

Ingegneria del Software — Soluzione del Tema 07/02/2022

Esercizio 1

Si consideri la classe Java `JobsDB` per la gestione di domande e offerte di lavoro (classi `JobRequest` e `JobOffer`).

```
public class JobsDB {
    // Ritorna l'elenco delle richieste salvate.
    public /*@ pure @*/ Set<JobRequest> getRequests();

    // Ritorna l'elenco delle offerte salvate
    // in ordine cronologico dalla meno alla piu' recente.
    public /*@ pure @*/ List<JobOffer> getOffers();

    // Aggiunge una richiesta di lavoro.
    // Lancia una DuplicateException se la richiesta e' gia' presente.
    public void addRequest(JobRequest req) throws DuplicateException;

    // Aggiunge un'offerta di lavoro e ritorna l'insieme di tutte le richieste
    // di lavoro compatibili con l'offerta (senza rimuoverle)
    public Set<JobRequest> addOffer(JobOffer offer);

    // Ricerca un'offerta di lavoro compatibile con la richiesta req.
    // Se esiste almeno un'offerta di lavoro compatibile con la richiesta,
    // ritorna la prima offerta compatibile (meno recente) e la rimuove.
    // Altrimenti, ritorna null.
    public JobOffer getOffer(JobRequest req);
}

public /*@ pure @*/ class JobRequest {
    // Ritorna true se e solo se l'offerta e' compatibile con la richiesta.
    public boolean matches(JobOffer offer);
    ...
}
```

Domanda a)

Si specifichi in JML il metodo `addOffer`.

Soluzione

Definiamo `unchangedRequests`, `unchangedOffers` come segue.

```
//@ unchangedRequests <==> (getRequests().size() == \old(getRequests().size()) &&
//@   getRequests().containsAll(\old(getRequests()))
//@
//@ unchangedOffers <==> (getOffers().size() == \old(getOffers().size()) &&
//@   (\forall int i; i>=0 && i<getOffers().size();
//@     \old(getOffers()).get(i).equals(getOffers().get(i)) )
```

Possiamo ora definire `addOffer` come segue, ricordandosi di mantenere l'ordinamento cronologico di `getOffers()`:

```
//@requires offer != null
//@
//@ensures unchangedRequests &&
//@ getOffers().size()==\old(getOffers().size())+1 &&
//@ (\forall int i; i>=0 && i<\old(getOffers().size());
//@   getOffers().get(i).equals(\old(getOffers().get(i)) ) &&
//@ getOffers().get(get(offers().size()-1).equals(offer) &&
//@
//@ \result!=null && (\forall JobRequest req; ; \result.contains(req) <==>
//@   (\old(getRequests()).contains(req) && req.matches(offer)) )
public Set<JobRequest> addOffer(JobOffer offer);
```

Domanda b)

Si specifichi in JML il metodo `getOffer`.

Soluzione

```
//@requires req != null
//@
//@ensures
//@ (\exists Offer o; \old(getOffers()).contains(o); req.matches(o)) ==> (
//@     unchangedRequests && getOffers().size() == \old(getOffers()).size() - 1 &&
//@     (\exists int i; i >= 0 && i < \old(getOffers()).size());
//@     req.matches(\old(getOffers()).get(i)) &&
//@     (\forall int j; j >= 0 && j < i; !req.matches(\old(getOffers()).get(j))) &&
//@     (\forall int j; j >= 0 && j < i; getOffers().get(j) == \old(getOffers()).get(j)) &&
//@     (\forall int j; j > i && j < \old(getOffers()).size();
//@         getOffers().get(j-1) == \old(getOffers()).get(j)) &&
//@     \result.equals(\old(getOffers()).get(i)) ) &&
//@ )
//@ (!\exists Offer o; \old(getOffers()).contains(o); req.matches(o)) ==>
//@     (unchangedOffers && unchangedRequests && \result == null)
public JobOffer getOffer(JobRequest req);
```

Domanda c)

Si consideri un'implementazione di `JobsDB` che fa uso di un `Set` per memorizzare le richieste di lavoro e di una `List` per memorizzare le offerte di lavoro, come mostrato di seguito.

```
public class JobsDB {
    private Set<JobRequest> requests;
    private List<JobOffer> offers;
    ...
}
```

Per tale implementazione, fornire l'invariante di rappresentazione (representation invariant, RI) e la funzione di astrazione (abstraction function, AF).

Soluzione

L'invariante di rappresentazione deve semplicemente indicare che le due collezioni non sono nulle e non contengono valori nulli.

```
private invariant
requests != null && offers != null && !requests.contains(null) && !offers.contains(null)
```

La funzione di astrazione deve indicare che i valori ritornati da `getRequests` e `getOffers` sono tutti e soli quelli contenuti nelle collezioni.

```
private invariant
getOffers().containsAll(offers) && offers.containsAll(getOffers()) &&
(\forall int i; i >= 0 && i < offers.size(); getOffers().get(i) == offers.get(i)) &&
getOffers().size() == offers.size()
```

Domanda d)

Si consideri una classe `RecentJobsDB` che modifica il metodo `addOffer(JobOffer offer)` (override) facendo in modo che ritorni solo le richieste di lavoro aggiunte nell'ultima settimana. `RecentJobsDB` può essere definita come sottoclasse di `JobsDB` in accordo con il principio di sostituzione? Motivare la propria risposta.

Si consideri invece una classe `RecentJobsDB2` che aggiunge un metodo `addOffer(JobOffer offer, int days)` (overload) che si comporta come `addOffer(JobOffer offer)` ma ritorna solo le richieste di lavoro aggiunte negli ultimi days giorni. `RecentJobsDB2` può essere definita come sottoclasse di `JobsDB` in accordo con il principio di sostituzione? Motivare la propria risposta.

Soluzione

`RecentJobsDB` non può essere definita come sottoclasse di `JobsDB` in quanto il metodo `addOffer` ridefinito indebolisce la post condizione del metodo originale, ritornando meno risultati.

`RecentJobsDB2` può invece essere definita come sottoclasse di `JobsDB` in quanto si tratta di un'estensione pura (aggiunta di un metodo) che, modificando la classe come il metodo `addOffer` originale, non può violare alcuna proprietà pubblica.

Esercizio 2

Si consideri nuovamente un'implementazione di JobsDB che fa uso di un Set per memorizzare le richieste di lavoro e di una List per memorizzare le offerte di lavoro, come mostrato di seguito.

```
public class JobsDB {  
    private Set<JobRequest> requests;  
    private List<JobOffer> offers;  
    ... }
```

Domanda a)

Si completi l'implementazione considerando solo i metodi getRequests, getOffers, addRequest, addOffer e garantendo thread-safety, ovvero facendo in modo che diversi thread possano invocare i metodi in parallelo senza creare problemi di concorrenza. **Si favoriscano soluzioni che massimizzano le possibilità di esecuzione parallela da parte dei thread che invocano i diversi metodi.**

Soluzione

```
public class JobsDB {  
    ...  
    public Set<JobRequest> getRequests() {  
        synchronized(requests) { return new HashSet<>(requests); } }  
  
    public List<JobOffer> getOffers() {  
        synchronized(offers) { return new ArrayList<>(offers); } }  
  
    public void addRequest(JobRequest req) throws DuplicateException {  
        synchronized(requests) {  
            if (requests.contains(req)) throws new DuplicateException();  
            requests.add(request); } }  
  
    public Set<JobRequest> addOffer(JobOffer offer) {  
        synchronized(offers) { offers.add(offer); }  
        synchronized(requests) {  
            return requests.stream()  
                .filter(req -> req.matches(offer))  
                .collect(Collectors.toSet());  
        } } }  
}
```

Si osservi che abbiamo evitato di esporre il rep, restituendo ogni volta una copia.

Domanda b)

Si implementi ora anche il metodo getOffer facendo in modo che, invece di ritornare null, sospenda il chiamante nel caso in cui nessuna offerta risulta compatibile con la richiesta. Risulta necessario modificare anche le implementazioni di altri metodi? Se sì, come?

Soluzione

Occorre anche aggiungere una notifyAll al metodo addOffer, come mostrato di seguito.

```
public class JobsDB {  
    ...  
    public JobOffer getOffer(JobRequest req) {  
        synchronized(offers) {  
            while (offers.stream().noneMatch(offer -> req.matches(offer))) {  
                offers.wait();  
            }  
            JobOffer offer = offers.stream().findFirst(offer -> req.matches(offer)).get();  
            offers.remove(offer);  
            return offer;  
        }  
    }  
  
    public Set<JobRequest> addOffer(JobOffer offer) {
```

```

    synchronized(offers) {
        offers.add(offer);
        offers.notifyAll();
    }
    synchronized(requests) {
        return requests.stream()
            .filter(req -> req.matches(offer))
            .collect(Collectors.toSet());
    }
}

```

Esercizio 3

Si consideri la seguente classe di cui viene qui omessa l'ovvia implementazione:

```

public class Student {
    private ...
    public Student(String name, int year, ...) {...}
    public String name() {...}
    public int yearOfBirth() {...}
    public String countryOfBirth() {...}
    public String personalCode() {...}
}

```

Sia data una `List<Student> s` che contiene una lista di studenti opportunamente inizializzata.

Rispondere alle domande seguenti usando **esclusivamente** i costrutti della *programmazione funzionale* (*senza quindi usare while, if.. else, forEach, ecc.*)

Domanda a)

Completare l'istruzione seguente per assegnare alla variabile `youngForeigners` i codici persona degli studenti non italiani nati dopo il 2000 presenti nella lista `s`.

```
List<String> youngForeigners =
```

Soluzione

```

List<String> youngForeigners = s.stream()
    .filter(student -> !student.countryOfBirth().equals("Italy"))
    .filter(student -> student.yearOfBirth() > 2000)
    .map(student -> student.personalCode())
    .collect(Collectors.toList());

```

Domanda b)

Memorizzare in una variabile `nameOfYoungestChinese` lo studente cinese più giovane della lista `s`. Successivamente, *solo se presente*, stampare a schermo il suo nome (senza utilizzare il costrutto `if.. else` di Java).

Soluzione

```

final Optional<Student> nameOfYoungestChinese = s.stream()
    .filter(student -> student.countryOfBirth().equals("China"))
    .reduce((stud1, stud2) -> stud1.yearOfBirth() > stud2.yearOfBirth() ? stud1 : stud2)

nameOfYoungestChinese.ifPresent(student -> System.out.println(student.name()));

```

Esercizio 4

Si considerino le seguenti classi Java:

```
class Person {
    protected String name;
    public Person(String name) { this.name = name; }
    public void drive(Car c) {
        System.out.println(name+" driving "+c.model());
    }
}
class Pilot extends Person {
    public Pilot(String name) { super(name); }
    public void drive(RacingCar c) {
        System.out.println("Pilot "+name+" driving "+c.model());
    }
    public void drive(RacingCar c, Pilot coPilot) {
        System.out.println("Pilot "+name+" driving "+c.model()+" with "+coPilot.name);
    }
}
class RallyPilot extends Pilot {
    public RallyPilot(String name) { super(name); }
    public void drive(Car c) {
        System.out.println("Rally pilot "+name+" driving "+c.model());
    }
}

class Car {
    public String model() { return "a car"; }
}
class RacingCar extends Car {
    public String model() { return "a racing car"; }
}
```

e il seguente frammento di codice le cui righe sono state numerate per riferimento:

```
1  Person p = new Pilot("Andrea");
2  Pilot pp = new RallyPilot("Laura");
3  Car c[] = new Car[2];
4  c[0] = new Car();
5  c[1] = new RacingCar();
6  RacingCar rc = new RacingCar();
7
8  p.drive(c[0]);
9  p.drive(c[1]);
10 p.drive(rc);
11 p.drive(rc, pp);
12 pp.drive(c[0]);
13 pp.drive(c[1]);
14 pp.drive(rc);
15 pp.drive((RacingCar)c[0], pp);
16 pp.drive(c[1], pp);
17 pp.drive((RacingCar)c[1], pp);
18 pp.drive(rc, pp);
```

Domanda a)

Scrivere il numero di riga delle istruzioni (se ne esistono) che generano un errore in compilazione o a run-time, chiarendo in quale momento (compile-time vs. run-time) viene sollevato l'errore e spiegandone brevemente le ragioni.

Soluzione

La riga 11 non compila: p è staticamente di classe Person e tale classe non possiede il metodo invocato.

La riga 15 restituisce una ClassCastException a tempo di esecuzione, c[0] è dinamicamente di classe Car.

La riga 16 non compila: `c[1]` è staticamente di classe `Car` e il metodo `drive` con due parametri della classe `Pilot` vuole come primo parametro una `RacingCar`.

Domanda b)

Supponendo di eliminare le eventuali righe che generano errori indicate sopra, si scriva, per ogni riga rimanente, il valore stampato in uscita.

Soluzione

```
Andrea driving a car
Andrea driving a racing car
Andrea driving a racing car
Rally pilot Laura driving a car
Rally pilot Laura driving a racing car
Pilot Laura driving a racing car
Pilot Laura driving a racing car with Laura
Pilot Laura driving a racing car with Laura
```

Politecnico di Milano

Ingegneria del Software - a.a. 2008/09

Appello del 9 Febbraio 2010

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi □, Ghezzi □, San Pietro □

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 7)

Scriverela specifica JML del seguente metodo statico:

```
public static ArrayList<Integer>
sommeElementi(ArrayList<Integer> a)
```

Il metodo riceve come parametro un array list di interi, non vuoto e eventualmente contenente elementi ripetuti. Il metodo restituisce un altro array list che contiene:

- in prima posizione la somma di tutti gli elementi di a
- dalla seconda posizione in poi, gli elementi di a sommando eventuali ripetizioni. Ad esempio, se a contenesse: 5, 3, 7, 11, 3, 7, 3, l'array restituito dovrebbe contenere: 36, 5, 9, 14, 11 dove 9 è la somma delle tre occorrenze del numero 3 e 14 corrisponde ai due 7.

```
/* @requires a != null && a.size() > 0; */
```

```
/* @ensures \result != null && \result.size() > 0 &&
```

Primo elemento di \result

```
    @ \result.get(0) == (\sum int i; i >= 0 && i < a.size(); a.get(i)) &&
```

Tutti gli elementi di \result hanno senso

```
    @ (\forall int i; i > 0 && i < \result.size());
    @ (\exists int j; j >= 0 && j < a.size());
    @.....\result.get(i) == a.get(j) * (\count int k; k >= 0 && k
    <a.size(); a.get(k) == a.get(j)))
```

Tutti gli elementi di a sono parte di \result

```
    @ (\forall int i; i >= 0 && i < a.size());
        @ (\exists int j; j > 0 && j < \result.size());
        @.....a.get(i) == \result.get(j) / (\count int k; k >= 0 && k
        <a.size(); a.get(k) == a.get(i)))
```

```
public static ArrayList<Integer>
sommeElementi(ArrayList<Integer> a)
```

Esercizio 2 (punti 13)

Un editor di figure geometriche consente di disegnare figure composte da nodi e da archi orientati che connettono i nodi. Ciascun nodo ha le coordinate x,y del suo centro.

Un arco orientato connette due nodi, detti sorgente e destinazione.

La specifica JML prevede due classi, Nodo e Arco, così definite.

```
public class Nodo {  
    //@ensures (* costruisce un Nodo di centro (x,y)*);  
    public Nodo (int x, int y);  
    //@ensures (*restituisce coord. x *);  
    public /*@ pure @*/int getX();  
    //@ensures (*restituisce coord. y *);  
    public /*@ pure @*/int getY();  
    //@ensures getX()==\old(getX())+dx && getY()==\old(getY())  
    +dy;  
    public void muovi(int dx, int dy)  
}  
  
public /*@ pure @*/ class Arco {  
    //@ensures (*costruisce un arco orientato da start a end*);  
    public Arco(Nodo start, Nodo end);  
    //@ensures (* restituisce l'origine dell'arco *);  
    public Nodo origine();  
    //@ensures (* restituisce la destinazione dell'arco *);  
    public Nodo destinazione();  
}
```

a) Si vuole descrivere, tramite una specifica, una classe Figura, che è costituita da un insieme di nodi e da un insieme di archi, ciascuno dei quali collega una coppia di nodi (non necessariamente distinti) della Figura. Completare, negli spazi bianchi, la specifica data di seguito. Si noti che:

- a1) Il metodo addNodo aggiunge un nodo n, a patto che n non abbia entrambe le coordinate uguali a quelle di nodi già presenti nella Figura.
- a2) Il metodo addArco(n1, n2) aggiunge, qualora non vi sia già un arco da n1 a n2, un nuovo arco da n1 a n2, e lo restituisce come risultato.
- a3) Il metodo arcoSeEsiste(sorg,dest) restituisce l'arco orientato che va dal nodo sorg al nodo dest, se l'arco esiste nella Figura, altrimenti restituisce il valore convenzionale null.
- a) Il numero massimo di nodi che si possono aggiungere a una figura è fissato al momento della costruzione della Figura (inizialmente vuota).

E' possibile aggiungere, se ritenuto utile, opportune eccezioni ai metodi.

```
public class Figura {  
    /*@ensures (* restituisce la lista dei nodi della figura, in un ordine  
qualsiasi, senza ripetizioni*);  
    @*/  
    public ArrayList<Nodo> nodi()  
    /*@ensures (* restituisce la lista degli archi della figura, in un ordine  
qualsiasi, senza ripetizioni*);  
    @*/  
    public ArrayList<Arco> archi()  
    /*@ requires maxsize>0;  
        ensures (*Costruisce una Figura vuota che può contenere fino  
a                                maxSize  
nodi*); @*/  
    public Figura(int maxsize);  
    /*@ requires !(\exists Nodo n1; n!=n1 && nodi().contains(n1)  
           n1.getX()==n.getX() && n.getY()==n1.getY());  
        ensures \old(archi()) == archi() && nodi().contains(n) &&  
           (\forall Nodo n1; n1!=n;  
           \old(nodi().contains(n1)) <=> nodi().contains(n1));  
    @*/  
  
    public void addNodo(Nodo n);  
  
    //@@requires n1!=null &&n2!=null;  
    //@@ensures      \old(nodi()) == nodi() &&  
        \result.origine()==n1 &&  
        \result.destinazione() = n2 &&  
        archi().contains(\result) &&  
        (\forall Arco a; a!=\result;  
        \old(archi().contains(a)) <=> archi().contains(a));  
    public Arco addArco(Nodo n1, Nodo n2);  
  
    //@@requires a!=null;  
    //@@ensures \old(nodi()) == nodi() && !archi().contains(a) &&  
        (\forall Arco a1; a1!=a;  
        \old(archi().contains(a1)) <=> archi().contains(a1));  
    public void deleteArco(Arco a);  
  
    //@@requires sorg!=null && dest != null;  
    //@@ensures (\exists Arco a; archi().contains(a));
```

```

        a.sorgente() == sorg && a.destinazione()==dest)
    ? \result.sorgente()==sorg && \result().destinazione()==dest)
    : \result==null;
public /* pure */ Arco arcoSeEsiste (Nodo sorg, Nodo dest);
}

```

- b) Descrivere un opportuno public invariant per la specifica di Figura. Argomentare che tale invariante è verificato dalla specifica o, nel caso non lo sia, spiegarne il motivo e mostrare che cosa dovrebbe essere modificato nelle specifiche date affinché sia verificato

L'invariante consta di due parti: nella prima si afferma che non vi sono due nodi che hanno le stesse coordinate; nella seconda che non vi sono due archi incidenti sugli stessi nodi.

```

public invariant
(\forallall Nodo n; nodi().contains(n);
 !(\exists Nodo n1; n!=n1 && nodi().contains(n1);
      n1.getX()==n.getX() && n.getY()==n1.getY())) &&
(\forallall Arco a; archi.contains(a);
  nodi().contains(a.origine()) && nodi().contains(a.destinazione()) &&
  !(exists Arco a1; a1!=a && archi().contains(a1);
      a1.origine() == a.origine() && a1.destinazione()==a.destinazione())))

```

La specifica è organizzata in modo tale da verificare l'invariante, grazie alle postcondizioni di addNodo e di AddArco e deleteArco, ma solo sotto l'ipotesi che la classe Nodo sia immutabile. Tuttavia, la classe Nodo non è mutabile ed è quindi possibile modificare le coordinate di un nodo già inserito in Figura, potendo quindi rendere falsa la prima parte dell'invariante. Basterebbe quindi rendere immutabile la classe Nodo.

c) E' data la seguente implementazione (parziale) di Figura.

```
public class Figura {  
    private ArrayList<Nodo> listaNodi;  
    private String [][] matriceArchi;  
    private int numNodi;  
    public Figura(int maxsize) {  
        listaNodi = new ArrayList<Nodo>();  
        matriceArchi = new Arco [maxSize] [maxSize] ;  
    }  
    public void addNodo(Nodo n){  
        if (numNodi == matriceArchi.size()) return;  
        for (int i == 0; i< listaNodi.size() {  
            n1= listaNodi.get(i);  
            if (n1.getX() == n.getX() && n1.getY()== n.getY())  
return;  
        }  
        listaNodi.add(n);  
        numNodi++;  
    }  
    public Arco addArco(Nodo n1, Nodo n2){  
        int sorg= listaNodi.indexOf(n1);  
        int dest = listaNodi.indexOf(n2);  
        matriceArchi [sorg][dest] = new Arco(n1,n2);  
        return matriceArchi [sorg][dest];  
    }  
    public deleteArco(Arco a) {  
        int sorg= listaNodi.indexOf(a.sorgente());  
        int dest = listaNodi.indexOf(a.destinazione());  
        matriceArchi [sorg][dest] = null;  
    }  
}
```

Scrivere l'invariante di rappresentazione della classe Figura e implementare il metodo arcoSeEsiste(...).

```
private invariant listaNodi != null && matriceArchi!= null &&  
0 <= numNodi <= matrice Archi.size() &&  
(\forallall int i; 0<=i && i<matrice Archi.size();  
    matrice Archi[i].size()== matriceArchi.size()) &&  
(\forallall int i; 0<=i && i<numNodi;  
    (\forallall int j; 0<=j && j<numNodi; matriceArchi [i][j] !=null  
==>  
        listaNodi.get(i) == matriceArchi[i][j].origine() &&  
        listaNodi.get(j) == matriceArchi[i][j].destinazione()));  
  
public Arco arcoSeEsiste(nodo n1, nodo n2) {  
    i = listaNodi.indexOf(n1); j = listaNodi.indexOf(n2);  
    if (i== -1 || j== -1) return null;  
    return matriceArchi[i][j];
```

}

- d) Si consideri l'implementazione parziale di una classe PoligonaleChiusa, riportata qui sotto. Una PoligonaleChiusa è una sequenza di almeno 2 nodi e altrettanti archi, che ritorna sempre al nodo iniziale. La figura seguente è un esempio con 5 nodi e 5 archi, in cui il nodo iniziale è marcato con un pallino.

o

```

public PoligonaleChiusa extends Figura {
    private Nodo primoNodo;
    private Nodo ultimoNodo;
    public PoligonaleChiusa(Nodo n1, Nodo n2, int maxSize)
throws PoligonaleErrataException {
        super(maxSize);
        super.addNodo(n1);
        super.addNodo(n2);
        if (!nodi().contains(n1) || !nodi().contains(n2)) throw
new PoligonaleErrataException;
        super.addArco(new Arco(n1,n2));
        super.addArco(new Arco(n2,n1));
        primoNodo = n1;
        ultimoNodo = n2;
    }
//ogni nodo nuovo è aggiunto come ultimo nodo e collegato al
primo:
    public void addNodo(Nodo n) {
        super.addNodo(n);
        if (!nodi().contains(n)) return;
        super.deleteArco(arcoSeEsiste(ultimoNodo, primoNodo));
        super.addArco(new Arco(ultimoNodo, n));
        super.addArco(new Arco(n, primoNodo));
        ultimoNodo = n;
    }
    public Nodo getPrimoNodo() {return primoNodo;}
    public Nodo getLastNodo() {return ultimoNodo;}
...
}

```

In base all'implementazione data rispondere alla seguente domande:

d1) Il costruttore PoligonaleChiusa(...) deve prevedere delle precondizioni? Se sì, quali?

E' sufficiente: $n1 \neq \text{null} \&\& n2 \neq \text{null} \&\& \text{maxSize} \geq 2$

d2) E' possibile specificare la classe PoligonaleChiusa, completando eventualmente le parti mancanti, in modo che il principio di sostituzione sia verificato? Motivare accuratamente la risposta.

No, perché ad esempio addArco non può aggiungere un arco fra due nodi qualora questo non esista: è vietato dalla definizione di PoligonaleChiusa (ossia dal suo invarianto pubblico). Quindi si violerebbe o la regola dei metodi (se addArco "si rifiutasse" di aggiungere l'arco) o la regola delle proprietà (se addArco aggiungesse l'arco la proprietà di essere una PoligonaleChiusa non sarebbe verificata).

Analogamente con il metodo addNodo: nel momento in cui si aggiunge un nodo, si deve anche aggiungere anche un arco, eliminando nel contempo il vecchio arco dall'ultimo nodo al primo. La postcondizione di addNodo deve quindi modificare l'insieme degli archi, violando la postcondizione originale. Il costruttore invece va bene, in quanto i costruttori non si ereditano e la figura definita dal costruttore verifica l'invariante di Figura.

Per completezza, riportiamo qui anche l'invariante pubblico di PoligonaleChiusa:

```
/*@also public invariant archi().size() = nodi().size() &&
    arcoSeEsiste(ultimoNodo,primoNodo) != null &&
    (\forall Nodo n; nodi().contains(n); //ogni nodo ha 1 solo arco entrante e
    1 uscente:
        (\#num_of Nodo n1; nodi().contains(n1); ArcoSeEsiste(n,n1)!=null)
    ==1) &&
        (\#num_of Nodo n2; nodi().contains(n2); ArcoSeEsiste(n2,n)!=null)
    ==1));
```

```
@*/
```


Esercizio 3 (punti 5)

Un'azienda è divisa in diverse sedi, dove lavorano dipendenti sia tecnici sia amministrativi. Ogni lavoratore percepisce uno stipendio, ma può anche venir licenziato, dare le dimissioni, oppure chiedere di cambiare sede. Gli amministrativi possono avere avanzamenti di carriera, mentre per i tecnici questo significa un cambiamento di mansione. Le due tipologie di dipendenti adottano modalità diverse per il calcolo dello stipendio. Ogni sede ha un budget proprio e provvede agli stipendi dei propri dipendenti in modo autonomo; l'azienda vuole solo poter sapere il numero totale dei propri dipendenti. Si definisca un opportuno diagramma delle classi per rappresentare l'organizzazione appena descritta evidenziando gli attributi e i metodi necessari.



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da

Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Appello di Ingegneria del Software

13 Luglio 2004

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h15m.
6. Punteggio totale a disposizione: 26/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio 1 (punti 10)

- Utilizzando opportune formule che definiscono le pre e post-condizioni, si fornisca la specifica di un'astrazione procedurale che in input riceve due array x e y di interi.

a) L'astrazione restituisce true se x è una permutazione di y, false altrimenti, sotto l'ipotesi che ne' x ne' y abbiano elementi duplicati. Se almeno uno dei due array è null, allora viene lanciata l'eccezione unchecked NullPointerException.

Es. x = [1,5, 2], y = [2, 5, 1] risultato = true

x = [1,5, 2], y = [2, 5, 0] risultato = false

x = [1,5, 2], y = [2, 5, 2] risultato = indefinito

```
public static boolean risultato(int[]x, int[]y) {
```

- In base alla specifica fornita, si definiscano cinque dati numerici significativi a supporto del test funzionale (black-box testing) in base ai criteri di copertura delle combinazioni proposizionali e dei valori limite, specificando inoltre i risultati attesi per ciascun dato.

