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.

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 huvudklass.

```
public class Computer {
    private Memory memory;
    private Program program;
    private ProgramCounter programCounter;
    public void run() {
        programCounter.setCounter(0);
        while (programCounter.getCounter() >= 0) {
            Instruction instruction = program.get(programCounter.getCounter());
            programCounter.increment();
            instruction.execute(memory, programCounter);
        }
    }
}
```

Använd *Strategy*-mönstret för att ge möjlighet att logga de instruktioner som exekveras på olika sätt. Modifiera klassen Computer och lägg till klasser/gränssnitt så att man kan bestämma vilken strategi som skall användas innan man anropar den parameterlösa metoden `run()`. Lösningen ges som Java-kod och skall innehålla två strategier: en som sparar exekverade instruktioner i en lista och en som inte gör någonting.

2 I samma modell finns ett minne som innehåller heltal med 64 bitar och klassen Add som representerar en additionsinstruktion.

```
public class Memory {
  private long memory[];
  public long getValue(int address) {
    return memory[address];
  }
  public void set(int address, long value) {
    memory[address] = value;
  }
  public int size() {
    return memory.length;
  }
}
```

```
public class Add implements Instruction {
  private int  address1, address2, address3;
  public void execute(Memory memory, ProgramCounter programCounter) {
    long value1 = memory.getValue(address1);
    long value2 = memory.getValue(address2);
    memory.set(address3, value1 + value2);
  }
}
```

Gör om designen så att man har ett minne där varje adress svarar mot ett heltal givet av 128 bitar. Den nya designen skall vara anpassad för liknande förändringar. Lösningen redovisas med ett klassdiagram med modifierade och tillfogade klasser och gränssnitt. Diagrammet skall innehålla alla attribut och metoder som behövs för att man skall kunna implementera `execute` i Add-klassen. Metoderna skall uppvisa typerna för parametrar och eventuellt returvärde..

3 I Add-klassen i uppgift 2 finns också

```
public String toString() {
  StringBuilder builder = new StringBuilder();
  builder.append("ADD ");
  builder.append(address1);
  builder.append(" ");
  builder.append(address2);
  builder.append(" ");
  builder.append(address3);
  return builder.toString();
}
```

Det finns en analog Mul-instruktionsklass med operationskoden `MUL`. Använd *Template Method*-mönstret för att eliminera duplicerade attribut och duplicerad kod. Lösningen redovisas med Java-kod.

4 I satslogiken använder man uttryck med variabler och de logiska operatorerna $\wedge$, $\vee$ och $\neg$ som utläses 'och', 'eller' resp. 'icke'. Operatorena $\wedge$ och $\vee$ har två operander och $\neg$ har en operand. Följande är exempel på sådana uttryck: p, $p \wedge q$, $q \vee p$, $\neg p$, $(p \wedge q) \vee r$ och $(p \wedge \neg q) \vee r$. Gör en objektorienterad modell för sådana uttryck. Lösningen redovisas som ett klassdiagram där alla attribut, associationer och generaliseringar syns. Inga metoder behöver anges.

5 Man vill använda ett grafiskt användargränssnitt för att följa vad som händer i minnet under exekveringen av modellen i uppgift 2.

```
public class MemoryView extends JPanel {
  private Memory memory;
  private JLabel label [];

  public MemoryView(Memory memory) {
    this.memory = memory;
    int size = memory.size();
    label = new JLabel[size];
    setLayout(new GridLayout(size / 8 + 1, 8, 2, 2));
    for (int i = 0; i < size; i++) {
      label[i] = new JLabel();
      label[i].setText("     0");
      add(label[i]);
    }
  }
}
```

Använd *Observer*-mönstret för att koppla ihop modell och vy. Om du har löst uppgift 2 så skall du utgå från din lösning och bara uppdatera vyn för den minnescell som modifierats via Add-instruktionen. Annars skall du utgå från den givna Memory-klassen. Lösningen redovisas med Java-kod för de inblandade klasserna. Den skall tydligt visa hur kopplingen mellan vy och modell etableras och aktiveras. Metoden `notifyObsevers(Object)`, som också finns i klassen Observable, får inte användas.

I `java.util` finns

```
public interface Observer {
  public void update(Observable observable, Object object);
}

public class Observable {
  public void addObserver(Observer observer) ...
  protected void setChanged() ...
  public void notifyObservers() ...
}
```