Lösningsförslag: Övning 5.

Uppgift 1

Klassen är inte en singleton eftersom flera instanser kan skapas av klassen.

Ett sätt att göra klassen till en singleton är enligt:

```
public class GameUtils {
   private static final GameUtils instance = new GameUtils();
   private GameUtils() {}  // Prohibits further instanciations by clients.
   public static GameUtils getInstance() {
      return instance;
   }
   public GameTile[][] newBoard(final int width, final int height, final GameTile baseTile) {
      GameTile[][] board = new GameTile[width][height];
      fillBoard(board, baseTile);
      return board;
   }
   public void fillBoard(final GameTile[][] board, final GameTile baseTile) {
      for (GameTile[] row : board) {
         for (int j = 0; j < row.length; j++) {
            row[j] = baseTile;
         }
      }
   }
}
```

Denna klass kan dock inte serialiseras (implementera interfacet Serializable, som vi inte behandlat ännu). För att deserialiseringen inte skall skapa nya instanser måste implementationen förändras.

Detta problem kan undvikas genom att göra klassen till en *enum med endast ett element*. För ett mer utvecklat resonemang, se gärna Item 3 i Joshua Bloch's "Effective Java".

```
public enum GameUtils {
   INSTANCE; // Marks end of enum members.

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

Kommentarer:

Det finns dock inget att tjäna på att göra klassen till en singleton eftersom det inte finns något tillstånd som behöver initieras vid ett bestämt tillfälle. Men eftersom klassen bara innehåller statiska metoder bör den inte gå att instansiera överhuvudtaget (vilket vi åstadkommit genom att göra den till en singleton).

Uppgift 2

```
public class Counter {
   private int value;
   private State currentState;
   private State ascendingState;
   private State descendingState;

   public Counter() {
      ascendingState = new AscendingState();
      descendingState = new DescendingState();
      setCountingUp();
   }

   public void set(int value) {
      this.value = value;
   }

   public int get() {
      return value;
   }

   public void setCountingUp() {
      currentState = ascendingState;
   }

   public void setCountingDown() {
      currentState = descendingState;
   }

   public void step() {
      value = currentState.step(value);
   }
}

public interface State {
    int step(int value);
}

public class AscendingState implements State {
   public int step(int value) {
      return value + 1;
   }
}

public class DescendingState implements State {
   public int step(int value) {
      return value - 1;
   }
}
```

Uppgift 3

```
public interface RealFunction {
   double fun(double x);
}

public class Integrator {
   public static double integrate(RealFunction f, double a, double b, int steps) {
      double sum = 0.0;
      double previous = f.fun(a);
      double delta = (b - a) / steps;
      for (int i = 1; i <= steps; i++) {
         double x = a + (b - a) * i / steps;
         double current = f.fun(x);
         /* Compute the integral through Simpson's 3/8 rule.*/
         sum += delta / 8
                  * (previous + current + 3 * (f.fun((2 * (x - delta) + x) / 3.0) + f.fun((3 * x - delta) / 3.0)));
         previous = current;
      }
      return sum;
   }
}

public class LogLog implements RealFunction {
   @Override
   public double fun(double x) {
      return Math.log(Math.log(x));
   }
}

public class PowXX implements RealFunction {
   @Override
   public double fun(double x) {
         return Math.pow(x, x);
   }
}

public class SinSin implements RealFunction {
   @Override
   public double fun(double x) {
      return Math.sin(Math.sin(x));
   }
}

public class TestIntegrator {
   public static void main(String[] args) {
      RealFunction loglog = new LogLog();
      System.out.println(Integrator.integrate(loglog, 10, 20, 1000));

