Lösningsförslag till tentamen *P r el i m i n ä r*

Kurs	**Objektorienterad programutveckling, fk**
Tentamensdatum	**2018-01-12**
Program	**DAI2**
Läsår	**2017/2018, lp 2**
Examinator	**Uno Holmer**

Uppgift 1 (8 p)

```
public class QueueAdapter implements Queue {
    private OldQueueImpl adaptee = new OldQueueImpl();
    public boolean isEmpty() {
        return adaptee.hasMore() == 0 ? true : false;
    }
    public void add(int x) {
        adaptee.put(x);
    }
    public int poll() throws NoSuchElementException {
        if ( isEmpty() )
            throw new NoSuchElementException(
                        "QueueAdapter: poll called on empty queue");
        int result = adaptee.look();
        adaptee.remove();
        return result;
    }
}
...
Queue q = new QueueAdapter();
...
```

Uppgift 2 (7 p)

```
public class FileTree {
    public static void main(String[] args) {
        System.out.println(getSizeOfFileTree(new File(args[0])));
    }

    public static long getSizeOfFileTree(File file) {
        long totalSize = 0;
        if ( file.isFile() )
            totalSize = file.length();
        else if ( file.isDirectory())
            for ( File f : file.listFiles() )
                totalSize += getSizeOfFileTree(f);

        return totalSize;
    }
}
```

Uppgift 3 (1+6 p)

a)
Designen bryter mot designprincipen *law of demeter* – do't talk to strangers.

b)
```
public class B {
    private C theCobject;
    public B() {
        theCobject = new C();
    }
    public void update(int x) {
        theCobject.update(x);
    }
}

public class A {
    private B theBobject;

    public A() {
        theBobject = new Brefactored();
    }
    public void dummy() {
        theBobject.update(42);
    }
}
```

Uppgift 4 (8 p)

```
public class WheatherStation implements Iterable<WheatherData> {
    private WheatherData[] hourData;

    public WheatherStation() {
        hourData = new WheatherData[24];
    }
    ...
    public Iterator<WheatherData> iterator() {
        return new WheatherIterator();
    }

    private class WheatherIterator implements Iterator<WheatherData> {
        private int current = 0;
        @Override
        public boolean hasNext() {
            return current < hourData.length;
        }

        @Override
        public WheatherData next() {
            if ( ! hasNext() )
                throw new NoSuchElementException();
            return hourData[current++];
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

Uppgift 5 (8 p)

Låt X > Y betyda att specifikationen X är starkare än Y. Då ser vi att B > A och dessutom A > C, B > C och D > C. A och D löser inte samma problem och inte heller B och D.
Om fältet är sorterat gäller motsvarande eftersom eftervillkoren är samma som ovan: F > E, E > G, F > G och H > G. Om fältet är sorterat gäller också att H > E, F > H och H > F och vi kan rangordna samtliga efter avtagande styrka, t.ex. F > H > E > G. Är förvillkoret uppfyllt blir flera av eftervillkoren ekvivalenta och därmed specifikationerna, t.ex. F och H. Är fältet sorterat hamnar ju det minsta duplikatet alltid först.

Uppgift 6 (3+4 p)

a)
Om stacken definieras med implementationsarv kan vi låta `foo` kasta typen till `List` och därefter hantera parametern som en lista. Det kanske konstruktören av klassen inte hade tänkt på?

```
private static void foo(Stack<String> s) {
    List<String> alias = (List<String>)s;
    String temp = alias.get(0);
    alias.set(0,alias.get(2));
    alias.set(2,temp);
}
```

Anm. Om parametertypen ändras till `List<String>` kompilerar inte anropet av `foo`.

b)

```
public class StackImpl2<E> implements Stack<E> {
    private ArrayList<E> theStack = new ArrayList<E>();

    public void push(E x) {
        theStack.add(x);
    }
    public void pop() throws NoSuchElementException {
        if ( isEmpty() )
            throw new NoSuchElementException();
        theStack.remove(theStack.size()-1);
    }
    public E top() {
        if ( isEmpty() )
            throw new NoSuchElementException();
        return theStack.get(theStack.size()-1);
    }

    public boolean isEmpty() {
        return theStack.isEmpty();
    }
}
```

Uppgift 7 (7 p)

Låt `Middle` passa undantaget vidare och `main` hantera det.

