Esercizi Design Pattern

Esercizio 1

1) Quale design pattern è implementato nel seguente codice?
2) Indicare i tipi appropriati per “Tipo1”, “Tipo2” e “Tipo3”
3) Che tipo di modificatore è “mod1” ?
4) Qual è la funzionalità del metodo “metodo1()” e quale dovrebbe essere il suo nome?
5) Implementare il codice che istanzia opportunatamente la classe e chiama i metodi implementati

```
public class Balancer{
        private String[] hosts = new String[]{"host1", "host2", "host3"};
        private int x;
        private static Tipo1 b = new Balancer();

        private Balancer() {
                x = 0;
        }

        public Tipo2 getHost(){
                if(x == hosts.length)
                        x = 0;
                return hosts[x++];
        }

        public mod1 Tipo3 metodo1(){
                return b;
        }
}
```

Esercizio 2

1) Disegnare il diagramma UML delle classi per il codice mostrato
2) Qual è il design pattern implementato?
3) Qual è il ruolo delle interfacce Arma e Munizioni?
4) Qual è il ruolo delle classi Fucile, MunizioniPesanti e Armeria
5) Implementare il codice delle classi Pistola e MunizioniLeggere
6) Implementare il codice che istanzia le classi opportune e chiama i metodi definiti in Arma

```
public interface Arma{
        public String getTipo();
        public int getDannoArea();
        public int getDannoMirato();
}

public interface Munizioni {
        public int getMoltiplicatoreDanno();
}

public class Fucile implements Arma {
        private Munizioni m;
```

```
        public Fucile(Munizioni m){
                this.m = m;
        }
        public String getTipo(){
                return "Fucile";
        }
        public int getDannoArea(){
                return 100*m.getMoltiplicatoreDanno();
        }
        public int getDannoMirato(){
                return 20*m.getMoltiplicatoreDanno();
        }
}

public class MunizioniPesanti implements Munizioni{
        public int getMoltiplicatoreDanno(){
                return 5;
        }
}

public class Armeria {
        public static Arma getFucilePesante(){
                return new Fucile(new MunizioniPesanti());
        }

        public static Arma getPistolaOrdinanza(){
                return new Pistola(new MunizioniLeggere());
        }
}
```

Esercizio 3

1) Qual è il design pattern implementato?
2) Il design pattern presenta problemi di implementazione? Se si, quali?
3) Che ruolo hanno le classi Partita, Giocatore, CheatBuster, Server e RegistroGiocatori?
4) Disegnare il diagramma UML delle classi

```
public class Partita{
        CheatBuster cb = new CheatBuster();
        Server s = new Server(10);
        RegistroGiocatori rg = new RegistroGiocatori();

        public void registra(Giocatore g){
                rg.registra(g);
        }

        public void partecipa(Giocatore g){
                boolean isRegistrato = rg.isRegistrato(g);
                boolean isCheater = cb.isCheater(g);
                boolean serverPieno = s.isPieno();

                if(isRegistrato && !isCheater && !serverPieno)
                        s.addGiocatore(g);
        }

        public Server getServer(){
                return s;
        }
}
```

Esercizio 4

1) Qual è il design pattern implementato? Quale variante?
2) Che ruolo hanno la classe Goblin, IGoblin e PelleVerde?
3) Implementare l'interfaccia IGoblin
4) Disegnare il diagramma UML delle classi
5) Implementare un client che utilizzi opportunatamente le classi del design pattern
6) Modificare il codice in modo da ottenere Lazy initialization
7) Modificare il codice in modo da ottenere un'altra variante nota

```
public class Goblin implements IGoblin{
        PelleVerde pv = new PelleVerde();

        public void tiraFreccia(float d){
                int d2 = Math.round(d * 10);
                pv.scagliaPezzoDiLegno(d2);
        }
}

public class PelleVerde {
        public void scagliaPezzoDiLegno(int d){
                System.out.println("- danno: " + d);
        }
}
```

Esercizio 5

1) Qual'è il design pattern implementato?
2) Indicare i tipi appropriati per T1, T2
3) Indicare i nomi appropriati per i metodi m1() e m2()
4) Che ruolo hanno le classi AssistenteVocale e SmartLight?
5) Completare il codice della classe T1 (se necessario) e scrivere il codice nell'interfaccia T2
6) Cosa bisogna modificare per poter ottenere una variante nota?

```
public class AssistenteVocale extends T1 {
        List<ComandoVocale> storicoComandi = new ArrayList<>();
        public void registra(ComandoVocale cv) {
                storicoComandi.add(cv);
                this.m1();
        }
        T2 getUltimoComando() {
                return storicoComandi.get(storicoComandi.size()-1);
        }
}

public abstract class T1 {
        private List<T2> l = new ArrayList<>();
        public void m1() {
                for(T2 x : l)
                        x.m2();
        }
}

public class SmartLight implements T2 {
        private AssistenteVocale assistente;
        public void m2() {
                ComandoVocale cv = assistente.getUltimoComando();
                elaboraComando(cv);
        }
}
```