- Si consideri la seguente implementazione del caso (a) (*non necessariamente corretta* rispetto alla specifica). Si identifichi il minimo numero di dati di test, specificandone valore numerico e risultato atteso, per garantire la copertura al 100% del codice secondo il criterio delle diramazioni.

```
public static boolean risultato(int[]x, int[]y) {
    boolean temp=true;
    for (int i=0; i < x.size() && temp; ++i) {
        temp= false;
        for (j=0; j<y.size(); ++j)
            if (x[i] ==y[j] && !temp) temp = true;
    }
    return temp;
}
```

Esercizio 2 (punti 16)

Si consideri la seguente specifica dell'astrazione sui dati PilaDoppiaTesta.

```
public class PilaDoppiaTesta {  
    /*OVERVIEW: Tipo mutabile, che rappresenta una pila con due estremità (sinistra e destra), che consente l'inserimento sia in testa a sinistra che in coda a destra. L'estrazione avviene esclusivamente dalla testa di sinistra, mantenendo l'ordine di inserzione. Gli elementi inseriti sono di tipo Object, ma diversi da null.  
    Un oggetto tipico è [o1, o2, ..., on], dove o1 è la testa sinistra della pila, on è la testa destra.  
*/  
  
    public PilaDoppiaTesta ()  
        //EFFECTS: costruisce la pila vuota  
    public void pushLeft (Object x)  
        //REQUIRES: x<>null  
        //EFFECTS: posto this = [o1, o2, ..., on], n>=0, this_post = [x, o1, o2, ..., on]  
        //MODIFIES this  
  
    public void pushRight (Object x)  
        //REQUIRES: x<>null  
        //EFFECTS: posto this = [o1, o2, ..., on], n>=0, this_post = [o1, o2, ..., on, x]  
        //MODIFIES this  
  
    public Object pop() throws EmptyException;  
        //EFFECTS: if this è vuota throw EmptyException  
        //           else, posto this = [o1, o2, ..., on], this_post = [o2, ..., on] e o1 è restituito  
        //MODIFIES this  
  
    public Iterator LeftToRight();  
        //EFFECTS: restituisce un generatore che itera su tutti gli elementi di this,  
        //           nell'ordine da sinistra a destra  
        //REQUIRES: this non può essere modificato mentre il generatore è in uso.  
  
    public Iterator RightToLeft();  
        //EFFECTS: restituisce un generatore che itera su tutti gli elementi di this,  
        //           nell'ordine da destra a sinistra  
        //REQUIRES: this non può essere modificato mentre il generatore è in uso.  
  
    public int size(); //EFFECTS: restituisce il numero di elementi nella pila  
}
```

(continua esercizio 2)

- a) Si specifichi, come sottoclasse di PilaDoppiaTesta, una variante in cui il metodo PushLeft lancia un'eccezione checked di tipo InvalidObject qualora sia passato un argomento x che vale null. La variante rispetta il principio di sostituzione?

```
public VariantePilaDoppiaTesta extends PilaDoppiaTesta {  
    public void pushLeft (Object x)
```

(continua esercizio 2)

- b) Si specifichi, come sottoclasse di PilaDoppiaTesta, un ADT DoppiaPila che possiede un'operazione supplementare popRight(), che è come pop ma estraе e restituisce l'elemento più a destra nella pila.
DoppiaPila rispetta il principio di sostituzione?

```
public class DoppiaPila extends PilaDoppiaTesta {
```

(continua esercizio 2)

- c) Si completi la seguente implementazione della classe PilaDoppiaTesta, specificando AF e RI. Note: Il metodo iterator() dei Vector restituisce un generatore agli elementi a partire dalla posizione 0. Non vi è un simile metodo predefinito che restituisce un generatore dall'ultima posizione alla posizione 0: il metodo RightToLeft() va quindi implementato ad hoc, definendo anche la classe del generatore.

L'implementazione è corretta rispetto alla specifica? Giustificare la risposta.

.....

}

(continua esercizio 2)

- d) Si desidera ora utilizzare la classe PilaDoppiaTesta per implementare una PilaSenzaSimili di Object, che si comporta come una pila qualunque ma in cui non è possibile inserire oggetti “simili” a oggetti già presenti in Pila. Per stabilire se due oggetti sono simili, la Pila utilizza l’approccio related subtype, tramite il metodo simili di un’interfaccia Verificatore. Il Verificatore da utilizzare è stabilito alla chiamata del costruttore di PilaSenzaSimili. Le specifiche di PilaSenzaSimili e del Verificatore sono:

```
public class PilaSenzaSimili {  
    //OVERVIEW: una Pila di Object in cui non si possono inserire elementi “simili” a quelli presenti nella pila.  
    public PilaSenzaSimili (Verificatore v)  
        // REQUIRES: v != null  
        // EFFECTS: costruisce una pila vuota  
        public Object pop()//EFFECTS: restituisce ed elimina da this l’elemento in cima  
        public void push(Object o) throws SimilarException  
            //EFFECTS: if o è simile a uno qualunque degli elementi già presenti nella Pila, lancia SimilarException,  
            //else inserisce o in cima alla pila; v  
}
```

```
public interface Verificatore {  
    public boolean simili(Object o1, Object o2);  
    //EFFECTS: return true se o1 e o2 sono simili, false in tutti gli altri casi.  
    //Devono valere le proprietà riflessiva e simmetrica:  
    // se o1.equals(o2) allora deve valere simili(o1,o2);  
    // se simili(o1,o2) allora simili(o2,o1).  
}
```

Completare l’implementazione seguente della classe PilaSenzaSimili.

```
public class PilaSenzaSimili {  
    //AF(c) = .....  
    //RI(c) = .....  
    private PilaDoppiaTesta p;  
    private Verificatore v;  
    public PilaSenzaSimili(Verificatore v) (p = new PilaDoppiaTesta(); this.v = v;}  
    public Object pop() {return p.pop();}  
    public void push(Object o) throws SimilarException {  
        .....  
        .....  
        .....  
        .....  
        p.pushLeft(o);  
    }  
}
```

Considerare poi la classe Double, corrispondente al tipo primitivo double. Si codifichi una classe RifiutaDoubleVicini che implementi l’interfaccia Verificatore e permetta di inserire oggetti nella Pila con il seguente criterio di similitudine: due oggetti x1 e x2 sono simili se entrambi x1 e x2 sono Double e inoltre vale $\text{Math.abs}(x1-x2) < 0.00001$, oppure (nel caso in cui non siano entrambi Double) se x1 e x2 sono “equals”. Mostrare un esempio di costruzione di una pila.



Politecnico di Milano

Anno accademico 2012-2013

Ingegneria del Software – Appello del 05 Settembre 2013

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1 (punti 16)

Si consideri un'applicazione per il prestito di libri. Ogni libro è presente in diverse copie; ogni utente iscritto al sistema può richiedere il prestito di uno dei libri presenti a catalogo, con il vincolo che al massimo può detenere tre libri complessivamente e comunque non più di una copia per ogni libro.

Si considerino le classi seguenti:

```
public class Libro {  
    ...  
  
    public int copie()  
        // ritorna il numero di copie esistenti del libro, in prestito o disponibili.  
  
    public ArrayList<Prestito> prestitiAttivi()  
        // ritorna l'insieme di prestiti attivi  
  
    public ArrayList<Utente> utentiInAttesa()  
        // ritorna la lista d'attesa in ordine cronologico  
  
    public Prestito richiestaPrestito(Utente u) throws CopieNonDisponibiliException, UtenteNonAbilitatoException  
        // ritorna la richiesta di un prestito  
  
    public void restituzionePrestito(Utente u) throws UtenteNonHaPresoInPrestitoException;  
        // permette la restituzione della copia del libro prestata ad un utente.  
}  
  
public class Prestito {  
    ...  
  
    public Libro libro()  
        // ritorna il libro chiesto in prestito  
  
    public Utente utente()  
        // ritorna l'utente che ha chiesto il prestito.  
}
```

Il metodo `richiestaPrestito` è definito in questo modo: Se esiste almeno una copia disponibile, e l'utente può richiederne il prestito non avendo superato il massimo, il prestito viene generato e restituito, si decrementa il numero di copie disponibili e l'utente può iniziare la lettura; se invece l'utente non può richiederne il prestito viene generata l'eccezione `UtenteNonAbilitatoException`; Se infine non esiste una copia disponibile, viene generata l'eccezione `CopieNonDisponibiliException`. In quest'ultimo caso, l'utente viene messo in lista d'attesa (gestita con una semplice politica *first-in first-out*) e sarà poi avvisato non appena una copia dovesse essere disponibile.

Il metodo `restituzionePrestito` corrisponde alla restituzione della copia del Libro da parte di un utente; lancia l'eccezione `UtenteNonHaPresoInPrestitoException` qualora l'utente non risulti fra i detentori di una copia.

- Come modifichereste le classi `Libro` e `Prestito` per notificare il primo utente in lista d'attesa non appena una copia del libro fosse restituita?

b) Si definiscano in JML le specifiche per i metodi `richiestaPrestito` e `restituzionePrestito`

- c) Si supponga che esista una classe Catalogo, che contiene tutti i libri della bilioteca.

```
public class Catalogo {  
    ...  
  
    private ArrayList<Libro> elenco; //contiene tutti e soli i libri.  
  
    public Iterator<Libro> disponibili()  
        //restituisce un iteratore che itera solo sui libri disponibili, in un ordine qualsiasi  
}
```

Si implementi l'iteratore `disponibili()`, definendo anche la classe che implementa l'iteratore e i suoi metodi. La definizione di opportuni metodi *helper* potrebbe semplificare la definizione dell'iteratore.

Esercizio 2 (punti 6)

Notoriamente, il cibo viene mangiato. Alcuni cibi vengono mangiati solo crudi, altri solo dopo essere stati cotti, altri ancora potrebbero essere mangiati sia crudi che cotti. Volendo definire un'opportuna gerarchia di cibi, quali classi definireste?

1. È possibile definire una gerarchia di ereditarietà che rispetti il principio di sostituibilità di Liskov?
 2. Se voleste definire una classe *astratta* quale gerarchia definireste? Quali metodi potreste implementare nella classe astratta?
 3. E se voleste definire un’interfaccia?

Esercizio 3 (punti 4)

Si consideri il seguente frammento di codice Java e si definisca l'output prodotto, motivando brevemente la risposta.:

```
public abstract class Figura {
    public void componi(Figura f) {
        System.out.println(this + "/" + f);
    }
}

public class Quadrato extends Figura {
    public String toString() {
        return("Quadrato");
    }

    public void componi(Quadrato f) {
        System.out.println(this + "+" + f);
    }
}

public class Rettangolo extends Quadrato {
    public String toString() {
        return("Rettangolo");
    }

    public void componi(Rettangolo f) {
        System.out.println(this + "/+" + f);
    }
}

public class Cerchio extends Figura {
    public String toString() {
        return("Cerchio");
    }
}

public class Ellisso extends Cerchio {
    public String toString() {
        return("Ellisso");
    }
}

public class Test {
    public static void main(String[] args) {
        Figura f;
        Quadrato q1, q2;
        Rettangolo r;
        Cerchio c;
        f = new Rettangolo();
        q1 = new Quadrato();
        q2 = new Rettangolo();
        r = new Rettangolo();
        c = new Ellisso();
        q1.componi(r);
        q1.componi(q2);
        q1.componi(c);
        q1.componi(f);
        r.componi(q2);
    }
}
```

Esercizio 4 (punti 6)

Si consideri il seguente frammento di codice Java:

```
public static int f(int i, int d) {
    int c = 1;

    while (c < 3) {
        if (i > c && d != 0) i = c/d;
        else return i;
        c++;
    }
    return i;
}
```

1. Si definisca il diagramma del flusso di controllo.
2. Si definisca l'insieme minimo di casi di test per coprire: (a) le istruzioni, (b) le decisioni (*branch*) e (c) i cammini.

Appello 24 luglio 2017



Politecnico di Milano
Anno accademico 2016-2017

Ingegneria del Software

Cognome:

LAUREANDO

Nome:

Matricola:

Sezione (segnarne una): Cugola

Mottola

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica di un'interfaccia Deque. Una Deque è una sequenza di elementi che consente inserimento ed eliminazione sia all'inizio che alla fine della sequenza. La specifica dell'interfaccia e' data nel seguito attraverso semplici commenti testuali.

```
public interface Deque<E> extends Collection<E>{
    /*OVERVIEW: Collezione mutabile di oggetti, di tipo E,
     * organizzati in una sequenza x1, x2, ... xn
     * Rimozione e inserimento di elementi possono avvenire
     * solo in prima o ultima posizione.
     */

    //inizializza this alla Deque vuota;
    public Deque()

    //osservatori:

    //restituisce il numero di elementi di this
    public /*@ pure @*/ int size();

    // restituisce, senza rimuoverlo, il primo elemento,
    //ma lancia eccezione se this e' vuoto;
    public /*@ pure @*/ E getFirst() throws NoSuchElementException;

    // restituisce, senza rimuoverlo, l'ultimo elemento,
    //ma lancia eccezione se this e' vuoto;
    public /*@ pure @*/ E getLast() throws NoSuchElementException;

    //mutators:

    //aggiunge x in prima posizione, ma lancia eccezione se x e' null;
    public void addFirst(E x) throws NullPointerException;

    //aggiunge x in ultima posizione, ma lancia eccezione se x e' null;
    public void addLast(E x) throws NullPointerException;

    //elimina il primo elemento, ma lancia eccezione se this e' vuoto
    public void removeFirst() throws NoSuchElementException;

    //elimina l'ultimo elemento, ma lancia eccezione se this e' vuoto
    public void removeLast() throws NoSuchElementException;

}
```

Si rammenta che l'interfaccia Collection prevede gli usuali metodi booleani contains(Object o), containsAll(Collection<?> c) e inoltre il metodo toArray() qui di seguito specificato:

```
//restituisce un array che contiene tutti gli elementi di this
//nella sequenza dal primo all'ultimo
public /*@ pure @*/ Object[] toArray() .
```

Domanda a

Considerati come dati i metodi puri, specificare in JML i metodi addLast, removeFirst.

Soluzione

```
/*@ensures x!=null && getLast()==x && size()==\old(size()+1) &&
```

```

//@\forall int i; 0<=i && i<size()-1;
//@\this.toArray()[i]==\old(this.toArray()[i]);
//@signals (NullPointerException e) x==null;
public void addLast(E x) throws NullPointerException

//@ensures x!=null && getFirst()==this.toArray()[0] && size()==\old(size()-1) &&
//@\forall int i; 0<=i && i<size();
//@\this.toArray()[i]==\old(this.toArray()[i+1]);
//@signals (NoSuchElementException e) \old(size()==0);
public void removeFirst() throws NoSuchElementException

```

Domanda b

Si consideri la seguente classe `ArrayDeque`, di cui nel seguito è descritta solo un parte. L'idea dell'implementazione è di avere due indici, `head` e `tail`, che rappresentano la posizione, rispettivamente, del primo e dell'ultimo elemento all'interno di un array `elements`. Gli indici si muovono in modo circolare, con `head` che viene decrementato quando si aggiunge un elemento in testa (e incrementato quando si rimuove) mentre `tail` è incrementato quando si aggiunge un elemento in coda (e decrementato quando si rimuove).

Il campo `head` rappresenta la posizione del primo elemento in coda. I successivi elementi sono “alla sua destra” (considerando la matematica in modulo). Il campo `tail` rappresenta invece la posizione del primo elemento vuoto alla fine della coda. I precedenti elementi sono “alla sua sinistra” (considerando la matematica in modulo). Quindi se `size()>0` allora `elements[head]` è il primo elemento, `elements[tail-1]` è l'ultimo.

Quando, dopo un inserimento in testa o in coda, `head` e `tail` diventano uguali, la coda è piena: la dimensione dell'array viene allora raddoppiata, tramite la chiamata di un metodo privato `doubleCapacity()` che costruisce un array di capacità doppia, vi ricopia tutti gli elementi e poi pone `elements` uguale al nuovo array. Il metodo, al termine dell'esecuzione, garantisce che valga la condizione

head==0 && tail==\old(elements.length)
ossia che la `head` sia inizializzata a 0 e la `tail` sia portata al primo elemento libero del nuovo array.

```

public class ArrayDeque<E> implements Deque<E> {
    // Overview: Implementazione basata su array flessibili dell'interfaccia Deque.
    // Si usa un array circolare, con due indici, head e tail

    private E[] elements;
    private int head; // indice del primo elemento
    private int tail; // indice dell'ultimo elemento
    private int num; // numero degli elementi in coda
    //costruttore
    public ArrayDeque() {
        elements = (E[]) new Object[16];
        head = tail = num = 0;
    }

    public void addFirst(E e) {
        if (e == null) throw new NullPointerException();
        head--;
        if (head<0) head = elements.length -1;
        elements[head] = e;
        if (head == tail) doubleCapacity();
        num++;
    }

    public void addLast(E e) {
        if (e == null) throw new NullPointerException();
        elements[tail] = e;
        tail = (tail + 1) % elements.length;
        if (tail == head) doubleCapacity();
        num++;
    }
}

```

```

}

public E getFirst() {
    if (num == 0) throw new NoSuchElementException();
    return elements[head];
}

public E removeFirst() {
    if (num == 0) throw new NoSuchElementException();
    E result = elements[head];
    elements[head]=null;
    head = (head + 1) % elements.length;
    num--;
    return result;
}

public int size() {
    return num;
}
...
}

```

Si descrivano l'invariante di rappresentazione e la funzione di astrazione della classe `ArrayDeque`, ad esempio tramite due opportuni private invariant.

Soluzione

L'invariante di rappresentazione specifica le condizioni sull'array `elements`, sul range tra `head` e `tail`, e infine specifica il numero di elementi `num` nei due casi: quando `head < tail` (gli elementi sono tutti compresi fra la posizione `head` e la posizione `tail - 1`) e quando `head > tail` (gli elementi sono dalla posizione `head` alla fine dell'array, e fra 0 e la posizione `tail - 1`). Inoltre, tutti gli elementi della coda sono diversi da null.

```

private invariant
elements!=null && 0<=tail && tail<elements.length &&
0<=head && head<elements.length &&
(head==tail <==> num==0) &&
(head<tail ==> num==tail-head) &&
(head>tail ==> num==(elements.length - head) + tail &&
(\forallall i; 0<=i && i<num; elements[(head+i) % elements.length]!=null);

```

La funzione di astrazione può essere rappresentata semplicemente specificando quali sono gli elementi restituiti dal metodo `toArray` e quale è la dimensione della Deque:

```

private invariant
size()==num && this.toArray().length==num &&
(\forallall i; 0<=i && i<num; this.toArray[i] == (head+i) % elements.length);

```

o anche implementando (con analogo meccanismo) il metodo `toString`.

Esercizio 2

Si deve progettare un'applicazione di posta elettronica. Il programma mantiene l'insieme dei messaggi ricevuti dall'utente, organizzato in un insieme di cartelle. Ogni cartella ha un nome, un ordine di visualizzazione dei messaggi (per data, per titolo o per mittente) e contiene messaggi o altre cartelle. È possibile aggiungere messaggi ad una cartella, rimuoverli, o spostarli in un'altra cartella. Ogni messaggio ha una serie di campi di informazione (mittente, titolo, lista dei destinatari, lista degli utenti che devono ricevere una copia –campo CC–, data di invio, codice identificativo), un testo e, optionalmente, una serie di allegati. Ogni allegato può

essere un testo, un'immagine, un programma eseguibile o un generico allegato binario. Ogni allegato può essere mostrato oppure salvato su file. Mostrare un allegato vuol dire mostrarlo a video nel caso di testo o immagini, oppure eseguirlo nel caso si tratti di un programma. Gli allegati binari generici non possono essere mostrati a video. Un messaggio può essere mostrato (il che equivale a mostrare i campi di informazione ed il testo del messaggio), salvato su file, oppure ritrasmesso ad un altro insieme di utenti. Infine è possibile rispondere al messaggio.

Domanda A.

Si progetti, in UML, l'insieme di classi che descrivono la struttura dati di tale programma di posta.

Domanda B.

Si fornisca il diagramma delle sequenze per lo scenario che prevede l'arrivo di un nuovo messaggio con una immagine allegata, l'inserimento nella cartella INBOX, la visualizzazione del messaggio e del relativo allegato e il successivo spostamento nella cartella Amici.

Domanda C.

Si estenda il diagramma UML iniziale per considerare il caso in cui il programma di posta possa gestire anche messaggi SMS, con le relative caratteristiche e differenze rispetto ai messaggi di posta, supponendo però che anche gli SMS possano essere classificati dall'utente inserendoli in opportune cartelle (distinte da quelle dei messaggi di posta).

Soluzione

Esercizio 3

Si consideri la seguente classe Java:

```
public class DataStore {  
    private List<Integer> data1, data2;  
    public DataStore() {  
        data1 = new ArrayList<Integer>();  
        data2 = new ArrayList<Integer>();  
    }  
    public void add1(int elem) {  
        synchronized(data1) { data1.add(elem); }  
    }  
    public void remove1(int elem) {  
        synchronized(data1) { data1.remove(elem); }  
    }  
    public void add2(int elem) {  
        synchronized(data2) { data2.add(elem); }  
    }  
    public void remove2(int elem) {  
        synchronized(data2) { data2.remove(elem); }  
    }  
}
```

Domanda a

Le istanze della classe sono “thread safe”, ovvero i metodi possono essere invocati da thread diversi senza che questo crei problemi? In caso negativo, come dobbiamo modificare il codice per garantire il corretto accesso alle istanze della classe da parte di thread diversi?

Soluzione

La sincronizzazione è corretta.

Domanda b

Si aggiunga il metodo `void copia12()` che copia il contenuto della lista `data1` nella lista `data2`. Al termine dell'esecuzione le due liste devono contenere gli stessi elementi. Si garantisca la corretta sincronizzazione del metodo in presenza di thread multipli.

Soluzione

```
public void copia12() {  
    synchronized(data1) {  
        synchronized(data2) {  
            data2.clear();  
            data2.addAll(data1);  
        }  
    }  
}
```

Domanda c

Si aggiunga il metodo `void copia12Async()` che opera come il precedente metodo `copia12` ma esegue in un thread parallelo rispetto al chiamante.

Soluzione

```
public void copia12Async() {  
    new Thread() {  
        public void run() { copia12(); }  
    }.start();  
}
```

Esercizio 4

Si consideri il seguente metodo Java:

```
public int cercaSequenzaCrescentePiuLunga (int vett[]) {  
    int i = 0, cont=0, max = 0;  
    for (i = 1; i < vett.length; i++) {  
        if (vett[i]>vett[i-1]) {  
            cont++;  
            if (cont > max) {  
                max = cont;  
            }  
        } else {  
            cont = 0;  
        }  
    }  
    return max;  
}
```

1. Si definisca il diagramma di flusso di controllo.
2. Si fornisca un insieme di cardinalità minima di casi di test per la copertura delle decisioni.
3. Quale altro criterio di test viene soddisfatto dai casi di test individuati al punto precedente?

Soluzione

E' sufficiente un singolo dato di test, costituito da un vettore di 4 elementi come ad esempio:

[0, 1, 0, 1]

per esercitare tutte le decisioni (branch). La lunghezza minima del vettore deve essere 4, in quanto il ciclo deve essere eseguito almeno 3 volte per coprire i tre branch al suo interno.

Ovviamente la copertura delle decisioni implica sempre anche quella delle istruzioni. In questo caso, e' soddisfatto anche il criterio di copertura delle condizioni (in quanto tutte le espressioni condizionali sono atomiche).

Appello 5 febbraio 2016



Politecnico di Milano
Anno accademico 2014-2015

Ingegneria del Software

Cognome:

LAUREANDO

Nome:

Matricola:

Sezione (segnarne una): Baresi

Ghezzi

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la classe Mazzo di carte (da bridge). L'universo di tutte le possibili carte può contenere anche carte di altro tipo per esempio carte da UNO. Ogni mazzo contiene inizialmente 52 carte, 13 per ogni seme (cuori = 0, quadri = 1, fiori = 2 e picche = 3) e 4 per ogni valore (da 1 a 13). Si definisca in JML:

1. l'invariante privato della classe Mazzo, ipotizzando che esista una classe Carta e che il rep sia un `ArrayList<Carta> carteMazzo`. La classe Carta fornisce anche due metodi `int seme()` e `int valore()` che restituiscono il seme e il valore di ogni carta come interi (secondo la numerazione esposta sopra).
2. la specifica del metodo `pesca()` che restituisce una Carta a caso dal mazzo.
3. la specifica del metodo `mescola()` che cambia l'ordine delle carte.

Per rispondere ai punti 2 e 3, si supponga che Mazzo offra un metodo `carte()` che restituisce le carte rimaste nel mazzo come un `ArrayList<Carta>`.

Sempre usando JML, sarebbe poi possibile specificare che due, o più, invocazioni successive del metodo `pesca()` restituiscono carte diverse?

SOLUZIONE:

PUNTO 1;

```
//@private invariant
//@ carteMazzo!=null && carteMazzo.size()<=52
// nessuna carta e' null
//@ (\forall int i; i>=0&&i<carteMazzo.size(); carteMazzo.get(i)!=null)
// ogni carta contenuta deve essere di uno dei semi consentiti
//@ (\forall Carta c; carteMazzo.contains(c); c.seme()>=0 && c.seme()<4)
// ogni carta contenuta deve essere di uno dei valori consentiti
//@ (\forall Carta c; carteMazzo.contains(c); c.valore()>=0 && c.valore()<14)
// ogni carta appare al piu' una volta
//@ (\forall int i; i>=0 && i<carteMazzo.size(); \forall int j; j>i && j<carteMazzo.size();
//      (carteMazzo.get(i).seme() != carteMazzo.get(j).seme() ||
//       carteMazzo.get(i).valore() != carteMazzo.get(j).valore()))
```

PUNTO 2

```
//@requires carte.size()> 1
//@ensures \result!=null &&
// la carta c'era prima
//@ (\old(carte()).contains(\result)) &&
// la carta viene rimossa
//@ !(carte().contains(\result))
// le altre carte devono rimanere uguali
//@ (\forall Carta c; \old(carte()).contains(c) &&
//@ (c.valore() != \result.valore() ||
//@ c.seme() != \result.seme()); carte().contains(c)) &&
// non posso aggiungere carte duplicate (vincolo sulla dimensione)
//@ carte().size()==\old(carte().size()-1)
```

PUNTO 3

```
//@requires carte.size()> 2
//@ensures
// la dimensione non cambia (in realta' implicata dalle seguenti)
//@ \old(carte().size())==carte().size()
```

```

// tutte le carte di prima ci sono anche adesso
//@ (\forall Carta c; \old{carte()}.contains(c));
//@ (\exists Carta d; carte().contains(d));
//@ d.valore() == c.valore() && d.seme() == c.seme());
// tutte le carte di adesso c'erano anche prima
//@ (\forall Carta c; carte().contains(c));
//@ (\exists Carta d; \old{carte()}.contains(d));
//@ d.valore() == c.valore() && d.seme() == c.seme());

```

No, in JML è possibile specificare le pre e post condizioni riguardanti la singola esecuzione del metodo. La proprietà potrebbe essere tuttavia implicata (come nel caso corrente).

Esercizio 2

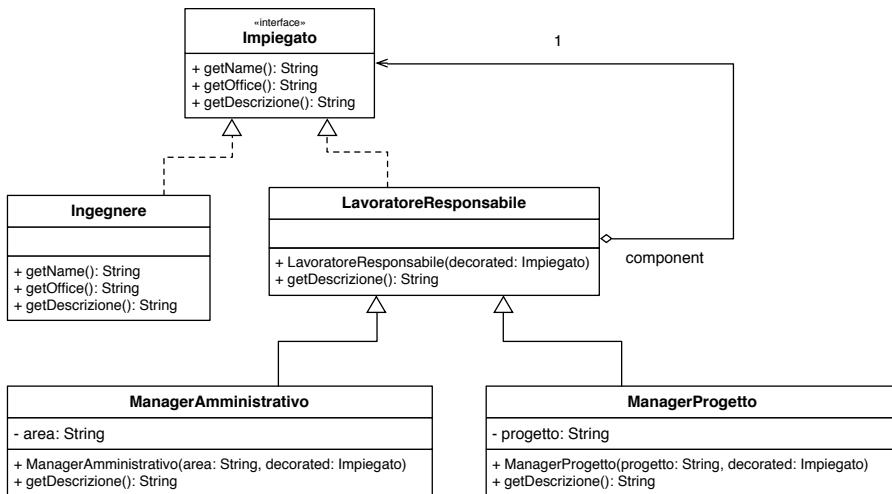
Si consideri l'insieme di impiegati di un'azienda. Gli impiegati espongono tre metodi `String getName()` e `String getOffice()` che ritornano nome e ufficio degli impiegati e `String getDescrizione()` che ritorna le mansioni dell'impiegato.

Ci sono vari tipi di impiegato, per esempio gli ingegneri. Le responsabilità degli impiegati (si considerino per semplicità solo gli ingegneri) possono cambiare dinamicamente. In particolare, un ingegnere può avere la responsabilità di manager amministrativo o manager di progetto. Il comportamento del metodo `String getDescrizione()` viene modificato opportunamente. Per esempio, se un ingegnere `ing` è manager amministrativo di un'area `A` la stringa "Manager di `A`" viene concatenata alla descrizione ritornata dall'invocazione del metodo `getDescrizione()`. Se a `ing` viene anche aggiunta la funzionalità di manager amministrativo dell'area `B`, ottenendo l'oggetto `ing1`, il metodo `getDescrizione()` invocato su `ing1` concatena la stringa "Manager di `B`" alla descrizione di `ing`. Infine se a `ing1` viene aggiunta la funzionalità manager del progetto `P1`, ottenendo l'oggetto `ing2`, la stringa ottenuta invocando il metodo `getDescrizione()` sull'oggetto `ing2` sarà "Oltre ad essere <XXX> sono Manager di `P1`", dove `XXX` è la stringa ritornata dal metodo `getDescrizione()` di `ing1`. Come evidenziato dagli esempi, un ingegnere può essere manager amministrativo di più aree e/o manager di più progetti.

1. Si modelli, attraverso un diagramma delle classi UML e un design pattern opportuno, una soluzione al problema sopra presentato.
2. Si scriva anche la struttura del codice Java risultante. Ovvero, si definiscano le classi identificate al passo precedente, le loro relazioni e le intestazioni dei metodi principali.
3. Si scriva il codice Java che crea un'istanza di un oggetto ingegnere con funzionalità di manager amministrativo per l'area A e per l'area B e project manager del progetto P1.

SOLUZIONE:

1. È possibile usare un pattern Decorator.



