Lösningsförslag till tentamen *P r e l i m i n ä r*

Kurs	**Objektorienterad programutveckling, fk**
Tentamensdatum	**2019-01-18**
Program	**TKIEK-2,TKTFY-3,TKTEM-3**
Läsår	**2018/2019, lp 2**
Examinator	**Uno Holmer**

Uppgift 1 (8 p)

```
public class Client extends SomeBaseClass implements Analyzable {
    private Analyzer analyzer;
    public Client() {
        analyzer = new Analyzer();
    }
    @Override
    public boolean isGood(int x) {
        return analyzer.isGood(x);
    }
    @Override
    public void use(int x) {
        analyzer.use(x);
    }
    public void decide(int x) {
        if ( isGood(x) )
            use(x);
    }
}
```

Uppgift 2 (1+6 p)

a)

Klassen `CustomerClient` bryter mot principerna DIP och OCP. Klassen beror av specifika menyklasser och måste modifieras om till exempel måltiderna brunch, mellanmål eller nattvickning tillkommer. Lösningen nedan kan dock inte hantera mellanmål eftersom sådana är notoriskt svåra att tidsbestämma. Huruvida klassen även bryter mot Law of Demeter är kanske ett gränsfall.

b)

```
public class MenuFactory {
    public static Menu createMenu(int hour) {
        if ( hour >= 5 && hour < 11 )
            return new BreakfastMenu();
        else if ( hour < 15 )
            return new LunchMenu();
        else if ( hour < 21 )
            return new DinnerMenu();
        else
            return null;
    }
}
public class CustomerClient {
    public List<Dish> getDishesBelow(int hour,int priceLimit) {
        Menu menu = MenuFactory.createMenu(hour);
        if ( menu != null ) {
            List<Dish> dishes = new ArrayList<>();
            for ( int i = 0; i < menu.getNoOfDishes(); i++ )
                if ( menu.getDish(i).getPrice() <= priceLimit )
                    dishes.add(menu.getDish(i));
```

```
            return dishes;
        }
        return null;
    }
}
```

Uppgift 3 (5+2 p)

a)

```
public class ReadOnlyIterator<E> implements Iterator<E> {
    private Iterator<E> it;
    public ReadOnlyIterator(Iterator<E> it) {
        this.it = it;
    }
    @Override
    public boolean hasNext() {
        return it.hasNext();
    }
    @Override
    public E next() {
        return it.next();
    }
    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

b)

```
public class Club implements Iterable<Membership> {
    private List<Membership> members = new ArrayList<>();
    public void join(Membership m) {
        members.add(m);
    }
    public Iterator<Membership> iterator() {
        return new ReadOnlyIterator(members.iterator());
    }
}
```

Uppgift 4 (9 p)

Det är tre principer som bestämmer vilken metod som anropas i fallen 1-12 (nästa sida):

1. *Statisk typ, statisk bindning.* För klassmetoder bestämmer referensvariabelns statiska typ vilken metod som anropas. I första hand väljs metoden (vid kompileringen) i den statiska typen om den finns där, annars i den närmast ovanliggande basklassen till denna som definierar metoden.
2. Kompilatorn utgår från referensvariabelns statiska typ för att hitta en lämplig metodsignatur som matchar parametertyperna i anropet. I första hand väljs metoden (vid kompileringen) i den statiska typen om den finns där, annars i den närmast ovanliggande basklassen till denna som definierar metoden.
3. *Polymorfism, dynamisk typ, dynamisk bindning*. När kompilatorn matchat anropet mot möjliga metodsignaturer och bestämt vilken metod som är tillämpbar börjar sökningen (under exekveringen) efter en lämplig metod i referensvariabels dynamiska typ (det utpekade objektets typ). Hittas inte metoden där fortsätter sökningen uppåt i klasshierarkin tills metoden hittas.

Utskrifterna blir:

```
1. Base.f(1)
2. Base.f(2,3)
3. Base.g(4)
4. Base.g(5,6)

5. Base.f(1)
6. Base.f(2,3)
7. Sub.g(4)
8. Base.g(5,6)

9. Sub.f(1)
10. Base.f(2,3)
11. Base.h(Base(bepa))
12. Sub.h(Sub(cepa))
```

Uppgift 5 (2+4 p)

a) Metoden `indexOfMin` har starkast specifikation i `MinFindable1` eftersom den har ett svagare förvillkor och ett starkare eftervillkor.

b)

1. `MinFinder1` är inte en äkta subtyp till `MinFindable1`
   Specifikationen är tydlig om vad som skall hända om fältreferensen `a` är `null` eller pekar på ett tomt fält, men om `a` är `null` kastar metoden `NullPointerException` och om `a` är tom kastar den `ArrayIndexOutOfBoundsException`.
2. `MinFinder2` är inte en äkta subtyp till `MinFindable1`.
   Metoden returnerar det *största* indexet för det minsta elementet, inte det minsta indexet som specifikationen kräver. Metoden kastar inte det undantag som specifikationen kräver. Istället returnerar den -1 som kan resultera i ett indexfel hos klienten, vilket inte är acceptabelt.
3. `MinFinder3` är en äkta subtyp till `MinFindable1`.
   Den strikta olikheten < i villkorssatsen garanterar att det minsta indexet för det minsta elementet returneras.
4. `MinFinder4` är inte en äkta subtyp till `MinFindable1`.
   Samma brister som `MinFinder1` för `null` eller tomt fält och liksom `MinFinder2` returnerar den det största indexet.
5. `6,7,8. MinFinder1` ,…, `MinFinder4` är äkta subtyper till `MinFindable2`.
   Metoderna ger i samtliga fyra fall rätt resultat om förvillkoret är uppfyllt eftersom de returnerar det minsta eller det största indexet för det minsta elementet och specifikationen tillåter vilket som helst av de minsta. Är förvillkoret inte uppfyllt är det ospecificerat vad metoden skall göra och klienten får då ta konsekvenserna.

Uppgift 6 (4+3 p)

a)

