Lösningar till tentamen i EDAF60

29 oktober 2018

Lösning 1

(a) I tur och ordning har vi:

- *Single Responsibility Principle*: Varje paket/klass/metod skall ha ansvar för en sak, och ha hela ansvaret för den.
- *Open Closed Principle*: Vi vill kunna lägga till ny funktionalitet genom att lägga till nya klasser, utan att behöva ändra gamla klasser.
- *Liskov Substitution Principle*: Kraven på objekt av en subklass får varken vara strängare eller mindre stänga än kraven på superklassen.
- *Interface Segregation Principle*: Skriv inte interfaces som är större än nödvändigt, dela hellre upp i mindre interfaces.
- *Dependency Inversion Principle*: Programmera mot abstrakta interfaces, inte mot konkreta klasser.

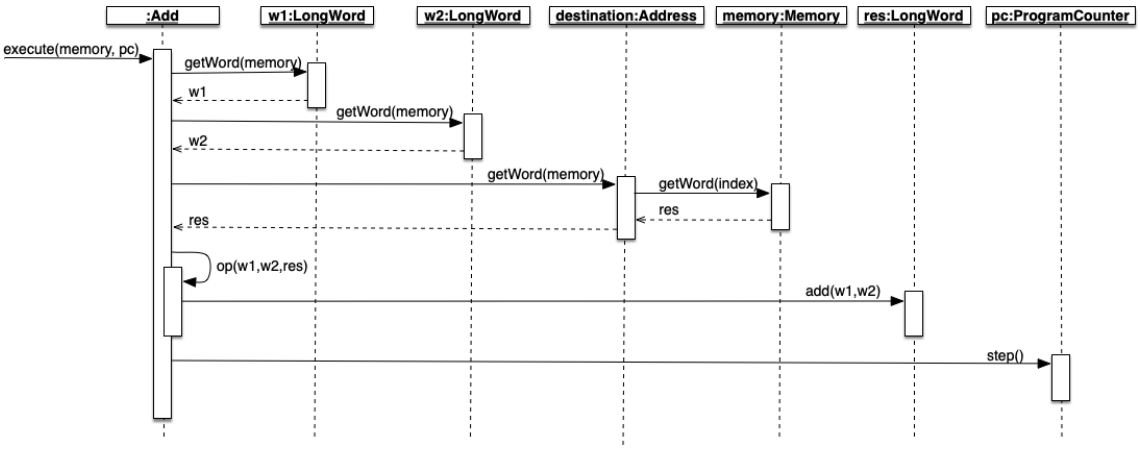
(b) *Cohesion*: Ett mått på hur väl sammanhängande något (paket/klass/metod) är – vi strävar efter *hög* cohesion.

(c) *Coupling*: Ett mått på hur mycket olika delar i ett system beror av varandra – vi strävar efter *låg* coupling.

(d) *Don't Repeat Yourself*: Upprepa inte kod, varje sak som görs i ett system skall göras på ett ställe.

Lösning 2

Sekvensdiagram:



Lösning 3

(a) Enklast är att använda ett `boolean`-attribut för tillståndet, vi kan dessutom introducera en hjälpmetod `ref` för att slippa de annars lite bökiga explicita typ-konverteringarna (ni behöver inte ha hjälpmetoden):

```
public class BooleanWord implements Word {

    private boolean value;

    BooleanWord  (boolean value) {    // denna är paketskyddad, så att
```

```
        this.value = value;          // vår factory kan anropa den
    }

    private BooleanWord ref(Word other) {
        return (BooleanWord)other;
    }

    public void add(Word left, Word right) {
        value = ref(left).value || ref(right).value;
    }

    public void mul(Word left, Word right) {
        value = ref(left).value && ref(right).value;
    }

    public void copy(Word other) {
        value = ref(other).value;
    }

    public boolean equals(Word other) {
        return value == ref(other).value;
    }

    public String toString() {
        if (value) {
            return ”False”;
        } else {
            return ”True”;
        }
    }
}
```

