Tentamen i Objektorienterad modellering och diskreta strukturer

Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet. Man får förutsätta att det finns standardkonstruerare i alla klasser. De behöver ej redovisas i lösningar.

Hjälpmedel: Martin: Agile Software Development
Andersson: Diskreta strukturer
Andersson: UML–syntax
Föreläsningsbilderna F01-06.pdf
Holm: Java snabbreferens

---

1 Följande klass finns i ett program som hanterar medlemmar i en golfklubb.

```
public class Member {
    public final static int SENIOR = 1, JUNIOR = 2, PASSIVE = 3;
    private int membership;
    private String name;
    public String payment() {
        switch (membership) {
            case SENIOR:
                return name + "den avgift du ska betala är" + 3000;
            case JUNIOR:
                return name + "den avgift du ska betala är" + 1800;
            case PASSIVE:
                return name + "den avgift du ska betala är" + 500;
            }
        return "";
    }
}
```

Programmet bryter mot Open/Closed-principen, då man inte kan lägga till fler typer av medlemskap utan att ändra i metoden payment.

a. Gör en bättre design enligt Open/Closed-principen. Använd Template Method-mönstret för att eliminera duplicerad kod i metoden payment. (Strategy-mönstret ska inte användas.) Lösningen redovisas med Java-kod.

b. Lösningen enligt (a) gör i detta fall det svårt att byta medlemskap för en existerande medlem. Utgå från grunduppgiften igen och gör en bättre design, men använd denna gång i stället Strategy-mönstret. Eliminera duplicerad kod i payment.

Strategy-mönstret gör det möjligt att ändra typ av medlemskap för en medlem genom att ändra dess strategi. Implementera en metod setMembership(parameter) i Member som sätter medlemskapet till önskad typ av medlemskap (bestäms av parametern till metoden som här anges "parameter"). Lösningen redovisas med Java-kod.

2 Nedanstående är två klasser som, tillsammans med ett huvudprogram, håller reda på ett bankkontos saldo och ett godtyckligt antal alarmnivåer som gör "alert" när saldot understiger deras limitnivå.

```
package accountmanager;
import java.util.*;

public class AccountManager {
        int saldo;
        Set<Alarm> alarms = new HashSet<Alarm>();

        public void deposit(int amount) {
                saldo += amount;
        }
        public void withdraw(int amount) {
                saldo -= amount;
                Iterator<Alarm> iterator = alarms.iterator();
                while (iterator.hasNext()) {
                        Alarm aAlarm = iterator.next();
                        if (saldo < aAlarm.getLimit()) {
                                aAlarm.alert();
                        }
                }
        }

        public void addAlarm (Alarm alarm) {
                alarms.add(alarm);
        }
}
```

```
package accountmanager;
public class Alarm {
        protected int limit;
        @Override
        public int hashCode() {
                return limit;
        }
        public Alarm(int limit) {
                this.limit = limit;
        }
        public int getLimit() {
                return limit;
        }
        public void alert() {
                // omissions
        }
}
```

Designen bryter mot principen enkelt ansvar och lokalitetsprincipen. Korrigera detta med användande av Observer-mönstret. De utelämnade kodavsnitten skall lämnas oförändrade. Lösningen ges som Java-kod och skall visa hur och var följande metoder används.

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

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

3 Ett klientprogram använder sig av en kommunikationskanal för att skicka och ta emot textsträngar.

```
public class Client {
    protected CommunicationChannel myCC = new FastCommunicationChannel();

    private void send(String text) {
        myCC.send(text);
    }
    private String receive() {
        return myCC.receive();
    }
}

public interface CommunicationChannel {
    public void send(String text);
    public String receive();
}

public class FastCommunicationChannel implements CommunicationChannel {

    public void send(String text) {
        // omissions
    }
    public String receive() {
        String text = null;
        // omissions
        return text;
    }
}
```

Utöka programmet så att man kan slå på och av loggning av trafiken utan att påverka eventuella inre tillstånd i `CommunicationChannel`-instansen. Lägg till lämpliga klasser och metoder, samt implementera metoderna `public void startLog(DB db)` och `public void stopLog()` i `Client`.

