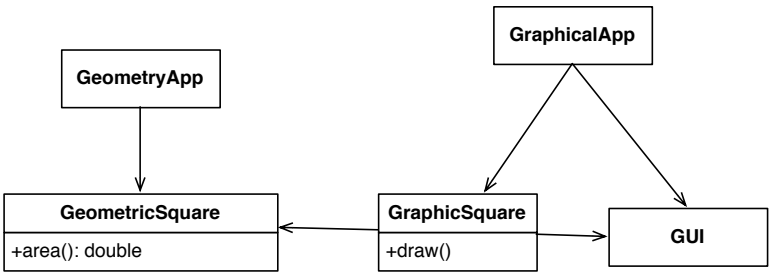
Lösningsförslag till tentamen i EDAF25 Objektorienterad modellering och design – Helsingborg

2015–06–04

1. a) Figure 1



Figur 1: Klasser med separata ansvar

b) Ja, en klass som utifrån användarens perspektiv har flera olika ansvarsområden leder till en bräcklig design. Antag att GeometryApp aldrig behöver metoder för att representera en kvadrat grafiskt men att GraphicalApp har behov av dylika metoder. Om dessa behov vid något tillfälle förändras så att vi vill ändra implementationen i Square men inte tänker på kopplingen till GeometryApp kan det leda till oväntade beteenden i den senare.

2. a) Man skulle kunna argumentera för att vi bryter mot OCP eftersom det inte går att lägga till ytterligare fordon utan att förändra i klassen VehicleTest. Man kan också ana ett brott mot DIP eftersom de beroenden som finns går mot konkreta klasser snarare än mot abstraktioner. Dock är det svårt att utifrån kontexten i detta lilla exempel avgöra om dessa är högnivå eller lågnivå-klasser. Koden innehåller en del duplicerad kod vilket i sin tur innebär brott mot en princip som vi inte gått igenom i kursen nämligen DRY-principen.

b) Template method pattern

```
public abstract class Vehicle {
    private boolean status = false;

    private void start(){
        this.status = true;
    }

    protected abstract void drive();

    protected abstract void stop();

    public final void testVehicle(){
        start();
        if(this.status){
            drive();
            stop();
        }
    }
}

public class Car extends Vehicle {

    protected void drive() {
        System.out.println("I'm driving a car!");

    }

    protected void stop() {
        System.out.println("Car stopped!");

    }
}

public class Truck extends Vehicle {

    protected void drive() {
        System.out.println("I'm driving a truck!");
    }

    protected void stop() {
        System.out.println("Truck stopped!");

    }
}

public class VehicleTest {
    public static void main(String args[]){

        Vehicle car = new Car();
        testVehicle(car);

        Vehicle truck = new Truck();
        testVehicle(truck);
    }
    public static void testVehicle(Vehicle v){
        v.testVehicle();
    }
}
```

c) Strategy pattern

```
public interface DrivingStrategy {
    public void drive();
}

public class CarDrivingStrategy implements DrivingStrategy {

    public void drive() {
        System.out.println("I'm driving a car!");
    }
}

public class TruckDrivingStrategy implements DrivingStrategy {

    public void drive() {
        System.out.println("I'm driving a truck!");
    }

}

public interface StoppingStrategy {
    public void stop();
}

public class TruckStoppingStrategy implements StoppingStrategy {

    public void stop() {
        System.out.println("Truck stopped!");
    }
}

public class CarStoppingStrategy implements StoppingStrategy {

    @Override
    public void stop() {
        System.out.println("Car stopped!");
    }
}

public class Vehicle{
    private boolean status = false;
    private DrivingStrategy drivingstrategy;
    private StoppingStrategy stoppingstrategy;

    public Vehicle(DrivingStrategy ds, StoppingStrategy ss){
        drivingstrategy = ds;
        stoppingstrategy = ss;
    }

    public void setDrivingStrategy(DrivingStrategy ds){
        this.drivingstrategy = ds;
    }

    public void setStoppingStrategy(StoppingStrategy ss){
        this.stoppingstrategy = ss;
    }

    public void start(){
        this.status = true;
    }
```