Soluzioni Esercizi

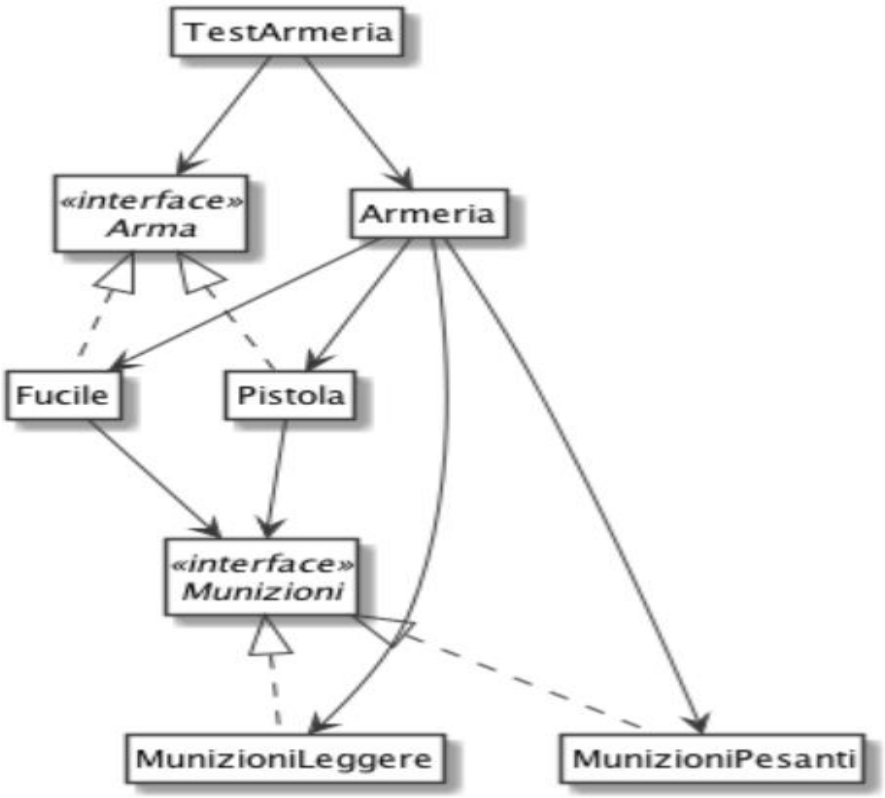
Esercizio 1

1) Singleton
2) Balancer, String, Balancer
3) Static
4) È il metodo che restituisce l'unica istanza presente, tipicamente è chiamato *getInstance()*
5)

```
public class TestBalancer {
        public static void main(String[] args) {
                Balancer b = Balancer.getInstance();
                for(int i=0; i<6; i++) {
                        System.out.println("Call: " + b.getHost());
                }
        }
}
```

Esercizio 2

1)



2) Factory Method
3) Arma svolge il ruolo di **Product**, Munizioni è un'interfaccia da cui dipende Fucile. Munizioni permette di realizzare *Dependency Injection*

4) Fucile svolge il ruolo di **ConcreteProduct**; MunizioniPesanti implementa Munizioni e si può iniettare dentro Fucile, Armeria svolge il ruolo di **ConcreteCreator**

5)

```
public class Pistola implements Arma {
	private Munizioni m;
	public Pistola(Munizioni m){
		this.m = m;
	}
	public String getTipo(){
		return "Pistola";
	}
	public int getDannoArea(){
		return 20 * m.getMoltiplicatoreDanno();
	}
	public int getDannoMirato(){
		return 50 * m.getMoltiplicatoreDanno();
	}
}

public class MunizioniLeggere implements Munizioni {
	public int getMoltiplicatoreDanno(){
		return 2;
	}
}
```

