen till den modifierade Word-klassen. Du behöver inte införa något gränssnitt för olika sorters Word-klasser.

2 Följande klass finns i ett program som simulerar händelser.

```
public class Event {
    public final static int ARRIVAL = 0,
        DEPARTURE = 1, MEASUREMENT = 2;
    private int kind, time;
    public void execute(Context context) {
        switch (kind) {
            case ARRIVAL:
                context.arrival(time);
                break;
            case DEPARTURE:
                context.departure(time);
                break;
            default:
                context.measurement(time);
        }
    }
}

public String toString() {
    switch (kind) {
        case ARRIVAL:
            return "ARRIVAL" + " " + time;
        case DEPARTURE:
            return "DEPARTURE" + " " + time;
        default:
            return "MEASUREMENT" + " " + time;
    }
}
```

Programmet bryter mot *Open/Closed*-principen. Man kan inte lägga till fler sorters händelser utan att ändra i metoderna. Gör en bättre design och använd *Template Method*-mönstret för att eliminera duplicerad kod i `toString`-metoden. Lösningen redovisas med Java-kod.

- 3 I följande klass går det inte att förändra formeln för att beräkna avkastningen (**revenue**) utan att kompilera om klassen.

```
public class Account {
    private float balance;
    public float revenue(int days) {
        float interest = 4.0;
        return balance*days*interest/365;
    }
    // other methods omitted
}
```

Gör om designen med användning av *Strategy*-mönstret så att man under exekveringen kan byta algoritm för att beräkna avkastningen. Man får förutsätta att algoritmen bara använder beloppet (**balance**) och antalet dagar (**days**). Metoden **revenue** används av andra klasser som man inte kan ändra på. Lösningen redovisas med en modifierad **Account**-klass, den klass som implementerar algoritmen ovan samt övriga klasser som används i den nya versionen av **Account**.

- 4 I läroboken för fördjupningskursen finns väsentligen följande implementering av ett binärt träd.

```
public class BinaryTree {
    protected static class Node {
        protected Object data;
        protected Node left, right;
        protected Node(Object data) {
            this.data = data;
        }
        protected void printNodes() {
            if (left != null) {
                left.printNodes();
            }
            System.out.println(data);
            if (right != null) {
                right.printNodes();
            }
        }
    }

    protected Node root;

    public BinaryTree() {
        root = null;
    }

    protected BinaryTree(Node root) {
        this.root = root;
    }

    public BinaryTree(Object data,
        BinaryTree left, BinaryTree right) {
        root = new Node(data);
        if (left != null) {
            root.left = left.root;
        } else {
            root.left = null;
        }
        // similar code for right subtree omitted
    }

    public void printNodes() {
        if (root != null) {
            root.printNodes();
        }
    }
}
```

Gör en bättre design och implementering med användande av *Null Object*-mönstret. Lösningen får inte innehålla några villkorsuttryck.

- 5 I följande design av ett kalkylprogram finns ett cirkulärt beroende mellan paketen expr och model. Inför ett gränssnitt så att det cirkulära beroendet elimineras. Lösningen redovisas med den modifierade Java-koden.

```
package model;
import expr.Expr;
public class Sheet {
    private Map<Address, Slot> map = new HashMap<Address, Slot>();
    public double value(Object key) {
        Slot slot = map.get(key);
        // omissions
    }
}

public interface Slot {
    public double value(Sheet sheet);
}

package expr;
import model.Sheet;

class AddressExpr extends Expr {
    private Address address;
    public double value(Sheet sheet) {
        return sheet.value(address);
    }
}
```