Tentamen i Objektorienterad modellering och design

Lösningar

.

1.

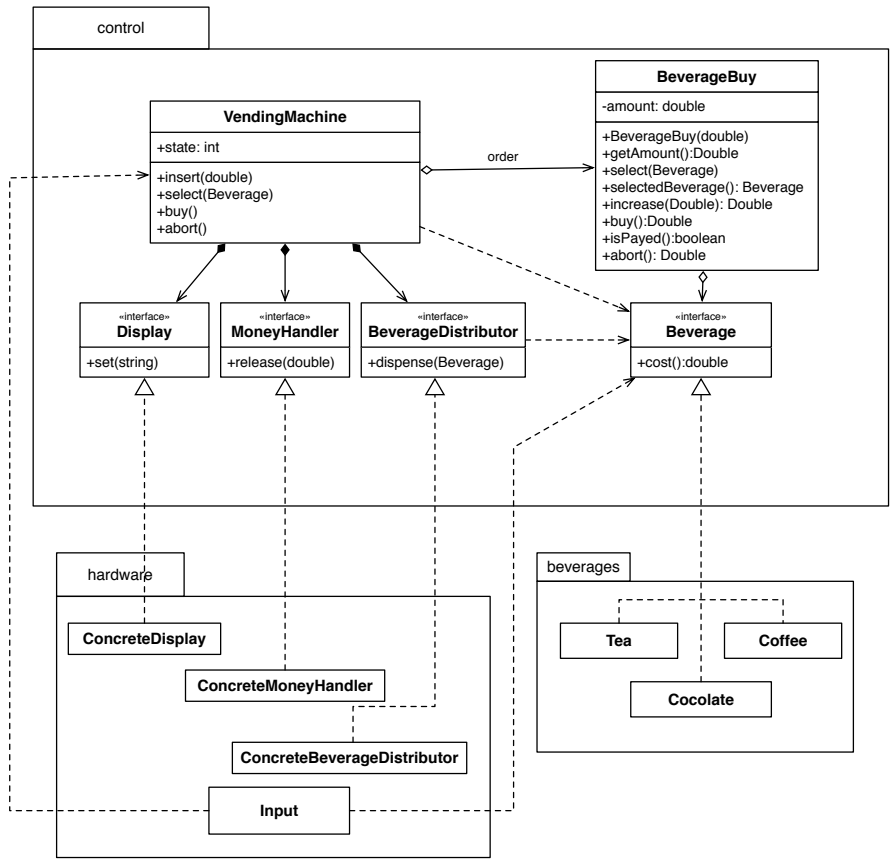
```
public class VendingMachine {

        final static int IDLE = 0;
        final static int PAYING = 1;
        final static int SELECTING = 2;

        private int state = IDLE;

        private BeverageBuy order;
        private Display display = new Display();
        private BeverageDistributor distributor = new BeverageDistributor();
        private MoneyHandler moneyHandler = new MoneyHandler();

        public void insert(double amount){
                switch(state){
                        case(IDLE):{
                                order = new BeverageBuy(amount);
                                state = PAYING;
                                display.set(order.getAmount().toString());
                                break;
                        }
                        case(PAYING):{
                                display.set(order.increase(amount).toString());
                                break;
                        }
                        case(SELECTING):{
                                state = PAYING;
                                display.set(order.increase(amount).toString());
                                break;
                        }
                }
        }

        public void select(Beverage beverage){
                switch(state){
                        case(IDLE):{break;}
                        case(PAYING):{
                                order.select(beverage);
                                state = SELECTING;
                                display.set(order.selectedBeverage().toString());
                                break;
                        }
                        case(SELECTING):{
                                order.select(beverage);
                                state = SELECTING;
                                display.set(order.selectedBeverage().toString());
                                break;
                        }
                }
        }

        public void buy(){
                switch(state){
                        case(IDLE):{break;}
                        case(PAYING):{break;}
                        case(SELECTING):{
                                if(order.isPayed()){
                                        distributor.dispense(order.selectedBeverage());
                                        moneyHandler.release(order.buy());
                                        state = IDLE;
                                        break;
```

```
                                }
                                else{
                                        state = PAYING;
                                        display.set(order.getAmount().toString());
                                }
                                break;
                        }
                }
        }

        public void abort(){
                switch(state){
                        case(IDLE):{break;}
                        case(PAYING):{
                                moneyHandler.release(order.abort());
                                state = IDLE;
                                display.set("please insert coin");
                                break;
                        }
                        case(SELECTING):{
                                moneyHandler.release(order.abort());
                                state = IDLE;
                                display.set("please insert coin");
                                break;
                        }
                }
        }
}
```

2. **a**. se figur 1

b. Metoderna buy() och isPayed() behöver skrivas om.

3.
```
public interface Beverage {
        public double cost();
        public String getDescription();
}

public abstract class BeverageDecorator implements Beverage {
        private Beverage beverage;

        public BeverageDecorator(Beverage bev){
                beverage = bev;
        }

        public double cost() {
                return beverage.cost() + additionalCost();
        }

        public String getDescription() {
                return beverage.getDescription() +" + " + extra() ;
        }

        protected abstract double additionalCost();
        protected abstract String extra();
}

public class Coffee implements Beverage {
        public double cost() {
                return 10;
        }
        public String getDescription() {
                return "coffee";
        }
}

// Klasserna Tea och Chocolate analogt

public class Milk extends BeverageDecorator {
```



Figur 1: Klassdiagram kaffeautomat

```
        public Milk(Beverage beverage){
                super(beverage);
        }

        protected double additionalCost() {
                return 3;
        }

        protected String extra() {
                return "milk";
        }
}

// Klassen Sugar analogt
```

	Påstående	**Anledning**	**Svar A,B,C, D,E**	**Motivering**
T1	Syftet med SRP är att hålla nere storleken på klasserna.	Genom att använda SRP sprider man ut funktionalitet över flera klasser.	E	SRP säger inget om storleken på ett ansvarsområde. Att tillämpa SRP innebär att man samlar funktionalitet kopplat till ett gemensamt ansvar på ett ställe vilket i många fall kan innebära färre och större klasser.
T2	Det enda sättet att följa OCP för en klass är genom att definiera subklasser för den.	Subklasser kan utvidga basklassens beteende utan att man behöver ändra i dess kod.	D	Delegation till ett interface som vid t.ex. strategimönstret är ett annat sätt att uppnå OCP
T3	Brott mot LSP kan leda till bräcklig kod.	LSP innebär att man ser till att man inte förändrar beteendet hos den basklass man utökar.	A	Att koden är bräcklig innebär att det är svårt att göra ändringar utan att introducera nya oväntade fel. Ofta beror det på att underliggande antaganden inte längre är sanna efter att man lagt till nya features. Brott mot LSP är ett sätt att förändra de ursprungliga förutsättningarna.
T4	Aggregering är att föredra framför komposition om objektet vars beteende man tänker använda har ett egenvärde utanför objektet som använder det.	Vid aggregering instansieras det aggregerade objektet oftast i ägarens konstruktor	C	Vid aggregering instansieras det aggregerade objektet oftast utanför det användande objektet.
T5	I observatörsmönstret är observatörer löst kopplade till ett observerbart objekt	Det observerbara objektet vet inte något om observatörerna mer än att de implementerar observatörsinterfacet.	A	Lösa kopplingar handlar om avsaknad av kunskap. Beroende på hur man väljer att implementera observatörerna (med eller utan kunskap om det observerade objektet) kan man även åstadkomma lösa kopplingar i omvänd riktning.