Lunds Tekniska Högskola
Datavetenskap
Ulf Asklund

Tentamen i Objektorienterad modellering och design

Tentamen består av en teoridel om totalt 5 poäng och en problemdel innehållande 3 uppgifter med totalt 21 poäng. 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: UML–syntax
Holm: Java snabbreferens

Teorifrågor

Denna del innehåller uppgifter med påståenden och anledningar. För varje uppgift svara med ett av följande alternativ:

A Både påståendet och anledningen är korrekta uttalanden och anledningen förklarar påståendet på ett korrekt sätt.

B Både påståendet och anledningen är korrekta uttalanden, men anledningen förklarar inte påståendet.

C Påståendet är ett korrekt uttalande, men anledningen är falsk.

D Påståendet är falskt, men anledningen är ett korrekt uttalande.

E Både påståendet och anledningen är falska.

Det går bra att svara direkt i formuläret. **Glöm då inte att lämna in formuläret tillsammans med övriga lösningar.** (5p)

	Påstående	**Anledning**	**Svar A,B,C, D,E**
T1	Syftet med SRP är att hålla nere storleken på klasserna.	Genom att använda SRP sprider man ut funktionalitet över flera klasser.	
T2	Det enda sättet att följa OCP för en klass är genom att definiera subklasser för den.	Subklasser kan utöka basklassens beteende utan att man behöver ändra i dess kod.	
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	
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.	
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.	

Problem

Ett företag vill utveckla mjukvara för att styra en kaffeautomat som finns tillgänglig för studenter och lärare på Campus. Utvecklingsteamet som har tagit sig an uppgiften har påbörjat design och implementering.

Längst bak i tentan finns en bilaga som innehåller:

- En kort beskrivning av funktionaliteten hos den tilltänkta kaffeautomaten
- En initial design i form av ett tillståndsdiagram (Fig 2) och ett klassdiagram (Fig 1)
- Javakod för en av de beskrivna klasserna.

1 Implementera klassen VendingMachine i control-paketet i enlighet med klassdiagrammet och tillståndsdiagrammet i bilagan. VendingMachine delegerar delar av logiken till klassen BeverageBuy som redan är skriven och given i bilagan. Lösningen redovisas med Java-kod.

(7p)

2 Uppdragsgivaren i föregående uppgift vill ha en mer flexibel design och önskar dels öppna upp för möjligheten att lägga till fler drycker och dels för möjligheten byta ut hårdvaran, i båda fall utan att behöva ändra i control-paketet. Ändra designen så att control-paketet inte längre har beroenden till hardware-paketet. Välj också ett lämpligt mönster som löser problemet med dryckerna. Låt Beverage vara en del av detta mönster istället för en enum-typ.

a. Lösningen redovisas med ett fullständigt klassdiagram. För klasserna VendingMachine och BeverageBuy behöver inte metoder och attribut räknas upp. För övriga klasser gäller att all väsentlig information ska finnas med.

b. Namnge och motivera ditt val av mönster.

c. Vilka metoder i klassen BeverageBuy (se java-kod i bilagan) behöver skrivas om med tanke på förändringen av Beverage? Svara med metodnamn.

(6p)

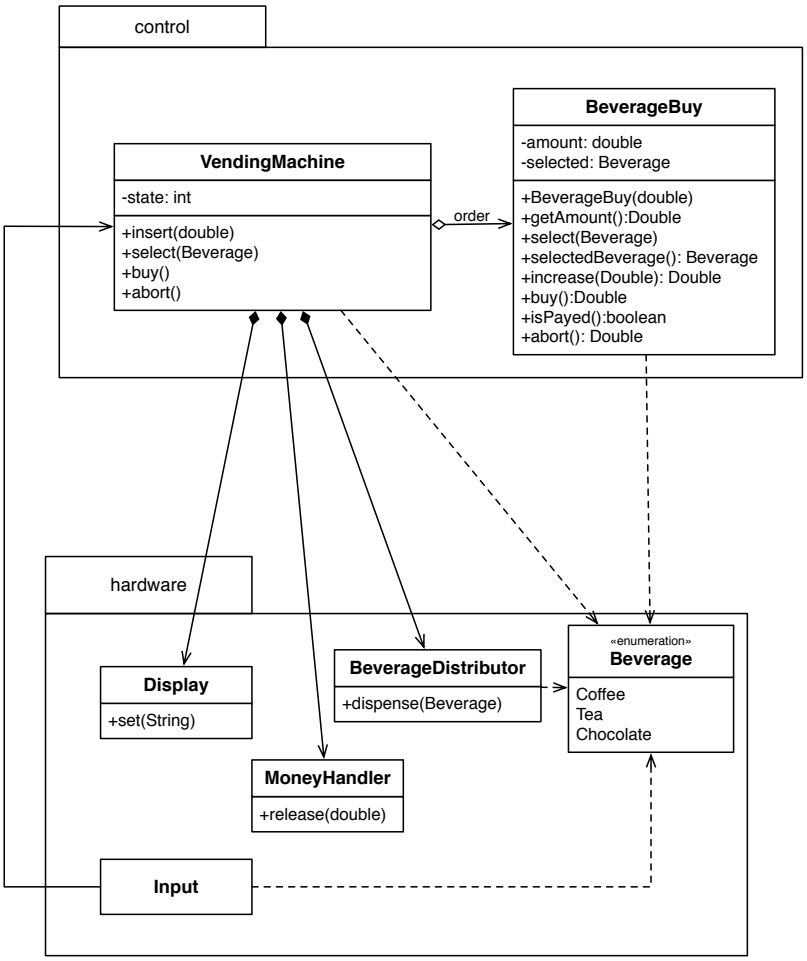
3 Nu börjar utvecklingsteamet tänka vidare kring möjliga utvidgningar och får för sig att erbjuda tillbehör till dryckerna. De skissar på ett exempel där extra socker ökar priset på den valda drycken med 2kr och extra mjölk kostar 3kr. Tillbehören kan inte beställas separat utan endast som tillägg till någon av de tre ursprungliga dryckesvalen (kaffe, te eller choklad).

