Esame di Ingegneria del Software e Programmazione ad Oggetti

Riportare i seguenti dati

Cognome:	_____	Nome:	_____
Matricola:	_____	Ordinamento (509/270):	_____

Indicare l'esame che si sta sostenendo

Denominazione Esame	*Acronimo*	*Ordinamento*	*Crediti*	
Ingegneria del Software	IS	509	5	☐
Programmazione ad Oggetti	PO	509	5	☐
Ingegneria del Software e Programmazione ad Oggetti	IS-PO	270	9	☐
Integrazione di uno degli esami di cui sopra *(indicare quale e per quanti crediti)*			_	☐

Marcare con una X la denominazione esatta dell'esame che si sta sostenendo.

Organizzazione dell'esame

L'esame è composito di due parti (parte IS, e parte PO) e 4 domande/esercizi (possibilmente ulteriormente suddivisi in più punti). Devono rispondere a tutte le domande della parte IS gli studenti che intendono sostenere l'esame di IS oppure IS-PO. Devono svolgere tutti gli esercizi della parte PO gli studenti che intendono sostenere l'esame di PO oppure IS-PO.

Regole per lo svolgimento dell'esame

Durata: 60 minuti per la parte IS + 60 minuti per la parte PO. Non si può consultare documentazione. Usare solo ed esclusivamente fogli di carta bianca. **Riconsegnare tutto: testo d'esame, fogli di bella, fogli di brutta.** Barrare i fogli di brutta prima di consegnarli e ricopiare i risultati negli spazi bianchi del testo d'esame se previsti.

Parte IS

(*solo studenti 509 che sostengono IS e studenti 270 che sostengono IS-PO*)

Domanda 1 *(vale il 50% di questo esercizio)*

Descrivere brevemente il *Design Pattern "Decorator"*, discutendo i principali benefici di questo pattern. Scrivere il codice java di un esempio di utilizzo di questo pattern. Elencare almeno un utilizzo di questo pattern all'interno della libreria standard java, ad es. nel package `java.io`.

Domanda 2 *(50%)*

Descrivere brevemente il *Design Pattern "Template Method"*. Fornire un esempio completo di suo utilizzo in java comprensivo di test di unità, preferibilmente utilizzando il framework JUnit.

Parte PO

(*solo studenti 509 che sostengono PO e studenti 270 che sostengono IS-PO*)

Esercizio 3 *(vale il 60% di questa parte)*

Una software house sta sviluppando un'applicazione per manipolare alcune forme geometriche. Tra le classi significative segnaliamo `Cerchio`, `Quadrato`, ed `Equilatero`, che devono essere scritte di modo da implementare la seguente interfaccia che rappresenta una forma geometrica di colore e dimensione data.

```
package esercizio3;

public interface Forma {

  public String getColore();

  public int getDimensione();

}
```

3a — *(vale il 35% di questo esercizio)* Scrivere il codice delle tre classi `Cerchio`, `Quadrato`, ed `Equilatero`, di modo che implementino l'interfaccia `Forma` e sia possibile creare oggetti che modellino forme geometiche di dimensione e colore specificate all'atto della costruzione di nuovi oggetti. Completare le classi in maniera tale che il seguente test di unità funzioni correttamente, facendo attenzione alla nota accanto all'asserzione finale:

```
package esercizio3;

import static org.junit.Assert.*;

import java.util.HashSet;
import java.util.Set;

import org.junit.Before;
import org.junit.Test;

public class UnitTest1 {

  private Set<Forma> set;

  @Before
  public void setUp() throws Exception {
    this.set = new HashSet<Forma>();
  }

  @Test
  public void test() {
    Forma r1 = new Equilatero( 1, "Rosso"); // r1 ed...
    Forma r2 = new Equilatero( 1, "Rosso"); // ..r2 "contano" 1
    Forma r3 = new Quadrato( 2, "Nero");
    Forma r4 = new Cerchio( 2, "Nero");
    set.add(r1);
    set.add(r2);
    set.add(r3);
    set.add(r4);

    assertEquals(3, set.size()); /* N.B. 3 e NON 4 */ // <----------
  }

}
```

3b — *(25%)* Scrivere il codice di una classe astratta `AbstractForma`, che affianca ed implementa l'interfaccia `Forma` consentendo di minimizzare il codice replicato nelle tre sclassi `Cerchio`, `Quadrato`, ed `Equilatero` già scritte. Riscrivere le tre classi dopo le modifiche fatte per utilizzare questa nuova classe astratta.

3c — *(25%)* si è deciso di catalogare le forme geometriche per tipologia facendo uso di questa classe:

```
package esercizio3;

import java.util.Set;

public class RaggruppatoreDiForme {

  private Raccoglitore raccoglitore;

  public void raggruppa(Cerchio c) {
    this.raccoglitore.addCerchio(c);
  }

  public void raggruppa(Equilatero e) {
    this.raccoglitore.addEquilatero(e);
  }

  public void raggruppa(Quadrato q) {
    this.raccoglitore.addQuadrato(q);
  }

  public Raccoglitore raggruppa(Set<Forma> forme) {
    this.raccoglitore = new Raccoglitore();
    for(Forma r : forme) {
      r.accetta(this);
    }
    return this.raccoglitore;
  }

}
```