2. public interface Impiegato {
 String getName();
 String getDescrizione();
 String getOffice();
 }

 public class Ingegnere implements Impiegato {
 @Override
 public String getName(){ ... }

 @Override
 public String getDescrizione(){ ... }

 @Override
 public String getOffice(){ ... }

 }

 public class LavoratoreResponsabile implements Impiegato {

 private Impiegato decorated;

 public LavoratoreResponsabile(Impiegato decorated){ ... }

 @Override
 public String getName(){ ... }

 @Override
 public String getDescrizione(){ ... }

 @Override
 public String getOffice(){ ... }

 }

 }

 public class ManagerAmministrativo extends LavoratoreResponsabile {

```

private String area;

public ManagerAmministrativo(String area, Impiegato decorated){ ... }

@Override
public String getDescrizione(){
    return "Manager di "+area+super.getDescrizione();
}

}

public class ManagerProgetto extends LavoratoreResponsabile {
    private String progetto;

    public ManagerProgetto(String progetto, Impiegato decorated){ ... }

    @Override
    public String getDescrizione(){
        return "Oltre ad essere <"+super.getDescrizione()+">" +
            "+ sono anche Manager di "+progetto;
    }

}

```

3. Ingegnere base = new Ingegnere(...);
 Impiegato managerP = new ManagerProgetto("P1", base);
 Impiegato managerBP = new ManagerAmministrativo("B", managerP);
 Impiegato managerABP = new ManagerAmministrativo("A", managerBP);

Esercizio 3

Un gruppo di 50 studenti deve sostenere un esame orale all'università. Tre docenti sono a disposizione degli studenti, ma chiaramente possono interrogare uno studente per volta. Per semplicità, si supponga che ogni esame abbia una durata casuale. Alla fine dell'esame ogni docente propone un voto allo studente e, se questo lo accetta, il voto viene verbalizzato.

Si completi il seguente programma (concorrente) Java che simuli la situazione appena descritta. Il singleton Aula svolge il ruolo di coordinatore. Lo Studente cercherà di sostenere l'esame, ma dovrà mettersi in attesa se tutti i docenti fossero impegnati. Il Docente inizierà un esame e, dopo un tempo variabile, proporrà un voto. Lo Studente potrà accettarlo e quindi il voto verrà verbalizzato dal Docente.

Si noti che lo studente deve completare le intestazioni dei metodi, ove necessario, e il corpo dei metodi `iniziasiEsame`, `terminaEsame` e `partecipaAEsame`.

```

import java.util.Stack;

public class Aula {
    public Stack<Docente> freeDocenti;

    public Aula(){
        this.freeDocenti=new Stack<Docente>();
        this.freeDocenti.push(new Docente());
        this.freeDocenti.push(new Docente());
        this.freeDocenti.push(new Docente());
    }
    public static void main(String[] args) {

```

```

        Aula a=new Aula();
        for(int i=0; i<50; i++) {
            new Studente().partecipaAEsame(a);
        }
    }

public          Docente iniziaEsame() throws InterruptedException{

}

public          void terminaEsame(Docente d) {

}

public          void sostieniEsame(Studente s) throws InterruptedException{

    Docente d=iniziaEsame();
    int voto=d.valuta(s);
    if(voto>18 && s.accetta(voto)) {
        d.verbalizza(s, voto);
    }
    terminaEsame(d);
}
}

import java.util.Random;

public class Docente {

    public          int valuta(Studente s) throws InterruptedException{
        Thread.sleep(new Random().nextInt(300));
        return new Random().nextInt(30);
    }
    public          void verbalizza(Studente s, int voto){
    }
}

import java.util.Random;

public class Studente {

```

```

        boolean accetta(int voto){
            return new Random().nextBoolean();
        }

        void partecipaAEsame(Aula a) {
            new Thread(new Runnable() {
                ...
            })
        }
    }

package prova;

import java.util.Stack;

public class Aula {
    public Stack<Docente> freeDocenti;

    public Aula(){
        this.freeDocenti=new Stack<Docente>();
        this.freeDocenti.push(new Docente());
        this.freeDocenti.push(new Docente());
        this.freeDocenti.push(new Docente());
    }
    public static void main(String[] args) {

        Aula a=new Aula();
        for(int i=0; i<50; i++){
            new Studente().partecipaAEsame(a);
        }
    }

    public synchronized Docente iniziaEsame() throws InterruptedException{
        while(freeDocenti.isEmpty()){
            wait();
        }
        return freeDocenti.pop();
    }
    public synchronized void terminaEsame(Docente d) {

```

```

        freeDocenti.push(d);
        this.notifyAll();
    }
    public void sostieniEsame(Studente s) throws InterruptedException{

        Docente d=iniziaEsame();
        int voto=d.valuta(s);
        if(voto>18 && s.accetta(voto)){
            d.verbalizza(s, voto);
        }
        terminaEsame(d);
    }
}

package prova;

import java.util.Random;

public class Docente {

    public int valuta(Studente s) throws InterruptedException{
        Thread.sleep(new Random().nextInt(300));
        return new Random().nextInt(30);
    }
    public void verbalizza(Studente s, int voto){
    }
}

package prova;

import java.util.Random;

public class Studente {
    public boolean accetta(int voto){
        return new Random().nextBoolean();
    }
    public void partecipaAEsame(Aula a){
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    a.sostieniEsame(Studente.this);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }).start();
    }
}

```

Esercizio 4

Si consideri il seguente metodo statico Java:

```
public static int foo(int a, int b) {
```

```
int n = 3;

if (a == 0 || b == 0)
    return 1;
else
    while (a%n != 0)
        if (a > b) a -= b;
    return b;
}
```

e si definisca:

1. il diagramma del flusso di controllo;
2. un insieme minimo di casi di test che coprano tutte le istruzioni;
3. un insieme di casi di test che copra tutti i branch;

Quali (probabili) errori sono evidenziati dall'insieme di test definito al punto precedente?

PUNTO A.

Lasciato al lettore

PUNTO B.

(A=1, B=0), (A=4, B=1)

PUNTO C.

(A=1, B=0), (A=4, B=1), (A=4, B=2)

Probabili errori: infinite loop per i valori A=4, B=2.

Appello 11 luglio 2017



Politecnico di Milano
Anno accademico 2015-2016

Ingegneria del Software

Cognome:

LAUREANDO

Nome:

Matricola:

Sezione (segnarne una): Cugola

Mottola

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente classe Java `Digrafo<T>`. Essa descrive un ADT che rappresenta un grafo orientato, i cui nodi hanno un'etichetta del tipo generico `T`. Un nodo particolare, detto radice, è distinto dagli altri. Tutti i nodi del grafo sono raggiungibili dalla radice tramite un cammino orientato.

I nodi del digrafo sono oggetti di una classe immutabile, detta `Nodo<T>`, non meglio precisata. Tale classe contiene solo l'etichetta del nodo, mentre non contiene riferimenti ai nodi ad esso collegati (che sono mantenuti separatamente in una opportuna struttura dati all'interno della classe `Digrafo<T>`).

```
public class Digrafo<T> {
    //OVERVIEW: Un digrafo connesso con un nodo speciale detto radice, tale che per ogni nodo
    //c'e' un cammino dalla radice al nodo.
    //Da ogni nodo escono esattamente 0 oppure R>0 archi, dove R (detto grado del grafo) e'
    //una costante fissata al momento della costruzione del grafo.
    //Ogni nodo e' etichettato con un valore di tipo T.

    public final int R;

    //@ensures (* costruisce un digrafo con radice n, con R==grado,
    //senza altri nodi ne' archi, lancia eccezione se grado e' scorretto o n nullo *)
    public Digrafo(Nodo<T> n, int grado) throws NonValidoException; ,

    //observers:

    //@ensures (* \result e' la radice del digrafo *)
    public /*@ pure @*/ Nodo<T> radice();

    //@ensures (* \result e' il numero di nodi del digrafo *);
    public /*@ pure @*/ int size();

    //@ensures (* \result vale true se, e solo se, il nodo n e' nel digrafo *);
    public /*@ pure @*/ boolean contains( Nodo<T> n);

    //@ensures (* \result e' l'insieme che contiene gli R nodi del grafo collegati
    //con un arco uscente dal nodo n, o l'insieme vuoto se non vi sono archi uscenti*);
    public /*@ pure @*/ Set<Nodo<T>> out(Nodo<T> n);

    //@ensures (* \result e' l'insieme di tutti e soli i nodi
    //collegati con un arco entrante nel nodo n*);
    public /*@ pure @*/ Set<Nodo<T>> in(Nodo<T> n);

    //@ensures (* restituisce true se esiste cammino dal nodo n al nodo h
    public /*@ pure @*/ boolean raggiungibile(Nodo<T> n, Nodo<T> h)

    //modifiers:

    //@ensures (* elimina tutti gli archi e tutti i nodi del grafo, salvo la radice *)
    public void cancellaGrafo()

    //@ensures (* Collega ciascuno dei nodi dell'insieme insNodi, supposto non vuoto,
    //con un un arco uscente dal nodo n.
    //@ I nodi in insNodi, che non sono gi'\a presenti nel grafo, sono ad esso aggiunti.
    //@ Lancia eccezione, senza modificare il grafo, se n non esiste nel digrafo oppure
    //@ se l'aggiunta degli archi renderebbe non valido il grado.
    public void collega(Nodo<T> n, Set<Nodo<T>> insNodi) throws NonValidoException;

}
```

Domanda a

Utilizzando la notazione JML, descrivere la specifica del costruttore e del metodo collega. Si ipotizzi che le specifiche di tutti gli altri metodi della classe siano già state definite.

Soluzione:

```
//@ensures n!=null && grado>0 && R == grado && radice()==n && contains(n) &&
//@           in(n).size()==0 && out(n).size()==0 && size()==1;
//@signals (NonValidoException e) n==null || grado<=0;
public Digrafo(Nodo<T> n, int grado) ;

//@ensures n!=null && insNodi!=null &&
//@           \old(contains(n) && insNodi.size()==R && out(n).size()==0) &&
//@           (\forall Nodo<T> h; insNodi.contains(h); out(n).contains(h) &&
//@           in(h).contains(n)) && out(n).size()==R &&
//@           (\forall Nodo<T> h; contains(h) <=> \old(contains(h) || insNodi.contains(h)));
//@
//@signals (NonValidoException e) (n==null || insNodi==null ||
//@           !\old(contains(n) && insNodi.size()==R && out(n).size()==0)) &&
//@           (\forall Nodo<T> h; \old(contains(h) <=> contains(h)));
//@
public void collega(Nodo<T> n, Set<Nodo<T>> insNodi) throws NonValidoException;
```

Domanda b

Si definiscano, tramite un opportuno invarianto astratto, le proprietà descritte nell'overview, che sono qui ricordate: Un digrafo connesso con un nodo speciale detto radice, tale che per ogni nodo c'e' un cammino dalla radice al nodo. Da ogni nodo escono esattamente 0 oppure R>0 archi.

Soluzione:

L'invariante asserisce che R è il grado del grafo e che il grafo è connesso.

```
//@public invariant size()>=1 &&
//@   (\forall Nodo<T> h; this.contains(h); (out(h).size()==0 || out(h).size()==R)) &&
//@   raggiungibile(h);
```

Domanda c

Si consideri ora la specifica della seguente classe:

```
public class DigrafoFlessibile<T> extends Digrafo<T> {

  \\OVERVIEW: Un Digrafo in cui e' possibile eliminare tutti gli archi uscenti da un nodo

  //@ensures (* Elimina tutti gli archi uscenti da n
  //@ Se il nodo n non esiste nel grafo, allora il metodo lancia eccezione*);
  public void elimina(Nodo<T> n) throws NonValidoException;

}
```

La classe `DigrafoFlessibile<T>` può ereditare da `Digrafo<T>` in base al principio di sostituzione di Liskov? Motivare la risposta.

Soluzione:

No, eliminando un nodo il grafo potrebbe non essere piu' connesso, violando la regola delle proprietà.

Domanda d

Si consideri la seguente implementazione della classe `DigrafoFlessibile<T>`, che utilizza una classe `Arco`, definita nello stesso package. Si noti che la classe `DigrafoFlessibile<T>` e' definita con gli identici metodi di `Digrafo<T>`, con in piu' il solo metodo `elimina`.

```
public class DigrafoFlessibile<T> {  
    //rep:  
    private ArrayList<Nodo<T>> nodi;  
    private ArrayList<Arco> archi;  
  
    public DigrafoFlessibile<T>(Nodo<T> n, int grado) {  
        if (n== null || grado<=0) throw new NonValidoException();  
        R=grado;  
        nodi = new ArrayList<Nodo<T>>();  
        archi = new ArrayList<Arco>();  
        nodi.add(n);  
    }  
    ...implementazione di tutti i metodi dell'interfaccia.  
}  
  
class Arco {  
    Nodo<T> sorgente;  
    Nodo>T> destinazione;  
}
```

L'implementazione memorizza in `nodi` tutti i nodi del grafo, con la radice in prima posizione, e in `archi`, tutti e soli gli archi, disposti in un ordine non precisato.

Si scriva l'invariante di rappresentazione della classe `DigrafoFlessibile<T>`.

Soluzione:

L'invariante richiede che gli archi colleghino solo nodi memorizzati in `nodi` e che per ogni nodo in `nodi` ci siano esattamente 0 oppure `R` archi di cui il nodo è il primo elemento.

```
//@private invariant nodi!=null && archi!= null && nodi.size()>=1 &&  
//@  (\forall Arco a; archi.contains(a); nodi.contains(a.sorgente) && nodi.contains(a.destinazione));  
//@  (\forall Nodo<T> n; nodi.contains(n);  
//@      (\#Arco a; archi.contains(a); a.sorgente==n) ==R ||  
//@      (\#Arco a; archi.contains(a); a.sorgente==n) ==0;
```

Esercizio 2

Si consideri la seguente classe Java:

```
public class Coppia {  
    private int myX, myY;  
    public Coppia() {  
        myX=0; myY=0;  
    }  
    public void set(int x, int y) {  
        if( (x+y)>=0 ) {  
            myX = x;  
            myY = y;  
        }  
    }  
    public int getSomma() {  
        return myX+myY;  
    }  
}
```

Domanda a

In presenza di thread multipli è corretto affermare che il metodo `getSomma` restituisce sempre un valore maggiore o uguale a 0? Motivare la propria risposta.

Soluzione

La coppia di assegnamenti nel metodo `set` non viene eseguita in maniera atomica, due thread diversi potrebbero quindi invocare il metodo con due coppie di valori che prese come tali superano il test $x+y>=0$ ma danno luogo ad una somma negativa se i valori vengono “mischiati”. Ad esempio, il primo thread potrebbe invocare `set(100, -10)`. Subito dopo l’assegnamento `myX=x`, che in questo caso assegna 100 a `myX`, il secondo thread potrebbe ottenere il controllo, invocando `set(-100, 200)`. Anche questo thread passerebbe il controllo eseguendo gli assegnamenti previsti per cui avremmo: `myX=-100` e `myY=200`. A questo punto il primo thread potrebbe riprendere l’esecuzione, impostando `myY=-10`. Al termine del processo avremmo: `myX=-100`, `myY=-10`. La somma dei due valori risulta ovviamente negativa.

Domanda b

In caso di risposta negativa alla precedente domanda, come bisogna modificare il codice per garantire che il metodo `getSomma` restituisca sempre un valore maggiore o uguale a 0?

Soluzione

È sufficiente rendere `synchronized` **entrambi** i metodi `set` e `getSomma`.

Domanda c

Modificare la classe `Coppia` aggiungendo un metodo `aspettaZero()` che sospende il thread chiamante fin tanto che la somma `myX+myY` non sia pari a 0 (se necessario, oltre ad aggiungere il nuovo metodo, si modifichino i metodi esistenti).

Soluzione

```
public class Coppia {  
    private int myX, myY;  
    public Coppia() {  
        myX=0; myY=0;  
    }  
    public synchronized void set(int x, int y) {  
        if( (x+y)>=0 ) {  
            myX = x;  
            myY = y;  
            if( (x+y)==0 ) notifyAll();  
        }  
    }  
}
```

```
}

public synchronized int getSomma() {
    return myX+myY;
}

public synchronized void aspettaZero() {
    while( (x+y)!=0 ) wait();
}

}
```

Esercizio 3

Si considerino le seguenti definizioni di classi:

```
abstract public class Animal {  
    abstract public void greeting();  
}  
public class Cat extends Animal {  
    public void greeting() {  
        System.out.println("Meow!");  
    }  
}  
public class Dog extends Animal {  
    public void greeting() {  
        System.out.println("Woof!");  
    }  
    public void greeting(Dog another) {  
        System.out.println("Wooooooooof!");  
    }  
}  
public class BigDog extends Dog {  
    public void greeting() {  
        System.out.println("Woow!");  
    }  
    public void greeting(Dog another) {  
        System.out.println("Wooooooowwww!");  
    }  
}
```

Si consideri ora la seguente esecuzione di un ipotetico metodo, e si descrivano *i*) eventuali errori segnalati dal compilatore, e *ii*) una volta rimosse le righe che eventualmente generano errori di compilazione, quale sia l'output prodotto a video. Per ciascuna riga prodotta a video, si spieghi inoltre, in base alla gerarchia delle classi e ai tipi statici e dinamici delle diverse variabili riferimento, il motivo per cui viene eseguito uno specifico metodo tra quelli disponibili.

```
Animal animal1 = new Cat();  
animal1.greeting();  
Animal animal2 = new Dog();  
animal2.greeting();  
Animal animal3 = new BigDog();  
animal3.greeting();  
Animal animal4 = new Animal();  
animal4.greeting();  
  
BigDog bigDog1 = new BigDog();  
Dog dog2 = (Dog)animal2;  
BigDog bigDog2 = (BigDog)animal3;  
Dog dog3 = (Dog)animal3;  
  
dog2.greeting(dog3);  
dog3.greeting(dog2);  
dog2.greeting(bigDog2);  
bigDog2.greeting(dog2);  
bigDog2.greeting(bigDog1);
```

Soluzione:

La riga `Animal animal4 = new Animal()` genera un errore di compilazione: la classe `Animal` è astratta. Di conseguenza, anche la riga successiva `animal4.greeting()` deve essere rimossa perché il codice possa compilare.

Le prime tre invocazioni di greeting () sono banali: eseguono il codice del metodo corrispondente al tipo dinamico, nell'ordine Cat, Dog, e BigDog. Le ultime cinque invocazioni di greeting (parametro) stampano rispettivamente:

```
Wooooooooooof !
Woooooooooooo !
Woooooooooooo !
Woooooooooooo !
Woooooooooooo !
```

Di queste, il caso degno di nota è il secondo: sebbene dog3 abbia un tipo statico Dog, il riferimento animal3 ha un tipo dinamico BigDog, per cui le regole di ereditarietà e polimorfismo determinano che sia il codice di quest'ultimo ad essere seguito. Notare che in tutti i casi il parametro di ingresso viene ignorato.

Esercizio 4

Si consideri il metodo seguente e si generi un insieme di casi di test che soddisfi il criterio di copertura delle diramazioni (branch coverage).

```
public int foo(int a, int b) {  
    int temp;  
    if ( a==b ) {  
        return 0;  
    } else {  
        if ( a>b ) {  
            b=a*a;  
            temp=0;  
        } else {  
            temp=1;  
        }  
        while ( a<b ) {  
            temp=temp+1;  
            a=a+2;  
        }  
    }  
    return temp;  
}
```

Soluzione:

Sono sufficienti tre casi di test: (1,1), (2,1) e (1,2).



Politecnico di Milano

Anno accademico 2012-2013

Ingegneria del Software – Appello del 17 Settembre 2013

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Un'azienda di *catering* vuole proporre ricevimenti personalizzati per clienti onnivori, vegetariani e vegani. I clienti onnivori mangiano tutto, i vegetariani non mangiano carne e i vegani non mangiano né carne né derivati animali. L'azienda gestisce anche l'archivio dei cibi che offre, organizzati secondo la stessa classificazione adottata per i clienti. Si noti che un cibo può essere un prodotto di base, ad esempio frutta o verdura, o un piatto composto da diversi ingredienti (cibi) di base. Alcuni ingredienti possono essere mangiati anche da soli, mentre altri devono essere combinati per ottenere il prodotto finale.

Volendo organizzare i propri clienti per essere sicuri che ognuno mangi solo i cibi adatti, quante e quali classi definireste per cliente e cibo? Come mettereste in relazione clienti e cibi? Ovviamamente, l'azienda non vuole gestire archivi separati per le diverse tipologie di cibi e clienti, ma vuole un solo archivio per i clienti e un solo archivio per i cibi, essendo però sicura che i vincoli imposti dalle diverse tipologie di clienti siano rispettati per costruzione.

- a) Si utilizzi un diagramma delle classi UML per spiegare la soluzione adattata.

- b) Supponendo che esistano una classe `Cliente` e una classe `Catering` (che include l'archivio dei clienti), si implementi un metodo `clientiVegani()` che restituisce un `Iterator<Cliente>` che consenta di scandire l'insieme dei soli clienti vegani. A tale scopo, si richiede di definire sia la parte del rep della classe `Catering` che contiene l'archivio sia la classe che implementa `Iterator<Cliente>`.
- c) Supponendo che, ad un certo punto, l'azienda voglia adottare un sistema di notifiche *push* verso i propri clienti per informarli delle nuove pietanze offerte (ovviamente sempre rispettando le diverse tipologie di clienti), come modifichereste il diagramma delle classi definito in precedenza? Quali metodi aggiungereste o cambiereste?

Esercizio 2

Si consideri il seguente programma Java

```
public class Figura {
    public void stampa(Figura f) {
        System.out.println("Figura");
    }
}

public class Quadrato extends Figura {
    public void stampa(Quadrato q) {
        System.out.println("Quadrato");
    }
}

public class Rettangolo extends Quadrato {
    public void stampa(Rettangolo r) {
        System.out.println("Rettangolo");
    }
    public void stampa(Quadrato r) {
        System.out.println("Rettangolo particolare");
    }
    public void stampa(Figura f) {
        System.out.println("Figura particolare");
    }
}

public class Test {
    public static void main(String args[]) {
        Figura f1, f2;
        Quadrato q = new Rettangolo();
        Rettangolo r = new Rettangolo();
        f1 = new Quadrato();
        f2 = new Rettangolo();

        f1.stampa(f2); .....
        q.stampa(r); .....
        f1.stampa(q); .....
        q.stampa(f1); .....
        q.stampa(q); .....
    }
}
```

Scrivere a lato delle precedenti istruzioni di stampa il risultato prodotto durante l'esecuzione. Cosa cambierebbe se Rettangolo estendesse Figura e non Quadrato? Motivare brevemente le risposte

Esercizio 3

Si consideri il seguente metodo generico `intersezione`, che prende come argomento due `ArrayList<T>` `x` e `y`, e un `Comparator` `c`. Gli `ArrayList` `x` e `y` contengono elementi di tipo generico `T`, che non sono modificati dal metodo, ma nessun valore null. Il metodo restituisce un nuovo `ArrayList<T>`, che contiene tutti e soli gli elementi che sono presenti sia in `x` che `y`, disposti in ordine crescente nell'`ArrayList` secondo l'ordinamento previsto dal `Comparator`.

```
public static <T> ArrayList<T> intersezione(ArrayList<T> x,  
                                              ArrayList<T> y, Comparator<? super T> c)
```

a) Si specifichi in JML il metodo `intersezione`. Si ricorda che l'interfaccia `Comparator` ha la seguente dichiarazione:

```
public interface Comparator<T> {  
    int compare (Object obj1, Object obj2) throws ClassCastException;  
    //\signals ClassCastException se obj1 e obj2 non sono di tipi confrontabili  
    //con il Comparator corrente.  
    //\result e' -1 se obj1<obj2, 0 se obj1.equals(obj2), +1 altrimenti  
}
```

Esercizio 4

Si consideri il seguente frammento di codice Java, in cui le varie istruzioni sono state numerate per comodità:

```
public static int test(int x, int y) {  
    1. int z = 5;  
    2. do {  
        3.     if (z < x && y != 1)  
        4.         x = z / (y - 1);  
        5.     else return z;  
        6.     z--;  
    7. }  
    8. while (z > 2);  
    9. return z;  
}
```

1. Si definisca il diagramma del flusso di controllo.
2. Si definisca l'insieme minimo di casi di test per coprire:
 - a) le istruzioni;
 - b) le condizioni
 - c) i cammini.

Soluzione del Tema

Esercizio 1

Si consideri la classe Java HotelDB per gestire la prenotazione di stanze di hotel.

```
public class HotelDB {
    // Ritorna l'elenco di tutti gli hotel contenuti.
    public /*@ pure @*/ Set<Hotel> getHotels();

    // Ritorna la lista di tutte le stanze che (i) possono ospitare almeno people persone,
    // (ii) si trovano vicino a location, (iii) sono libere il giorno day.
    // La lista e' in ordine crescente di prezzo (dalla stanza meno costosa alla piu' costosa).
    public /*@ pure @*/ List<Room> findRooms(int people, Location location, Day day);

    // Prenota la stanza room per il giorno day.
    // Solleva una AlreadyBookedException se la stanza non e' disponibile il giorno specificato.
    public void book(Room room, Day day) throws AlreadyBookedException;
}

public class Hotel {
    // Ritorna tutte le stanze presenti nell'hotel.
    public /*@ pure @*/ Set<Room> getRooms();

    // Ritorna true sse l'hotel e' vicino a location.
    public /*@ pure @*/ boolean isCloseTo(Location location);
}

public class Room {
    // Ritorna true sse la stanza e' disponibile il giorno day.
    public /*@ pure @*/ boolean isAvailable(Day day);

    // Ritorna il numero massimo di persone che la stanza puo' ospitare.
    public /*@ pure @*/ int getMaxPeople();

    // Ritorna il prezzo della stanza.
    public /*@ pure @*/ float getPrice();

    // Permette di prenotare la stanza nel giorno precisato.
    // Richiede che la stanza non sia gia' prenotata per lo stesso giorno.
    public void book(Day day);
}
```

Domanda a)

Si specifichi in JML il metodo findRooms(). **Soluzione**

```
/*@ requires people > 0 && location != null && day != null
*/
/*@ ensures \result != null &&
/*@ (\forall Room r; r != null; \result.contains(r) <==> (
/*@   (\exists Hotel h; getHotels().contains(h) && h.isCloseTo(location);
/*@     h.getRooms().contains(r) ) &&
/*@     r.isAvailable(day) && r.getMaxPeople() >= people) ) &&
/*@ (\forall int i; i >= 1 && i < \result.size());
/*@   \result.get(i-1).getPrice() <= \result.get(i).getPrice())
public /*@ pure @*/ List<Room> findRooms(int people, Location location, Day day);
```

NB: La porzione

(\forall Room r; r != null; \result.contains(r) <==> (\exists Hotel h; ...))

può essere sostituita dalla forma seguente:

(\forall Hotel h; ... (\forall Room r; r != null && ...; \result.contains(r) <==> (...)))

. Le due forme in generale non sono equivalenti, ma in questo caso particolare lo sono in quanto è evidentemente sottointeso che ogni stanza puo' essere contenuta in un solo hotel.

Domanda b)

Si specifichi in JML il metodo book () .

Soluzione

```
//@ requires room != null && day != null &&
//@ (\exists Hotel h; getHotels().contains(h); h.getRooms().contains(room))
//@
//@ ensures \old(room.isAvailable(day)) && !room.isAvailable(day) &&
//@ getHotels().size() == \old(getHotels().size()) &&
//@ getHotels().containsAll(\old(getHotels())) &&
//@ (\forall Day d; d != null;
//@   (\forall Room r;
//@     (\exists Hotel h; getHotels().contains(h); h.getRooms().contains(r));
//@     (d != day || r != room) ==> r.isAvailable(d) == \old(r.isAvailable(d)) ) )
//@
//@ signals (AlreadyBookedException e) && !\old(room.isAvailable(day)) &&
//@ getHotels().size() == \old(getHotels().size()) &&
//@ getHotels().containsAll(\old(getHotels())) &&
//@ (\forall Day d; d != null;
//@   (\forall Room r;
//@     (\exists Hotel h; getHotels().contains(h); h.getRooms().contains(r));
//@     r.isAvailable(d) == \old(r.isAvailable(d)) ) )
public void book(Room room, Day day);
```

Domanda c)

Si consideri un'implementazione che utilizza una List per memorizzare gli hotel, come mostrato di seguito. Per tale implementazione si fornisca un invarianto di rappresentazione e una funzione di astrazione.

```
public class HotelDB {
    private List<Hotel> hotels;
    ...
}
```

Soluzione

L'invariante di rappresentazione richiede semplicemente che la lista non sia nulla, non contenga elementi nulli e non contenga duplicati.

```
//@ private invariant hotels != null && !hotels.contains(null) &&
//@ (\forall int i; i>=1 && i<hotels.size(); !hotels.get(i).equals(hotels.get(i-1)) )
```

La funzione di astrazione definisce getHotels () a partire dalla lista. Tutti gli altri metodi sono definiti a partire da getHotels () .

```
//@ private invariant
//@ (\forall Hotel h; h != null; hotels.contains(h) <=> getHotels().contains(h))
```

Domanda d)

Si consideri una classe HotelDB2 che aggiunge un metodo bookAvailableRoom () alla classe HotelDB. Il metodo si comporta esattamente come book () ma non solleva alcuna eccezione nel caso in cui la stanza non sia disponibile. Può HotelDB2 essere definito come sottoclasse di HotelDB in accordo con il principio di sostituzione? Motivare la propria risposta.

Soluzione

HotelDB2 può essere definito come sottoclasse di HotelDB in quanto aggiunge un metodo senza modificare i metodi esistenti (e le loro pre-condizioni e post-condizioni). Il metodo aggiunto inoltre non viola alcun invariante della classe.