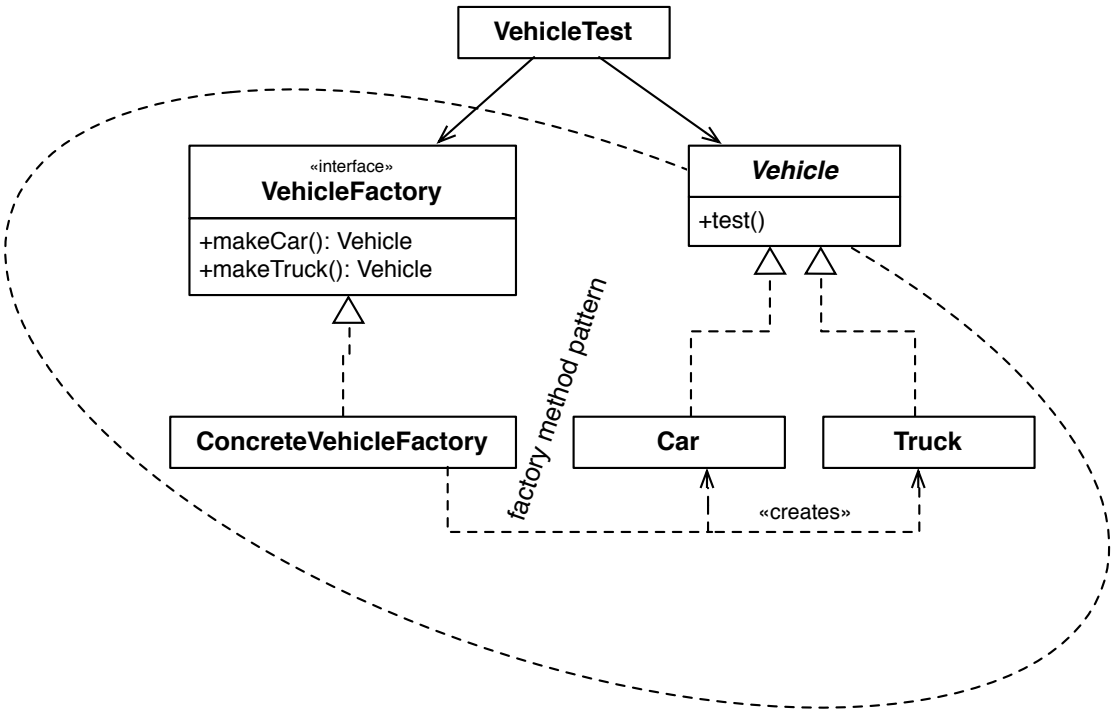
```
    public void drive(){
        drivingstrategy.drive();
    }

    public void stop(){
        stoppingstrategy.stop();
    }

    public void testVehicle(){
        start();
        if(this.status){
            drive();
            stop();
        }
    }
}

public class VehicleTest {
    public static void main(String args[]){
        Vehicle car = new Vehicle(new CarDrivingStrategy(), new CarStoppingStrategy());
        testVehicle(car);
        Vehicle truck = new Vehicle(new TruckDrivingStrategy(), new TruckStoppingStrategy());
        testVehicle(truck);
    }
    public static void testVehicle(Vehicle v){
        v.testVehicle();
    }
}
```