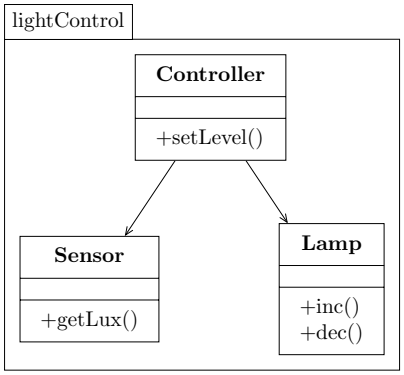
Själva loggningen ska ske på en databas som skickas med som parameter till metoden startLog. Databasen har följande gränssnitt:

```
public interface DB {
    public void appendSent(String text);
    public void appendReceived(String text);
}
```

Du kan utgå från att startLog och stopLog används korrekt, t ex inte anropar stopLog två gånger utan startLog emellan.

Lösningen ska vara baserad på Decorator-mönstret och presenteras i form av Java-kod samt en instansmodell för tidpunkten då loggning är påslagen.

4 Nedan ges ett klassdiagram för ett system som reglerar ljusnivån i ett rum. Logiken ligger i klassen Controller som genom att öka resp. sänka lampans effekt kan hålla rätt ljusnivå. Aktuell ljusnivå i rummet fås genom att anropa sensorns metod getLux().

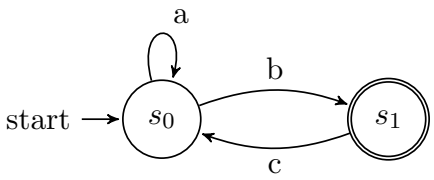


Programmet har en dålig design i det avseende att regulatorn har beroende till den ”lägre” hårdvarunivån bestående av sensorn och lampan. Gör om designen på ett sådant sätt att Controller blir oberoende av dessa klasser. Dela även upp ditt nya system i två paket så att det paket Controller ingår i är oberoende av hårdvarunivån.

Lösningen redovisas i form av ett klassdiagram där alla klasser och interface finns med, inklusive sina metoder. Även paketstrukturen ska framgå.

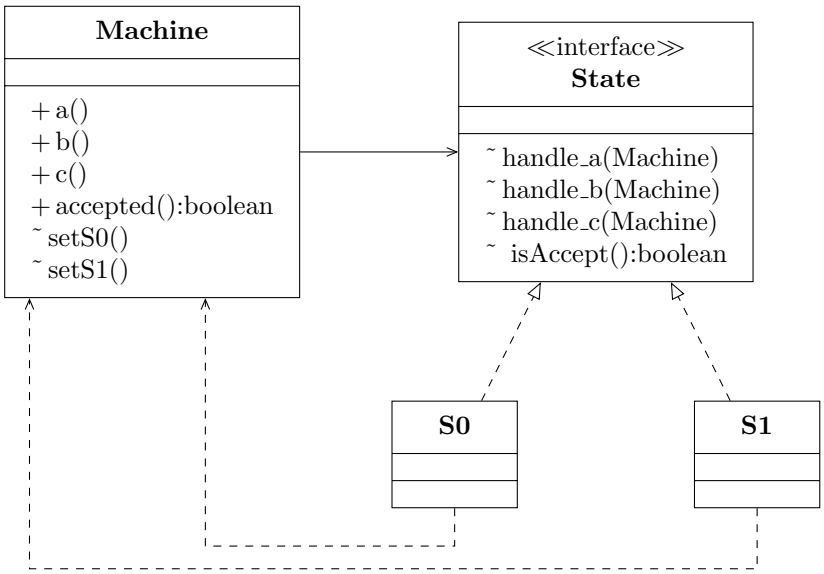
5 State pattern är en utökning av Strategy pattern där strategin kan påverka sin kontext.

Vi vill implementera tillståndsmaskinen



enligt state pattern. Implementationen av maskinen ska vara sådan, att den har en metod `boolean accepted()`, som returnerar `true` om och endast om maskinen är i accept-tillståndet s_1.

