Tentamen i Objektorienterad modellering och diskreta strukturer

Tentamen består av 6 uppgifter som vardera ger 5 poäng. För godkänt betyg kommer att krävas högst 16 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: Diskreta strukturer
Andersson: UML–syntax
Föreläsningsbilderna F01-14.pdf
Java snabbreferens och Inferensregler för Naturlig härledning 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 **a.** Låt $F(x, y)$ betyda att x är far till y, $M(x, y)$ att x är mor till y och $E(x, y)$ att x och y är samma person. Konstruera ett predikat som är sant precis då x och y är syskon. Två personer är syskon om de har samma föräldrar men är olika personer.

b. Komplettera följande härledningsträd för $\{\neg p \rightarrow \neg q\} \vdash q \rightarrow p$ och indikera vilken inferensregel som använts i varje nod.

$$\dfrac{\dfrac{\dfrac{\neg p \rightarrow \neg q \qquad [\neg p]}{\qquad\qquad\qquad\qquad [q]}}{\qquad\qquad\qquad}}{\qquad\qquad}$$

6 Följande är en grammatik för en *term*:

```
term     ::= ID | function
function ::= NAME '(' termList ')'
termList ::= term (',' term)*
```

Konstruera en parser för termer. Du behöver bara redovisa metoderna som svarar mot de tre grammatikreglerna. Fel skall indikeras med `RuntimeException`. Parsern skall inte bygga den abstrakta representationen utan bara kontrollera att indatasträngen är syntaktiskt korrekt. Du får förutsätta att det finns två attribut i parser-klassen

```
private int token;
private Scanner scanner;
```

där `token` förutsätts innehålla nästa token-värde och `Scanner` definieras av

```
public interface Scanner {
    public static final int EOF = -1, ID = -2, NAME = -3;
    public int nextToken();
    public String token();
}
```

`nextToken` hämtar nästa token från strängen och returnerar en av de negativa konstanterna om den hittat filslut (EOF), ett variabelnamn (ID) eller ett funktionsnamn (NAME). Annars returneras ASCII-koden för nästa tecken. Aktuellt token returneras av `token()`.

Koden blir enklare om du använder

```
private void accept(int expected) {
     if (token == expected) {
         token = scanner.nextToken();
     } else {
         throw new RuntimeException("expecting: " + expected + " found: "
                 + token);
     }
 }
```