      RealFunction f =
          new RealFunction() { //anonym klass
            @Override
            public double fun(double x) {
               return Math.cos(Math.sin(x));
            }
          };
      System.out.println(Integrator.integrate(f, 10, 20, 1000));
   }
}
```

Kommentarer:

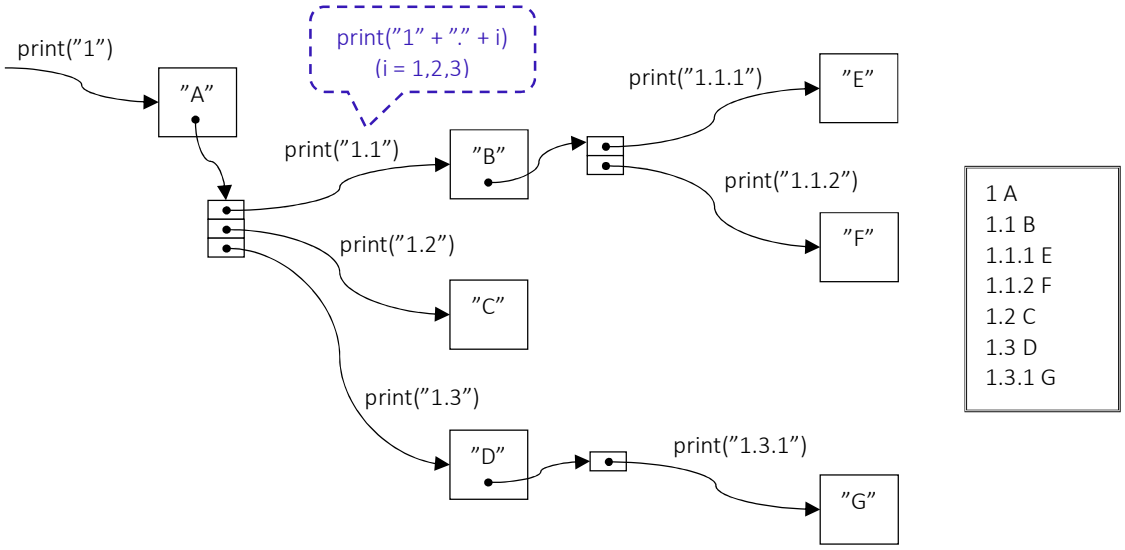
Strategy-mönstret bygger på principen *encapsulate what varies*. I detta exempel är det funktionen som skall integreras som varierar och lösningen är att kapsla in funktionen i en separat klass.

Uppgift 4

```
import java.util.ArrayList;
public class Flower implements Observable {
   private final String name;
   private ArrayList<Observer> observers;

   public Flower(String name) {
      this.name = name;
      observers = new ArrayList<Observer>();
   }

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

   public void removeObserver(Observer o) {
       observers.remove(o);
   }
   public void notifyObservers() {
      for(Observer o : observers) {
         o.update(this);
      }
   }

   public void visit(Bee b) {
      System.out.println(name + " is being visited by " + b.getName());
   }

   public void bloom() {
      System.out.println(name + " is blooming!");
      notifyObservers();
   }
}

public class Bee implements Observer {
   private final String name;

   public Bee(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void update(Observable obs) {
      if (obs instanceof Flower) {
         Flower f = (Flower) obs;
         f.visit(this);
      }
   }
}
```

Uppgift 5

```
import java.util.*;
public class Section {
   public static final String SPACE = " ";
   private String title;
   private List<Section> subsections;
   public Section(String title) {
       this.title = title;
       subsections = new ArrayList<Section>();
   }

   public void add(Section m) {
       subsections.add(m);
   }

   public void print() {
       print("1");
   }

   private void print(String sectionNumber) {
       printTitle(sectionNumber,title);
       printSubsections(sectionNumber);
   }

   private void printTitle(String sectionNumber,String title) {
       System.out.println(sectionNumber + SPACE + title);
   }

   private void printSubsections(String sectionNumber) {
       int i = 1;
       for ( Section s : subsections ) {
           s.print(sectionNumber + "." + i);
           i++;
       }
   }
}
```



Uppgift 6

```
public class NoRepeatDie implements Die {
    private Die die;
    public NoRepeatDie(Die die) {
        this.die = die;
    }
    public void roll() {
        die.roll();
    }
    public int getValue() {
        return die.getValue();
    }
    public void noRepeatRoll() {
        int v = die.getValue();
        do {
            die.roll();
        } while ( die.getValue() == v );
    }
}
```

Uppgift 7

a)

```
public final class Car implements Comparable<Car> {
  //Code not shown here
  // Compare as lazily as possible.
  public int compareTo(Car c) {
      int diff = modelYear - c.modelYear;
      if (diff != 0) {
          return diff;
      }
      diff = manufacturer.compareTo(c.manufacturer);
      if (diff != 0) {
          return diff;
      }
      return modelName.compareTo(c.modelName);
  }