Esercizio 2

Si consideri la seguente versione semplificata delle classi HotelDB e Hotel.

```
class HotelDB {
    private final List<Hotel> hotels;

    public HotelDB(List<Hotel> hotels) {
        this.hotels = new ArrayList<>(hotels);
    }

    public void book(int id) {
        if (hotels.get(id).getAvailableRooms() > 0) {
            hotels.get(id).bookOneRoom();
        }
    }
}

class Hotel {
    private int totalRooms;
    private int availableRooms;

    public synchronized int getAvailableRooms() { return availableRooms; }

    public synchronized int getTotalRooms() { return totalRooms; }

    public synchronized void bookOneRoom() { availableRooms--; }

    public synchronized void cancelOneBooking() { availableRooms++; }
}
```

Domanda a)

L'implementazione fornita risulta correttamente sincronizzata? In caso contrario, indicare come correggere l'implementazione, favorendo soluzioni che massimizzino l'esecuzione parallela di thread che accedono ai metodi della classe HotelDB.

Soluzione

L'implementazione *non* risulta correttamente sincronizzata, in quanto il controllo della disponibilità di stanze e la successiva prenotazione non avvengono in maniera atomica. Di conseguenza il numero di stanze disponibili in un hotel potrebbe andare sotto 0. Per correggere, si può modificare l'implementazione come mostrato di seguito.

```
class HotelDB {
    ...

    public void book(int id) {
        Hotel h = hotels.get(id);
        synchronized(h) {
            if (h.getAvailableRooms() > 0) {
                h.bookOneRoom();
            }
        }
    }
}
```

Domanda b)

Si implementi un metodo `cancel(int id)` in HotelDB. Tale metodo agisce sull'hotel in posizione `id`: se per tale hotel il numero di stanze disponibili risulta inferiore al numero di stanze totali (`getAvailableRooms() < getTotalRooms()`), il metodo cancella una prenotazione dall'hotel invocando il suo metodo `cancelOneBooking()`, altrimenti non fa nulla.

Si modifichi quindi il metodo `book(int id)` facendo in modo che, in caso non ci sia disponibilità nell'hotel indicato, il metodo sospenda il chiamante in attesa di una cancellazione. Si specifichi, qualora fosse necessario, quali altri metodi richiedono modifiche.

Si favoriscano soluzioni che massimizzano l'esecuzione parallela di thread che accedono ai metodi di HotelDB.

Soluzione

Occorre anche modificare il metodo `book()` come mostrato di seguito, per fare in modo che invochi `notifyAll()` per risvegliare i thread in attesa.

```

class HotelDB {
    ...
    public void cancel(int id) {
        Hotel h = hotels.get(id);
        synchronized(h) {
            if (h.getAvailableRooms() < h.getTotalRooms()) {
                h.cancel();
                h.notifyAll();
            }
        }
    }

    public void book(int id) {
        Hotel h = hotels.get(id);
        synchronized(h) {
            while (h.getAvailableRooms() == 0) {
                h.wait();
            }
            h.bookOneRoom();
        }
    }
}

```

Esercizio 3

```

1 public static int foo(int n) {
2     int x = n%2;
3     int y = n%3;
4     while (n != 0) {
5         if (x*y == 2 || n < 0) return x;
6         else n--;
7     }
8     return y;
9 }

```

Si determini un insieme minimo di casi di test per ciascuno dei casi seguente di copertura.

1. Copertura delle istruzioni (statement coverage)
2. Copertura delle decisioni (edge coverage)
3. Copertura delle decisioni e delle condizioni (edge and condition coverage)

Soluzione

1. Copertura delle istruzioni (statement coverage): n=2 (per entrare nel ramo else) e n=5 (per entrare nel ramo then)
2. Copertura delle decisioni (edge coverage): come sopra.
3. Copertura delle decisioni e delle condizioni (edge and condition coverage) oltre ai casi precedenti serve n=-1 per coprire il secondo disgiunto nell'OR.

Esercizio 4

Si considerino le seguenti classi Java:

```
class Device {
    public String toString() { return "a generic device"; }
}

class Watch extends Device {
    public String toString() { return "a watch"; }
}

class Smartwatch extends Watch {
    public String toString() { return "a smartwatch"; }
}

class Person {
    protected String name;
    public Person(String name) { this.name = name; }
    public void buy(Device d, double euro) {
        System.out.println(name+" buy "+d+" for "+euro+" euros");
    }
}

class Professor extends Person {
    public Professor(String name) { super(name); }
    public void buy(Device d, int euro) {
        System.out.println("Prof. "+name+" buy "+d+" for exactly "+euro+" euros");
    }
}

class FullProfessor extends Professor {
    public FullProfessor(String name) { super(name); }
    public void buy(Smartwatch d, int euro) {
        System.out.println("Full prof. "+name+" buy "+d+" for exactly "+euro+" euros");
    }
}
```

e il seguente frammento di codice le cui righe sono state numerate per riferimento:

```
1 Device d1 = null, d2 = new Device();
2 Watch w = new Smartwatch();
3 Smartwatch sw = new Smartwatch();
4 Person p = new Person("Marco");
5 Professor pp = new FullProfessor("Andrea");
6 FullProfessor fp;
7
8 d1 = sw;
9 fp = pp;
10 p.buy(d1, 15);
11 p = pp;
12 p.buy(d1, 15);
13 pp.buy(w, 15);
14 fp.buy(w, 15.0);
15 fp = new FullProfessor("Luca");
16 fp.buy(w, 15.0);
17 fp.buy(w, 15);
```

Domanda a)

Scrivere il numero di riga delle istruzioni (se ne esistono) che generano un errore in compilazione o a run-time, chiarendo in quale momento (compile-time vs. run-time) viene sollevato l'errore e spiegandone brevemente le ragioni.

Soluzione

La riga 9 non compila perché fp è staticamente di tipo FullProfessor mentre pp è staticamente di tipo Professor e non si può assegnare una sopraclass ad una sottoclass. Si noti che dinamicamente pp sarebbe del tipo corretto (FullProfessor) ma il compilatore guarda solo ai tipi statici. In conseguenza dell'errore precedente non va considerata neppure la gira 14.

Domanda b)

Supponendo di eliminare le eventuali righe che generano errori indicate sopra, si scriva, per ogni riga rimanente, il valore stampato in uscita.

Soluzione

Ricordando che il compilatore guarda solo ai tipi statici e sceglie di conseguenza la “firma” del metodo da invocare, mentre il run-time cerca eventuali ridefinizioni (“override”) del metodo scelto dal compilatore a partire dal tipo dinamico, si ha il seguente risultato:

Riga 10 Marco buy a smartwatch for 15.0 euros. p è staticamente di classe Person, classe che possiede un solo metodo buy che, previa conversione automatica dell’intero 15 al double 15.0, può essere invocato con i parametri passati.

Riga 12 Andrea buy a smartwatch for 15.0 euros. Come sopra. Infatti, la classe FullProfessor, tipo dinamico dell’oggetto invocato, non ridefinisce il metodo buy (Device d, double euro) ma aggiunge (“overload”) un metodo simile con parametri di tipo diverso, e lo stesso accade alla sopraclassse Professor.

Riga 13 Prof. Andrea buy a smartwatch for exactly 15 euros. In questo caso il tipo statico di pp è Professor, il compilatore sceglie quindi il metodo buy(Device d, int euro) aggiunto in Professor e compatibile con il tipo (statico) dei parametri attuali. Dinamicamente la classe FullProfessor, tipo dinamico di pp, non ha ridefinito tale metodo ma ha aggiunto (“overload”) un ulteriore metodo con primo parametro di tipo Smartwatch (e non Device come nella sopraclassse).

Riga 16 Luca buy a smartwatch for 15.0 euros. Il tipo statico di fp è FullProfessor, a partire da questa classe il compilatore cerca un metodo compatibile con i tipi (statici) dei parametri attuali: non può scegliere il metodo buy(Smartwatch d, int euro), aggiunto in FullProfessor, perchè il secondo parametro attuale è un double che non può essere convertito a int; per ragioni analoghe non può scegliere neppure buy(Device d, int euro), aggiunto in Professor; di conseguenza il compilatore sceglie buy(Device d, double euro) definito nella classe Person. Dinamicamente non ci sono ridefinizioni (“overload”) di questo metodo lungo la catena di ereditarietà del tipo dinamico (FullProfessor), di conseguenza si chiama il metodo definito in Person.

Riga 17 Prof. Luca buy a smartwatch for exactly 15 euros. Come sopra il compilatore cerca un metodo compatibile con i tipi (statici) dei parametri attuali a partire dal tipo statico FullProfessor: non può scegliere il metodo buy(Smartwatch d, int euro), aggiunto in FullProfessor, perchè il primo parametro attuale è (staticamente) di tipo Watch, non compatibile con SmartWatch; può invece scegliere il metodo buy(Device d, int euro) perchè entrambi i parametri attuali sono compatibili con i parametri formali. All’atto della chiamata non ci sono, nella classe FullProfessor, tipo dinamico di fp, ridefinizioni del metodo scelto (il metodo definito in FullProfessor è un caso di “overloading”), quindi si risale la catena di ereditarietà e si esegue il metodo della classe Professor.

Si noti che in tutti i ragionamenti precedenti abbiamo trascurato di dire che in fase di stampa del “device” si sceglie sempre la versione del metodo `toString` ridefinita dal tipo dinamico del parametro (in questo caso non è presente “overloading” ma solo “overriding”).

Ingegneria del Software – a.a. 2006/07

Appello del 10 luglio 2007

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 6)

Il seguente metodo statico:

```
public static int[] fusioneOrdinata (int[ ] a, int[ ] b) {... }
```

effettua la fusione ordinata dei due array a e b, ossia fa in modo che, dopo la chiamata, l'array restituito al chiamante contenga tutti e soli gli elementi contenuti, prima della chiamata, nei due array a e b, e inoltre risulti ordinato.

Si scriva una specifica, in JML del metodo fusioneOrdinata nei due seguenti casi:

a- si assume che i due array a e b non abbiano elementi ripetuti, e siano disgiunti (nessun valore int può essere memorizzato in entrambi)

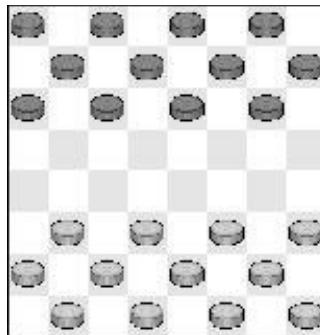
```
//@assignable \nothing
//@requires a!= null && b!= null &&
// ogni valore int memorizzato in a compare una sola volta in a e non compare in b
//@  (\forall int i; 0<=i&&i<a.length; (\num_of int j; 0<=j<a.length; a[j]==a[i])==1 && (\num_of int k;
0<=k<b.length; b[k]==a[i])==0)
// ogni valore int memorizzato in b compare una sola volta in b e non compare in a
//@  (\forall int i; 0<=i&&i<b.length; (\num_of int j; 0<=j<b.length; b[j]==b[i])==1 && (\num_of int k;
0<=k<a.length; a[k]==b[i])==0)
//@ensures \result != null &&
//\result è ordinato
//@  (\forall int i; 0<=i<\result.length-1; \result [i]<= \result [i+1]) &&
// tutti gli elementi di a stanno in \result
//@      (\forall int i; 0<=i<a.length; (\exists int j; 0<=j<\result.length; a[i]== \result[j])) &&
// tutti gli elementi di b stanno in \result
//@      (\forall int i; 0<=i<b.length; (\exists int j; 0<=j<\result.length; b[i]== \result[j])) &&
// \result non contiene elementi spuri
//@      \result.length = a.length + b.length
```

b- si ammettono anche i casi in cui i due array a e b abbiano elementi ripetuti, e non siano disgiunti

```
//@assignable \nothing
//@requires a!= null && b!= null &&
//@ensures \result != null &&
//\result è ordinato
//@  (\forall int i; 0<=i<\result.length-1; \result[i]<= \result[i+1]) &&
//il numero di volte in cui un qualsiasi numero compare in \result è pari alla somma
// del numero di volte incui compare in a e del numero in cui compare in b
//@  (\forall int i; 0<=i && i<\result.length;
//      (\num_of int j; 0<=j<\result.length; \result[j]==\result[i])) ==
//          (\num_of int j; 0<=j<a.length; a[j]== \result[i])+ (\num_of int j; 0<=j<b.length; b[j]== \result[i])
// \result non contiene elementi spuri: ora è diventato pleonastico
//@      \result.length = a.length + b.length
```

Esercizio 2 (punti 12)

Si consideri una versione semplificata del gioco della dama (italiana) in cui le pedine possono solo muovere in avanti. **Scacchiera** e **Pedina** sono le classi principali della soluzione presentata. Il gioco, al quale prendono parte due giocatori, si svolge su una scacchiera formata da 64 caselle: 32 bianche e 32 nere. Le pedine sono 24: 12 bianche e 12 nere e si dispongono sulle caselle scure come indicato in figura.



Le righe e le colonne della scacchiera sono numerate da 1 a 8 dall'alto in basso e da sinistra a destra. La prima pedina nera è quindi in posizione 1:1 (riga:colonna), mentre l'ultima pedina bianca è in posizione 8:8.

La classe **Scacchiera** fornisce i seguenti metodi:

- **public int numPezzi(int colore)** restituisce il numero di pedine sulla scacchiera di un certo colore (0 bianco e 1 nero).
- **public ArrayList<Pedina> pedine(int colore)** restituisce i pezzi sulla scacchiera come una lista di elementi.
- **public ArrayList<Pedina> diagonale(Pedina p, char direzione)** restituisce la sequenza di pedine sulla diagonale sinistra (direzione = s, dal basso a dx all'alto a sx) oppure destra (direzione = d, dal basso a sx all'alto a dx) viste dalla pedina p.

La classe **Pedina** fornisce i seguenti metodi :

- **public int colore()** restituisce il colore della pedina.
- **public int riga()** restituisce la riga in cui la pendina si trova nel momento della chiamata del metodo.
- **public int colonna()** restituisce la colonna in cui la pedina si trova nel momento della chiamata del metodo.

- a. Si scriva la post-condizione del metodo pubblico **nuovaPartita()** della classe **Scacchiera**

il metodo numPezzi è ridondante rispetto al metodo size() applicato al risultato di pedine().

//@ requires true;

```
//@ ensures
// presi i due colori
(\forallall int i; 0 <= i <= 1;
 // il numero di pezzi è uguale a 12
 numPezzi(i) == 12 &&
 // tutte le pedine
 (\forallall int j; 0 <= j < pedine(i).size());
 // stanno sulle caselle nere
 pedine(i).get(j).riga() + pedine(i).get(j).colonna() % 2 == 0 &&
 // e stanno sulle righe previste
 pedine(i).get(j).colore() == 0 ? 6 <= pedine(i).get(j).riga() <= 8 : 1 <= pedine(i).get(j).riga() + <= 3))
```

- b. Si scriva la pre- e post-condizione del metodo **public bool mossaPossibile(Pedina p, char direzione)** della classe **Scacchiera**. Ogni pedina ha due sole possibilità di movimento di una posizione in diagonale: le pedine bianche si possono muovere di una casella verso l'alto a sinistra (direzione vale s) oppure a destra (direzione vale d), quelle nere verso il basso a sinistra o a destra. La casella di destinazione deve essere libera da pedine. Ad esempio, se una pedina bianca in posizione 6:6 volesse muoversi in direzione s (sinistra), questo significherebbe spostarla in posizione 5:5 a patto che la casella sia libera. Il metodo ritorna true se la mossa può essere effettuata correttamente, false altrimenti.

```
// se p non esiste o se non ci sono pedine dell'altro colore, non ha senso porsi il problema
//@ requires p!=null && numPedine(1-p.colore()) > 0 &&
```

```
1<=p.riga()<=8 && 1<=p.colonna()<=8 && 0<=p.colore()<=1 &&
(direzione == 's' || direzione == 'd')
```

```
// ipotizzando che il metodo diagonale() restituisca sempre la diagonale,
// al più con elementi null se la cella è libera
// e che che il primo elemento restituito da diagonale sia
// la pedina stessa
```

```
//@ ensures diagonale(p, direzione).size() > 1 &&
diagonale(p, direzione).get(1)==null ? \result == true: \result == false
```

- c. Si scriva la pre- e post-condizione del metodo **void muovi(Pedina p, char direzione)** della classe **Scacchiera**, sapendo la mossa deve essere possibile e l'invocazione del metodo modifica la posizione di p che è anche l'unica pedina che cambia posizione.

```
//@ requires mossaPossibile(p, direzione);
```

```
//@ ensures p.colore()==0? (p.riga()==\old(p.riga))-1 &&
direzione=='s'? p.colonna()==\old(p.colonna)-1:
p.colonna()==\old(p.colonna)+1:
(p.riga()==\old(p.riga))+1 &&
direzione=='s'? p.colonna()==\old(p.colonna)+1:
p.colonna()==\old(p.colonna)-1) &&
```

```
// p è l'unica pedina che si muove
```

```
(forall int i; 0<=i<=1;
(forall int j; 0<=j<pedine(i).size(); pedine(i).get(j) != p ->
pedine(i).get(j).riga() == \old(pedine(i).get(j).riga()) &&
pedine(i).get(j).colonna() == \old(pedine(i).get(j).colonna())))
```

d. Si scriva l'invariante privato della classe **Scacchiera**, ipotizzando il REP indicato qui sotto e ricordando che le pedine possono stare solo sulle caselle nere e che le pedine di ogni colore non possono essere più di 12:

```
private Pedina[8][8] griglia;

//@ private invariant (\forall int k, 0<=k<=1,
//  (\sum int i; 0<=i<8;
//   (\num_of int j; 0<=j<8; griglia[i][j].colore()==k))<=12) &&
// qui avremmo anche potuto usare anche il metodo pubblico numPezzi()

//  (\forall int i; 0<=i<8;
//   (\forall int j; 0<=j<8; griglia[i][j]!=null->(i+j)%2==1))
```

Esercizio 3 (punti 7)

Si consideri la classe **Poly**, definita a lezione. Si supponga che in essa siano definiti anche i metodi **valuta()** e **soluzioneMinima()**. Il primo metodo calcola il valore float del polinomio corrispondente al parametro. Ad esempio, **valuta(3.0)**, quando è chiamato sul Poly $-2x^2+3x+2$, restituisce -7. Il secondo metodo restituisce la radice reale minima del Poly, se essa esiste, ossia il più piccolo valore reale x' per cui $\text{valuta}(x') == 0$. Per il Poly dell'esempio, **soluzioneMinima()** restituisce $x' = -0.5$ (l'altra soluzione è 2.0). La soluzione calcolata è sempre esatta (nel senso che è calcolata fino alla massima precisione possibile con il tipo float). Per il teorema di Ruffini, è noto che se il grado del polinomio supera quattro, la soluzione non può essere calcolata in modo esatto tramite operazioni razionali ed estrazioni di radice. Pertanto, il metodo è definito solo se il grado del polinomio non supera 4.

```
public /*@ pure */ class Poly {  
    ...  
    //@ensures (*\result è la valutazione del polinomio quando la variabile vale var *);  
    public float valuta(float var);  
  
    //@requires this.deg() <=4;  
    //@ensures valuta(\result) ==0 && !(exists float f; f<\result; valuta(f) ==0);  
    //@signals (NessunaSoluzioneRealeException e) !(exists float f; ; valuta(f) ==0);  
    public float soluzioneMinima() throws NessunaSoluzioneRealeException  
}
```

Un'estensione di Poly calcola la soluzioneMinima anche quando il grado è superiore a 4. In tal caso, dovrà ovviamente ricorrere a tecniche di calcolo numerico. Pertanto, la soluzione sarà approssimata. In particolare, si richiede che la valutazione del risultato differisca dallo zero al più di 0.001.

```
public /*@ pure */ class NuovoPoly extends Poly {  
    //@also  
    //@requires true;  
    //@ensures -0.001<=valuta(\result) <=0.001 &&  
    //@ !(exists float f; f < \result - 0.001; -0.001<=valuta(f) <=0.001);  
    public float soluzioneMinima() throws NessunaSoluzioneRealeException;  
}
```

La classe NuovoPoly verifica il principio di sostituzione? Mostrare in dettaglio come mai ciò avviene, costruendo le pre e post condizioni complete per il metodo **soluzioneMinima()** in base alle regole del linguaggio JML. La risposta sarà considerata valida solo se correttamente motivata.

La regola delle proprietà è ovviamente verificata (in quanto il metodo **soluzioneMinima()** è solo un osservatore, e non può quindi influenzare proprietà dello stato astratto).

Apparentemente, la postcondizione di **NuovoPoly.soluzioneMinima()** è più debole della postcondizione di **Poly.soluzioneMinima()**. Tuttavia, la regola dei metodi è verificata, perché la nuova postcondizione si applica solo ai casi non previsti dalla precondizione originale (cioè solo per $this.deg() > 4$). Per il caso con $this.deg() \leq 4$, è sufficiente che la nuova postcondizione sia non contraddittoria con la postcondizione originale. In dettaglio, la regola JML delle specifiche ereditate costruisce la seguente specifica complessiva per il metodo **NuovoPoly.soluzioneMinima()**

```
//@requires this.deg()<=4 || true  
//@ensures (this.deg()<=4 ==> valuta(\result) ==0 && !(exists float f; f<\result; valuta(f) ==0))  
//@     && (true ==> -0.001<=valuta(\result) <=0.001 &&  
//@             !(exists float f; f<\result-0.001; -0.001<=valuta(f) <=0.001);
```

che può essere riscritta come:

```
//@requires true  
//@ensures (this.deg()<=4 ==> valuta(\result) ==0 && !(exists float f; f<\result; valuta(f) ==0)) &&  
//@     -0.001<=valuta(\result) <=0.001 && !(exists float f; f<\result-0.001; -0.001<=valuta(f
```

Pertanto, se il grado è minore o uguale a 4, la postcondizione garantisce che la soluzione (se esiste) sia esatta (e quindi a fortiori verifichi anche il vincolo sull'"approssimazione"); negli altri casi, la soluzione sarà solo approssimata. La regola dei metodi è pertanto verificata.

Appello del 22 Settembre 2014



Politecnico di Milano
Anno accademico 2010-2011

Ingegneria del Software

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Esercizio 1

Si deve progettare una classe `Clock`, che realizza oggetti che rappresentano ora, minuto e secondo. La classe offre le operazioni `click(x)`, che fa avanzare il tempo di `x` secondi, e le operazioni `getHour`, `getMin` e `getSec`, che restituiscono l'ora, il minuto e il secondo di un oggetto `Clock`. La classe implementa pure l'interfaccia `Comparable`, di cui si riporta la definizione:

```
public interface Comparable<T> {
    //@ensures (* if this<arg then \result == -1
    //@ else if this == arg then \results == 0,
    //@     else \result == 1 *);
    public int compareTo(T arg);
}
```

Domanda 1

Fornire la specifica JML del metodo `click`. Il metodo non solleva eccezioni.

Domanda 2

Definire una rappresentazione (rep) per la classe. Specificare l'eventuale rep invariant e fornire un'implementazione della funzione di astrazione, come implementazione dell'operazione `toString`.

Domanda 3

Definire un'implementazione del metodo `compareTo`.

Esercizio 2

Si consideri la seguente specifica di una classe Lista. Una Lista è una sequenza di elementi che consente ricerca, inserimento ed eliminazione. (Per l'interfaccia Comparable, si faccia riferimento all'Esercizio 1)

```
public class Lista<T extends Comparable<T>> {  
    /*OVERVIEW: Collezione mutabile di oggetti, di tipo T,  
     * organizzati in una sequenza.  
     * Oggetto tipico di dimensione n: x0 x2 ... xn-1,  
     * in cui x0 e' il primo elemento e xn-1 l'n-esimo.  
     * Rimozione e inserimento di elementi possono avvenire  
     * in una posizione qualunque.  
     */  
  
    //@ensures (*inizializza this alla Lista vuota*);  
    public Lista()  
  
    //osservatori puri:  
  
    //@ensures (* \result == numero di elementi di this *)  
    public /*@ pure @*/ int size()  
  
    //@ensures (* \result <==> c'e' elemento equals(x) in this *)  
    public /*@ pure @*/ boolean contains(T x)  
  
    //@requires i>=0 && i < size();  
    //@ensures (* \result == l'elemento di this di posizione i-esima;  
    public /*@ pure @*/ T get(int i )  
  
    //mutators:  
  
    //@requires size()>0;  
    //@ ensures (* elimina e restituisce il massimo elemento  
    //@ (secondo compareTo) in this *);  
    public T extractMax()  
  
    //@ensures (* aggiunge x in una qualunque posizione della lista *);  
    public void insert(T x)  
  
    //@requires size()>0;  
    //@ ensures (* elimina da this una comparsa di un elemento equals(x) *)  
    public void remove(T x)  
}
```

Domanda 1

Specificare in JML i metodi `insert` e `extractMax`, evitando di inserire vincoli non richiesti. Ad esempio, non si forzi la scelta di una posizione specifica in cui inserire l'elemento nel caso dell'operazione `insert`.

Domanda 2

Si consideri una classe `ListaSenzaRipetizioni`, definita come `Lista`, ma con il vincolo che non possono esistere elementi ripetuti. Pertanto l'operazione `insert` non inserisce un elemento se questo è già presente nella lista.

1. È possibile definire `ListaSenzaRipetizioni` come sottoclasse di `Lista` o viceversa rispettando il principio di sostituzione?
2. Fornire la specifica JML dell'operazione `insert`.
3. Definire un invarianto astratto per la classe che specifica l'assenza di elementi ripetuti.

Esercizio 3

Si supponga di disporre di una classe con metodi che accettano parametri e restituiscono risultati in unità di misura cosiddette *British Imperial*, cioè piedi, miglia, etc. Si vuole riusare questa classe, ma utilizzando parametri e risultati che siano espressi nel sistema *Metrico*. Spiegare se e come il problema può essere risolto utilizzando design pattern opportuni. Fornire anche una bozza della soluzione usando un diagramma delle classi UML.

Esercizio 4

Si consideri il seguente frammento di programma che opera su interi:

```
1. ''legge n da input'';
2. if (n>0) {
3.     j = n*(n-1);
4.     while (j>0) {
5.         ----//qui codice che non cambia n e j
6.         j--;
7.     }
8.    ----;
9. }
```

1. Si definisca un insieme minimo di casi di test che coprano tutte le istruzioni (si considerino le parti tralasciate ---- come se fossero una singola macro-istruzione);
2. Si definisca un insieme minimo di casi di test che coprano tutti i branch.
3. Si consideri un cammino che esegue il ciclo due volte. Si scriva l'espressione logica che deve essere soddisfatta per la percorrenza di detto cammino e si identifichi un valore di input che causa la percorrenza dello stesso.

Esercizio 5

Si consideri una tabella di dati, realizzata come singleton. Ogni riga della tabella contiene una chiave intera ed una stringa come valore. La tabella offre le seguenti operazioni:

1. **searchAndGet** riceve come parametro una chiave e restituisce tutte le stringhe memorizzate nella tabella di dati con quella chiave, cancellandola dalla tabella stessa.
2. **insert** riceve un dato (stringa), con chiave associata, e lo memorizza.

La tabella viene usata in un ambiente concorrente da più thread. Poiché essa ha una dimensione predefinita, quando risulta piena, l'invocazione dell'operazione `insert` sospende il task che invoca l'operazione in attesa che venga liberato spazio dall'invocazione di `searchAndGet`.

Si fornisca un'implementazione Java del singleton Tabella.



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da

Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Il Prova di Ingegneria del Software

2 Luglio 2003

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
 2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità.
Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
 3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
1. Non è possibile lasciare l'aula conservando il tema della prova in corso.
 2. Tempo a disposizione: 1h30.
 3. Punteggio totale a disposizione: 13/30.

Esercizio 1

Si consideri la seguente specifica di un ADT:

```
public class Sorter {  
    public static int searchSorted (int[] a, int x);  
        //REQUIRES: a != null e ordinato per valori crescenti  
        //EFFECTS: se x ∈ a ritorna un indice i per cui a[i]==x, altrimenti -1  
}
```

Si consideri inoltre il seguente main che invoca il metodo searchSorted:

```
public static void main (String [] args) {  
//qui compare la definizione dei metodi getData(N), che legge N dati e li restituisce in un array,  
//getDatum, che legge un intero e lo restituisce come risultato, e output, che stampa in uscita un intero  
    int[] data = getData(10); int datum = getDatum(); int result;  
    result = searchSorted(data, datum);  
    output(result);  
}
```

1. Si supponga che la chiamata del metodo *getData* consenta di produrre i seguenti valori del vettore data:

- a) null
- b) [0, 13, 14, 27, 8, 33]

Spiegare perché l'esecuzione del metodo *searchSorted* potrebbe generare un errore a run-time o un risultato scorretto e spiegare a quale metodo debba essere attribuita la responsabilità dell'errore.

risp.: la responsabilità è di chi chiama il metodo, in quanto la precondizione del metodo specifica che l'array sia non nullo e ordinato.

2. Al fine di operare un test funzionale del metodo *searchSorted*, quali dati di test generereste come valori di data e datum?

risp.: un caso in cui datum appartiene e un caso in cui datum non appartiene all'array. Per testare i casi limiti, il caso in cui l'elemento da cercare sia il primo o l'ultimo dell'array.

