Övning 5.

Denna vecka ska vi titta på designmönstren Singleton, State, Observer, Composite och Decorator, samt gränssnittet Comparator.

Uppgift 1

Syftet med designmönstret *Singleton* är att kunna garantera att *endast en instans* av en viss klass existerar. Oftast skall också denna instans kommas åt utifrån klassen. Nedan ges en klass med enbart klassmetoder. Är denna klass en singleton? Om inte, hur ska vi förändra klassen så att den blir en singleton? Spelar det någon roll om klassen är en singelton eller inte?

```
public class GameTile {
   // code not show here
}

public class GameUtils {
   public static GameTile[][] newBoard(final int width, final int height, final GameTile baseTile) {
      GameTile[][] board = new GameTile[width][height];
      fillBoard(board, baseTile);
      return board;
   }
   public static void fillBoard(final GameTile[][] board, final GameTile baseTile) {
      for (GameTile[] row : board) {
         for (int j = 0; j < row.length; j++) {
            row[j] = baseTile;
         }
      }
   }
}
```

Uppgift 2

Betrakta nedanstående klass:

```
public class Counter {
   private int value;
   private boolean countingUp;
   public Counter() {
      value = 0;
      countingUp = true;
   }
   public void set(int value) {
      this.value = value;
   }
   public int get() {
      return value;
   }
   public void step() {
      if (countingUp)
         value = value +1;
      else
         value = value - 1;
   }
   public void setCountingUp() {
      countingUp = true;
   }
   public void setCountingDown() {
      countingUp = false;
   }
}
```

Gör om designen genom att använda designmönstret *State*.

Uppgift 3

Designmönstret Strategy är en lämplig teknik att ta till då bara delar av en implementation varierar. Gränssnittet Comparator är ett exempel inom Javas standard-API på en strategi. I en typisk sorteringsalgoritm är det lämpligt att bryta ut själva jämförelsen av elementen från resten av algoritmen, därigenom blir jämförelsekriteriet enkelt utbytbart.

Betrakta klassen Integrator nedan:

```
public class Integrator {
    public enum Function {
        FUNCTION1, FUNCTION2, FUNCTION3
    }
    public static double integrate(Function f, double a, double b, int steps) {
        double sum = 0.0;
        double previous = fun(f, a);
        double delta = (b - a) / steps;
        for (int i = 1; i <= steps; i++) {
            double x = a + (b - a) * i / steps;
            double current = fun(f, x);
            // Compute the integral through Simpson's 3/8 rule.
            sum += delta / 8 * (previous + current  + 3 * (fun(f, (2 * (x - delta) + x) / 3.0)
                                + fun(f, (3 * x - delta) / 3.0)));
            previous = current;
        }
        return sum;
    }
    private static double fun(Function f, double x) {
        switch (f) {
            case FUNCTION1: { return Math.log(Math.log(x)); }
            case FUNCTION2: {return Math.pow(x, x); }
            case FUNCTION3:{ return Math.sin(Math.sin(x)); }
            default:
                throw new IllegalArgumentException("Illegal function!");
        }
    }
}
```

Metoden integrate gör en numerisk skattning av integralen

$$\int_a^b f_k(x)dx$$

där $f_k(x)$ är någon av funktionerna $f_1(x) = \log(\log(x))$, $f_2(x) = x^x$, $f_3(x) = \sin(\sin(x))$.

Ett allvarligt problem med klassen Integrator är den inte stödjer *Open-Closed principen*. Vi måste förändra klassen varje gång vi vill integrera en ny funktion, trots att metoden för själva integrationen inte förändras. Vi kan därmed inte skapa nya funktioner under körning.

Skriv om klassen Integrator så att den istället för fasta funktioner tar en strategi som parameter till metoden integrate. Du behöver bara skapa en korrekt strategi samt förändra metoden integrate på lämpligt vis.

Tips: Enum-typen Function ska inte finnas med i lösningen.