eller, ännu enklare:

```
public class BooleanWord implements Word {

    private boolean value;

    BooleanWord  (boolean value) {   // denna är paketskyddad, så att
        this.value = value;          // vår factory kan anropa den
    }

    private boolean valueOf(Word other) {
        return ((BooleanWord)other).value;
    }

    public void add(Word left, Word right) {
        value = valueOf(left) || valueOf(right);
    }

    public void mul(Word left, Word right) {
        value = valueOf(left) && valueOf(right);
    }

    public void copy(Word other) {
        value = valueOf(other);
    }

    public boolean equals(Word other) {
```

```
            return value == valueOf(other);
        }

        public String toString() {
            if (value) {
                return ”False”;
            } else {
                return ”True”;
            }
        }
    }
```

Man får inget poängavdrag om man skriver:

```
    public void add(Word left, Word right) {
        value = ((BooleanWord)left).value || ((BooleanWord)right).value;
    }
```

och liknande på andra ställen (även om det är en mild form av DRY).

(b)

```
    public class BooleanWordFactory implements WordFactory {

        public Word word(String s) {
            return new BooleanWord(!s.equals(”0”));
        }
    }
```

Lösning 4

(a)

```
    interface Observer {

        void update(Observable obs, Object obj);
    }

    class Observable {

        private List<Observer> observers = new LinkedList<>();
        private boolean changed = false;

        public void addObserver(Observer observer) {
            observers.add(observer);
        }

        public void setChanged() {
            changed = true;
        }

        public boolean hasChanged() {
            return changed;
        }

        public void clearChanged() {
            changed = false;
        }

        public void notifyObservers(Object arg) {
            if (!changed) {          // Denna test behöver ni
                return;              // inte göra i era lösningar
            }
```

```
            for (var observer : observers) {
                observer.update(this, arg);
            }
            clearChanged();
        }

        public void notifyObservers() {
            notifyObservers(null);
        }
    }
```

(b) Det är några saker som vi måste göra för att låta Observer-mönstret hantera hela kedjan av händelser:

- `MotionDetector` måste utvidga `Observable`.
- `Alarm` måste implementera `Observer`.
- `MotionDetector` måste anropa

  ```
  setChanged();
  notifyObservers();
  ```

  i `motionDetected`-metoden.
- `Alarm` måste anropa `start()` i sin `update`-metod.

Vi har sedan två alternativa lösningar:

- vi kan antingen låta samtliga alarm lyssna på samtliga detektorer, eller
- låta `SurveillanceSystem` agera som 'spindeln i nätet', så att den lyssnar på samtliga detektorer, och att samtliga alarm lyssnar på den.

Dessa lösningar ger lika många poäng på tentan (den senare lösningen kräver färre kopplingar, men vi hade inga sådana krav i uppgiftstexten).

För att koppla samtliga larm till samtliga detektorer kan vi spara alla larm och detektorer som attribut i `SurveillanceSystem`:

```
private List<MotionDetector> detectors = new LinkedList<>();
private List<Alarm> alarms = new LinkedList<>();
```

och sedan i `add`-metoderna skriva:

```
public void add(MotionDetector detector) {
    detectors.add(detector);
    for (var alarm : alarms) {
        detector.addObserver(alarm);
    }
}
```

respektive

```
public void add(Alarm alarm) {
    alarms.add(alarm);
    for (var detector : detectors) {
        detector.addObserver(alarm);
    }
}
```

För att istället låta `SurveillanceSystem` själv lyssna på alla detektorer, och väcka alla alarm, så är det några saker vi måste göra:

- Vi måste göra vårt `SurveillanceSystem` till `Observable`, och anmäla alla alarm som lyssnare.
- Vi måste göra vårt `SurveillanceSystem`, *eller någon del av det*, till `Observer`, och anmäla detta som lyssnare på detektorerna.

