Programmazione a Oggetti
Metodologie di Programmazione
1 Giugno 2011

Nome: _____ Matricola: _____

Metodologia di Programmazione [ ]
Programmazione a Oggetti [ ]

Istruzioni

- Scrivete il vostro nome sul primo foglio.
- Indicare se la vostra prova è riferita all'esame MP o PO
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.

LASCIATE IN BIANCO:

1	2	3	4	5	6	7	8	9	10	TOT

1 Esercizio

Definite l'implementazione del metodo `equals` per la classe `Point` qui di seguito.

```
class Point
{
    double x, y;
    public Point(double x, double y) { ... }


    // ... altri metodi
}
```

La definizione deve assicurare che, dato un riferimento `p : Point`, il risultato dell'invocazione `p.equals(q)` sia `true` se e solo se `q` è un `Point` con coordinate uguali a quelle di `p`. [3pt]

2 Esercizio

Siano date le seguenti definizioni

```
class SpaceShip
{
  public void CollideWith(Asteroid a) { a.CollideWith(this); }
}

class GiantSpaceShip extends SpaceShip {}

class Asteroid
{
  public void CollideWith(SpaceShip s)
  { system.out.println("Asteroid hit a SpaceShip"); }

  public void CollideWith(GiantSpaceShip s)
  { system.out.println("Asteroid hit a GiantSpaceShip"); }
}

class ExplodingAsteroid extends Asteroid
{
  public void CollideWith(SpaceShip s)
  { system.out.println("ExplodingAsteroid hit a SpaceShip"); }

  public void CollideWith(GiantSpaceShip s)
  { system.out.println("ExplodingAsteroid hit a GiantSpaceShip"); }
}
```

Considerate ora le seguenti dichiarazioni:

```
  Asteroid theAsteroid = new Asteroid();
  ExplodingAsteroid theExplodingAsteroid = new ExplodingAsteroid();

  SpaceShip theSpaceShip = new SpaceShip();
  GiantSpaceShip theGiantSpaceShip = new GiantSpaceShip();

  Asteroid theAsteroidReference = theExplodingAsteroid;
  SpaceShip  theSpaceShipReference = theGiantSpaceShip;
```

Descrivete l'output delle seguenti chiamate: [3pt]

```
  theAsteroid.CollideWith(theSpaceShipReference);

  theAsteroidReference.CollideWith(theSpaceShipReference);

  theSpaceShipReference.CollideWith(theAsteroid);

  theSpaceShipReference.CollideWith(theAsteroidReference);
```

3 Esercizio

Scrivete il codice di una classe `TwoStripes` che realizza una immagine rettangolare composta da due fasce verticali, contigue e di uguale larghezza, associate ciascuna ad un colore. Il costruttore della classe ha la seguente firma:

```
public TwoStripes(int width, int height,
                  PictureColor firstStripe,
                  PictureColor secondStripe);
```

La classe deve implementare l'interfaccia definita qui di seguito:

```
public interface Picture
{
   // restituisce il colore della coordinata (x,y)
   public PictureColor getColor(int x, int y);
   // restituisce la larghezza dell'immagine
   public int getWidth();
   // restituisce l'altezza dell'immagine
   public int getHeight();
}
```

[2pt]

4 Esercizio

Considerate il seguente sistema di classi:

```
public interface StockBroker
{
   public abstract String getAdvice();
}

public class Moodys implements StockBroker
{
   public String getAdvice() { return "buy Microsoft!";}
}

public class MerrillLynch implements StockBroker
{
  public String getAdvice() { return "sell Google!"; }
}

public class Investor
{
  private Stockbroker _sb;
  public Investor(Stockbroker sb) { _sb = sb; }

  public void setStockBroker(Stockbroker sb) {_sb = sb; }

  public String tradeStock()
  {
    return "I am going to "+_sb.getAdvice();
  }
 }
```

Descrivere quale(i) *design pattern(s)* viene (vengono) realizzato (i) dal sistema di classi appena descritto. [2pt]

Quale è l'output del seguente frammento di codice?

```
Investor i = new Investor(new Moodys());
System.out.println(i.tradeStock());
i.setStockbroker(new MerrillLynch());
System.out.println(i.tradeStock());
```

[1pt]

5 Esercizio

Consideriamo la realizzazione del concetto di funzione ottenuta mediante la seguente definizione di interfaccia.

```
public interface Fun
{
  // Applicata all'input x, restituisce un oggetto
  // come  risultato.
  public Object apply(Object x);
}
```

Data `f:Fun` ed `o:Object`, l'espressione `f.apply(o)` denota il risultato ottenuto applicando `f` all'argomento `o`. Ad esempio, consideriamo le due "funzioni" seguenti:

```
public class F implements Fun
{
  public Object apply(Object x)
  { return "F("+x.toString()+")"; }
}
public class G implements Fun
{
  public Object apply(Object x)
  { return "G("+x.toString()+")"; }
}
```

Ora, l'espressione `new F().apply("x")` restituisce la stringa `F(x)`; similmente, l'espressione `new G().apply("x")` restituisce la stringa `G(x)`.

