Tentamen i Objektorienterad modellering och design

Tentamen består av 5 uppgifter som vardera ger 5 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 F01-06,14.pdf
Java snabbreferens finns att låna hos skrivningsvakten.

---

1 Följande design bryter mot *open/closed*-principen. Gör en design som ej bryter mot denna princip. För full poäng krävs att det inte finns duplicerade attribut. Klasserna `Memory` och `Address` skall användas på oförändrat sätt.

```
public class Instruction {
  private static final int  COPY = 0, ADD = 1;
  private int opCode;
  private List<Address> addressList;

  public void execute(Memory memory) {
    switch (opCode) {
    case COPY:
      memory.copy(addressList.get(0), addressList.get(1));
      break;
    case ADD:
      memory.add(addressList.get(0), addressList.get(1), addressList.get(2));
    }
  }
}
```

Lösningen presenteras med Java-kod för nya och modifierade klasser och ett klassdiagram för alla inblandade klasser. Konstruerare behöver inte redovisas. Diagrammet skall presentera alla relationer, attribut och metoder som syns i Java-koden.

Uppgift 2 utgör en fortsättning på denna uppgift. Det är tillåtet att presentera en gemensam lösning för båda uppgifterna.

2 I den ursprungliga `Instruction`-klassen i föregående uppgift kan man tillfoga en `toString`-metod:

```
  public String toString() {
    StringBuilder builder = new StringBuilder();
    switch (opCode) {
    case COPY:
      builder.append("COPY");
      break;
```

```
    case ADD:
      builder.append("ADD");
    }
    for (Address address : addressList) {
      builder.append(' ').append(address);
    }
    return builder.toString();
  }
```

Anpassa metoden till den nya designen och använd *Template method*-mönstret för att eliminera duplicerad kod. Lösningen presenteras med Java-kod.

3 Metoden `sort` i följande klass sorterar elementen i den ordning som bestäms av `compareTo` i `String`.

```
public class StringList extends ArrayList<String> {
  public void sort() {
    for (int i = 0; i < size(); i++) {
      for (int j = i + 1; j < size(); j++) {
        if (get(i).compareTo(get(j)) > 0) {
          // omissions
        } else {
          // omissions
        }
      }
    }
  }
}
```

Modifiera klassen så att man kan välja jämförelseoperation med hjälp av *Strategy*-mönstret. Lösningen redovisas med Java-kod för alla klasser och gränssnitt som behövs för att man skall kunna sortera med samma jämförelseoperation som ovan.

4 Konstruera en modell för att representera en Java-klass eller ett Java-gränssnitt. Modellen skall innehålla namnet på klassen/gränssnittet, vad som utvidgas och vilka gränssnitt som implementeras. En klass utvidgar alltid exakt en klass (`Object` om inget anges) och implementerar noll eller flera gränssnitt. Ett gränssnitt utvidgar inget eller ett gränssnitt.

Modellen skall inte innehålla information om modifierare (public, etc.), attribut, metoder, inre klasser eller generiska typer. Lösningen redovisas med Java-kod med attribut och konstruerare och en implementering av `toString` som returnerar en sträng analog med

```
class A extends B implements C, D
```

För full poäng krävs en design som undviker duplicerad kod.

5 I ett kalkylprogram finns det rutor som kan innehålla aritmetiska uttryck. Ett uttryck kan innehålla adresser som refererar till uttryck i andra rutor vilket öppnar för risken att få cirkulära beroenden. Implementera ett objektorienterat sätt som signalerar att ett sådant beroende kommer att inträffa genom att kasta ett `RuntimException`. Lösningen behöver inte hantera felet på något annat sätt, som t ex att återställa det ursprungliga uttrycket. Den påtänkta lösningen kan åstadkommas genom att lägga till sju rader kod.

Alla uttrycksklasser implementerar gränssnittet

```
public interface Expr {
    public double value(Sheet sheet);
}
```

Kalkylarket representeras med klassen

```
public class Sheet {
    private HashMap<Address, Expr> map;
    public void put(Address address, Expr expr) {
        map.put(address, expr);
    }
    public double value(Address address) {
        // returns the value of the expression at the given address.
    }
}
```