  public boolean equals(Object o) {
      // Ok to use instanceof since the class is declared "final".
      if (o instanceof Car) {
          Car c = (Car) o;
          return c.manufacturer.equals(manufacturer) && c.modelName.equals(modelName)
                  && c.modelYear == modelYear;
      }
      return false;
  }
}
```

Kommentarer:

Att en klass implementerar Comparable innebär att objekten av klassen bli jämförbara med varandra och att det därmed kommer att existera en *naturlig ordning* på objekten. Informellt kan man tänka att objekten ska kunna läggas på rad i storleksordning genom att par av objekt jämförs med varandra. Detta bör vara en *total ordning* (*linjär ordning*):

En mängd S är totalt ordnad under ≤ om följande villkor gäller för alla element a, b och c i S:

- Om $a \leq b$ och $b \geq a$ så $a = b$ (antisymmetri)
- Om $a \leq b$ och $b \leq c$ så $a \leq c$ (transitivitet)
- $a \leq b$ eller $b \geq a$ (totalitet)

Enligt kontraktet för compareTo skall anropet a.compareTo(b) returnera:

1. ett positivt heltal, om a > b
2. 0, om a == b
3. ett negativt heltal, om a < b

Observera hur subtraktion modelYear - c.modelYear används för att skapa ett returvärde från compareTo() utan att använda **if**-satser eller liknande konstruktioner.

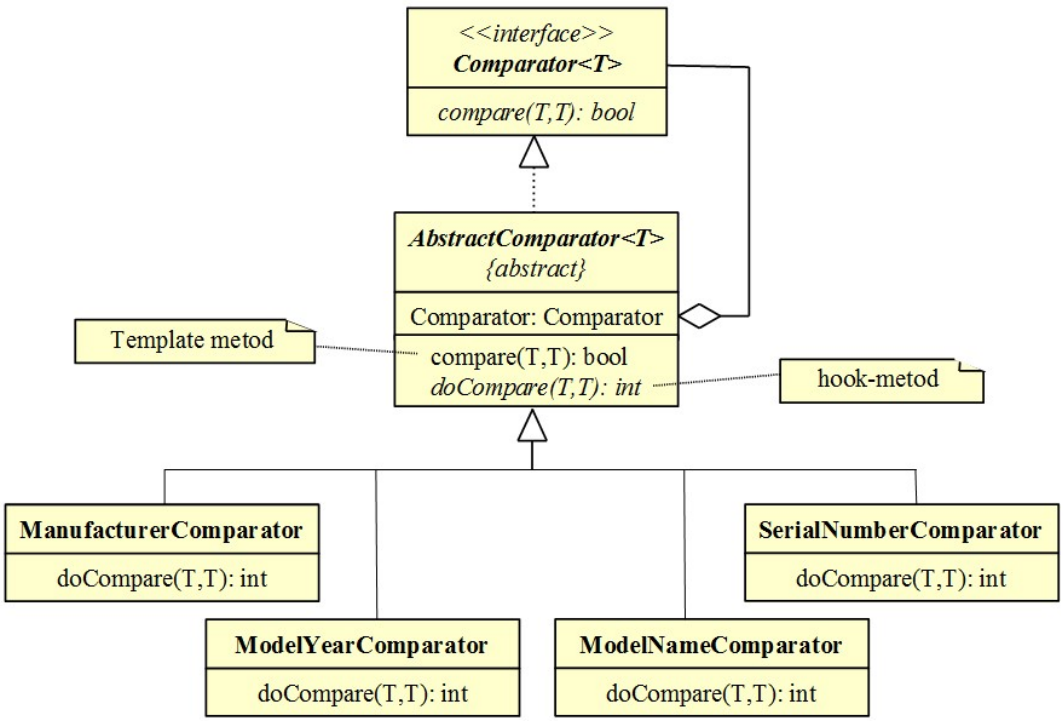
b)

```
import java.util.Comparator;
public class CarComparator implements Comparator<Car> {
  public int compare(Car c1, Car c2) {
      int diff = c1.getManufacturer().compareTo(c2.getManufacturer());
      if (diff != 0) {
          return diff;
      }
      diff = c1.getModelYear() - c2.getModelYear();
      if (diff != 0) {
          return diff;
      }
      return c1.getModelName().compareTo(c2.getModelName());
  }
}
```

Kommentarer:

För Comparator gäller det till stor del att tänka på motsvarande saker som när Comparable ska implementeras. (med metoden compare() som motsvarighet till compareTo()), se därför kommentarerna till föregående uppgift.

c)




```
import java.util.Comparator;
public abstract class AbstractComparator<T> implements Comparator<T> {
   // The decorator comparator
   private final Comparator<T> comparator;
   private final boolean ascending;

   protected AbstractComparator(Comparator<T> comparator, boolean ascending) {
      this.comparator = comparator;
      this.ascending = ascending;
   }