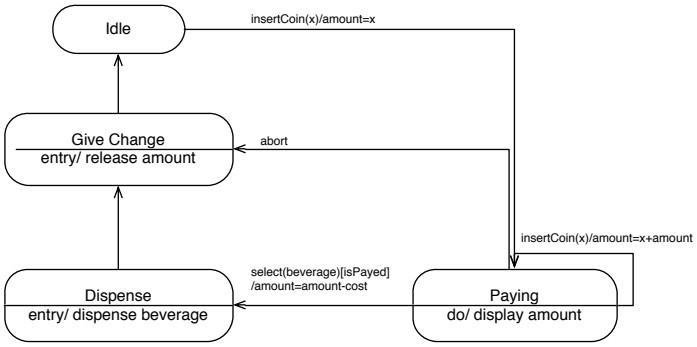
d) Factory



Figur 2: Factory-mönstret

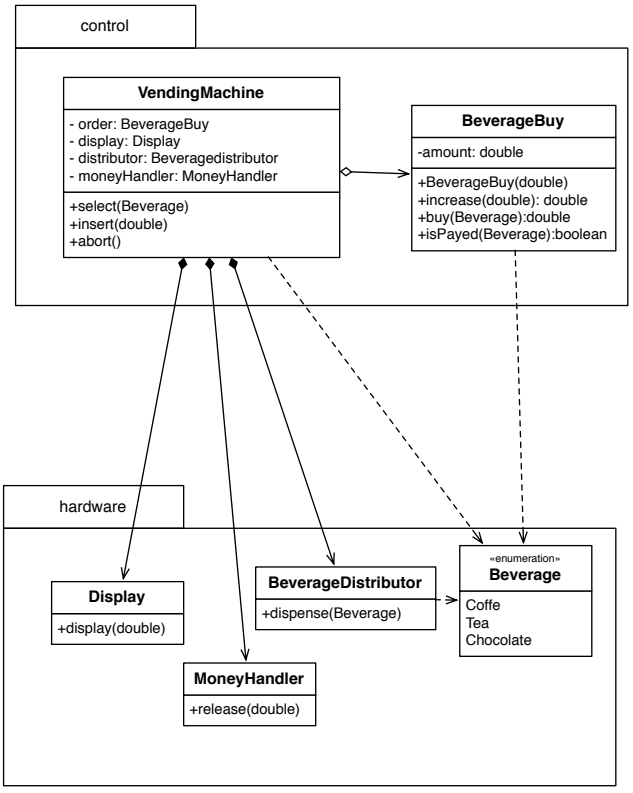
```
public class VehicleTest {
        VehicleFactory factory = new ConcreteVehicleFactory();
        Vehicle car = factory.makeCar();
        testVehicle(car);
        Vehicle truck = factory.makeTruck();
        testVehicle(truck);
    }
    public static void testVehicle(Vehicle v){
        v.test();
    }
}
```

3. a) Figure 3



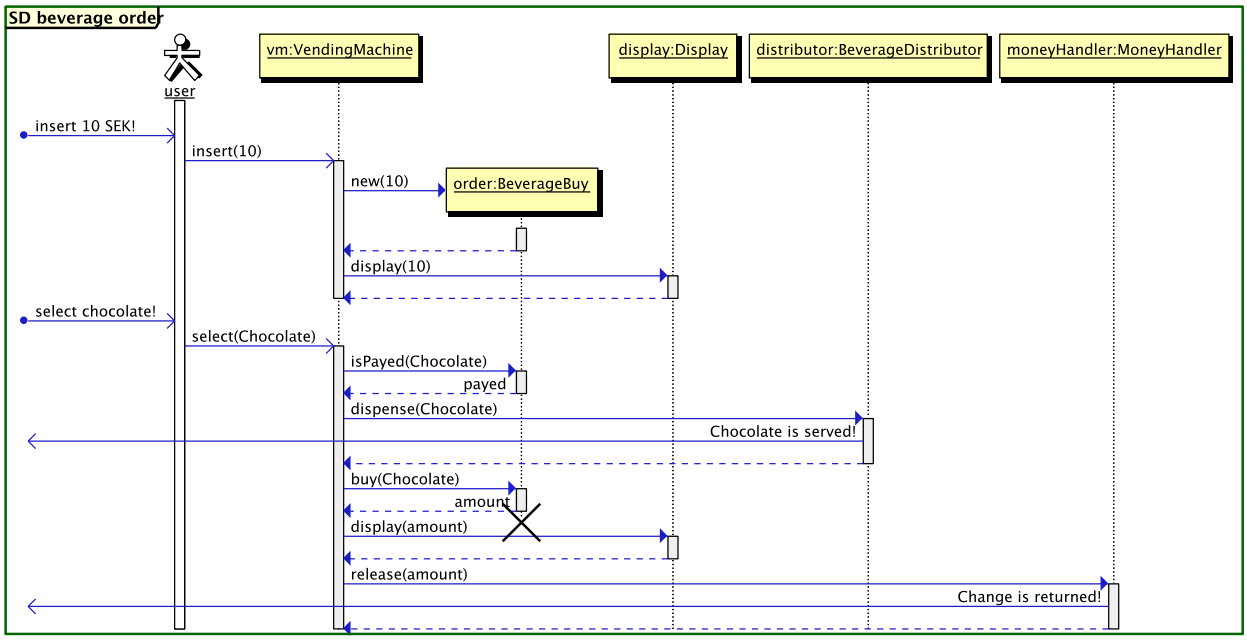
Figur 3: Tillståndsdiagram kaffeautomat

b) Figure 4



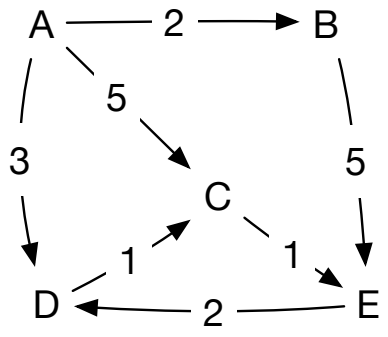
Figur 4: klassdiagram kaffeautomat

c) Figure 5



Figur 5: tillståndsdiagram kaffeautomat

4. Figure 6
   a) DF: A, B, E, D, C



Figur 6: Graf

   b) Efter 5 iterationer är alla 5 noderna markerade som besökta. Den kortaste vägen som identifieras är A,D,C,E och har längden 5.