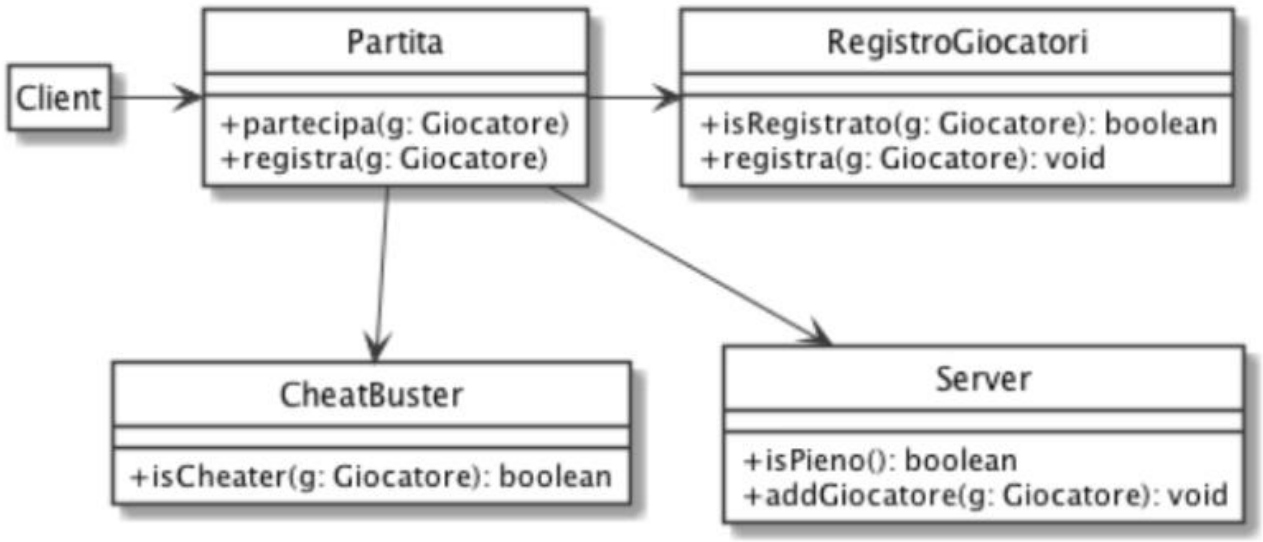
6)

```
public class TestArmeria {
	public static void main(String[] args){
		Arma a = Armeria.getFucilePesante();
		Arma b = Armeria.getPistolaOrdinanza();

		System.out.println("Danno ad area " + a.getTipo() + ": " +
a.getDannoArea());
		System.out.println("Danno mirato " + b.getTipo() + ": " +
b.getDannoMirato());
	}
}
```

Esercizio 3

1) Façade
2) Il façade non dovrebbe esporre le classi interne al sottosistema: il metodo getServer() andrebbe eliminato
3) Partita ha il ruolo di facade; Giocatare è una classe di supporto esterna al sottosistema; CheatBuster, Server e RegistroGiocatori sono classi del sottosistema nascosto dal Facade
4)

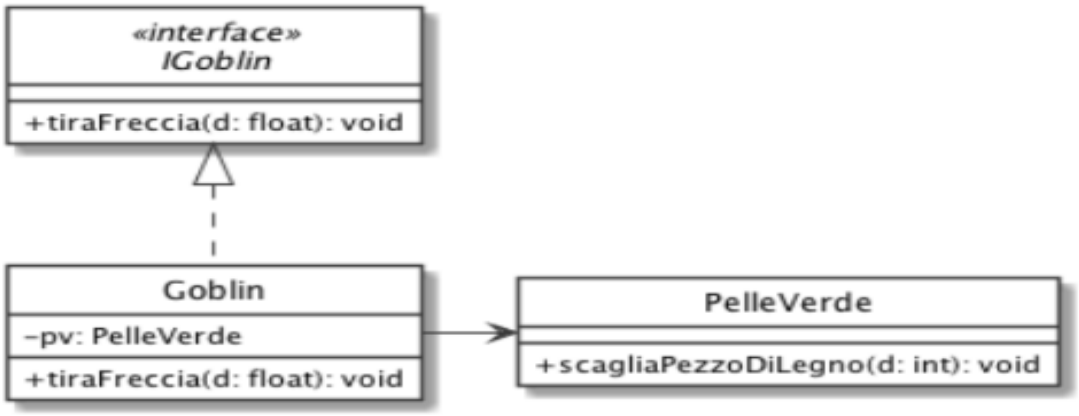


Esercizio 4

1) Adapter, variante Object Adapter
2) Goblin ha il ruolo di Adapter; IGoblin ha il ruolo di Target, PelleVerde ha il ruolo di Adaptee
3)

```
public interface IGoblin {
        public void tiraFreccia(float d);
}
```

4)



5)

```
// il risultato atteso è la stampa di: "danno: 52"
public void client() {
        IGoblin g = new Goblin();
        g.tiraFreccia(5.23);
}
```

6)

```
public class Goblin implements IGoblin {
        PelleVerde vd = null;

        public void tiraFreccia(float d){
                if(pv == null) pv = new PelleVerde();
                int d2 = Math.round(d * 10);
                pv.scagliaPezzoDiLegno(d2);
        }
}
```

7)

```
public class Goblin implements IGoblin extends PelleVerde {
        public void tiraFreccia(float d){
                int d2 = Math.round(d * 10);
                this.scagliaPezzoDiLegno(d2);
        }
}
```

Esercizio 5

1) Observer, variante Pull
2) Subject e Observer
3) Tipicamente notify() e update() (N.B.: in java notify() è già definito nella classe Object, quindi bisogna usare un nome o una firma diversa, es. notifyObservers())
4) ConcreteSubject e ConcreteObserver
5)

```
public abstract class Subject {
        private List<Observer> obList = new ArrayList<>();
        public void attach(Observer ob) {
                if(!obList.contains(ob))
                        obList.add(ob);
        }
        public void detach(Observer ob) {
                if(obList.contains(ob))
                        obList.remove(ob);
        }
        public void notifyObservers() {
                for(Observer ob : obList)
                        ob.update();
        }
}
```

```
public interface Observer {
        public void update();
}
```

6) Modificare il codice usando i tipi Publisher, Subscriber, SubmissionPublisher e Subscription di Java 9 → Reactive Streams