```
public class Utilities {
    public static long fac(int n) throws IllegalArgumentException {
        if ( n < 0 )
            throw new IllegalArgumentException(
                        "fac called with negative argument");
        else {
            long result = 1;
            for ( int i = 1; i <= n; i++ )
                result *= i;
            return result;
        }
    }
}
public class Middle {
    public long f(int n) throws IllegalArgumentException {
        return Utilities.fac(2*n) + 42;
    }
}
public class Main {
    public static void main(String[] args) {
        Middle m = new Middle();
        try {
            System.out.println(m.f(Integer.parseInt(args[0])));
        }
        catch ( IllegalArgumentException e ) {
            System.out.println(e.getMessage());
        }
    }
}
```

Uppgift 8 (8 p)

Designen bryter mot *DIP*, *OCP*, *Information Expert*. Tillämpa *Strategy*-mönstret och faktorisera ut en basklass `BoatInsurance` med gemensamma attribut och placera resten i två subkllasser för resp. försäkringstyp. Basklassen kan göras abstrakt, även om den saknar abstrakta metoder. Av kravet i uppgiften att döma skall metoden `totalPremiums` kunna hantera vilken samling som helst vilket leder till att parametertypen till metoden måste vidgas till åtminstone `Collection`. Som en konsekvens behöver `BoatInsurance` får likhets-, hash- och jämförelsemetoder, så att objekt kan lagras i olika typer av samlingar. För att tillåta argument av mer specifika typer, t.ex. `HashSet<MotorBoatInsurance>`, krävs att typen för parametern `col` innehåller ett begränsningsuttryck: `Collection<? extends BoatInsurance>`.

```
public abstract class BoatInsurance
implements Comparable<BoatInsurance> {
    private String insuranceId;
    private String name;
    private int length;         // cm
    private int value;          // Euro

    public BoatInsurance(String insuranceId,String name,
                         int length,int value) {
        this.insuranceId = insuranceId;
        this.name = name;
        this.length = length;
        this.value = value;
    }
    public String getId() { return insuranceId; }
```

```
    public String getName() { return name; }
    public int getValue() { return value; }
    public int getLength() { return length; }
    public void setValue(int value) { this.value = value; }
    public int getInsurancePremium() {
        return (int)(value*0.01 + Math.max(0,length-500));
    }
    public final boolean equals(Object o) {
        if ( o == this )
            return true;
        if ( o instanceof BoatInsurance ) {
            BoatInsurance other = (BoatInsurance)o;
            return insuranceId.equals(other.insuranceId);
        }
        return false;
    }
    public int hashCode() {
        return insuranceId.hashCode();
    }
    public int compareTo(BoatInsurance other) {
        return insuranceId.compareTo(other.insuranceId);
    }
}
public class MotorBoatInsurance extends BoatInsurance {
    private int enginePower;    // Kw

    public MotorBoatInsurance(String insuranceId,String name,
                              int length,int value,int enginePower) {
        super(insuranceId,name,length,value);
        this.enginePower = enginePower;
    }
    public int getEnginePower() { return enginePower; }

    public int getInsurancePremium() {
        return super.getInsurancePremium() +
                  10*Math.max(0,enginePower-15);
    }
}
public class SailingBoatInsurance extends BoatInsurance {
    private int draught;        // cm

    public SailingBoatInsurance(String insuranceId,String name,
                                int length,int value,int draught) {
        super(insuranceId,name,length,value);
        this.draught = draught;
    }
    public int getDraught() { return draught; }

    public int getInsurancePremium() {
        return super.getInsurancePremium() +
                   20*Math.max(0,draught-150);
    }
}

public class Main {
    public static void main(String[] args) {
        // Dessa exempelsamlingar krävs ej i lösningen.
        List<BoatInsurance> ll = new LinkedList<>();
        ll.add(new MotorBoatInsurance("1234","Titanic",
                     785,320000,180));
        ll.add(new SailingBoatInsurance("5678","Blue Bird",
                     1200,170000,187));
        System.out.println(totalPremiums(ll));
```

```
        HashSet<MotorBoatInsurance> hs = new HashSet<>();
        hs.add(new MotorBoatInsurance("1234","Titanic",
                    785,320000,180));
        hs.add(new MotorBoatInsurance("8741","Jupiter",
                    830,645000,240));
        System.out.println(totalPremiums(hs));

        Set<BoatInsurance> ts = new TreeSet<>();
        ts.add(new MotorBoatInsurance("1234","Titanic",
                    785,320000,180));
        ts.add(new SailingBoatInsurance("5678","Blue Bird",
                    1200,170000,187));
        System.out.println(totalPremiums(ts));
    }
    public static int totalPremiums(Collection<? extends BoatInsurance> col){
        int sum = 0;
        for ( BoatInsurance i : col)
            sum += i.getInsurancePremium();
        return sum;
    }
}
```