Två mönster som kan kombineras för att lösa detta är dekoratör (decorator) och mallmetod (template method). Visa hur. Låt klassen *BeverageDecorator* vara en dekoratör av klassen *Beverage* (som då alltså inte kan vara en enum-typ längre). BeverageDecorator innehåller också mallmetoder för de konkreta klasserna Milk och Sugar. Ett objekt av typen Beverage ska kunna tillhandahålla sin totala kostnad *cost()* och en beskrivning av sig själv *getDescription()* (d.v.s. en lista över sitt innehåll). Skriv java-kod för klasserna i den beskrivna lösningen (Beverage, BeverageDecorator, Coffee, Tea, Chocolate, Milk och Sugar). (8p)

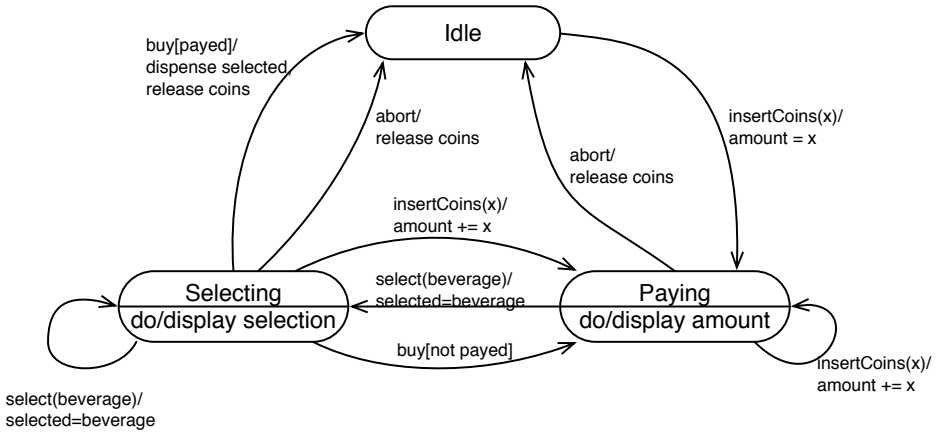
Bilaga - VendingMachine

Spec: I automaten kan man köpa tre olika sorters varma drycker: kaffe, te eller varm choklad. Kaffet kostar 10kr, choklad kostar 8kr och te kostar 5kr. Automaten har 5 knappar (3 för dryckesval, en för att avbryta köpet och en för att verkställa köpet) Utöver dessa insignaler finns ett myntinkast som läser av värdet på instoppade mynt. Ett normalt köp går till som följer: 1) Kunden stoppar pengar i automaten. 2) Värdet av instoppade mynt visas i automatens display. 3) Kunden väljer dryck. 4) Kunden bekräftar köpet. 5) Drycken serveras. 6) Eventuell växel betalas tillbaks. Kunden kan också välja att avbryta köpet och ska då få alla instoppade pengar tillbaka.

Design: Systemet består av två paket: ett paket *control* som innehåller logiken och ett paket *hardware* som utgör ett gränssnitt mot hårdvaran. I controlpaketet ansvarar klassen *VendingMachine* för interaktionen med hårdvaran och delegerar till *BeverageBuy* att hålla reda på status för en påbörjad beställning. Alla insignaler till maskinen (d.v.s. knapptryckningar och myntinkast) hanteras av klassen *Input* i kontrollpaketet som i sin tur anropar de publika metoderna i klassen VendingMachine.



Figur 1: Klassdiagram kaffeautomat



Figur 2: Tillståndsdiagram kaffeautomat

```
class BeverageBuy {
        private double amount;
        private Beverage selected;

        public BeverageBuy(double amount){
                this.amount = amount;
        }

        public Double getAmount(){
                return amount;
        }

        public void select(Beverage bev){
                selected = bev;
        }

        public Beverage selectedBeverage(){
                return selected;
        }

        public Double increase(Double amount){
                this.amount += amount;
                return this.amount;
        }

        public Double buy(){
                switch(selected){
                        case COFFEE:{
                                amount = amount - 10;
                                break;
                        }
                        case CHOCOLATE:{
                                amount = amount - 8;
                                break;
                        }
                        case TEA:{
                                amount = amount - 5;
                                break;
                        }
                }
                selected = null;
                return amount;
        }
```

```
        public boolean isPayed(){
                double price = 0;
                switch(selected){
                        case COFFEE:{
                                price = 10;
                                break;
                        }
                        case CHOCOLATE:{
                                price = 8;
                                break;
                        }
                        case TEA:{
                                price = 5;
                                break;
                        }
                }
                return amount >= price;
        }

        public Double abort(){
                double released = amount;
                amount = 0;
                selected = null;
                return released;
        }
}
```