3. Si consideri la seguente implementazione del metodo *searchSorted*. Si supponga di volerne operare la copertura dei cammini nel caso di array di dimensione 10. Qual è il numero di cammini possibili?

```
public static int searchSorted (int[] a, int x)  
    //uses linear search  
    for (int i =0; i<a.length; i++) {  
        if (a[i]==x) return 1;  
        else if (a[i]>x) return -1;  
    }  
    return -1;
```

risp.: 21: ci sono fino a 10 iterazioni del ciclo. Per ogni iterazione vi sono due modi per uscire dal ciclo. Allora il numero di cammini è 20, più un cammino corrispondente al caso in cui si esce dal ciclo solo dopo avere esaminato tutti gli elementi.

Esercizio 2

- a) Un'applicazione d è utilizzata per produrre paghe e contributi in un'azienda. I gestori del sistema decidono di sostituire la vecchia stampante con una nuova stampante più veloce. L'applicazione deve essere mantenuta per potere utilizzare la nuova stampante. Motiva sinteticamente come classificheresti questo intervento di manutenzione.

risp. manutenzione adattativa

- b) Considera una classe che definisce un tipo di dato astratto che realizza information hiding (la rep non è visibile all'esterno). Descrivi sinteticamente un tipo di manutenzione perfettiva che potrebbe essere alla base della modifica.

risp. potrebbe essere manutenzione perfettiva per migliorare l'efficienza.

Esercizio 3

Si definisca un iteratore evenNumbers che fornisce in successione i numeri interi pari.

risp. si veda l'esempio allPrimes di Liskov.

Esercizio 4

Si considerino le seguenti specifiche:

```
public class CharBuffer {  
    //OVERVIEW: Un buffer (cioè una coda limitata) di caratteri. I caratteri non possono essere rimossi  
    //dal buffer. Il contenuto del buffer può essere visualizzato con toString()  
    //costruttori:  
    public CharBuffer() //EFFECTS: inizializza this al buffer vuoto
```

```
    public void insert(char x) throws BufferPienoException  
        //EFFECTS: if this è pieno lancia BufferPienoException else inserisce x in this  
        public String toString()//EFFECTS: restituisce una rappresentazione testuale di this  
    }
```

```
public class CharBuffer1 extends CharBuffer {  
    //OVERVIEW: Un buffer (cioè una coda limitata) di caratteri minuscoli. I caratteri non possono //essere rimossi  
    //dal buffer. Il contenuto del buffer può essere visualizzato con toString()  
    //costruttori:  
    public CharBuffer1() //EFFECTS: inizializza this al buffer vuoto  
    public void insert(char x) throws BufferPienoException  
        //EFFECTS: if this è pieno lancia BufferPienoException else  
        // se x e' minuscolo inserisce x in this, se non e'minuscolo non inserisce nulla  
    }
```

a) Il principio di sostituzione viene violato da CharBuffer1? La risposta deve essere giustificata.

risp: si, perche la post di insert non inserisce caratteri minuscoli e quindi e' piu' debole. Le altre regole, -segnature e proprieta'- sono invece verificate.

Si consideri ora la seguente specifica:

```
public class CharBuffer2 extends CharBuffer{  
    //costruttori:  
    public CharBuffer2(Character x) //EFFECTS: inizializza this al buffer che contiene il carattere contenuto in x  
    //metodi:  
    public void insert(Character x) throws BufferPienoException  
        //EFFECTS: if this è pieno lancia BufferPienoException else inserisce in this il carattere contenuto in x se e'  
        //minuscolo (se non e' minuscolo, non inserisce nulla).  
    }
```

b) Il principio di sostituzione viene violato da CharBuffer2? La risposta deve essere giustificata.

Risp: No, in quanto la insert(character x) e' definita per overloading e quindi non viola la regola della precondizione. Il costruttore di CharBuffer2, inoltre, può essere ridefinito liberamente (non viola la regola della segnatura).

Esercizio 5

Si consideri il seguente frammento UML che esprime l'associazione tra un calciatori e squadre. Si vuole arricchire il modello per esprimere le seguenti proprietà:

- 1) una squadra deve avere almeno 11 calciatori, e non esistono vincoli al numero di calciatori tesserati e un calciatore puo' essere associato al piu' a 1 squadra;
- 2) in alternativa al caso 1), l'associazione descrive tutte le squadre per cui un calciatore ha giocato. Si vuole anche che venga specificata, per ogni squadra per cui il calciatore ha giocato, l'ultima stagione calcistica (anno) in cui cio' e' avvenuto e il numero di goals segnati.
- 3) Nel caso 1), si introduca anche l'entita' Allenatore e la sua associazione con Squadra, specificando che una squadra deve avere esattamente un allenatore e un allenatore puo' allenare al piu' una squadra.



Traccia risposta: I casi 1 e 3 sono ovvi. Per il caso 2 occorre introdurre opportuni attributi per la l'associazione (anno goals), oltre a modificare le cardinalità.

Esercizio 1

Si vuole progettare un'applicazione Java per gestire aste online. A tale scopo, si è specificata la classe `AuctionManager`, riportata più avanti (molto semplificata rispetto alla realtà, p.es. nella gestione delle scadenze o nel tracciamento dei partecipanti).

Ad ogni cliente è associato un customer id univoco e segreto (modellato per semplicità con un `long`), con cui i clienti possono sia vendere che fare offerte. La gestione dei clienti è gestita da classi opportune, che sono ignorate qui.

Per mettere in vendita un oggetto, si deve passare al metodo `addItem` un oggetto della classe immutabile `Item`, oltre al customer id del venditore. È responsabilità del venditore creare e descrivere l'oggetto corrispondente, tramite i metodi della classe `Item` (in gran parte ignorati qui):

```
public /*@ pure @*/ class Item {
    public Item(long sellerId, ...altri parametri ignorati)
        ...altri metodi ignorati
    // restituisce il customer id di chi ha creato l'oggetto
    public long sellerId();
    // Restituisce la base d'asta per l'item
    public double getStartingPrice();
}

public class AuctionManager {
    // Costruisce un gestore di aste, senza ancora alcun oggetto in vendita.
    public AuctionManager();

    // Il cliente con codice sellerId mette in vendita l'oggetto item, non ancora messo in asta,
    // Se l'item corrisponde ad un oggetto già in asta, rilancia WrongItemException;
    public void addItem(Item item, long sellerId) throws WrongItemException;

    // Il venditore con sellerId che ha messo in vendita l'oggetto item, accetta
    // la migliore offerta corrente. Il metodo ritorna la migliore offerta corrente
    // e ritira (rimuove) l'oggetto dall'asta.
    // Se non vi sono state offerte, lancia eccezione NoBidException.
    // WrongItemException è lanciata se il sellerId non corrisponde all'item
    // o se l'item non risulta in asta.
    public double knockDown(Item item, long sellerId) throws WrongItemException, NoBidException;

    // Il cliente con buyerId propone un'offerta di valore newBid, superiore alla massima
    // corrente (altrimenti si lancia BidTooLowException), per l'oggetto item in asta.
    // Se l'item non è in asta si lancia WrongItemException
    public void bid(Item item, long buyerId, double newBid) throws WrongItemException,
        BidTooLowException;

    // Restituisce l'insieme di tutti gli oggetti attualmente in asta
    public /*@ pure @*/ Set<Item> auctionedItems();

    // Restituisce la migliore offerta corrente per l'item (se è in asta).
    // Se non vi sono state offerte restituisce la base d'asta.
    // Se l'item non è in asta si lancia WrongItemException
    public /*@ pure @*/ double getBestBid(Item item) throws WrongItemException;
}
```

Domanda a)

Specificare in JML il metodo bid. Si consiglia di introdurre abbreviazioni per rendere la specifica più semplice e leggibile.

Soluzione

otherItemsUnchanged(item) sta per

```
(\forall Item it; auctionedItems().contains(it) && it!=item;
  getBestBid(it)==\old(getBestBid(it)) &&
  \old(auctionedItems()).contains(it)) &&
(\forall Item it; \old(auctionedItems()).contains(it) && it!=item
  auctionedItems().contains(it))

unchanged(item) sta per

getBestBid(item)==\old(getBestBid(item)) &&
auctionedItems().contains(item) && \old(auctionedItems()).contains(item)

/*@ ensures \old(auctionedItems()).contains(item) && auctionedItems().contains(item) &&
   newBid>\old(getBestBid(item)) && newBid==getBestBid(item) &&
   otherItemsUnchanged(item)

   signals(WrongItemException e) !\old(auctionedItems().contains(item)) &&
   !auctionedItems().contains(item) && otherItemsUnchanged(item)

   signals(BidTooLowException e) newBid<=getBestBid(item) &&
   unchanged(item) && otherItemsUnchanged(item)
*/
public void bid(Item item, long buyerId, double newBid) throws WrongItemException, BidTooLowException;
```

Domanda b)

Specificare il metodo knockDown.

Soluzione

```
/*@ ensures item.sellerId()==sellerId && \old(auctionedItems().contains(item)) &&
   !auctionedItems().contains(item) && \result==\old(getBestBid(item)) &&
   \old(getBestBid(item))>getStartingPrice(item) && otherItemsUnchanged(item)

   signals(WrongItemException e) (otherItemsUnchanged(item) &&
   !(\old(auctionedItems()).contains(item) && !auctionedItems().contains(item)) || 
   (item.sellerId()!=sellerId && unchanged(item))

   signals(NoBidException e) \old(getBestBid(item))==getStartingPrice(item) &&
   otherItemsUnchanged(item) && unchanged(item);
*/
public knockDown(Item item, long sellerId) throws WrongItemException, NoBidException;
```

Domanda c)

Si consideri l'implementazione seguente del rep:

```
private Set<Item> currentItems; // gli item all'asta
private Map<Item, Double> bestBids; // l'offerta migliore per ciascun item;
```

Si ricorda che una `Map<K, V>` è una struttura dati che associa a chiavi di tipo K (Item in questo caso) valori di tipo V (Double in questo caso). Una Map fornisce, fra gli altri, i metodi osservatori `get(K key)`, che restituisce il valore univoco associato alla chiave key, e `containsKey(K key)`, che restituisce true se esiste un valore associato alla chiave key. Altri metodi sono ad esempio `size()` e `isEmpty()`, di significato ovvio.

Si scrivano l'invariante di rappresentazione e la funzione di astrazione relative a questo rep.

Soluzione

RI:

```
private invariant
currentItems!=null && bestBids!=null &&
!bestBids.containsKey(null) &&
(\forall Item it; currentItems.contains(it); bestBids.containsKey(it) &&
(\forall Item it; bestBids.containsKey(it)); currentItem.contains(it))
```

AF:

```
private invariant
auctionedItems().size()==currentItems.size() && auctionedItems().containsAll(currentItems) &&
(\forall Item it; currentItems.contains(it); getBestBid(it)==bestBids.get(it))
```

Domanda d)

Si vuole aggiungere alla classe AuctionManager un metodo per consentire al venditore, qualora non vi siano state offerte, di introdurre uno sconto sulla base d'asta:

```
public void discount(Item item, long sellerId, double discount)
    throws WrongItemException, AlreadyBiddedException;
```

È possibile definire, senza violare il principio di sostituzione, una sottoclasse di AuctionManager che abbia anche questo metodo? Motivare la risposta.

Soluzione

Non è possibile in quanto il nuovo metodo viola un'importante proprietà evolutiva della classe: invocazioni successive del metodo `getBestBid()` per uno stesso `item` ritornano valori monotoni non decrescenti. L'introduzione del metodo `discount()` permetterebbe invece di ridurre il valore ritornato da due invocazioni di `getBestBid()`, eseguite rispettivamente prima e dopo l'invocazione di `discount()`. NB: Lo sconto non modifica la base d'asta memorizzata nell'`item` (in quanto immutabile), ma il valore ritornato da `getBestBid()` quando non vi sono ancora state offerte: `getBestBid(item)` in questo caso dovrebbe ritornare `item.getStartingPrice() * discount(item)` invece di `item.getStartingPrice()`.

Esercizio 2

Si consideri la seguente classe `ScoreBoard` per memorizzare i voti di un insieme di studenti passato in fase di costruzione:

```
public class ScoreBoard {
    private String[] students;
    private int[] scores;
    public ScoreBoard(String[] stud) {
        students = new String[stud.length];
        for(int i=0; i<stud.length; i++) students[i] = stud[i];
        scores = new int[stud.length];
    }
    public int getScore(String stud) {
        for(int i=0; i<students.length; i++) {
            if(students[i].equals(stud)) return scores[i];
        }
        return -1;
    }
    public void setScore(String stud, int score) {
        for(int i=0; i<students.length; i++) {
            if(students[i].equals(stud)) scores[i] = score;
        }
    }
}
```

Domanda a)

Si introduca l'opportuno codice di sincronizzazione nei metodi `getScore` e `setScore` al fine di massimizzare il parallelismo, consentendo a thread diversi di accedere in parallelo ai dati di studenti diversi, evitando conflitti nell'accesso ai dati (voto) del medesimo studente.

Soluzione

L'array `students` è immutabile. Possiamo scorrerlo senza rischio di conflitti in accesso e possiamo usarlo come array di oggetti su cui sincronizzarsi quando si deve leggere o scrivere un voto.

```

public class ScoreBoard {
    private String[] students;
    private int[] scores;
    public ScoreBoard(String[] stud) {
        students = new String[stud.length];
        for(int i=0; i<stud.length; i++) students[i] = stud[i];
        scores = new int[stud.length];
    }
    public int getScore(String stud) {
        for(int i=0; i<students.length; i++) {
            if(students[i].equals(stud)) synchronized(students[i]) { return scores[i]; }
        }
        return -1;
    }
    public void setScore(String stud, int score) {
        for(int i=0; i<students.length; i++) {
            if(students[i].equals(stud)) synchronized(students[i]) { scores[i] = score; }
        }
    }
}

```

NB: la sincronizzazione sulla String stud, invece, puo' portare ad errori di sincronizzazione, in quanto Java non garantisce che due stringhe uguali siano sempre lo stesso oggetto! Java usa automaticamente lo "string interning" per espressioni del tipo:

String s1=="John", String s2=="John" (quindi s1==s2),

ma non per String s3=new("John") (quindi s1!=s3).

Domanda b)

Si modifichi il metodo `getScore` affinchè sospenda il chiamante se il voto dello studente cercato è minore o uguale a zero. Si specifichi se e come sia necessario modificare altri metodi.

Soluzione

```

public class ScoreBoardSync1 {
    private String[] students;
    private int[] scores;
    public ScoreBoardSync1(String[] stud) {
        students = new String[stud.length];
        for(int i=0; i<stud.length; i++) students[i] = stud[i];
        scores = new int[stud.length];
    }
    public int getScore(String stud) throws InterruptedException {
        for(int i=0; i<students.length; i++) {
            if(students[i].equals(stud))
                synchronized(students[i]) {
                    while(scores[i]<=0) students[i].wait();
                    return scores[i];
                }
        }
        return -1;
    }
    public void setScore(String stud, int score) {
        for(int i=0; i<students.length; i++) {
            if(students[i].equals(stud))
                synchronized(students[i]) {
                    scores[i] = score;
                    students[i].notifyAll();
                }
        }
    }
}

```

Esercizio 3

```
public static int f(int n) {
    if (n <= 0 || n > 6) return 0;
    if (n%2 != 0) n = n + 1;
    int a = 0;
    while (n > 1) {
        a = a + n;
        n = n - 2;
    }
    return a;
}
```

Si determini per la funzione `f()` un insieme minimo di casi di test per ciascuno dei seguenti criteri di copertura.

- (a) Copertura delle istruzioni (statement coverage)
- (b) Copertura delle decisioni (edge coverage)
- (c) Copertura dei cammini (path coverage)

Soluzione

- (a) Sono necessari (e sufficienti) due casi, uno che entri nel primo `if` e uno che copra tutte le successive istruzioni, sia quelle all'interno del corpo del secondo `if`, sia quelle all'interno del `while`. Il primo caso deve avere `n <= 0` oppure `n > 6`, ad esempio `n = 0`. Il secondo caso deve avere `n` dispari, ad esempio `n = 1`.
- (b) Ai casi precedenti è necessario aggiungere un caso con `n` pari, per valutare negativamente la condizione del secondo `if`, ad esempio `n = 2`.
- (c) In tutto sono possibili 7 cammini:
 - (1) Un primo cammino entra solo nel primo `if`, ad esempio con `n = 0`.
 - (2) Con `n = 1` si entra nel secondo `if` e si entra nel ciclo `while` 1 volta.
 - (3) Con `n = 2` NON si entra nel secondo `if` e si entra nel ciclo `while` 1 volta.
 - (4) Con `n = 3` si entra nel secondo `if` e si entra nel ciclo `while` 2 volte.
 - (5) Con `n = 4` NON si entra nel secondo `if` e si entra nel ciclo `while` 2 volte.
 - (6) Con `n = 5` si entra nel secondo `if` e si entra nel ciclo `while` 3 volte.
 - (7) Con `n = 6` NON si entra nel secondo `if` e si entra nel ciclo `while` 3 volte.

Esercizio 4

Si considerino le seguenti classi Java, in cui viene omessa l'ovvia implementazione dei metodi osservatori `getMembers()`, `getAge()`, `getName()`.

```
public class Group {
    public List<Person> getMembers() { ... }
}

class Person {
    public String getName() { ... }
    public int getAge() { ... }
}
```

Si risponda ai seguenti quesiti facendo uso *esclusivamente* di costrutti della programmazione funzionale.

Domanda a)

Si completi l'implementazione del seguente metodo per fare in modo che stampi l'età media delle persone appartenenti al gruppo. Se il gruppo è vuoto, il metodo non deve stampare nulla.

```
public static void printAverage(Group group) {
    ...
}
```

Soluzione

```
public static void printAverage(Group group) {
    group.getMembers().stream()
        .mapToInt(p -> p.getAge())
        .average()
        .ifPresent(System.out::println);
}
```

Domanda b)

Si completi l'implementazione del seguente metodo per fare in modo che ritorni una lista contenente i nomi di tutte le persone maggiorenne presenti all'interno dei gruppi in `groups`.

```
public static List<String> namesInGroups(List<Group> groups) {
    return ...
}
```

Soluzione

```
public static List<String> namesInGroups(List<Group> groups) {
    return groups.stream()
        .flatMap(g -> g.getMembers().stream())
        .filter(p -> p.getAge() > 18)
        .map(p -> p.getName())
        .collect(Collectors.toList());
}
```



Politecnico di Milano

Anno accademico 2013-2014

Ingegneria del Software – Appello del 23 luglio 2014

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi

Ghezzi

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Esercizio 1

Durante la realizzazione di un gioco multi-giocatore, occorre definire una struttura dati, adatta a rappresentare un labirinto. Il labirinto è costituito da stanze, collegate da porte magiche. Vi è una stanza di ingresso al labirinto e un certo numero di stanze di uscita dallo stesso. Ogni stanza ha un nome (una stringa) ed è dotata di almeno una porta per passare istantaneamente in una stanza diversa. Anche ogni porta ha un nome; se da una stanza x esiste una porta verso una stanza y , allora questa porta è unica.

Il labirinto in Java è rappresentato da una classe immutabile, i cui esemplari sono costruiti a partire da un file opportuno, caratteristica che per semplicità qui ignoriamo. Di seguito è riportata la specifica delle classi rilevanti.

```
public /*@ pure @*/ class Stanza {

    //s e' l'insieme delle porte in uscita a this
    public Stanza(Set<Porta> s, String nome);

    //@ensures (* \result e' l'insieme di porte in uscita dalla stanza this*)
    public Set<Porta> porteOut();

    //@ensures (* \result e' il nome della stanza this*)
    public String nome()
}

public /*@ pure @*/ class Porta {

    //x e' la stanza verso cui si viene trasportati
    public Porta(Stanza x, String nome)

    //@ensures (* \result e' la stanza a cui si viene trasportati passando attraverso this.*)
    public Stanza versoStanza();

    //@ensures (* \result e' il nome della porta this*)
    public String nome()
}

public /*@ pure @*/ class Labirinto {

    public Labirinto(...strutture adeguate a caricare un labirinto non vuoto...);

    //@ensures (* \result e' true se, e solo se, esiste una porta che
    //            collega direttamente la stanza x alla stanza y*)
    public boolean esistePorta(Stanza x, Stanza y)

    //@ensures (* \result e' la stanza di ingresso al labirinto*)
    public Stanza ingresso()

    //@ensures (* \result e' l'insieme di tutte le stanze del labirinto*)
    public Set<Stanza> stanze()

    //@ensures (* \result e' il sottoinsieme di stanze() che sono di uscita dal labirinto*)
    public Set<Stanza> uscite()

    //@ensures (*\result restituisce un iteratore che itera su tutte le porte di this,
    //            una alla volta, in ordine qualunque*)
    Iterator<Porta> porte()
}
```

Si rammenta che un Set è una collection di Java, in cui non vi possono essere elementi duplicati. Si può accedere agli elementi di un set esclusivamente tramite i consueti metodi `contains`, `containsAll`, `isEmpty` e `size`.

Domanda 1 Specificare in JML il metodo `esistePorta`.

SOLUZIONE: Ipotizziamo, qui e nel seguito, che i metodi delle varie classi non ritornino mai il valore null, ne' null sia un valore ammissibile per una porta o una stanza del labirinto.

```
//equires stanze().contains(x) && stanze().contains(y);
//@ensures \result ==
//@      (\exists Porta p; x.porteOut().contains(p); p.versoStanza().equals(y));
```

Domanda 2 Si vuole definire un `LabirintoStregato`, simile al `Labirinto`, ma in cui alcune delle `Porte` sono rimpiazzate da una `PortaStregata`. Una `PortaStregata` è identica a una `Porta`, salvo che la stanza a cui conduce è scelta casualmente fra tutte le stanze. Le classi `PortaStregata` e `LabirintoStregato` sono definite nel modo seguente:

```
public class PortaStregata extends Porta {
    public PortaStregata(Set<Stanza> stanze, String nome);
    @override
    //@ensures (*\result e' una stanza scelta a caso *);
    public Stanza versoStanza();
}

public class LabirintoStregato extends Labirinto {
    // ridefinizione del solo costruttore, che permette di
    // costruire un labirinto in cui una o piu' porte sono
    // stregate
}
```

Queste ridefinizioni verificano il principio di sostituzione? Giustificare la risposta.

SOLUZIONE:

La classe `PortaStregata` verifica la regola dei metodi, in quanto la postcondizione garantisce di restituire una stanza, che e' tutto quello che viene promesso dalla postcondizione originale. Tuttavia, la regola delle proprietà è violata, in quanto la classe `PortaStregata` diventa mutabile, ossia non è più pura: la stanza restituita è un invariante della classe originale. La classe `LabirintoStregato` non viola naturalmente la regola dei metodi, ma anch'essa viola la regola delle proprietà. Ad esempio, l'invariante che non ci sono in una stanza due porte che conducono a una stessa stanza, può essere violato, in quanto in modo causale in certi momenti due porte potrebbero riferirsi alla stessa stanza.

Domanda 3 Si consideri una generalizzazione del concetto di Porta, ipotizzando che ogni porta possa avere anche un effetto sulle caratteristiche dell'avatar che la attraversa. Per esempio, una Porta Magica consente all'avatar di assumere alcuni poteri magici per un certo tempo; una Porta Premio incrementa il punteggio del giocatore; una Porta Resurrezione fornisce una vita supplementare al giocatore; una Porta Maledizione procura una maledizione che rispedisce il giocatore a un livello precedente, ecc. Il tipo delle porte potrebbe cambiare a *run-time* e nuovi tipi di porta potrebbero essere definiti in base al particolare tipo di gioco che usa il labirinto. Quale potrebbe essere l'organizzazione di una struttura a classi che consente di realizzare un progetto flessibile ed estendibile? Non serve scrivere il codice Java, ma un diagramma delle classi UML potrebbe essere utile per rappresentare le diverse classi e le loro relazioni.

SOLUZIONE:

Si può utilizzare una Strategy. Supponendo che vi sia una classe Avatar, che descrive l'avatar di un giocatore, si potrebbe definire l'interfaccia:

```
interface ComportamentoPorta {
    //ensures (* agisce sull'avatar p *);
    public void effetto(Avatar p)
}
```

la classe Porta potrebbe essere così definita:

```
public class Porta {
    //questi metodi non cambiano:
    public Porta(Stanza x, String nome, ComportamentoPorta p)
    public Stanza versoStanza();
    public String nome();
    //un pezzo del rep:
    private ComportamentoPorta comportamento;
    //nuovo metodo per associare un comportamento
    public associaEffetto(ComportamentoPorta p) {
        this.comportamento == p;
    }
}
```

Ad esempio, il comportamento della Porta Premio potrebbe essere ottenuto così:

```
public class ComportamentoPortaPremio implements ComportamentoPorta {
    //ensures (* incremento il punteggio di p *)
    public void effetto (Avatar p);
}
```

Il pattern Strategy consente di modificare a run-time il comportamento della Porta, senza modificare la classe Labirinto. Un Abstract per la gerarchia delle porte non consente di cambiare facilmente il comportamento delle porte: occorre ogni volta una nuova istanza di Porta, copiarvi le altre caratteristiche della Porta esistente e infine memorizzare nel Labirinto la nuova porta al posto di quella vecchia (aggiungendo naturalmente i metodi opportuni per farlo): la struttura e' piu' goffa e poco flessibile.

Un Decorator puo' svolgere lo stesso ruolo di una Strategy, ma e' eccessivamente generale per risolvere questo problema particolare: un decorator e' utile quando ci sono piu' attributi/metodi che possono variare, anche a livello dell'interfaccia della classe, mentre lo Strategy e' da preferire quando si deve scegliere fra diverse implementazioni di un'operazione (a parita' di interfaccia).

Esercizio 2

Sia data per la classe `Labirinto` la seguente implementazione. Il numero totale di stanze del labirinto è memorizzato nel campo intero `numStanze`. Le stanze sono rappresentate da una map `mappaStanze`, che associa ad ogni stanza un intero progressivo (in un ordine arbitrario), partendo da 0. Lo 0 corrisponde sempre alla stanza di ingresso.

Il labirinto è rappresentato con una matrice quadrata `collegamenti`, di dimensione `numStanze × numStanze`. Nella posizione (i, j) della matrice è memorizzata, se esiste, la porta che collega la stanza di numero i con la stanza di numero j ; se la porta non esiste, allora è memorizzato il valore `null`.

```
private int numStanze;
private Map<Stanza, Integer> mappaStanze;
private Porta[][] collegamenti;
```

Si rammenta che l'interfaccia `Map<K, V>` prevede, fra l'altro, i seguenti metodi puri. Il metodo `V get (Object x)`, dato un `x` di tipo `K`, restituisce l'elemento di tipo `V` associato a `x`, se questo esiste (se non esiste, `get` restituisce `null`). Quindi se p.es. `x` è la stanza di ingresso, `get (x)` restituisce l'intero 0, ecc. Il metodo `size()` restituisce la dimensione della mappa (ossia il numero complessivo di stanze). Il metodo `keySet()` restituisce un `Set` che contiene tutte le chiavi (ossia tutte le stanze memorizzate nella mappa). Infine, il metodo `values()` restituisce una `Collection<V>` che contiene tutti i valori della mappa.

Domanda 1 Scrivere l'invariante di rappresentazione. Facoltativamente, si scriva anche la funzione di astrazione (utilizzando la tecnica preferita) della classe `Labirinto`.

SOLUZIONE:

```
//RI:  
private invariant collegamenti!= null && mappaStanze!= null &&  
//numStanze e' proprio la dim. della matrice collegamenti:  
    numStanze == collegamenti.length &&  
//la matrice collegamenti e' quadrata:  
    (\forall int i; 0<=i && i<numStanze; collegamenti[i].length == numStanze) &&  
//mappaStanze restituisce valori compresi fra 0 e numStanze:  
    (\forall Integer x;; mappaStanze.values().contains(x) <==> 0<= x && x< numStanze) &&  
//mappaStanze non contiene null:  
!mappaStanze.keySet().contains(null);
```

Si puo' descrivere anche la funzione di astrazione con un invariante privato, che correla i campi privati con i metodi osservatori:

```
private invariant  
//il numero di stanze corrisponde:  
    numStanze == stanze().size() &&  
//l'insieme di stanze della mappa e' proprio l'insieme di stanze del labirinto:  
    mappaStanze.keySet().equals(stanze()) &&  
//l'ingresso ha valore 0:  
    mappaStanze.get(ingresso()) == 0 &&  
//le porte nei collegamenti sono quelle corrette:  
    (\forall Stanza x; stanze().contains(x);  
        (\forall Stanza y; stanze().contains(x) && x!= y;  
            esistePorta(x,y) <==> collegamenti[mappaStanze().get(x)][mappaStanze().get(y)] !=null))
```

Domanda 2 Implementare l'iteratore `porte()`.

Esercizio 3

Si consideri il seguente programma

```
public static int gcd(int a, int b) { //valid for positive integers.  
    while (a !=b) {  
        if (a > b) a = a - b;  
        else b = a - b;  
    }  
    return a;  
}
```

1. Scrivere la condizione logica che deve essere soddisfatta dai valori in input (chiamati a e b) affinchè venga testato il cammino del programma che esegue il loop due volte, la prima eseguendo il ramo then e la seconda eseguendo il ramo else
2. Qualora la condizione sia soddisfacibile, trovare valori interi che causano l'esecuzione di questo cammino.
3. Si supponga di eliminare dal programma il ramo `else` dell'`if`. Definire per questo programma casi di test che coprano tutti i branch. Mostrare l'effetto dell'esecuzione del programma e la capacità di trovare errori mediante l'esecuzione dei programmi per i casi di test.