Om vi gör hela `SurveillanceSystem` till `Observer` så får vi:

```
class SurveillanceSystem extends Observable implements Observer {

    public void add(MotionDetector detector) {
        detector.addObserver(this);
    }

    public void add(Alarm alarm) {
        addObserver(alarm);
    }

    public void update(Observable obs, Object obj) {
        setChanged();
        notifyObservers();
    }
}
```

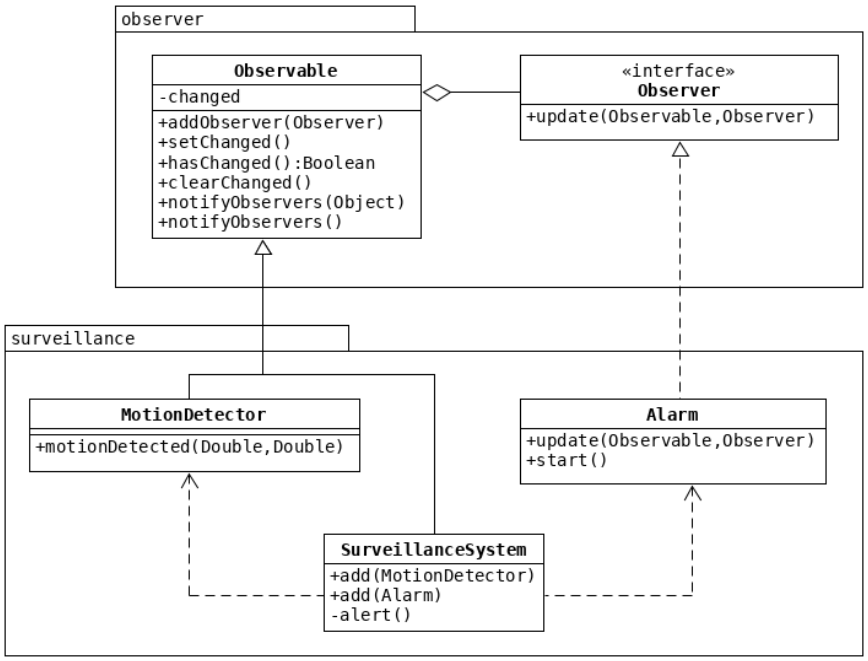
Alternativt kan vi låta ett lambda-uttryck implementera `Observer`, och får då:

```
class SurveillanceSystem extends Observable {

    public void add(MotionDetector detector) {
        detector.addObserver((obs,obj) -> alert());
    }

    public void add(Alarm alarm) {
        addObserver(alarm);
    }

    private void alert() {
        setChanged();
        notifyObservers();
    }
}
```

Dessa lösningar gör att vi inte behöver skriva någon annan kod för att kontrollera våra detektorer och alarm, men man kan få poäng även med andra lösningar.

(c) I Javas standardklasser ligger `Observer` och `Observable` i paketet `java.util` (vilket minskar graden av cohesion ytterligare lite), här lägger vi det i ett eget `observer`-paket (det behövde ni inte göra).

I diagrammet förutsätter vi lösningen ovan, så `SurveillanceSystem` implementerar inte själv `Observer` (det är ett lambda-uttryck inne i den som gör det) – om vi låter `SurveillanceSystem` själv implementera `Observer` så får vi dra ytterligare en streckad linje från `Observer`.



Lösning 5

(a) Vi får klasserna

```
class NormalRandom implements RealRandom {

    private Random rng = new Random();
    private double mu, sigma;

    public NormalRandom  (double mu, double sigma) {
        this.mu = mu;
        this.sigma = sigma;
    }

    public double next() {
        return mu + rng.nextGaussian() * sigma;
    }
}
```

och

```
class UniformRandom implements RealRandom {

    private Random rng = new Random();
    private double lower, upper;

    public UniformRandom  (double lower, double upper) {
        this.lower = lower;
        this.upper = upper;
    }

    public double next() {
        return lower + (upper - lower) * rng.nextDouble();
    }
}
```

Man kan använda *Template Method* för att lyfta upp `Random`-attributet i en gemensam superklass, men det ger ingen större vinst i detta fall (och vi vill inte lyfta upp de två reella talen som beskriver för-

delningarna, eftersom de har helt olika innebörd i de olika klasserna). Om ni har använt Template Method här, så är det helt OK, men det var inte nödvändigt.