En sådan implementation ges i nedanstående klassdiagram



Rita ett sekvensdiagram för ett program som avgör om maskinen accepterar insignalssekvensen `<a, b, c>` (d v s kör nedanstående kod). Antag att maskinen är i initialtillståndet, s_0, vid anropet av `m.a()`.

```
Machine m;
...
m.a();
m.b();
m.c();
boolean r = m.accepted();
```

6 Vi ska nu titta på tillståndsmaskinen i uppgift 5 som språkprocessor

a. Ange ett reguljärt uttryck för språket, $\mathcal{L}(M)$, som maskinen accepterar.

b. Ange alla fyra tecken långa strängar som ingår i $\mathcal{L}(M)$.

7 Visa $\{\neg(P \wedge Q)\} \vdash Q \to \neg P$. Svar får ges som beviskräd eller tabell. Som ledning ges skeletten nedan

$$\dfrac{\neg(P \wedge Q) \qquad \underline{[\text{—}] \quad [\text{—}]}}{\dfrac{}{Q \to \neg P}\,{\to_I}}$$

1	$\neg(P \wedge Q)$	P
2		
3		
4		
5		
6		
7	$Q \to \neg P$	$\to_I$, 2, 6

8 **a.** Ange ett predikatlogiskt uttryck (mängdbyggare) som definierar den den sammansatta relationen

$$(<)\ ;\ (>)$$

på de naturliga talen ($\mathbb{N}$)

b. Låt $(<_{\mathbb{N}_3})$ och $(>_{\mathbb{N}_3})$ vara relationerna *mindre än* och *större än* på mängden $\mathbb{N}_3 \triangleq \{0, 1, 2\}$. Ange vilka element som ingår i den sammansatta relationen

$$(<_{\mathbb{N}_3})\ ;\ (>_{\mathbb{N}_3})$$

9 Givet en grammatik

```
S ::= T ("#" T)*
T ::= V ("@" V)*
V ::= LETTER
```

bygg ett härledningsträd för strängen "`a # b @ c # d`"

Härledningsregler

Regler för $\wedge$

$$\frac{P \qquad Q}{P \wedge Q}\,[\wedge_I] \qquad \frac{P \wedge Q}{P}\,[\wedge_{E_1}] \qquad \frac{P \wedge Q}{Q}\,[\wedge_{E_2}]$$

Regler för $\rightarrow$

$$\frac{\begin{array}{c}[\cancel{P}]\\ \vdots\\ Q\end{array}}{P \rightarrow Q}\,[\rightarrow_I] \qquad \frac{P \qquad P \rightarrow Q}{Q}\,[\rightarrow_E]$$

Regler för $\vee$

$$\frac{P}{P \vee Q}\,[\vee_{I_1}] \qquad \frac{Q}{P \vee Q}\,[\vee_{I_2}] \qquad \frac{P \vee Q \qquad \begin{array}{c}[\cancel{P}]\\ \vdots\\ R\end{array} \qquad \begin{array}{c}[\cancel{Q}]\\ \vdots\\ R\end{array}}{R}\,[\vee_E]$$

Regler för $\neg$

$$\frac{\begin{array}{cc}[\cancel{P}] & [\cancel{P}]\\ \vdots & \vdots\\ R & \neg R\end{array}}{\neg P}\,[\neg_I] \qquad \frac{\neg\neg P}{P}\,[\neg_E]$$

Regler för $\forall$

$$\frac{P}{\forall x\,.\,P}\,[\forall_I] \qquad \frac{\forall x\,.\,P}{P[x\backslash t]}\,[\forall_E]$$

Regler för $\exists$

$$\frac{P[x\backslash t]}{\exists x\,.\,P}\,[\exists_I] \qquad \frac{\exists x\,.\,P \qquad \begin{array}{c}[\cancel{P[x\backslash y]}]\\ \vdots\\ Q\end{array}}{Q}\,[\exists_E]$$

$\begin{array}{c}[\cancel{P}]\\ \vdots\\ Q\end{array}$ i en regel betyder att man har gjort ett (hypotetiskt) antagande P för att härleda Q och att antagandet upphävs (strykes) när man använder regeln. Det hypotetiska antagandet får finnas noll eller flera gånger i beviset av Q.

När man använder $\forall_I$ så får beviset av P inte innehålla några icke upphävda antaganden om x.

I $\exists_E$ får y inte vara en fri variabel i Q eller förekomma i ett icke upphävt antagande i $\begin{array}{c}[\cancel{P[x\backslash y]}]\\ \vdots\\ Q\end{array}$