SOLUZIONE (1) e (2): La condizione logica è l'AND di $b < a < 2b$ (condizione per eseguire prima il then e poi l'else) e di $b == 0$ (condizione per uscire dal ciclo dopo due iterazioni). La condizione è ovviamente insoddisfacibile. Di fatto, è possibile selezionare valori per percorrere il cammino richiesto (p.es. $a=3$, $b=2$), ma poi si entra per forza in loop infinito. (3): Un caso di test deve soddisfare $a > b > 0$ per potere entrare almeno una volta nel while e nel ramo then, con la condizione che a sia multiplo di b per potere uscire dal ciclo. P.es. $a=2$, $b=1$: il programma esegue una volta il while, poi il ramo then: a quel punto $a=b=1$ e il programma esce; un altro esempio è $a=6$, $b=2$, che resta tre volte nel ciclo, ecc.

Un caso di test che copre l'altro ramo dell'`if` deve verificare $a < b$, ad esempio $a=2$, $b=3$. L'esecuzione con $a < b$ entra in loop infinito (in quanto ne' a ne' b sono modificati). Quindi il testing co il criterio di copertura delle diramazioni in questo caso è in grado di rilevare un errore significativo, che poteva non essere rilevato ad esempio con il criterio di copertura delle istruzioni (il test con $a=2$, $b=1$ ad esempio copre tutte le istruzioni, ma non rileva l'errore).

Esercizio 4

Si progetti una classe Java Contatore che permetta l'accesso concorrente ad un contatore che:

- Inizialmente vale 0 e può arrivare a 10;
- Quando il contatore arriva a 10, il metodo `reset()` consente di tornare a 0. L'operazione di reset deve essere possibile solo se il contatore vale 10. I metodi `incrementa()` e `decrementa()` consentono l'incremento e il decremento di una unità per volta. Le operazioni di incremento e decremento devono sempre mantenere il valore del contatore tra 0 e 10.

Suggerimento: si sfrutti opportunamente il modificatore `synchronized` per evitare inconsistenze nell'accesso all'oggetto condiviso.

SOLUZIONE

Ipotizziamo che quando il contatore sia arrivato a 10, l'eventuale incremento si sospenda fino a quando un decremento o un reset portino il contatore a un valore inferiore. Similmente per il reset e il decremento. La soluzione in questo caso è, ignorando le eccezioni `InterruptedException`:

```
public class Contatore {  
    private int c; //il contatore  
    public synchronized void incrementa {  
        while (c==10) wait();  
        c++;  
        notifyAll();  
    }  
    public synchronized void decrementa {  
        while (c==0) wait();  
        c--;  
        notifyAll();  
    }  
    public synchronized void reset {  
        while (c<10) wait();  
        c=0;  
        notifyAll();  
    }  
}
```

Ingegneria del Software – a.a. 2008/09

Appello del 25 Luglio 2008

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Totale

Esercizio 1 (punti 20)

Il gioco **Labirinto** è costituito da stanze, variamente collegate fra loro; il suo obiettivo è quello di trovare una via d'uscita, data una stanza iniziale in cui si trova il giocatore. Una classe astratta **Ambiente** fornisce l'astrazione comune alle stanze del labirinto e all'ambiente esterno in cui ci si deve portare per terminare il gioco. In altri termini, una stanza è un particolare ambiente *interno*, mentre il mondo esterno al labirinto è un particolare ambiente *esterno*. Mentre esistono più stanze, esiste solo un ambiente esterno.

1. (1 punto) Definire le intestazioni delle due classi concrete **Stanza** e **MondoEsterno**, che rappresentano gli elementi concreti di un labirinto.

2. Si consideri la classe **Labirinto**, che incapsula una struttura dati che rappresenta un insieme di ambienti e le loro connessioni. La classe fornisce un metodo osservatore **ambienti()** che restituisce gli ambienti del labirinto come **ArrayList<Ambiente>**, e un metodo osservatore **ambientiCollegati()** che restituisce in un **ArrayList<Ambiente>** l'insieme di ambienti collegati a un ambiente dato:

```
//@ ensures (*\result e' un elenco degli ambienti componenti il labirinto this *);
public /*@ pure @*/ ArrayList<Ambiente> ambienti ()

//@ ensures (*\result e' un elenco degli ambienti collegati direttamente a x*);
public /*@ pure @*/ abstract ArrayList<Ambiente> ambientiCollegati(Ambiente x);
```

2.1 (3 punti) Definire un invarianto per **Labirinto** che imponga le seguenti condizioni:

- a) Nessun ambiente può essere collegato a sé stesso.
- b) La relazione di collegamento è simmetrica, ovvero se l'ambiente x è collegato all'ambiente y allora y è collegato a x.
- c) ci deve essere sempre uno, e uno solo, mondo esterno.

NB: Si consiglia di utilizzare, anche nel seguito dell'esercizio, il metodo boolean contains(Object o) degli ArrayList, che restituisce true se, e solo se, o è un elemento di this. Non è necessario che l'invariante sia eseguibile.

Versione non eseguibile:

```
/*@public invariant
  (\forall Ambiente x; ambienti().contains(x);
   !ambientiCollegati(x).contains(x) // (a)
   (\forall Ambiente y; ambienti().contains(y) && x!=y;
    ambientiCollegati(x).contains(y) ==> ambientiCollegati(y).contains(x))) &&
  (\text{num\_of} MondoEsterno m; ; ambienti().contains(m)) ==1;
```

Versione eseguibile:

```
/*@public invariant
  (\forall int i; 0 <= i && i < ambienti().size();
   ambienti().get(i) != null &&
   !ambientiCollegati(ambienti().get(i)).contains(ambienti.get(i)) &&
   (\forall int j; 0 <= j && j < ambienti().size() && j != i;
    ambientiCollegati(ambienti().get(i)).contains(ambienti.get(j)) ==>
    ambientiCollegati(ambienti().get(j)).contains(ambienti.get(i))) &&
  (\text{num\_of} int h; 0 <= h && h < ambienti().size(); ambienti().get(h) instanceof Mondo Esterno)
  ==1;
```

2.2 (1 punto) Di che tipo di invariante si tratta? In quali momenti dell'esecuzione del programma deve essere vero il predicato definito dall'invariante?

Di un invariante astratto, valido all'entrata e all'uscita dai metodi pubblici (o anche da quelli privati che non siano helper)

2.3 (1 punto) Fornire la specifica JML del costruttore **Labirinto** vuoto, che crea un labirinto costituito solo dal mondo esterno

`\@ensures ambienti().size() == 1 && ambienti().get(0) instanceof MondoEsterno;`

2.4 (2 punti) Fornire la specifica JML del metodo **public void addAmbiente(Ambiente a)**, che aggiunge un ambiente al labirinto. Deve trattare il caso in cui **a** sia null, deve salvaguardare l'ipotesi che non ci sia più di un mondo esterno per labirinto e inoltre deve mantenere gli ambienti pre-esistenti nel Labirinto.

`\@requires a != null && (a instanceof mondo Esterno);
\@ensures (\forall Ambiente a1; ;
 ambienti().contains(a1) <==>
 (\old(ambienti().contains(a1)) || a == a1));`

In alternativa a requires, si possono lanciare eccezioni opportune.

2.5 Fornire la specifica JML del metodo **void addConnessione(Ambiente a1, a2)** che collega **a1** e **a2**, rispondendo alle 4 domande dettagliate che seguono. La specifica deve trattare il caso in cui **a1** e **a2** siano **null**, salvaguardando l'ipotesi che nessun ambiente è collegato a se stesso e che la relazione di collegamento sia simmetrica. Tutte le connessioni già esistenti tra ambienti restano inalterate.

- (3 punti) Fornire la specifica assumendo che sia irrilevante l'eventuale pre-esistenza di un collegamento tra **a1** e **a2**. Ciò che importa è che il collegamento sussista dopo l'esecuzione del metodo.
 - `\@requires a1!= null && a2!=null && a1 != a2 && ambienti.contains(a1) && ambienti.contains(a2);`
 - `\@ensures`
 - `(\forall Ambiente a3; ;`
 - `(\forall Ambiente a4) ; ;`
 - `ambientiCollegati(a3).contains(a4) <==>`
 - `(\old(ambientiCollegati(a3).contains(a4)) ||`
 - `a1==a3 && a2==a4 ||`
 - `a1==a4 && a2 == a3)`
- (1 punto) Che cosa cambia se invece si assume che sia a carico del client verificare a priori la non esistenza di un collegamento tra **a1** e **a2** prima di invocare il metodo. La non pre-esistenza del collegamento è pertanto una pre-condizione.
`\@requires ! ambientiCollegati(a1).contains(a2) && ...`
- (1 punto) Che cosa cambia nella specifica se invece nel caso di pre-esistenza di un collegamento tra **a1** e **a2** il metodo deve sollevare un'eccezione?

- `\@signals (GiaCollegatiException e)`
`(ambientiCollegati(a1).contains(a2) || (ambientiCollegati(a2).contains(a1));`
- (1 punto) È necessario che la specifica del metodo definisca esplicitamente la proprietà di simmetria del collegamento oppure ciò è garantito a priori dall'invariante di cui al punto 2?

La specifica fornita per i metodi deve rendere vero l'invariante astratto, e non viceversa.

2.6 (2 punti) Fornire la specifica JML del metodo **boolean raggiungibile (Ambiente a1, a2)**, che fornisce un risultato vero se, e solo se, a2 è raggiungibile da a1 mediante una connessione diretta o indiretta. Deve trattare il caso in cui a1 e a2 siano null. Per definizione, se a1 e a2 coincidono (ossia $a1==a2$) allora a2 è raggiungibile da a1 (la raggiungibilità è una relazione riflessiva).

`//@ensures a1==a2 || (\exists Ambiente a; ambientiCollegati(a1).contains(a) && raggiungibile(a,a2))`

2.7 (1 punto) Fornire la specifica JML del metodo **boolean esisteUscita(Ambiente a)**, che fornisce un risultato vero se, e solo se, da a è possibile uscire dal labirinto, raggiungendo l'ambiente esterno. Deve trattare il caso in cui a sia *null*.

`//@ensures (\exists MondoEsterno m; ambienti().contains(m); raggiungibile(a,m));`

3. (3 punti) Si supponga che il metodo **ambientiCollegati** della classe **Labirinto** restituisca, invece dell'**ArrayList**, un iteratore con il quale accedere tutti gli ambienti collegati a un certo ambiente. Definire la classe **Labirinto**, che esporta il nuovo metodo **ambientiCollegati**. Si mostri anche l'implementazione della classe **AmbientiGen** che realizza l'iteratore. Non è necessario mostrare l'implementazione dei metodi dell'iteratore, ma solo la specifica della classe **AmbientiGen**.

4. (2 punti) Fornire un diagramma UML che descrive Labirinto, Ambiente, Stanza e MondoEsterno.

Esercizio 2 (punti 3)

Si consideri il seguente metodo statico

```
public static boolean checkTr (int[][] m)
```

che presa in ingresso una matrice triangolare, restituisce true se la somma degli interi di ogni riga è inferiore alla somma degli interi della riga successiva. Le righe della matrice hanno lunghezze diverse: una matrice è triangolare se la lunghezza di ogni riga è uguale alla lunghezza della riga precedente +1. La prima riga ha lunghezza 1.

Si definisca la specifica del metodo statico nei seguenti due casi:

- Il controllo che la matrice sia triangolare è demandato al cliente del metodo; la proprietà di essere triangolare è pertanto una pre-condizione.
- Il metodo definisce una funzione totale; viene sollevata un'eccezione qualora la matrice non sia triangolare.

Ingegneria del Software – a.a. 2004/05

I Prova - 2 Maggio 2005

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi ?, Ghezzi ?, Morzenti ?, SanPietro ?

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h15m.
6. Punteggio totale a disposizione: 14/30.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 4)

Si consideri il seguente programma Java:

```
public class ClasseA {  
    protected static int num = 0;  
    protected int val;  
  
    public ClasseA(){  
        val = 0;  
    }  
  
    public void set(int a){  
        num++;  
        val = a;  
    }  
  
    public void stampa(){  
        System.out.print("num: " + num);  
        System.out.println(" val: " + val);  
    }  
  
    public static void main(String[] args) {  
        ClasseA o1, o2;  
  
        o1 = new ClasseA();  
        o2 = new ClasseA();  
  
        o1.set(3);  
        o2.set(4);  
        o1.stampa();  
        o2.stampa();  
    }  
}
```

Parte A

Cosa leggiamo sullo schermo dopo aver eseguito il programma?

```
num: 2 val: 3  
num: 2 val: 4
```

Parte B

Si supponga poi di aggiungere le due sottoclassi e le dichiarazioni delle variabili a, b e c che seguono

```
public class ClasseB extends ClasseA {...}  
public class ClasseC extends ClasseA {...}
```

```
ClasseA a;  
ClasseB b;  
ClasseC c;
```

Quali delle seguenti righe di codice darebbe un errore in compilazione? (mettere una X per ogni riga che si ritiene scorretta aggiungendo una breve spiegazione)

```
X a = new ClasseB();  
X b = new ClasseA(); // Non si può assegnare ad una variabile un oggetto di una  
// sua super classe  
X c = new ClasseC();  
X b = a; // come sopra  
X b = new ClasseB();  
X a = b;  
X c = b; // Il tipo di b non è sottotipo di quello di c, né viceversa
```

Parte C

Si supponga anche di aggiungere il metodo usaVal alla ClasseA:

```
public int usaVal(ClasseA a){  
    return (val+a.val);  
}
```

e di includere nelle classi ClasseB e ClasseC i seguenti frammenti di codice:

```
public class ClasseB extends ClasseA {  
...  
    public int usaVal(ClasseB b){  
        return (val*b.val);  
    }  
}  
  
public class ClasseC extends ClasseA {  
...  
    public int usaVal(ClasseA c){  
        return (val-c.val);  
    }  
}
```

Indicare, nei due casi della ClasseB e della classeC, se si tratta di ridefinizione del metodo usaVal o di overloading.

Si tratta di overloading nel primo caso, di ridefinizione nel secondo.

Si immagini che venga eseguito il seguente frammento di codice come parte del main della classe ClasseA.

```
ClasseA a = new ClasseA();  
ClasseB b = new ClasseB();  
ClasseC c = new ClasseC();  
  
a.set(2); b.set(4); c.set(8);  
  
System.out.println("risultato = " + a.usaVal(a));  
System.out.println("risultato = " + a.usaVal(b));  
System.out.println("risultato = " + b.usaVal(c));  
System.out.println("risultato = " + c.usaVal(b));
```

Indicare, per ogni invocazione del metodo usaVal nelle istruzioni println, quale è la classe in cui è definito il metodo usaVal eseguito, fornendo una breve giustificazione della risposta.

Vengono eseguiti i metodi definiti, rispettivamente, in ClasseA (unico metodo disponibile per a), ClasseA (idem c.s.), ClasseA (parametro attuale c fatto corrispondere al parametro formale di tipo ClasseA dell'usaVal di ClasseA, non può corrispondere al parametro formale di tipo ClasseB dell'usaVal di ClasseB), ClasseC (parametro attuale b fatto corrispondere al parametro formale di tipo ClasseA dell'usaVal di ClasseC).

Riportare il risultato dell'esecuzione delle istruzioni `println`.

```
risultato = 4
risultato = 6
risultato = 12
risultato = 4
```

Esercizio 2 (punti 4)

Si consideri la seguente specifica informale di un'astrazione procedurale.

```
//@ ensures (* restituisce true sse parola compare almeno una volta in testo*)
public static boolean sottoStringa(char[] testo, char[] parola)
```

Esempi: testo = [abbcccdedbccc] e parola = [bccc], risultato è true
Se testo = [abbcccdedcc] e parola = [bccd], risultato è false

Parte A

Si migliori la specifica aggiungendo opportune precondizioni JML sui due array affinché l'operazione sottoStringa sia sensata.

```
//@ requires ..... testo!= null &&
parola != null;
//@ ensures (* restituisce true sse parola compare almeno una volta in testo*)
public static boolean sottoStringa(char[] testo, char[] parola)
```

E' invece inutile aggiungere la precond. che parola sia più corta di testo, in quanto se ciò accadesse parola non sarebbe una sottostringa.

Parte B

Si trasformino le precondizioni di cui al punto (a) nel lancio di opportune eccezioni, specificandole in JML.

```
//@ ensures .....testo!= null && parola
!= null
//@ && (* restituisce true sse parola compare almeno una volta in testo*);
//@ signals ..... (NullPointerException
e) testo ==null || parola == null;
public static boolean sottoStringa(char[] testo, char[] parola) .....
throws NullPointerException
```

Parte C

Si riscriva la clausola ensures utilizzando JML, in modo completamente formale (senza commenti) facendo sì che la postcondizione risulti eseguibile.

```
//@ ensures .....testo!= null && parola
!= null && \result <==>
//@ ..... (\exists int i; 0<=i
&& i <=testo.length-parola.length;
//@ ..... (\forall int j;
0<=j && j<parola.length; testo[i+j] == parola[j]));
public static boolean sottoStringa(char[] testo, char[] parola) .....
```

Esercizio 3 (punti 6)

Si consideri una classe *SetterGetter*, i cui esemplari rappresentano multi-insiemi di oggetti di una classe astratta *ObjectWithKind*; questa sostanzialmente aggiunge a Object un campo costante "kind" intero positivo, che specifica il "tipo" degli oggetti (abbiamo così, per esempio, oggetti di tipo 7).

```
abstract class ObjectWithKind extends Object {  
    public final int kind;  
    //le implementazioni di questa classe garantiscono che kind  
    // sia >0; inoltre il suo valore non è modificabile  
}
```

Gli oggetti di classe *SetterGetter* sono multi-insiemi nel senso che possono contenere diversi oggetti di classe *ObjectWithKind* aventi lo stesso valore dell'attributo *kind*; tuttavia i multi-insiemi non ammettono ripetizioni dello stesso oggetto e pertanto ogni oggetto può comparire al più una sola volta in ogni multi-insieme.

La classe *SetterGetter* fornisce le seguenti operazioni:

1. costruttore di default, che inizializza a vuoto il multi-insieme;
2. operazione observer boolean *isIn(ObjectWithKind o)* che restituisce true se e solo se l'oggetto *o* è presente nel multi-insieme;
3. operazione observer int *curKind()* che dà il tipo di oggetto da estrarre dal multi-insieme, come impostato dal più recente *startKind()*; se non vi è stata alcuna invocazione di *startKind()* viene restituito il valore 0;
4. operazione observer int *howMany(int i)* che restituisce il numero di elementi di tipo *i* presenti nel multi-insieme;
5. operazione *startKind(x)*, con parametro intero positivo, che fa sì che da quel momento in avanti (fino a un'eventuale successiva *startKind()*) l'operazione *get()* prelevi un elemento il cui *kind* vale *x* eliminandolo dal multi-insieme.
6. operazione *add(o)* che inserisce nel multi-insieme un *ObjectWithKind o*;
7. operazione *get()* che preleva un elemento dal multi-insieme (vedi anche i punti precedenti); se non è possibile prelevare alcun elemento viene lanciata un'eccezione *NoObjectWithKindException*. Nel caso in cui esista più di un elemento che possa essere prelevato, non viene precisato quale debba essere scelto. Non è possibile prelevare alcun elemento prima che sia stata effettuata almeno un'invocazione del metodo *startKind()*.

Parte A

Si fornisca una specifica della classe *SetterGetter* sulla base della descrizione a parole fornita qui sopra. I metodi osservatori sono già specificati in modo informale mediante commenti JML. Si aggiungano le specifiche completamente formalizzate per tutti gli altri metodi.

```
public class SetterGetter {  
  
    // observers  
    //@ ensures (* \result è true se e solo se o è incluso in this *);  
    public /*@ pure @*/ boolean isIn(ObjectWithKind o){...}  
  
    //@ ensures (* \result restituisce il kind corrente, cioè quello definito  
    // dalla ultima startKind, oppure 0*);  
    public /*@ pure @*/ int curKind(){...}  
  
    //@ ensures (* \result è il numero degli oggetti in this con kind k *);  
    public /*@ pure @*/ int howMany(int k){...}  
  
    //@ ensures this.curKind() == 0 && (\forall int i; i>0; this.howMany(i) == 0) &&  
    //(\forall ObjectWithKind o; ; !this.isIn(o));  
    public SetterGetter(){...}  
  
    //@ requires x > 0;  
    //@ ensures this.curKind() == x && this.howMany(x) > 0;  
    //@
```

```

//@ signals (NoObjectWithKindException e) this.howMany(x) == 0;
public void startKind(int x) throws NoObjectWithKindException {...}

//@ requires o != null;
//@ ensures this.isIn(o) &&
//@(\forall ObjectWithKind x; x != o; this.isIn(x)<==> \old(this.isIn(x)));
public void add(ObjectWithKind o){...}

//@ ensures this.curKind()>0 &&\old(this.curKind())>0 &&
//@          \result.kind == this.curKind() &&
//@          !this.isIn(\result) && (\exists int i; \old(isIn(i)); \result==i)
//@          (\exists ObjectWithKind o; \old(this.isIn(o)); o == \result) &&
//@          (\forall ObjectWithKind o; o != \result; this.isIn(o) <==>
//@          \old(this.isIn(o)));
//@ signals (NoObjectWithKindException e) this.curKind()==0 || this.curKind()>0 &&
//@          \old(this.howMany(this.curKind()))==0;
public ObjectWithKind get() throws NoObjectWithKindException {...}
}

```

Parte B

Con riferimento a un'implementazione della classe SetterGetter basata sulle seguenti linee:

- ? un vector mi memorizza gli elementi presenti nel multi-insieme, disposti in un ordine qualunque, non precisato, che presumibilmente dipende dalla successione degli inserimenti e delle estrazioni;
- ? una variabile intera kind memorizza il valore corrente del tipo di elemento di cui si possono prelevare elementi; se non è definito alcun tipo di elemento da prelevare (non è stato mai invocato il metodo startKind) allora kind = 0;
- ? una variabile intera current individua nel vettore mi la posizione dell'ObjectWithKind prossimo da prelevare (il cui valore dell'attributo kind coincide con this.kind); se non è definito alcun tipo di elemento da prelevare (this.kind vale 0) essa può avere qualsiasi valore, mentre se il tipo di elemento da prelevare è definito (quindi this.kind!=0) ma non vi sono nel multi-insieme elementi di quel tipo (per esempio, perché sono stati tutti prelevati) current ha un valore <0 oppure >=mi.size().

Si definisca in JML il rep invariant e si implementi il metodo get.

```
public class SetterGetter {  
    private Vector mi;  
    private int kind;  
    private int current;  
  
    /*@ private invariant mi != null && kind >= 0 &&  
    (\forall int i; 0<=i && i<mi.size(); mi.get(i) instanceof ObjectWithKind &&  
    this.kind>0 ==>  
        (0<=current && current <mi.size() ==>((ObjectWithKind) mi.get(current)).kind  
        == this.kind) &&  
        !(0<=current && current <mi.size() ==>  
        !(\exists int i; 0<=i && i<mi.size(); ((ObjectWithKind) mi.get(i)).kind ==  
        this.kind)  
    */
```

L'ulteriore vincolo che tutti gli oggetti siano distinti (ad.es. $(\forall \text{int } i; 0 \leq i & i < \text{mi.size}(); (\forall \text{int } j; i < j & j < \text{mi.size}(); \text{mi.get}(i) \neq \text{mi.get}(j)))$) non è previsto in modo esplicito dalla descrizione a parole, ma è comunque ragionevole.

```
public ObjectWithKind get() throws NoObjectWithKindException {  
    if (kind == 0 || current<0 || current >= mi.size())  
        throw new NoObjectWithKindException();  
    ObjectWithKind o = (ObjectWithKind) mi.get(current);  
    mi.set(current, mi.lastElement());  
    mi.remove(mi.size()-1);  
    if (mi.size() == 0)  
        current = -1;  
    else  
        for (current = 0; current < mi.size() &&  
            ((ObjectWithKind) mi.get(current)).kind != kind; current++);  
    return o;
```

}



Politecnico di Milano

Anno accademico 2010-2011

Ingegneria del Software – Appello del 2 settembre 2011

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione.
È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 1

Un'OrganizzazioneBenStrutturata (OBS) è costituita da un insieme di Dipendenti (D) e da un ResponsabileOperativo (RO), che a sua volta è un dipendente. I dipendenti hanno un salario: quello del RO non può superare il più alto degli altri dipendenti di più del 30%. Il salario dei dipendenti può oscillare del + o - 20% rispetto a un salario "standard".

Le classi OrganizzazioneBenStrutturata e Dipendente sono specificate come segue, ipotizzando che Dipendente deleghi a una classe Persona la gestione dei dati anagrafici: serve una Persona per creare un nuovo Dipendente. Inoltre, per semplicità, si consideri che il RO sia definito solo al momento dell'inizializzazione e che non si possano eliminare dipendenti dalla OBS.

```
public /*@ pure @*/ class Dipendente {  
    // @ensures (*costruisce un nuovo dipendente, con salario salarioIniziale*)  
    public Dipendente(Persona p, int salarioIniziale);  
  
    // @ensures (*\result e' la Persona corrispondente a this *);  
    public Persona persona();  
  
    // @ensures (*\result e' il salario di this *);  
    public int salario();  
}  
  
public class OrganizzazioneBenStrutturata {  
    // definisce una nuova OBS con resp come RO e salario standard  
    // pari a salarioStandard  
    public OrganizzazioneBenStrutturata(ResponsabileOperativo resp,  
                                         int salarioStandard)  
  
    // @ensures (* \result e' true sse d e' parte di this, come RO o dipendente*)  
    public /*@ pure @*/ boolean isDipendente (Dipendente d)  
  
    // @ensures (* \result e' il responsabile di this*)  
    public /*@ pure @*/ ResponsabileOperativo responsabile()  
  
    // @ensures (*\result e' il salario standard per l'organizzazione*)  
    public /*@ pure @*/ int salarioStandard()  
  
    // aggiunge un dipendente con il salario assegnato  
    public void addDipendente(Dipendente d, int salario)  
  
    // aggiorna il salario del dipendente  
    public void cambiaSalario(Dipendente d, int nuovoSalario)  
}
```

Si chiede di:

1. Definire l'interfaccia della classe ResponsabileOperativo in modo opportuno. **Soluzione:**

```
public /*@ pure @*/ class ResponsabileOperativo extends Dipendente {  
    // @ensures (*costruisce un nuovo ResponsabileOperativo, con salario salarioIniziale*)  
    public ResponsabileOperativo(Persona p, int salarioIniziale);  
}
```

2. Esprimere in JML i vincoli di cui sopra per la classe OrganizzazioneBenStrutturata come invarianti astratti.

Soluzione: Definiamo per brevità un predicato:

```
//@ensures \result <==> d.salario() >= salStand*0.8 && d.salario() <= salStand*1.2
//@ && r.salario() >= d.salario()*1.3;
public boolean salarioOk(int salStand, ResponsabileOperativo r, salario s)
```

L'invariante pubblico, che considera oltre ai vincoli sui salari anche il vincolo (non esplicito, ma ovvio dal contesto) che non vi siano due dipendenti che sono la stessa persona, è:

```
//@public invariant (\forall Dipendente d; isDipendente(d) && d!=responsabile());
//@ salarioOk(salarioStandard(), responsabile(), d)
&& !(\exists Dipendente d1; isDipendente(d1); d!= d1 && d.persona() == d1.persona()));
```

3. Specificare in JML i metodi OrganizzazioneBenStrutturata, addDipendente e cambiaSalario. Le specifiche devono essere definite come operazioni parziali, aggiungendo eventuali vincoli mediante opportuna precondizione, allo scopo di verificare l'invariante astratto.

Soluzione:

```
//@requires resp != null && salarioStandard >0;
//@ensures this.responsabile() == resp && this.salarioStandard() == salarioStandard &&
//@ (\forall Dipendente d; isDipendente(d); d==resp); //non ci sono altri dipendenti
public OrganizzazioneBenStrutturata(ResponsabileOperativo resp,
                                      int salarioStandard)

//@requires d!=null && salario>0 && salarioOk(salarioStandard(), responsabile(), d)
//@ && !isDipendente(d)
//@ && !(\exists Dipendente d1; isDipendente(d1); d!= d1&&d.persona()==d1.persona());
//@ensures isDipendente(d) && (\forall Dipendente d1; d1!= d;
//@ \old(isDipendente(d1)) <=> isDipendente(d1);
public void addDipendente(Dipendente d, int salario)
```

Per la cambiaSalario occorre prestare attenzione al fatto che Dipendente è immutabile:

```
//@requires d!=null && nuovoSalario>0 && salarioOk(salarioStandard(), responsabile(), d)
//@ensures !isDipendente(d)
//@ && !(\exists Dipendente d1; isDipendente(d1); d1.persona()==d.persona() &&
//@ d1.salario == nuovoSalario);
public void cambiaSalario(Dipendente d, int nuovoSalario)
```

4. Specificare in JML cambiaSalario come operazione totale, in modo che l'invariante sia verificato.

Soluzione: Le requires vanno trasformate in eccezioni...

5. Si consideri una classe definita come segue:

```
public class DipendenteSpeciale {
    //@ensures (*costruisce un DipendenteSpeciale, con salario salarioIniziale*)
    public DipendenteSpeciale(Persona p, int salarioIniziale);
    ...qui gli stessi metodi salario() e persona() di Dipendente...
    //@ensures salario() == salario;
    public void nuovoSalario(int salario)
}
```

E' corretto definire la classe DipendenteSpeciale come erede di Dipendente? Cosa succederebbe all'invariante astratto di OBS se nella definizione di OBS si usasse DipendenteSpeciale invece di Dipendente? Giustificare le risposte.