che una volta attivata invocando il metodo `Raccoglitore raggruppa(Set<Forma> forme)` su di un insieme di forme permette di ottenere, per ciascuna tipologia di forme, il sottoinsieme delle sole forme di quella tipologia mediante l'utilizzo di un oggetto istanza della classe `Raccoglitore` restituito dal metodo stesso. `Raccoglitore` permette di interrogare il risultato del raggruppamento di forme per tipologia tramite dei metodi dedicati allo scopo: ad esempio `List<Quadrato> getQuadrati()`, ma anche, analogamente, `List<Cerchio> getCerchi()` e `List<Equilatero> getEquilateri()`. Il suo codice ci viene integralmente fornito qui di seguito:

```
package esercizio3;

import java.util.LinkedList;
import java.util.List;

public class Raccoglitore {

  private List<Cerchio> cerchi;
  private List<Equilatero> equilateri;
  private List<Quadrato> quadrati;

  public Raccoglitore() {
    this.cerchi = new LinkedList<Cerchio>();
    this.equilateri = new LinkedList<Equilatero>();
    this.quadrati = new LinkedList<Quadrato>();
  }

  public List<Cerchio> getCerchi()        { return this.cerchi;     }

  public List<Equilatero> getEquilateri() { return this.equilateri; }

  public List<Quadrato> getQuadrati()     { return this.quadrati;   }

  public void addCerchio(Cerchio c)       { this.cerchi.add(c);     }

  public void addEquilatero(Equilatero e) { this.equilateri.add(e); }

  public void addQuadrato(Quadrato q)     { this.quadrati.add(q);   }

}
```

Osservare che tre metodi `raggruppa()` di `RaggruppatoreDiForme` si avvalgono dei servizi offerti dai metodi "adder" di `Raccoglitore` come ad es. il metodo `void addCerchio(Cerchio c)`.

A sua volta `RaggruppatoreDiForme` ricorre, all'interno del suo metodo `Raccoglitore raggruppa(Set<Forma>)`, al metodo `void accetta(RaggruppatoreDiForme)` da aggiungere appositamente all'interfaccia `Forma` per

aiutare il suo lavoro di raggruppamento di forme per tipologia (e degli oggetti che riceve per tipo), come segue:

```
package esercizio3;

public interface Forma {

  public String getColore();

  public int getDimensione();

  public void accetta(RaggruppatoreDiForme raggruppatore);

}
```

Riscrivere integralmente il codice di `Cerchio`, `Equilatero`, e `Quadrato` (ed eventualmente anche di `AbstractForma`) affinché funzionino correttamente implementando la nuova versione dell'interfaccia `Forma` come atteso dal codice di `RaggruppatoreDiForme` sopra riportato.

3d — *(15%)* Completare il seguente test (nel punto evidenziato) con delle asserzioni affinché abbia successo e risulti completo nella verifica del codice di raggruppamento di forme per tipologia come prodotto nei punti precedenti:

```
package esercizio3;

import static org.junit.Assert.*;

import java.util.*;

import org.junit.Before;
import org.junit.Test;

public class UnitTest2 {

  private Set<Forma> forme;

  private RaggruppatoreDiForme raggruppatore;

  @Before
  public void setUp() throws Exception {
    this.forme = new HashSet<Forma>();
    this.raggruppatore = new RaggruppatoreDiForme();
  }

  @Test
  public void testRaggruppamento() {
    List<Equilatero> equilateri = Arrays.asList(new Equilatero[] {
        new Equilatero(3, "Rosso"),
        new Equilatero(2, "Rosso")
      });
    List<Cerchio>    cerchi     = Arrays.asList(new Cerchio[] {
        new Cerchio(1, "Blu")
      });
    List<Quadrato>   quadrati   = Arrays.asList(new Quadrato[] {
        new Quadrato(2, "Nero"),
        new Quadrato(4, "Marrone")
      });

    forme.addAll(equilateri); forme.addAll(cerchi); forme.addAll(quadrati);

    Raccoglitore raccoglitore = this.raggruppatore.raggruppa(forme);

    /* Domanda 3d: da completare in questo punto */ // <----------
  }

}
```

Esercizio 4 *(vale il 40% di questa parte)*

Con riferimento all'esercizio precedente ed in particolare alla classe `Raccoglitore` così modificata:

```
package esercizio3;

import java.util.LinkedList;
import java.util.List;

public class Raccoglitore {

  private List<Cerchio> cerchi;
  private List<Equilatero> equilateri;
  private List<Quadrato> quadrati;

  public Raccoglitore() {
    this.cerchi = new LinkedList<Cerchio>();
    this.equilateri = new LinkedList<Equilatero>();
    this.quadrati = new LinkedList<Quadrato>();
  }

  public List<Cerchio> getCerchi()        { return this.cerchi;     }

  public List<Equilatero> getEquilateri() { return this.equilateri; }

  public List<Quadrato> getQuadrati()     { return this.quadrati;   }

  public void addCerchio(Cerchio c)       { this.cerchi.add(c);     }

  public void addEquilatero(Equilatero e) { this.equilateri.add(e); }

  public void addQuadrato(Quadrato q)     { this.quadrati.add(q);   }

  public List<Forma> ordinaPerDimensione() {
    /* Domanda 4a: da completare in questo punto */
  }

  public Map<String, Set<Forma>> raggruppaPerColore() {
    /* Domanda 4b: da completare in questo punto */
  }

}
```

4a — *(35%)* Scrivere il codice del metodo `List<Forma> ordinaPerDimensione()` di `Raccoglitore` che restituisce la lista `java.util.List<Forma>` di forme presenti nel raccoglitore su cui viene invocato. Tale lista deve essere ordinata per dimensioni crescenti delle forme contenute.

4b — *(65%)* Scrivere il codice del metodo `Map<String, Set<Forma>> raggruppaPerColore()` di `Raccoglitore` che restituisce una `java.util.Map<String,Set<Forma>>` che associa ad ogni colore l'insieme di forme che possiedono quel colore all'interno del raccoglitore.