```
public class SortedValueMap<K,V> implements ValueMap<K,V> {
    private TreeMap<K,Collection<V>> map;
    public SortedValueMap() {
        map = new TreeMap<>();
    }
    public void addValue(K key,V value) {
        if ( ! map.containsKey(key) )
            map.put(key,new ArrayList<V>());
        map.get(key).add(value);
    }
    public Collection<V> getValues(K key) {
        return map.get(key);
    }
    public Set<K> getKeys(V value) {
        Set<K> result = new HashSet<K>();
        for (Map.Entry<K,Collection<V>> e : map.entrySet() )
            if ( e.getValue().contains(value) )
                result.add(e.getKey());
        if ( result.isEmpty() )
            return null;
        else
            return result;
    }
}
```

b)

```
public class CVSReader {
    public final static String FIELD_SEPARATOR = ",";
    public static ValueMap<String,Integer> readCSVFile(String fileName)
    throws IOException,NumberFormatException
    {
        ValueMap<String,Integer> map = new SortedValueMap<>();
        Scanner sc = new Scanner(new FileReader(fileName));
        while ( sc.hasNextLine() )
            addValues(map,sc.nextLine());
        sc.close();
        return map;
    }

    private static void addValues(ValueMap<String,Integer> map,String line) {
        String[] fields = line.split(FIELD_SEPARATOR);
        for ( int i = 1; i < fields.length; i++ )
            map.addValue(fields[0],Integer.parseInt(fields[i]));
    }
}
```

Uppgift 7 (2+4 p)

a) Punktobjektet i ett geometriskt objekt kan muteras via en sparad referens till objektet som gavs till konstruktorn, eller via en referens returnerad av en accessmetod:

```
Point p = new Point(10,20);
GeometricalObject g = new GeometricalObject(p);
p.setPosition(30,40);
Point q = g.getPosition();
System.out.println("q.x:"+q.getX()+",q.y:"+q.getY());  // q.x:30,q.y:40
q.setPosition(50,60);
Point r = g.getPosition();
System.out.println("r.x:"+r.getX()+",r.y:"+r.getY());  // r.x:50,r.y:60
```

b)

```
public class Point implements Cloneable {
    ...
    public Point clone() {
        try {
            return (Point)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
public class GeometricalObject {
    private Point position;
    public GeometricalObject(Point position) {
        this.position = position.clone();
    }
    public Point getPosition() {
        return position.clone();
    }
    // other methods omitted
}
```

Uppgift 8 (10 p)

Designen bryter mot principerna *DIP* och *OCP*. Såväl klassen `Expression` som metoden `getValue` måste skrivas om vid tillägg av ytterligare operatorer. Designen bryter också mot principen *Information Expert*. Metoden `getValue` borde tillhöra klassen `Expression` och inte `Main`. Vi refaktorerar båda klasserna och tillämpar *Strategy*-mönstret. Problemet kan lösas på några olika sätt. Här eliminerar klasserna `BinaryExpression` och `UnaryExpression` onödig kodduplicering i `Addition` m.fl. genom att de lagrar operanderna och är parametriserade med avseende på operatorn, som ges till konstruktorn och lagras i klassen. Jämför med hur t.ex. `TreeSet` kan ta ett jämförarobjekt av typen `Comparator` som parameter till konstruktorn (Strategy).

```
public class Constant implements Expression {
    private int value;
    public Constant(int value) {
        this.value = value;
    }
    public int getValue(Environment env) {  // Skönhetsfläck
        return value;
    }
}
public class Variable implements Expression {
    private String name;
    public Variable(String name) {
        this.name = name;
    }
    public int getValue(Environment env) {
        return env.getValue(name);
    }
}
public class BinaryExpression implements Expression {
    private BinaryOperator operator;
    private Expression leftOperand,rightOperand;
    public BinaryExpression(BinaryOperator operator,Expression leftOperand,
                                                    Expression rightOperand)
    {
        this.operator = operator;
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }
    @Override
    public int getValue(Environment env) {
       return
           operator.binOp(leftOperand.getValue(env),
                          rightOperand.getValue(env));
    }
}
public class Addition extends BinaryExpression {
    public Addition(Expression leftOperand,Expression rightOperand) {
        super(new AddOp(),leftOperand,rightOperand);
    }
}
public class UnaryExpression implements Expression {
    private UnaryOperator operator;
    private Expression operand;
    public UnaryExpression(UnaryOperator operator,Expression operand) {
        this.operator = operator;
        this.operand = operand;
    }
    @Override
    public int getValue(Environment env) {
```

```
        return operator.unOp(operand.getValue(env));
    }
}
public class Negation extends UnaryExpression {
    public Negation(Expression operand) {
        super(new NegOp(),operand);
    }
}
public class AddOp implements BinaryOperator {
    @Override
    public int binOp(int leftOperand,int rightOperand) {
        return leftOperand + rightOperand;
    }
}
public class NegOp implements UnaryOperator {
    @Override
    public int unOp(int operand) {
        return -operand;
    }
}
```