Soluzione: No, la classe DipendenteSpeciale e' mutabile, e le eventuali mutazioni possono essere osservate anche con il metodo salario() già definito in Dipendente: e' quindi violata la regola delle proprietà. Se nonostante questo si procedesse con la definizione di DipendenteSpeciale come erede di Dipendente (cosa comunque possibile in Java), ad esempio l'invariante pubblico della classe OrganizzazioneBenStrutturata potrebbe essere violato, aggiungendo un DipendenteSpeciale e poi modificandone il salario con nuovoSalario.

6. Si consideri il seguente rep per la classe OBS.

```
private RO ro;
private ArrayList<Dipendente> dipendenti;
private int salarioStandard;
```

Scrivere un invariante di rappresentazione adeguato e, in base alla conoscenza dell'invariante e del rep, implementare il metodo cambiaSalario.

Soluzione: Oltre a condizioni consuete sul rep, occorre stabilire se ro è in dipendenti oppure no. È più naturale che non vi sia, essendo già memorizzato a parte nel rep. Si noti che le condizioni su salarioOk sono parte dell'inv. astratto e non sono da ripetere nell'inv. di rappresentazione; così pure le relazioni fra il rep e i metodi osservatori (che costituiscono invece la funzione di astrazione).

```
//@ private invariant ro != null &&
//@ (\forall int i; 0 \leq i && i < dipendenti().size(); dipendenti != null) &&
//@ !dipendenti().contains(ro);
```

L'implementazione (per semplicità, nel caso con le precondizioni) deve considerare che Dipendente è classe immutabile, e che il responsabile Operativo è un dipendente come gli altri (ma non è memorizzato nell'ArrayList dipendenti):

```
public void cambiaSalario(Dipendente d, int nuovoSalario) {
    if (ro.equals(d)) {
        RO r = new ResponsabileOperativo(d.persona(), nuovoSalario);
        ro = r;
    }
    else {
        Dipendente d1 = new Dipendente(d.persona(), nuovoSalario);
        dipendenti.remove(d);
        dipendenti.add(d1);
    }
}
```

Esercizio 2

Definire, motivando la risposta, cosa stampa il seguente frammento di codice Java:

```
public class ClassA {
    private int a;

    public int value() {return a;}

    public void stampa(ClassA p) {
        System.out.println(this.value()+p.value());
    }
}

public class ClassB extends ClassA {
    public void stampa(ClassB p) {
        System.out.println(this.value()+p.value()+1);
    }

    public void stampa(ClassA p) {
        System.out.println(this.value()+p.value()+2);
    }
}

public class ClassC extends ClassB {
    public void stampa(ClassC p) {
        System.out.println(this.value()+p.value()+3);
    }

    public void stampa(ClassA p) {
        System.out.println(this.value()+p.value()+4);
    }
}

public class Test {
    public static void main(String[] args) {
        ClassA v1 = new ClassA();
        ClassA v2 = new ClassB();
        ClassA v3 = new ClassC();
        ClassB v4 = new ClassB();
        ClassC v5 = new ClassC();

        v1.stampa(v1);
        v1.stampa(v2);
        v3.stampa(v2);
        v4.stampa(v3);
        v5.stampa(v1);
        v5.stampa(v4);
        v4.stampa(v5);
    }
}
```

Soluzione: 0 0 4 2 4 1 1

Esercizio 3

Solitamente, quando si vogliono prelevare dei soldi da uno sportello bancomat, si compiono le seguenti operazioni:

- Il cliente inserisce la tessera nel lettore
- Il cliente digita il codice segreto
- Il sistema controlla che il codice sia corretto (e supponiamo sia corretto)
- Il sistema propone le diverse quantità prelevabili (6 diverse alternative)
- Il cliente sceglie l'ammontare desiderato
- Se richiesto, il sistema stampa la ricevuta
- Il sistema restituisce la tessera, dà i soldi e toglie la somma dal saldo del conto del cliente
- Il sistema termina l'operazione e si prepara per un'altra operazione

Identificare gli attori/oggetti coinvolti, utilizzando una granularità opportuna, attraverso un *diagramma delle classi* e rappresentare lo scenario con un *diagramma di sequenza*.

Ingegneria del Software – a.a. 2005/06

Appello del 8 settembre 2006

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi ?, Ghezzi ?, Morzenti ?, SanPietro ?

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Totale

Esercizio 1 (punti 6)

L'ADT immutabile Carta fornisce un'astrazione per una carta da briscola: asso ("1"), due ("2"), tre ("3"), quattro ("4"), cinque ("5"), sei ("6"), sette ("7"), fante ("F"), cavallo ("C") e re ("R") di spade ("S"), coppe ("C"), bastoni ("B") e denari ("D") per un totale di quaranta possibili carte distinte. L'ADT *MazzoCarte* fornisce un'astrazione per un mazzo di carte da briscola, ossia di quaranta carte. L'ADT fornisce i metodi puri *nelMazzo* e *carteNelMazzo*:

```
//@ requires true ;
//@ ensures (*restituisce true se c è parte del mazzo*) ;
public /*@pure@*/ boolean nelMazzo(Carta c)

//@ requires true ;
//@ ensures (*restituisce il numero di carte rimaste nel mazzo*) ;
public /*@pure@*/ int carteNelMazzo()
```

a- Si scriva la specifica del metodo pesca tenendo presente che:

- ✓ Si assume che il mazzo debba contenere almeno una carta
- ✓ Il numero di carte nel mazzo viene decrementato di uno
- ✓ La carta pescata non è più presente nel mazzo
- ✓ La carta pescata è l'unica carta eliminata dal mazzo

//@ requires

//@ ensures

public Carta pesca()

Soluzione:

requires carteNelMazzo() >=1
ensures \old(nelMazzo(result)) &&
 !nelMazzo(result) &&
 carteNelMazzo() == \old(carteNelMazzo()) -1 &&
 (\forall Carta c; c != result; nelMazzo(c) <==> \old(nelMazzo(c)));

b- Si trasformi la specifica del metodo pesca eliminando l'assuzione che il mazzo non sia vuoto e includendo il lancio di un'eccezione *ExceptionMazzoVuoto*. Si mostri solo la parte di specifica modificata rispetto a quella definita in (a).

//@ requires

//@ ensures

```
public Carta pesca() throws MazzoVuotoException
```

Soluzione:

requires true;

Aggiungere alla ensures: `\old{carteNelMazzo()} >= 1`

Aggiungere la clausola:

signals(MazzoVuotoException e) `\old{carteNelMazzo()} == 0` ;

c- Sapendo che l'ADT *Carta* fornisce i seguenti metodi puri per conoscere il valore e il seme di una carta:

```
//@ requires true ;
//@ ensures (*restituisce il valore di this*) ;
public /*@pure@*/ char valore()
```

```
//@ requires true ;
//@ ensures (*restituisce il seme di this*)
public /*@pure@*/ char seme()
```

e che il rep della classe *MazzoCarte* è :

```
private Carta[] m;
```

si scriva l'invariante privato per la classe *MazzoCarte* per imporre che un mazzo non può avere carte ripetute.

Soluzione:

```
private invariant
(forall int i; 0<=i&& i<m.length()-1;
 (forall int j; i<j&& j<m.length(); i.valore()!= j.valore() || i.seme()!= j.seme()));
```

Esercizio 2 (punti 4)

Supponendo che sia definita la classe :

```
public class InsiemeInteri
    private int[] insieme;
```

si scriva in Java un opportuno iteratore che restituisca in sequenza i numeri primi contenuti nell'array, definendo sia il metodo *primi()* di *InsiemeNumeri* che restituisce l'iteratore, sia la classe *PrimiGen* che definisce l'iteratore (ossia il generatore). L'iteratore deve scandire gli elementi dell'array e restituire solo i numeri primi. Ad esempio, se l'array contenesse i numeri 5, 7, 23, 45, 13, 48, 17, l'iteratore dovrebbe restituire i numeri 5, 7, 23, 13, e 17.

Soluzione :

```
public Iterator<Integer> primi() {
    return new PrimiGen(this) ;
}

private static class PrimiGen implements Iterator<Integer> {
    private InsiemeInteri i ;
    private int n ;
    PrimiGen(InsiemeInteri ins){
        i=ins ;
        n=0 ;
        while(n<i.insieme.length && !isPrime(i.insieme[n])) n++ ;
            /* isPrime metodo statico implementato in modo ovvio */
    }

    public boolean hasNext() { return n<i.insieme.length ;}

    public Integer next() throws NoSuchElementException {
        Integer res ;
        if (this.hasNext()) res = i.insieme[n] ;
        else throw new NoSuchElementException("PrimiGen") ;
        while(n<i.insieme.length && !isPrime(i.insieme[n])) n++ ;
        return res ;
    }
}
```

Esercizio 3 (punti 5)

Il metodo statico *controllaSomma()* riceve in ingresso un array *nums* (assunto non nullo) e due interi *som* e *pos* e restituisce un valore booleano. Il metodo restituisce true se la somma degli elementi dell'array da *pos* (compreso) alla fine dell'array è pari a *som*; restituisce false altrimenti. Inoltre, il metodo solleva l'eccezione *ExceptionFuoriRange* se *pos* supera i limiti dell'array.

Si fornisca la specifica JML del metodo. Si rammenta che la specifica deve considerare il caso in cui l'array sia nullo. Si ricorda che in JML esiste un costrutto (*\sum variable; condizione; espressione*) che fornisce la somma di tutti i valori dell'espressione ottenuti per valori della variabile che soddisfano la condizione).

//@ requires

//@ ensures

/@ signals

```
public static boolean controllaSomma(int [] nums, int pos, int som) throws ExceptionFuori-
RangeSol.
//@ requires num != null ;
//@ ensures (pos < nums.length) && \result == (som == (\sum int i ; pos <=i && i<nums.length);
nums[i])
//@ signals (ExceptionFuoriRange e) (pos >= nums.length) ;
public static boolean controllaSomma(int [] nums, int pos, int som) throws ExceptionFuoriRange
```

Esercizio 4 (punti 4)

Si consideri una classe Java *MiaClasse*:

```
public class MiaClasse {
    protected static int num = 0;
    protected int val;
    public MiaClasse(int a){
        val = a;
        num += a;
    }
}
```

a- Supponendo di creare due oggetti o1 e o2 in sequenza:

```
o1 = new MiaClasse(5);  
o2 = new MiaClasse(3);
```

quali valori assumono i campi val e num dei due oggetti?

Sol.

o1 = new MiaClasse(5);	val = 5 e num = 5
o2 = new MiaClasse(3);	val = 3 e num = 8

b- Aggiungendo due sottoclassi MiaSottoclasse1 e MiaSottoclasse2:

```
public class MiaSottoclasse1 extends MiaClasse {...}  
public class MiaSottoclasse2 extends MiaClasse {...}
```

```
MiaClasse a;  
MiaSottoclasse1 b;  
MiaSottoclasse2 c;
```

Quali delle seguenti righe di codice darebbe un errore in compilazione? (mettere una X per ogni riga che si ritiene scorretta aggiungendo una breve spiegazione)

```
a = new MiaClasse();  
b = new MiaClasse();  
c = new MiaSottoclasse1();  
b = a;  
a = b;  
b = new MiaSottoclasse2();
```

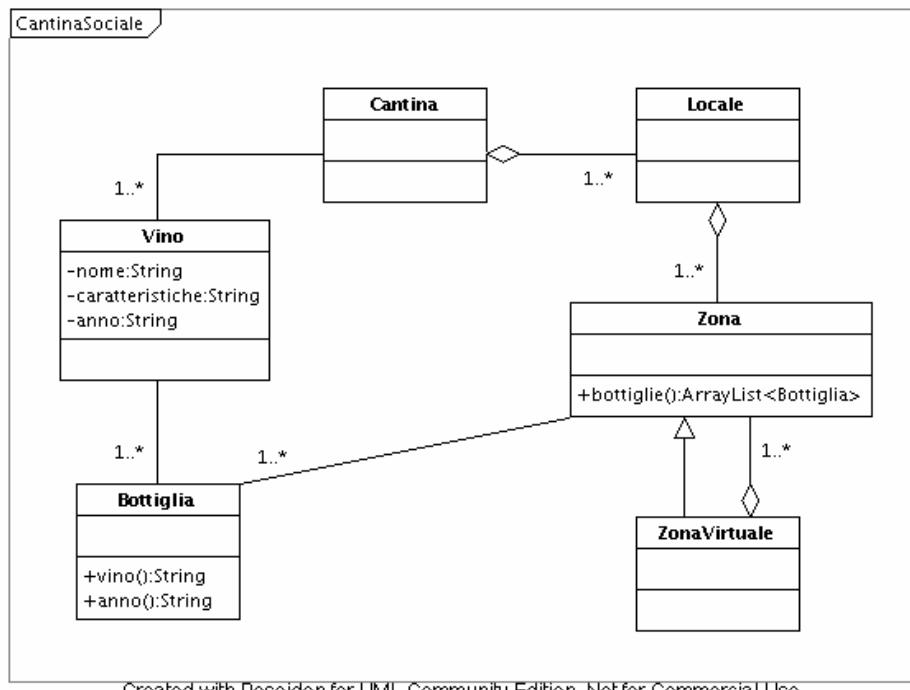
Sol.

```
a = new MiaClasse();  
b = new MiaClasse(); X assegnamento di un oggetto a una variabile di una sottoclasse  
della sua classe  
c = new MiaSottoclasse1(); X assegnamento di un oggetto a una variabile di una classe non  
antenato della sua classe  
b = a; X assegnamento di una variabile a una variabile di una sottoclasse della sua classe  
a = b;  
b = new MiaSottoclasse2(); X assegnamento di un oggetto a una variabile di una classe non  
antenato della sua classe
```

Esercizio 5 (punti 6)

Una cantina sociale offre diversi vini, identificati dal nome, dalle caratteristiche principali e dall'anno di produzione. La cantina sociale possiede diversi locali divisi in zone: tutte le bottiglie di una data annata di un certo vino sono tenute nella stessa zona, ma se il numero di bottiglie eccede la capacità di una zona, è possibile comporre le zone. In pratica, si definisce una nuova zona (virtuale) componendo zone più piccole.

- a- Si progetti il diagramma delle classi UML, evidenziando attributi e metodi principali.



Created with Poseidon for UML Community Edition. Not for Commercial Use.

- b- Si scriva in JML (o notazione similare) il vincolo che impone che tutte le bottiglie di una stessa annata di un certo vino devono stare nella stessa zona.

```

(!forall Bottiglia b1; this.bottiglie();
    (!forall Bottiglia b2; this.bottiglie() && b2 != b1;
        b1.vino() == b2.vino() && b1.anno() == b2.anno() ==>
        b1.zona()==b2.zona() ))
    
```

Ingegneria del Software – a.a. 2006/07

Appello del 14 settembre 2007

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Totale

Esercizio 1 (punti 6)

In presenza della seguente dichiarazione

```
interface Functions { //funzioni a valore float di una variabile float
    float value (float x); // calcola il valore di this in corrispondenza al parametro x
}
```

il metodo

```
public static float [] findFunctionZeros(functions f, float a, float b, char p) {... }
```

accetta come ingresso un oggetto *Functions* (assunto non nullo), due parametri *a* e *b* che individuano un intervallo non vuoto e un carattere *p* che può valere ‘l’ (per indicare “bassa precisione”) oppure ‘h’ (“alta precisione”) e restituisce un array di float che contiene valori che rappresentano zeri della funzione *f* compresi nell’intervallo tra *a* e *b*: se *p*=‘l’ allora il valore della funzione, per ognuno dei valori memorizzati nell’array restituito, è in valore assoluto <0.001; se *p*=‘h’ è <0.000001.

Si scriva una specifica, in JML del metodo, indicando la pre- e la post-condizione.

Soluzione

```
//@requires f!= null && a<=b && (p=='l' || p=='h')

//@ensures
//@ \result !=null &&
//@ (\forall int i; 0<=i && i<\result.length;
//@ \result[i]>=a && \result[i]<=b &&
//@ f.value(\result[i])> (p=='l'?-0.001:-0.000001) && f.value(\result[i])< (p=='l'?0.001:0.000001) )
```

Esercizio 2 (punti 12)

La classe **Grafo** rappresenta grafi i cui archi sono orientati e pesati e la sua interfaccia comprende il metodo *observer* **ArrayList<Nodo> nodi()**, che restituisce tutti i nodi del grafo. L'interfaccia della classe **Nodo** comprende i seguenti metodi *observer*:

- **Arco arco(Nodo dest)** restituisce l'arco che collega this a dest, se esiste ; null altrimenti.
- **ArrayList<Arco> archi(int dir)** restituisce gli archi che entrano (dir = 0) o escono (dir =1) dal nodo.
- **boolean raggiungibile(Nodo dest)** restituisce true se dest è raggiungibile da this e false altrimenti.

L'interfaccia della classe **Arco** comprende i seguenti metodi *observer*:

- **Nodo sorgente()** restituisce il nodo sorgente dell'arco considerato
- **Nodo destinatario()** restituisce il nodo destinatario dell'arco considerato
- **int peso()** restituisce il peso dell'arco.

- a. Si scrivano la pre- e post-condizione del metodo **void aggiungiArco(Nodo dest, int peso)** della classe **Nodo**, sapendo che _ l'oggetto this è il nodo sorgente dell'arco, una coppia di nodi può essere connessa da un solo arco e il peso del nuovo arco deve essere positivo. Si ricorda anche che dopo l'aggiunta, il nuovo arco deve comparire nell'insieme di archi che entrano/escono dai nodi passati come parametri.

```
// dest diverso da null
//@ requires dest != null;
// peso nuovo arco positivo
//@ requires peso > 0;
// non esiste un arco da this a dest
//@ requires !(exists int i; 0 <= i <= archi(1).size(); archi(1).get(i)==dest)

// esiste un arco che esce da this e entra in dest
//@ ensures !(exists int i; 0 <= i <= archi(1).size(); archi(1).get(i).destinatario()==dest)
// esiste un arco che entra in dest da this
//@ ensures !(exists int i; 0 <= i <= dest.archi(0).size(); dest.archi(0).get(i).sorgente()==this)
```

void aggiungiArco(Nodo dest, int peso)

- b. Si scriva la post-condizione del metodo **void rimuoviArco(Arco a)**, supponendo che appartenga alla classe **Grafo**. Si noti che un nodo che rimane isolato dopo la cancellazione dell'arco, deve pure essere cancellato.

```
/l'arco non deve piu' esistere
//@ ensures !(forall int i; 0 <= i <= nodi().size();
  ((forall int j; 0 <= j < nodi().get(i).archi(1).size();
    nodi().get(i).archi(1).get(j) == a) &&
   (forall int j; 0 <= j < nodi().get(i).archi(0).size();
    nodi().get(i).archi(0).get(j) == a));
// non ci devono essere nodi isolati
//@ ensures !(forall int i; 0 <= i <= nodi().size();
  nodi().get(i).archi(1).size() > 0 || nodi().get(i).archi(0).size() > 0);
```

- c. Si scriva in JML l'invariante pubblico della classe **Grafo** sapendo che un grafo non può mai avere nodi isolati e archi senza sorgente o destinatario.

```
//@ public invariant
// non ci devono essere nodi isolati
(forall int i; 0 <= i < nodi().size();
  nodi().get(i).archi(1).size() > 0 || nodi().get(i).archi(0).size() > 0) &&
// non ci devono essere archi senza sorgente o destinatario
(forall int i; 0 <= i < nodi().size();
  ((forall int j; 0 <= j < nodi().get(i).archi(1).size();
    nodi().get(i).archi(1).get(j).destinatario() != null) &&
   (forall int j; 0 <= j < nodi().get(i).archi(0).size();
    nodi().get(i).archi(0).get(j) != null));
```

- d. Si scrivano la pre- e post-condizione del metodo int distanzaMinima(Nodo dest) della classe **Nodo** che calcola la distanza minima tra l'oggetto this e il nodo dest, se dest è raggiungibile da this. Si consiglia di adottare una definizione ricorsiva del problema per semplificare la scrittura della post-condizione.

```
// il nodo deve essere raggiungibile
//@ requires raggiungibile(dest);

// se i due nodi sono connessi direttamente, la distanza minima e' il
// peso dell'arco che connette i due nodi, altrimenti dobbiamo
// identificare ogni nodo p collegato a this da un arco e la distanza
// minima e' il minimo tra le somme della distanza tra this e p e
// p.distanzaMinima(dest).
// Per semplicità ipotizzo che 1000 sia un valore fuori scala.

//@ ensures (\exists int i; 0 <= i < archi(1).size(); archi(1).get(i).destinatario() == dest)?
\result == arco(dest).peso():
\result == (\min int i; 0 <= i < archi(1).size(); archi(1).get(i).destinatario().raggiungibile()?
            arco(archi(1).get(i).destinatario()).peso() +
            archi(1).get(i).destinatario().distanzaMinima(dest):1000);

int distanzaMinima(Nodo dest)
```

Esercizio 3 (punti 3)

Si implementi un iteratore per la classe **Grafo**, dell'esercizio precedente, in modo che dato un nodo n, l'iteratore restituisca tutti i nodi raggiungibili da n dal più vicino al più lontano. Si ipotizzi che esista un metodo int distanza(Nodo dest) della classe Nodo che restituisce la distanza tra this e un nodo dato come parametro.

```
public class Grafo {
    ...
    public Iterator<Nodo> nodiRaggiungibili(Nodo n) {
        return new GenNodi(n);
    }

    //classe interna
    private static class GenNodi implements Iterator<Nodo> {
        private int pos; // posizione corrente
        private ArrayList<Nodo> nodiOrdinati; // struttura per i nodi ordinati

        GenNodi(Nodo n) {
            pos=0;
            // nodiOrdinati riempita in funzione di n.archi() e dei loro destinatari
        }

        public boolean hasNext() {return pos < nodiOrdinati().size()}

        public int next() throws NoSuchElementException {
            if (nodiOrdinati.size() > 0 && pos < nodiOrdinati.size())
                return nodiOrdinati.get(pos++);
            throw new NoSuchElementException();
        }
    }
}
```

Esercizio 4 (punti 4)

Si definisca un diagramma delle classi UML per rappresentare il seguente problema descrivendo con accuratezza le relazioni fra le classi e le loro cardinalità, ove necessario.

Una banca si vuole dotare di un nuovo sistema informativo per unificare la gestione dei prodotti bancari. Nello specifico, la banca tratta conti correnti tradizionali, conti correnti online, mutui, e portafogli titoli. Mentre i conti correnti sono sottoscrivibili da tutti, i mutui e i portafogli titoli sono riservati ai clienti intestatari di un conto corrente con una giacenza media di 2000 euro negli ultimi tre mesi.

Ciascun cliente della banca può scegliere di aprire alternativamente un conto corrente online, uno tradizionale, ma non entrambi. Il conto corrente online prevede esclusivamente l'accesso al conto tramite il sito Internet ed è necessariamente associato ad una carta Bancomat che è invece opzionale nel caso del conto tradizionale. Entrambi i conti possono poi essere associati a carta di credito e libretto assegni. Anche Carta di credito e libretto assegni sono riservati ai clienti titolari di un conto corrente con una giacenza media di 2000 euro negli ultimi tre mesi.

Nel caso del conto online, il versamento di denaro può avvenire solo tramite assegno o bonifico, mentre nel caso del conto tradizionale è anche possibile versare contanti. I conti correnti sono associati a delle spese fisse, e a delle spese per ciascuna operazione sul conto (prelievo, versamento, ecc.). Tuttavia sono previsti dei bonus per i clienti che effettuano un numero elevato di operazioni mensili.

Appello del 18 Luglio 2011



Politecnico di Milano
Anno accademico 2010-2011

Ingegneria del Software

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 1

Si consideri la seguente specifica di una classe Lista. Una Lista e' una sequenza di elementi che consente ricerca, inserimento e eliminazione.

```
public class Lista<T extends Comparable<T>> {
    /*OVERVIEW: Collezione mutabile di oggetti, di tipo T, organizzati in una sequenza.
     * Oggetto tipico di dimensione n: x1 x2 ... xn, in cui x1 e' il primo elemento.
     * Rimozione e inserimento di elementi possono avvenire in una posizione qualunque.
     */

    // @ensures (*inizializza this alla Lista vuota*);
    public Lista()

    //osservatori puri:

    // @ensures (* \result == numero di elementi di this *)
    public /*@ pure @*/ int size()

    // @ensures (* \result <==> c'e' elemento equals(x) in this *)
    public /*@ pure @*/ boolean contains(T x)

    // @requires i>=0 && i < size();
    // @ensures (* \result == i+1-esimo elemento di this, cosi' come sarebbe restituito
    // @ dall'iteratore elementi(), ossia get(0) e' il primo elemento, ecc. *)
    public /*@ pure @*/ T get(int i )

    //mutators:

    // @requires size()>0;
    // @ ensures (* elimina e restituisce il massimo elemento
    // @ (secondo compareTo) in this *);
    public T extractMax()

    // @ensures (* aggiunge x in una qualunque posizione della lista *);
    public void insert(T x)

    // @requires size()>0;
    // @ ensures (* elimina da this una comparsa di un elemento equals(x) *)
    public void remove(T x)
}

// @requires (*this non e' modificato mentre l'iteratore e' attivo*);
// @ensures (* \result e' un iteratore che itera, una sola volta, su ciascun elemento
// @ L'ordine degli elementi puo' cambiare solo se cambia la lista*);
public Iterator<T> elementi()
```

dove si rammenta la definizione dell'interfaccia Comparable:

```
public interface Comparable<T> {
    // @ensures (* if this<arg then \result == -1
    // @ else if this == arg then \result == 0,
```

```

//@      else \result == 1 *);
public int compareTo(T arg);
}

```

Si noti che l'iteratore elementi(), se richiamato piu' volte sulla stessa Lista non modificata, restituisce gli elementi sempre nello stesso ordine (ossia, e' consistente).

Domanda A

Specificare in JML le postcondizioni dei metodi insert, extractMax, remove.

Soluzione: Uno dei vari modi per scrivere le postcondizioni e' il seguente.

insert:

```

/*@ensures    size() == \old(size())+1 &&
@          (\forall T y; \old(contains(y));
@            (\num_of T z; contains(z); y.equals(z)) ==
@              (\num_of T w; \old(contains(w); y.equals(w))
@                + (x.equals(y) ? 1 : 0)
@ */

```

remove:

```

/*@ensures    size() == \old(size()) - (\old(contains(x)) ? 1 : 0) &&
@          (\forall T y; \old(contains(y));
@            (\num_of T z; contains(z); y.equals(z)) ==
@              (\num_of T w; \old(contains(w); y.equals(w))
@                - (x.equals(y) ? 1 : 0)
@ */

```

extractMax:

```

/*@ensures    size() == \old(size())-1 && \old(contains(\result)) &&
@          (\forall T y; \old(contains(y));
@            \result.compareTo(y)>=0 &&
@              (\num_of T z; contains(z); y.equals(z)) ==
@                (\num_of T w; \old(contains(w); y.equals(w))
@                  + (\result.equals(y) ? 1 : 0)
@ */

```

Domanda B

Si consideri una classe ListaOrdinata, definita come Lista, ma con la proprietà che gli elementi sono disposti nella lista in ordine crescente. La classe ha gli stessi metodi di Lista con la stessa specifica, salvo insert, che è specificato come segue:

```
//@ensures (* aggiunge x in this in una posizione fra un elemento <=x  
//e un elemento >=x (usando compareTo per il confronto*);  
public void insert(T x)
```

e salvo il metodo iteratore elementi(), che restituisce un iteratore che itera sugli elementi di this in ordine *crescente*. La get(i) deve sempre rispettare lo stesso ordine di elementi().

1. E' possibile definire ListaOrdinata come erede di Lista o viceversa rispettando il principio di sostituzione? Se la risposta alla precedente domanda e' si, definire in JML la specifica della classe erede.

2. Definire un invariante astratto per la classe che specifica l'ordinamento della lista.

Soluzione:

Ovviamente l'unica possibilità è che ListaOrdinata erediti da Lista, in quanto le postcondizioni di ListaOrdinata.insert e di ListaOrdinata.elementi() sono più forti delle corrispondenti di Lista. L'invariante e la specifica di ListaOrdinata sono le seguenti:

```
public class ListaOrdinata extends Lista {  
    //@public invariant (\forall int i; 0\le && i<size()-1; get(i).compareTo(get(i+1))<=0);  
  
    //@also  
    //@ensures (* \result è un iteratore che itera, una sola volta, su ciascun elemento di this in  
    public Iterator<T> elementi()  
  
    //@also  
    //@ensures (\exists int i;  
    //            0\le i && i < size();  
    //            get(i).equals(x) &&  
    //            (i+1<size() ==> x.compareTo(get(i+1))>=0) &&  
    //            (i-1\ge 0 ==> x.compareTo(get(i-1))<=0)  
    //);  
    public void insert(T x)  
}
```

NB: se insert è stata definita correttamente per Lista, l'inserimento dell'elemento è già garantito, ed è quindi sufficiente aggiungere la posizione corretta.