   // Template method
   @Override
   public int compare(T o1,T o2) {
      if (o1 == o2) {
         return 0;
      }
      if (comparator != null) {
         int result = comparator.compare(o1,o2);
         if (result != 0) {
            return result;
         }
      }
      int result = doCompare(o1,o2);
      return ascending ? result : -result;
   }

   // Subclasses should implement this hook method:
   protected abstract int doCompare(T o1,T o2);
}
```

```
import java.util.Comparator;
/**
 * Comparator for comparing car manufacturers
*/
public class ManufacturerComparator extends AbstractComparator<Car> {
    /**
    * @param comparator other decorator possible null
    * @param ascending
    */
    public ManufacturerComparator(Comparator<Car> comparator, boolean ascending) {
        super(comparator, ascending);
    }

    // Note : doCompare NOT compare
    @Override
    public int doCompare(Car c1, Car c2) {
        String name1 = c1.getManufacturer();
        String name2 = c2.getManufacturer();
        return name1.compareTo(name2);
    }
}

import java.util.Comparator;
/**
 * Comparator for comparing car model names
*/
public class ModelNameComparator extends AbstractComparator<Car> {
    /**
    * @param comparator other decorator possible null
    * @param ascending
    */
    public ModelNameComparator(Comparator<Car> comparator, boolean ascending) {
        super(comparator, ascending);
    }

    @Override
    public int doCompare(Car c1, Car c2) {
        String name1 = c1.getModelName();
        String name2 = c2.getModelName();
        return name1.compareTo(name2);
    }
}
```

```
import java.util.Comparator;
/**
 * Comparator for comparing car model years
*/
public class ModelYearComparator extends AbstractComparator<Car> {
   /**
   * @param comparator other decorator possible null
   * @param ascending
   */
   public ModelYearComparator(Comparator<Car> comparator, boolean ascending) {
      super(comparator, ascending);
   }

   @Override
   public int doCompare(Car c1, Car c2) {
      int year1 = c1.getModelYear();
      int year2 = c2.getModelYear();
      if (year1 == year2)
            return 0;
      else if (year1 > year2)
            return 1;
      return -1;
   }
}

import java.util.Comparator;
/**
 * Comparator for comparing car serial numbers
*/
public class SerialNumberComparator extends AbstractComparator<Car> {
   /**
   * @param comparator other decorator possible null
   * @param ascending
   */
   public SerialNumberComparator(Comparator<Car> comparator, boolean ascending) {
      super(comparator, ascending);
   }

   @Override
   public int doCompare(Car c1, Car c2) {
      String name1 = c1.getSerialNumber();
      String name2 = c2.getSerialNumber();
      return name1.compareTo(name2);
   }
}
```

Exempel: Sortering av en lista av bilar i stigande ordning på tillverkarnamn och om lika i stigande ordning på modellnamn och om lika i avtagande ordning på modellår.

```
import java.util.*;
public static void main(String[] args) {
   List<Car> cars = new ArrayList<Car>();
   cars.add(new Car("Volvo","Amazon",1969,"12345678"));
   cars.add(new Car("Saab","9000",1988,"3463658"));
   cars.add(new Car("Volvo","PV",1962,"34853648"));
   cars.add(new Car("Rover","Streetwise",2004,"7744229"));
   cars.add(new Car("Volvo","Amazon",1968,"10857043"));
   cars.add(new Car("Saab","900",1982,"12345678"));

   Comparator<Car> myCarComp =
      new ModelYearComparator(
         new ModelNameComparator(
            new ManufacturerComparator(null,true),true),false);

   Collections.sort(cars,myCarComp);

   for ( Car c : cars )
      System.out.println(c);
}
```

Utskriften blir:

```
manufacturer: Rover, modelName: Streetwise, modelYear: 2004, serialNumber: 7744229
manufacturer: Saab, modelName: 900, modelYear: 1982, serialNumber: 12345678
manufacturer: Saab, modelName: 9000, modelYear: 1988, serialNumber: 3463658
manufacturer: Volvo, modelName: Amazon, modelYear: 1969, serialNumber: 12345678
manufacturer: Volvo, modelName: Amazon, modelYear: 1968, serialNumber: 10857043
manufacturer: Volvo, modelName: PV, modelYear: 1962, serialNumber: 34853648
```