Sia ora data l'interfaccia `Logical`, che rappresenta oggetti con la capacità di applicare una di due possibili funzioni:

```
public interface Logical
{
  // Applica trueF or falseF ad x
  public Object select(Fun trueF, Fun falseF, Object x);
}
```

Definite due classi concrete, `True` e `False` che implementino l'interfaccia `Logical` e realizzino il seguente comportamento. Dato `b:Logical`, l'espressione `b.select(new F(), new G(), x)` restituisce "`F(x)`" se `b` è una istanza di classe `True`, mentre restituisce "`G(x)`" se `b` è una istanza di classe `False`.

NOTA BENE: Il vostro codice per le classi `True` e `False` non deve contenere alcuna costante di tipo stringa. [3pt]

6 Esercizio

Vogliamo definire un sistema di classi ed interfacce che rappresentino le relazioni tra le componenti di un circuito elettronico descritte qui di seguito:

- una *resistenza* è una componente che regola la quantità di corrente da cui è attraversata;
- un *condensatore* è una componente che immagazzina carica;
- un *transistor* è un amplificatore di segnale elettrico;
- un *circuito integrato* è una componente che internamente contiene resistenze, condensatori, transistors e possibilmente altri circuiti integrati che agiscono come un unico elemento;
- una *scheda* contiene una collezione di componenti.

isegnate il diagramma UML delle classi. [3pt]

7 Esercizio

Sia `Pred<T>` l'interfaccia definita come segue:

```
interface Pred<T>
{
     public boolean good(T x);
}
```

Completate il codice delle due classi `EList<T>` (la lista sempre vuota) e `NEList<T>` (una lista sempre non vuota) che implementano l'interfaccia `IList<T>` e realizzano l'interfaccia `IList<T>` definita qui di seguito:

```
public interface IList<T>
{
    // restituisce la lista di tutti gli elementi di
    // this che soddisfano p;  ovvero la lista degli
    // x tali che p.good(x) = true
    IList<T> filter(Pred<T> p);
}

public class EList<T> implements IList<T>
{
    public static final EList singleton = new EList();


}


public class NEList<T> implements IList<T>
{
    private T first;
    private IList<T> rest;

    public NEList(T f, IList<T> r) { first = f; rest = r; }

    public IList<T> filter(Pred<T> p)
    {




    }
}
```

[3pt]

8 Esercizio

I neuroni presenti nel cervello realizzano un sistema bio-elettrico molto sofisticato che trasmette l'effetto di uno stimolo mediante un impulso elettrico. La propagazione dell'impulso corrisponde ad una modifica dello "stato elettrico" del neurone, che si svolge in quattro fasi, ognuna delle quali ha una durata di circa un 1 millisecondo.

- *resting*: il neurone è a riposo, voltaggio = -70.0mV); se stimolato passa nello stato *rising*
- *rising*: il neurone passa ad un voltaggio = 0.0 mV; in questa fase eventuali stimoli sono ignorati; dopo un millisecondo passa nello stato *falling*; in
- *falling*: fase calante dello stimolo, corrispondente ad un voltaggio = -25.0mV; anche in questa fase il neurono non reagisce a stimoli; dopo un millisecondo passa nella fase *undershoot*
- *undershoot*: fase di riequilibrio, ad un voltaggio = -80.0mV; se stimolato passa allo stato *resting*, a cui passa comunque dopo un millisecondo;

Possiamo rappresentare i neuroni come oggetti della classe `Neuron` descritta qui di seguito in termini di tre metodi:

- `stimulate()`: stimola il neurone per attivare l'impulso elettrico
- `getVoltage()`: misura il voltaggio della membrana
- `wait1ms`: simula il passaggio di tempo ed il conseguente passaggio delle diverse fasi nella trasmissione dello stimolo;

```
public class Neuron
{
  private State _state = RestingState.Singleton;

  void setState(State state) { _state = state;  }
  public double getVoltage() { return _state.getVoltage(this); }
  public void wait1ms()   { _state.wait1ms(this);   }
  public void stimulate() { _state.stimulate(this); }
}
```

Definite la classe astratta `State` e quattro classi concrete `RestingState`, `RisingState`, `FallingState` e `UndershootState` che realizzano il comportamento descritto in precedenza; [3pt]

9 Esercizio

Data l'interfaccia albero binario:

```
public interface AlberoBinario<T> {

    AlberoBinario<T> getLeft();
    AlberoBinario<T> getRigth();
    T getInfo();
    boolean isEmpty();

    List<T> preordine();
    Iterator<T> iterator();
}
```

Definiamo la *visita in preordine* la visita dell'albero in cui viene visitata prima l'informazione della radice, poi ricorsivamente visitato in preordine il sottoalbero sinistro e successivamente il sottoalbero destro. Implementare i metodi `getLeft, getRight, getInfo` e `isEmpty` definendo la classe `AlberoBinarioImpl` con le variabili d'istanza necessarie. [2pt]

Implementare il metodo ricorsivo `preordine` che ritorna una lista con le informazioni dell'albero in preordine. [3pt]

10 Esercizio

Con riferimento all'interfaccia `AlberoBinario` dell'Esercizio 9, implementate il metodo `iterator` in modo restituisca un iteratore che visita l'albero in preordine **non usando** una lista di supporto in preordine ma usando uno stack. [5pt]

Disegnare il diagramma delle classi nel caso si adotti il Design Pattern NullObject. [2pt]