Uppgift 4

Betrakta klasserna Bee och Flower nedan

```
public class Bee {
   private final String name;
   public Bee(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
   //. . .
}

public class Flower {
   private final String name;
   public Flower(String name) {
      this.name = name;
   }
   public void visit(Bee b) {
      System.out.println(name + " is beeing visited by " + b.getName());
   }
   public void bloom() {
      System.out.println(name + " is blooming!");
   }
   //. . .
}
```

Komplettera dessa klasser så att designmönstret *Observer* realiseras. När en blomma blommar meddelar den detta till alla bin som är observatörer, och när ett bi som får reda på att en blomma blommar besöker biet blomman. Nedanstående huvudprogram illustrerar hur det hela är tänkt att fungera:

```
public static void main(String[] args) {
   Flower f = new Flower("Blossom");
   Bee b1 = new Bee("Bob");
   Bee b2 = new Bee("Fred");
   f.addObserver(b1);
   f.addObserver(b2);
   f.bloom();
}
```

Utskriften blir:

```
Blossom is blooming!
Blossom is beeing visited by Bob
Blossom is beeing visited by Fred
```

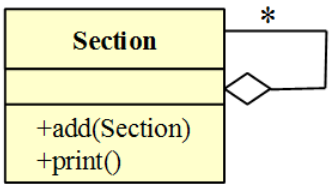
Java stöder designmönstret *Observer* genom att tillhandahålla interfacet Observer och klassen Observable, samt interfacet PropertyChangeListener och klassen PropertyChangeSupport. Du skall dock här göra en egen (!) implementation av designmönstret *Observer*. Följande, något förenklade, gränssnitt är givna:

```
public interface Observable {
   void addObserver(Observer o);
   void removeObserver(Observer o);
   void notifyObservers();
}

public interface Observer {
    void update(Observable obs);
}
```

Uppgift 5

I många ordbehandlingsprogram finns möjlighet att automatiskt producera en innehållsförteckning över alla sektioner i ett dokument. Alla rubriker på olika nivåer i dokumentet kommer med i innehållsförteckningen. Varje rad börjar med rubrikens sektionsnummer följt av själva rubriktexten. Sektioner kan vara hierarkiskt uppbyggda med undersektioner, som i sin tur kan innehålla undersektioner o.s.v. Sektionshierarkin i en innehållsförteckning kan representeras med designmönstret *Composite* enligt nedanstående UML-diagram:



Implementera klassen Section. Följande exempel visar hur utskriften skall se ut.

```
Section alg = new Section("Algorithms");
alg.add(new Section("Recursive algorithms"));
alg.add(new Section("Sorting algorithms"));
alg.add(new Section("Correctness of algorithms"));

Section c = new Section("Complexity of algorithms");
c.add(new Section("Time Complexity"));
c.add(new Section("Space Complexity"));
c.add(new Section("Worst Case behaviour"));
c.add(new Section("Average Case behaviour"));
alg.add(c);

Section ds = new Section("Data Structures");
ds.add(new Section("Abstract data types"));

Section lists = new Section("Lists");
lists.add(new Section("Array based representation"));
lists.add(new Section("Pointer based representation"));
ds.add(lists);

Section trees = new Section("Trees");
trees.add(new Section("General trees"));
Section btrees = new Section("Binary trees");
btrees.add(new Section("Expression trees"));
btrees.add(new Section("Huffman coding trees"));
btrees.add(new Section("Binary search trees"));
trees.add(btrees);
ds.add(trees);

Section algDS = new Section("Algorithms and Data Structures");
algDS.add(alg);
algDS.add(ds);

algDS.print();
```

Utskrift:

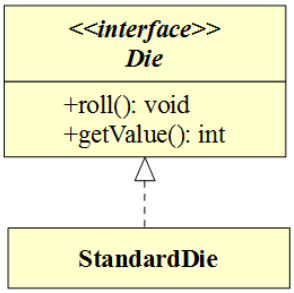
```
1 Algorithms and Data Structures
1.1 Algorithms
1.1.1 Recursive algorithms
1.1.2 Sorting algorithms
1.1.3 Correctness of algorithms
1.1.4 Complexity of algorithms
1.1.4.1 Time Complexity
1.1.4.2 Space Complexity
1.1.4.3 Worst Case behaviour
1.1.4.4 Average Case behaviour
1.2 Data Structures
1.2.1 Abstract data types
1.2.2 Lists
1.2.2.1 Array based representation
1.2.2.2 Pointer based representation
1.2.3 Trees
1.2.3.1 General trees
1.2.3.2 Binary trees
1.2.3.2.1 Expression trees
1.2.3.2.2 Huffman coding trees
1.2.3.2.3 Binary search trees
```

Tips: För att enkelt kunna hantera sektionsnumreringen rekommenderas att definiera en privat hjälpmetod, print(String), som anropas av print() på lämpligt sätt.