(b) Detta är ett typexempel på användning av *Decorator Pattern*, och det som vi vill dekorera är slumptalsgeneratorerna, dvs `RealRandom` – vi kan bespara oss lite framtida arbete genom att skriva en generell dekorator (vi använder här *Template Method*):

```
abstract class RealRandomDecorator implements RealRandom {

    private RealRandom random;

    public RealRandomDecorator  (RealRandom random) {
        this.random = random;
    }

    protected abstract void handle(double sample);

    public final double next() {
        var sample = random.next();
        handle(sample);
        return sample;
    }
}
```

Man kan få nästan full poäng även om man inte använder Template Method.

Det är nu enkelt att beräkna statistik – vi behöver egentligen inte ens spara alla värden, det räcker att spara summan av våra värden, summan av deras kvadrater, och antalet tal:

```
class RealRandomStatistics extends RealRandomDecorator {

    private double sumX, sumX2;
    private int n;

    public RealRandomStatistics  (RealRandom random) {
        super(random);
    }

    protected void handle(double sample) {
        sumX += sample;
        sumX2 += square(sample);
        n += 1;
    }

    private double square(double value) {
        return value * value;
    }

    public double average() {
        return sumX / n;
    }

    public double standardDeviation() {
        return Math.sqrt((sumX2 - n * square(average())) / (n - 1));
    }
}
```

Det går också bra att i `handle` spara alla tal i en lista, och sedan beräkna medelvärde och standardavvikelse genom att använda den givna formeln på elementen i listan.

För att logga skriver vi istället:

```
class RealRandomLogger extends RealRandomDecorator {

    private DB db;

    public RealRandomLogger  (RealRandom random, DB db) {
        super(random);
        this.db = db;
    }

    protected void handle(double sample) {
        db.addSample(sample);
    }
}
```

Vi kan köra en simulering och undersöka statistiken på följande sätt:

```
var rng = new UniformRandom(0, 4);
var bigBang = new BigBangSimulation();
var rngWithStats = new RealRandomStatistics(rng);
bigBang.run(rngWithStats);
System.out.printf("           Average: %8.4f\n", rngWithStats.average());
System.out.printf("Standard deviation: %8.4f\n", rngWithStats.standardDeviation());
```

Utvikning: Om vi både vill beräkna statistik och spara till en databas så kan vi kapsla in en dekorator i en annan:

```
// ...
var rng = new UniformRandom(0, 4);
var bigBang = new BigBangSimulation();
var rngWithStats = new RealRandomStatistics(rng);
var loggedRng = new RealTimeLogger(rngWithStats, db);
bigBang.run(loggedRng);
// ...
```

(c) Vi använder följande mönster:

- *Command Pattern*: Vi kapslar i våra slumptalsgeneratorer in den information som behövs för att vi senare skall kunna generera slumptal när vi anropar `next()`.
- *Decorator Pattern*: För att kapsla in och dekorera våra slumptalsgeneratorer med olika slags beräkningar och loggning.
- *Template Pattern*: När vi i `RealRandomDecorator` lyfter upp den kod som annars skulle varit gemensam för flera olika slags dekoratorer.
- *Strategy*: Vi använder en form av strategy när vi vid körningen bestämmer vilken typ av slumptal vi vill använda (definitionsmässigt är strategy när man bestämmer vilken algoritm som skall användas under körningen – ofta sparar man en referens till ett objekt som implementerar algoritmen som ett attribut, och det gör vi inte här, så man kan argumentera för att det inte är standardversionen av strategy-mönstret).

För full poäng på uppgiften räcker det att man anger tre av dessa.