Uppgift 6

I ett programsystem för att hantera tärningsspel finns gränssnittet

```
public interface Die {
   void roll();       // rolls the die
   int getValue();  // returns the face value
}
```



samt klassen

```
public class StandardDie implements Die {
   //code not shown here
}
```

som implementerat en vanlig sexsidig tärning.

Nu behöver man ha tillgång till tärningar som aldrig upprepar föregående utfall vid nästföljande kast. En sådan tärning ger t.ex. aldrig två sexor i följd. Din uppgift är att implementera klassen

```
public class NoRepeatDie implements Die
```

genom att använda designmönstert Decorator. Klassen NoRepeateDie har, förutom metoderna som specificeras av gränssnittet Die, även metoden

```
public void noRepeatRoll()
```

som skiljer sig från metoden roll genom att den aldrig ger samma värde två gånger i följd.

Uppgift 7

Betrakta klassen Car nedan:

```
public final class Car {
    private final String manufacturer;
    private final String modelName;
    private final int modelYear;
    private final String serialNumber;
    public Car(String manufacturer, String modelName, int modelYear, String serialNumber) {
        this.manufacturer = manufacturer;
        this.modelName = modelName;
        this.modelYear = modelYear;
        this.serialNumber = serialNumber;
    }
    public String getManufacturer() {
        return manufacturer;
    }
    public String getModelName() {
        return modelName;
    }
    public int getModelYear() {
        return modelYear;
    }
    public String getSerialNumber() {
        return serialNumber;
    }
    public String toString() {
        return " manufacturer: " + this.manufacturer + ", modelName: " +  this.modelName
                + ", modelYear: " + this.modelYear +  ", serialNumber: " +  this.serialNumber;
    }
}
```

a) Vi vill att objekt av klassen Car skall kunna sorteras i stigande ordning med avseende på följande variabler:

1. modelYear
2. manufacturer
3. modelName

Om två bilar är av samma årsmodell jämför man också tillverkare, om tillverkare också är lika, jämförs även modellnamn.

Skriv om klassen Car så att den implementerar gränssnittet Comparable. Metoden compareTo i Car skall ge ordningen enligt ovan.

The Java Language Specification rekommenderar att metoden compareTo() är konsistent med metoden equals(), d.v.s. för två objekt a och b där a.compareTo(b) == 0 skall gälla att a.equals(b) == true. Om detta inte uppfylls kan det t.ex. inträffa underligheter när objekten lagras i samlingar. Implementera därför även metoden equals i klassen Car.

b) I de fall då vi vill tillhandahålla mer än en ordning för en klass använder vi oss av gränssnittet Comparator (detta kan vi så klart också använda för att ordna objekt av en klass som inte redan implementerar Comparable).

Skriv en klass

public class CarComparator **implements** Comparator<Car>

som kan användas för att sortera objekten i klassen Car i stigande ordning med avseende på:

1. manufacturer
2. modelYear
3. modelName

Om två bilar har samma tillverkare jämför man också årsmodell, om årsmodellen också är lika, jämförs även modellnamn.

c) Objekten i Car kan dock sorteras efter många andra ordningar än den som implementeras av CarComparator ovan. Eftersom Car har 4 instansvariabler kan dessa ordnas på 4! = 24 olika sätt. Dessutom kan varje instansvariabel sorteras efter både stigande och fallande ordning. Det totala antalet ordningar som objekten i Car kan sorteras på blir därför 2^4 * 4! = 16 * 24 = 384 olika sätt. Det behövs således 384 olika implementationer av gränssnittet Comparator för att få alla totala ordningar för objekten i klassen Car.

Ett betydligt flexiblare sätt är att nyttja designmönstren *Decorator* och *Template Method*, genom att skapa en jämförare för varje instansvariabel och seriekoppla dessa efter behov. Implementera denna idé för klassen Car.