Domanda C

Si consideri una classe ListaStrana, definita come Lista, ma che memorizza gli elementi in un ordine “strano”. Vale a dire, gli elementi della lista sono disposti in un ordine che puo’ essere inizialmente crescente e poi decrescente (le due porzioni di lista crescente e decrescente possono essere nulle). La classe ListaStrana fornisce una operazione di inserimento analoga a quella definita in precedenza per ListaOrdinata, che inserisce un elemento nella porzione crescente di valori. Fornisce anche una operazione *insert1* che inserisce l’elemento nella parte decrescente di valori.

- 1. Definire un invariante astratto per la classe ListaStrana.**
- 2. E’ possibile definire ListaStrana come erede di ListaOrdinata o viceversa rispettando il principio di sostituzione?**
- 3. Definire le specifiche JML per insert e insert1.**

Soluzione: 1)

```
public invariant
size()>0 ==>
(\exists int j; 0<=j && j<size();
 \forall int i; 0<=i && i<size();
 get(i) != null &&
 (i<j ? get(i).compareTo(get(i+1))<=0 : get(i).compareTo(get(i+1))>=0));
```

Di fatto, la posizione *j* corrisponde al massimo elemento nella ListaStrana.

2) ListaOrdinata e’ un caso speciale di ListaStrana, ma non puo’ essere definita come sottoclasse che rispetti il principio di sostituzione in quanto essa eredita pure l’operazione *insert1* che non si applica a ListaOrdinata. Anche il viceversa non e’ possibile, in quanto ListaStrana violerebbe la proprieta’ dell’ordinamento di ListaOrdinata (bastano due chiamate a *insert1* per introdurre un “pezzo” decrescente).

3) *insert* contiene come postcondizione la stessa specifica di *Lista.insert*, ma inserisce solo nella porzione *j*, definita dalla stessa formula usata nell’invariante, che stabilisce l’ordinamento:

```
size() == \old(size())+1 && //questa e’ come in Lista.insert
 (\forall T y; \old(contains(y));
  (\num_of T z; contains(z); y.equals(z)) ==
  (\num_of T w; \old(contains(w); y.equals(w))
   + (x.equals(y) ? 1 : 0) &&
 \old(size()>0)==>(\exists int j; 0<=j && j<size(); //qui e’ come in invariante
  (\forall int i; 0<=i && i<size();
   (i<j ? get(i).compareTo(get(i+1))<=0 : get(i).compareTo(get(i+1))>=0)) &&
   (\exists int k; 0\le k && k <=j; get(k).equals(x)); //x sta in prima parte
```

La *insert1* e’ simmetrica.

Domanda D

Si consideri una classe ListaEstesa, definita come Lista, ma che definisce un altro ordine in cui si trovano gli elementi. A tale scopo, ListaEstesa fornisce, oltre alle operazioni di Lista, le seguenti operazioni:

```
//@ensures (* aggiunge x nella posizione immediatamente precedente al primo elemento  
//@ equals(y). Se y non esiste in this, allora inserisce x in ultima posizione *);  
public void insertAfter(T x, T y)
```

e ridefinisce element() in modo che restituisca tutti e soli gli elementi di this nell'ordine di *inserzione*.

- 1. Specificare insertAfter in JML.**
- 2. E' possibile definire ListaEstesa come erede di Lista?**
- 3. Come cambia la risposta alla Domanda B(1) se al posto di Lista si considera ListaEstesa?**

Soluzione:

```
//@ensures ( (\exists int i; 0 < i && i < size(); get(i).equals(y) && get(i-1).equals(x) ||  
! (\exists int i; 0 < i && i < size(); get(i).equals(y)) && get(size()-1).equals(x))  
&& (* qui stessa parte che in insert assicura che gli elementi sono rimasti gli stessi salvo l'inserimento *)  
public void insertAfter(T x, T y)
```

ListaEstesa puo' essere erede di Lista, in quanto Lista non prevede particolari vincoli sull'inserimento.

ListaOrdinata non puo' ereditare da ListaEstesa (in quanto il metodo insertAfter sarebbe ereditato), ma nemmeno il viceversa, in quanto ListaEstesa viola la regola delle proprietà consentendo tramite insertAfter di inserire in posizioni che non rispettano l'ordine.

Domanda E

Si consideri la seguente implementazione di ListaOrdinata (che si suppone definita “da zero”, ossia senza estendere Lista o ListaEstesa o ListaStrana).

```
\rep:  
private ArrayList<T> elems;  
private ArrayList<Integer> ordine;  
  
public ListaOrdinata ()  
    elems = new ArrayList<T>();  
}
```

Il vettore *elems* contiene tutti gli elementi della lista, in un ordine qualunque. Il vettore *ordine* contiene per ogni posizione i , $0 \leq i \leq \text{elems.size()} - 1$, la posizione del successore in *elems* di *elems.get(i)*. Per l’ordinamento, si usa sempre il *compareTo*. Se l’elemento in posizione i non ha un successore (ossia è il max), *ordine* in posizione i contiene -1.

Ad esempio, se *elems* contiene [A E B F C], allora *ordine* contiene [2 3 4 -1 1]

Scrivere l’invariante di rappresentazione in JML per ListaOrdinata.

Soluzione:

```
private invariant elems.size() == ordine.size() &&  
(\num_of int j; 0<= j && j <elems.size(); elems.get(j) ==-1) ==1 &&  
(\forallall int i; 0<= i && i <elems.size();  
    elems.get(i) !=null && ordine.get(i) !=null &&  
    (ordine.get(i) !=-1 ? ordine.get(i) >=0 &&  
        ordine.get(i) < ordine().size() &&  
        elems.get(i).compareTo(elems.get(ordine.get(i)))<=0 &&  
        !(\exists int j; j>=0 && j<elems.size();  
            j!= i && elems.get(i).compareTo(elems.get(j)<0 &&  
            elems.get(j).compareTo(elems.get(ordine.get(i)))>0))  
    : (\forallall int j; 0<=j && j<elems.size();  
        elems.get(i).compareTo(elems.get(j))>=0)) &&
```

Usando la notazione preferita (tramite oggetto tipico, oppure un private invariant in JML, oppure implementando il metodo *toString()* di ListaOrdinata in Java), **descrivere la funzione di astrazione della ListaOrdinata**.

Soluzione: Esprimiamo la funzione di astrazione ad esempio con un invariante che mette in relazione i metodi osservatori puri *get()* e *size()* con il rep.

```
private invariant  
size()== elems.size() &&  
get(size()-1) == elems(ordinate.indexof(-1)) &&  
(\forallall int i; 0<= i && i<size()-1; get(i) == ordine.indexof(i)-1);
```

Implementare il metodo insert() di ListaOrdinata. Soluzione:

```
elems.add(x); //aggiunge x come ultimo elemento  
if (elems.size()== 1) {  
    ordine.add(-1);  
    return;  
}  
int minimo = 0; //indice del minimo elemento (da trovare)-- all'inizio e' 0;  
(for int i=0; i<ordine.size(); ++i) {  
if (elems.get(i).compareTo(elems.get(minimo))<0)  
    minimo =i; //elems.get(i) e' il minimo dei primi i elementi  
    int succ = ordine.get(i); //succ contiene indice del successore di elems.get(i);  
    if (succ == -1 ) //se elems.get(i) e' il massimo elemento
```

```
if (elems.get(i).compareTo(x) == -1) { //se il massimo e' < x
    ordine.add(-1); //il nuovo max e' x, ossia l'ultimo elemento
    ordine(i)= elems.size()-1; //il succ di i e' proprio x, in ultima posiz.
    return;
}
else {
    if (elems.get(i).compareTo(x)<0 && elems.get(succ).compareTo(x)>0) {
        //se x e' compreso fra elems.get(i) e il suo successore:
        ordine.add(succ); //x ha come successore il succ. di elems.get(i)
        ordine.set(i,elems.size()-1); //elems.get(i) ha x come succ.
        return;
    }
}
//in uscita dal ciclo, x deve essere minore del minimo elemento;
ordine.add(minimo);
```

Domanda F

Dato il rep definito alla Domanda E, si definisca e implementi il metodo iteratore elementi() di ListaOrdinata.

Soluzione:

```
public Iterator<T> elementi() {return new elgen<T>(this);}

private static class elgen<T> implements Iterator<T> {
    private int current;
    private ListaOrdinata<T> lista;
    public elgen<T>(ListaOrdinata c) {
        lista = c;
        if (c.size() == 0) {//lista vuota
            current = -1;
            return;
        }
        current = 0; //ora cerca il minimo elemento
        for int i = 1; i < lista.ordine.size(); i++)
            if (elems.get(current).compareTo(elems.get(i)) > 0) current = i;
        //in uscita, current punta al minimo elemento di elems
    }
    public boolean hasNext() {
        return current >= 0;
    }
    public T next() throws NoSuchElementException {
        if (current < 0) throw new NoSuchElementException();
        T temp = elems.get(current);
        current = ordine.get(current);
        //quando current e' -1, temp e' il massimo, ossia l'ultimo elem. da restituire
        return temp;
    }
}
```

Esercizio 2

Si consideri il seguente frammento di programma

```
1. ``legge n da input'';
2. if (n>0) {
3.     j = 2*n;
4.     while (j>0) {
5.         j--;
6.     }
7. ----;
8. };
```

1. Si definiscano casi di test che coprano tutte le istruzioni (si considerino le parte tralasciate “—” come se fossero una singola macro-istruzione);
2. Si definiscano casi di test che coprono tutti i branch.
3. Si consideri il cammino: 1 2 3 4 5 6 4 7 (che attraversa il ciclo 1 sola volta) e si derivi sia una condizione logica sull'input n, che causi l'attraversamento del cammino, che valori specifici di input che ne causano l'attraversamento.
4. Si risolva lo stesso problema per il cammino 1 2 3 4 5 6 4 5 6 4 7 (che corrisponde ad eseguire il ciclo due volte)

Soluzione: 1) Basta prendere $n = 1$.
2) Basta prendere $n = 0$ e $n = 1$.
3) La condizione e' $0 < n \leq 0.5$. Se n e' un int allora non esiste alcun valore, se fosse un float basterebbe ad es. $n = 0.5$.
4) La condizione e' $0.5 < n \leq 1$, ossia basta prendere $n = 1$.

Ingegneria del Software — SOLUZIONE DEL TEMA 10 settembre 2020

Esercizio 1

Si consideri la seguente classe `ImmList<T>` che definisce una lista immutabile di elementi di tipo `T`.

```
public /*@ pure */ class ImmList<T> {
    // Ritorna true se e solo se la lista e' vuota
    public boolean empty();

    // Ritorna il primo elemento della lista
    // Lancia EmptyException se la lista e' vuota
    public T first() throws EmptyException;

    // Ritorna una nuova lista identica a this ma senza il primo elemento
    // In caso di lista vuota, ritorna una nuova lista vuota
    public ImmList<T> tail();

    // Ritorna una nuova lista che contiene el in ultima posizione
    // Lancia NullException se el e' null
    public ImmList<T> add(T el) throws NullException;

    // Ritorna il numero di occorrenze dell'elemento el nella lista
    public int count(T el)
}
```

Domanda a)

Si scriva la specifica JML del metodo `add`.

Soluzione

```
// @ensures el != null && \result != null &&
// @this.empty() ==>
//@  (\result.first().equals(el) && \result.tail().empty()) &&
//@! this.empty() ==>
//@  (\result.first().equals(first()) && \result.tail().equals(tail().add(el)))
//@
// @signals NullException el == null
public ImmList<T> add(T el) throws NullException;
```

Domanda b)

Si scriva la specifica JML del metodo `count`.

Soluzione

```
// @ensures
// @empty() ==> \result == 0 &&
//@! empty() ==>
//@  \result == tail().count(el) + first().equals(el) ? 1 : 0
public int count(T el);
```

Domanda c)

Si consideri una classe `ImmList2<T>` che aggiunge un metodo `remove` che restituisce una nuova lista identica a `this` ma senza l'ultimo elemento (in caso di lista vuota, restituisce una nuova lista vuota). È possibile definire `ImmList2<T>` come derivata da `ImmList<T>`, rispettando il principio di sostituzione?

Soluzione

Si, in quanto `ImmList2<T>` sarebbe un'estensione pura di `ImmList<T>`, che aggiunge un metodo senza modificare le specifiche degli altri, e non modifica alcuna proprietà della classe `ImmList<T>`.

Domanda d)

Si consideri una classe `ImmList3<T>` in cui il metodo `add` non lancia eccezioni ma permette l'inserimento di elementi nulli. È possibile definire `ImmList3<T>` come derivata da `ImmList<T>`, rispettando il principio di sostituzione?

Soluzione

No, per due ragioni. (1) Si violerebbe la regola dei metodi, dato che il mancato lancio dell'eccezione *non* rafforza la post-condizione del metodo `add`. (2) Si violerebbe la regola delle properità, invalidando la proprietà che la lista non possa contenere valori nulli.

Esercizio 2

Si consideri la seguente classe Java:

```
public class Bank {
    private List<String> accName;
    private List<Double> accBalance;
    public Bank() {
        accName = new ArrayList<String>();
        accBalance = new ArrayList<Double>();
    }
    public void addAccount(String clientName) throws DuplicateAccountException {
        synchronized(accName) {
            synchronized(accBalance) {
                if(accName.contains(clientName)) throw new DuplicateAccountException();
                accName.add(clientName);
                accBalance.add(0.0);
            }
        }
    }
    public double getBalance(String clientName) throws MissingAccountException {
        synchronized(accName) {
            synchronized(accBalance) {
                int pos = accName.indexOf(clientName);
                if(pos== -1) throw new MissingAccountException();
                return accBalance.get(pos);
            }
        }
    }
    public void deposit(String clientName, double amount) throws MissingAccountException {
        synchronized(accName) {
            synchronized(accBalance) {
                int pos = accName.indexOf(clientName);
                if(pos== -1) throw new MissingAccountException();
                double newBalance = accBalance.get(pos) + amount;
                accBalance.set(pos, newBalance);
            }
        }
    }
    public void withdraw(String clientName, double amount) throws MissingAccountException {
        synchronized(accName) {
            synchronized(accBalance) {
                int pos = accName.indexOf(clientName);
                if(pos== -1) throw new MissingAccountException();
                double newBalance = accBalance.get(pos) - amount;
                accBalance.set(pos, newBalance);
            }
        }
    }
}
```

Domanda a)

La classe è correttamente sincronizzata (ovvero non si verificano né deadlock né potenziali corse critiche)? Motivare la propria risposta.

Soluzione

La sincronizzazione è corretta (seppur irrituale). La doppia sincronizzazione annidata sui due attributi protegge correttamente l'accesso ai dati condivisi e non da luogo a deadlock perché ogni thread acquisisce i due lock nel medesimo ordine.

Domanda b)

Si modifichi il codice dei soli metodi addAccount e getBalance affinché il metodo getBalance, invece di sollevare una eccezione, sospenda il chiamante se il cliente non è presente nella banca, in attesa che il metodo addAccount aggiunga l'account. Si suggerisce di prestare particolare attenzione nel modificare i due metodi al fine di evitare di introdurre potenziali deadlock.

Soluzione

```
public void addAccount(String clientName) throws DuplicateAccountException {
    synchronized(accName) {
        synchronized(accBalance) {
            if(accName.contains(clientName)) throw new DuplicateAccountException();
            accName.add(clientName);
            accBalance.add(0.0);
        }
        accName.notifyAll();
    }
}

public double getBalance(String clientName) {
    synchronized(accName) {
        while(!accName.contains(clientName)){
            try { accName.wait(); } catch(InterruptedException ex) {ex.printStackTrace();}
            synchronized(accBalance) {
                int pos = accName.indexOf(clientName);
                return accBalance.get(pos);
            }
        }
    }
}
```

Si noti come la chiamata al metodo `wait` in `getBalance` sia fatta prima di acquisire il secondo lock, al fine di evitare potenziali deadlock.

Esercizio 3

Si consideri il seguente frammento di un programma sequenziale, dove y, z sono variabili intere:

```
1 int x=0;
2 int k=10;
3 while (x <= 10 && z>0) {
4     if (z <= y && k>=x)
5         y = y - z;
6     k--;
7     if (y > 0) x++;
8     else break;
9 }
```

Domanda a)

Si definisca il minimo numero di casi di test (per y, z) necessari e sufficienti per coprire tutte le istruzioni.

Soluzione

Ne basta uno: $y = 2, z = 1$.

Domanda b)

Si calcoli la condizione logica (path condition) che impone che il ciclo venga eseguito tre volte, uscendo alla terza iterazione eseguendo l'istruzione break.

Soluzione

$z > 0 \wedge y = 3z$

Domanda c)

Si definisca il minimo numero di casi di test necessari e sufficienti per coprire tutte le condizioni e le diramazioni (edge and condition coverage), se ciò è possibile.

Soluzione

Bastano tre casi. Il primo caso consiste in un valore negativo o nullo per z , p.es. $z = 0$.

Il secondo caso è quello già visto per la copertura delle istruzioni (copre il valore vero e falso di $z \leq y$ e di $y > 0$).

Il terzo caso di test deve rendere falsi $x \leq 10$ e $k \geq x$: a tale scopo, occorre eseguire il ciclo while per 11 volte con $y > 0$ in modo da eseguire ogni volta $x++$: basta scegliere $y > 6z > 0$, p. es. $y = 7, z = 1$. In questo modo al termine della sesta iterazione $k == 4, x == 5, y == 1$, quindi $k \geq x$ diventa falso: l'assegnamento di y alla riga 5 non può essere eseguito, quindi y resta positivo: dopo altre 5 iterazioni $x == 11$.



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da

Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Prova di recupero di Ingegneria del Software

Parte 1

Parte 2

16 Luglio 2003

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h 30 min per ciascuna delle due parti.
6. Punteggio totale a disposizione: 13/30 per ciascuna delle due parti.
7. **Recupero parte 1: esercizi da 1 a 4. Recupero parte 2: esercizi da 5 a 8. Recupero entrambe le parti: esercizi da 1 a 8.**

Esercizio 1

Si consideri la seguente gerarchia:

```
interface A {  
    void a1();  
    int a2(B b);  
}  
  
class B {  
    public void b1(){...}  
    public int b2(A a){...}  
}  
  
class C implements A extends B {  
    public void b1(int x) {...}  
    public int b2(A a) {...}  
  
    public void a1(){...}  
    public int a2(B b){...}  
}
```

Si considerino le seguenti dichiarazioni:

int i, j;
A y;
B z;
C w;

1. Le seguenti istruzioni sono corrette? Si scriva SI o NO a fianco di ciascuna e, nel caso la risposta sia NO, una sintetica motivazione.

y.a2(z);	SI
y.a2(w);	SI
y = w;	SI
i = z.b2(y);	SI
j = z.b2(w);	SI
w = y;	NO il tipo apparente di y (A) è soprattutto del tipo apparente di w (C)

2. Sarebbe corretto se al posto dell'ultima istruzione $w = y$; ci fosse l'istruzione
 $w = (C) y;$

Motivare sinteticamente la risposta.

Sì, sarebbe corretto, perché viene fatto un cast esplicito al sottotipo (C) del tipo apparente di y (A). Alla peggio avrei errore a run-time, se il cast non fosse possibile.

3. Si considerino le seguenti due istruzioni:

w.b1();
w.b1(3);

Sono corrette? Se si, quale metodo b1 viene chiamato da ciascuna delle due istruzioni? Motivare sinteticamente la risposta.

Sì, sono corrette in entrambi i casi, in quanto b1 senza argomenti viene definito da B (ed ereditato da C), mentre b1 con argomento intero viene definito da C. Nel primo caso il metodo chiamato è quello di B, che è l'unico che non vuole argomenti in ingresso, nel secondo è quello di C, che vuole in ingresso un intero.

4. Si consideri la sequenza $z = w; z.b1();$

Quale metodo b1 viene chiamato? Perchè?

Il metodo di B, in quanto non viene ridefinito da C, ma ereditato così come è.

E quale metodo b2 viene chiamato nel caso seguente?

$z = w; z.b2(y);$

Il metodo ridefinito da C, con il quale viene fatto binding dinamico a run-time.

Esercizio 2

Si consideri la seguente situazione di progetto: un'applicazione in Java è composta da una classe A che chiama un metodo `metB` di una classe B. L'interfaccia del metodo `metB` elenca possibili eccezioni che possono essere propagate al chiamante. Si vuole fare in modo che la chiamata del metodo `metB` contenuta in A effettui una gestione locale di tutte le eccezioni dichiarate nell'interfaccia di `metB` che `metB` può generare e che, terminata la gestione locale, propaghi a sua volta l'eccezione ricevuta da `metB` al proprio cliente.

Descrivere come deve essere fatta la classe A affinché ciò avvenga, mostrando in particolare un suo metodo `metA` che contiene la chiamata di `metB`.

Ogni metodo di A che chiama `metB` deve avere, attorno ad ogni chiamata di `metB`, un blocco `try...catch` che catturi le eccezioni sollevate da `metB`, le gestisca localmente, e poi le risollevi con una nuova `throw`. Inoltre, la signature di ogni metodo di A che chiama `metB` deve dichiarare di sollevare le eccezioni dichiarate da `metB`.

Per esempio, se `metB` solleva una sola eccezione `metBException`, un metodo `metA` potrebbe essere fatto in questa maniera:

```
metA() throws metBException {  
    ...  
    try{  
        metB();  
    } catch (metBException e) {  
        /* gestione locale dell'eccezione */  
        throw e;  
    }  
}
```

Esercizio 3

Specificare la seguente astrazione procedurale, fornendo pre e post condizioni (senza quindi implementare l'astrazione):

Dato un *array* x di Integer e un valore intero a, l'astrazione restituisce un *insieme* di valori interi in x la cui somma è a. L'insieme restituito è un'istanza della classe IntSet.

Si richiede di scrivere la specifica in due versioni:

1. la specifica *impone al chiamante* di verificare se effettivamente esiste un insieme di valori in x la cui somma sia a.

Risposta:

notazione: per un IntSet I e un array di Integer x, definiamo incluso(I, x) come:

forall i (I.isIn(i) => exists j a[j].intValue() = i)

e somma(I) come: $\sum_{i \in I} i$

IntSet somma(Integer[] x, int a)

REQUIRES: exists IntSet I : somma(I) == a && incluso(I,x)

EFFECTS: restituisce un IntSet I: somma(I) == a && incluso(I,x)

2. la specifica assume che l'insieme di valori richiesto sia presente nell'array, *gestendo come comportamento anomalo* il caso in cui ciò non accada.

Risposta:

Non c'è clausola requires. Effects diventa:

EFFECTS: if exists IntSet I : somma(I) == a && incluso(I,x) restituisci un IntSet I: somma(I) == a && incluso(I,x)

else throw SommaInsiemeNonValidaException

dove **SommaInsiemeNonValidaException** è definita opportunamente come erede della classe **Exception**.

Esercizio 4

Si consideri un tipo di dato astratto che rappresenta un conto corrente bancario. Sul conto è possibile effettuare le seguenti operazioni

- creazione di un nuovo conto, con nessuna operazione effettuata e saldo corrente nullo;
- *versamento*: si deposita una certa somma (per semplicità si assume che venga versato un numero intero di euro), il saldo complessivo del conto ne risulta incrementato in corrispondenza;
- *prelievo*: si ritira dal conto una certa somma (anche qui un numero intero di euro), e il saldo complessivo ne risulta decrementato;
- *saldo*: viene restituita la quantità di denaro attualmente presente nel conto;
- *riepilogo*: cancella dal conto corrente tutte le operazioni effettuate dalla creazione del conto o dal riepilogo precedente e aggiorna di conseguenza il *saldoUltimoRiepilogo*; delle operazioni effettuate prima del riepilogo non rimane più traccia, se non indirettamente nel *saldoUltimoRiepilogo*.

Si assuma per semplicità che il conto possa anche “andare in rosso” e non ci siano limitazioni sulla quantità di denaro che si può prelevare o versare.

- Si specifichi il tipo di dato astratto.
- Dovendo implementare la classe, se ne definisca la rappresentazione interna e l’Invariante di Rappresentazione; per la rappresentazione interna si memorizzi in opportuni attributi le operazioni di versamento e prelievo del mese corrente, il saldo dell’ultimo riepilogo e il saldo complessivo attuale, risultante da tutte le operazioni effettuate finora.
- Si implementi la classe.
- Si dimostri che, nell’implementazione fornita, l’Invariante di Rappresentazione è effettivamente verificato.

Specifiche:

```
class contoCorrenteADT {  
    //OVERVIEW: rappresenta un conto corrente, che comprende un saldo,  
    //           ed un elenco di operazioni dall'ultimo riepilogo, ed un  
    //           saldo dall'ultimo riepilogo effettuato.  
  
    public contoCorrenteADT(){  
        //EFFECTS: inizializza il conto corrente, ponendo  
        //           this.saldo = 0, this.operazioni = Ø  
        //           e this.saldoUltimoRiepilogo = 0  
    }  
  
    public void versamento (int somma){  
        //REQUIRES: somma > 0  
        //EFFECTS: this.saldo = this.saldo + somma  
        //           this.operazioni = this.operazioni ∪ "+ somma"  
    }  
  
    public void prelievo (int somma){  
        //REQUIRES: somma > 0  
        //EFFECTS: this.saldo = this.saldo - somma  
        //           this.operazioni = this.operazioni ∪ "- somma"  
    }  
  
    public int saldo (){  
        //EFFECTS: ritorna this.saldo  
    }  
}
```

```

public void riepilogo (){
    //EFFECTS: this.saldoUltimoRiepilogo =
    //          this.saldoUltimoRiepilogo +  $\sum$ (this.operazioni)
    //          this.operazioni =  $\emptyset$ 
}

}

```

Realizzazione:

```

class contoCorrenteADT {
    //OVERVIEW: rappresenta un conto corrente, che comprende un saldo,
    //           ed un elenco di operazioni dall'ultimo riepilogo, ed un
    //           saldo dall'ultimo riepilogo effettuato.

    //Rappresentazione interna:
    private int saldo;
    private int saldoUltimoRiepilogo;
    private Vector operazioni;

    //rep invariant:
    //operazioni != null &&
    //saldo = saldoUltimoRiepilogo +  $\sum$ (this.operazioni)

    public contoCorrenteADT(){
        //EFFECTS: inizializza il conto corrente, ponendo
        //          this.saldo = 0, this.operazioni =  $\emptyset$ 
        //          e this.saldoUltimoRiepilogo = 0
        saldo = 0;
        saldoUltimoRiepilogo = 0;
        operazioni = new Vector();
    }

    public void versamento (int somma){
        //REQUIRES: somma > 0
        //EFFECTS: this.saldo = this.saldo + somma
        //          this.operazioni = this.operazioni  $\cup$  "+ somma"
        saldo += somma;
        operazioni.add(new Integer(somma));
    }

    public void prelievo (int somma){
        //REQUIRES: somma > 0
        //EFFECTS: this.saldo = this.saldo - somma
        //          this.operazioni = this.operazioni  $\cup$  "- somma"
        saldo -= somma;
        operazioni.add(new Integer(-somma));
    }
}

```

```

public int saldo (){
    //EFFECTS: ritorna this.saldo
    return saldo;
}

public void riepilogo (){
    //EFFECTS: this.saldoUltimoRiepilogo =
    //          this.saldoUltimoRiepilogo +  $\sum$ (this.operazioni)
    //          this.operazioni =  $\emptyset$ 
    for(int i=0; i<operazioni.size(); i++) {
        saldoUltimoRiepilogo +=  

            ((Integer)operazioni.elementAt(i)).intValue();
    }
    operazioni = new Vector();
}
}

```

Il costruttore mantiene l'invariante di rappresentazione in quanto tutto viene messo a 0. I modificatori sono **versamento**, **prelievo** e **riepilogo**. In tutti e 3 i casi, se il rep invariant vale all'inizio, esso vale anche alla fine. Nel caso di **versamento**, **saldoUltimoRiepilogo** rimane invariato, ma **saldo** viene incrementato di **somma**, e ad **operazioni** viene aggiunto un "+**somma**" (simmetricamente per **prelievo**). Nel caso di **riepilogo**, invece, il contenuto di **operazioni** viene "trasferito" a **saldoUltimoRiepilogo**, ed il vettore viene azzerato, mantenendo così sempre l'invariante.

Esercizio 5

Sia supponga che sia data una classe `IntSet` che definisce i set di interi. Si vuole definire una classe che implementi l'insieme di interi dispari.

1. E' possibile farlo definendo una sottoclasse `OddIntSet` che ridefinisce l'operazione di inserimento mettendo una precondizione che richiede che il valore da inserire passato come parametro sia dispari? Giustificare sinteticamente sia l'eventuale risposta affermativa che quella negativa.

Risposta: no, perché in questo modo sarebbe violata la regola dei metodi, che richiede che il metodo ridefinito accetti (almeno) tutti gli ingressi che il metodo di `IntSet` accettava (“require no more”) (si rafforza la precondizione).

2. L'inserimento della precondizione a chi delega la responsabilità della verifica che la precondizione sia verificata?

Risposta: è il chiamante di un metodo che deve verificare di rispettare la precondizione del metodo che intende invocare.

3. E' corretto definire una sottoclasse `OddIntSet` che ridefinisce l'operazione di inserimento in modo che vengono accettati tutti i parametri in ingresso, ma se si cerca di inserire un valore non dispari viene sollevata un'eccezione `evenNumberException`? Giustificare sinteticamente sia l'eventuale risposta affermativa che quella negativa.

Risposta: no, perché se `evenNumberException` è definita come checked allora si viola la regola della segnatura, mentre se fosse unchecked verrebbe violata la regola dei metodi, stavolta perché il metodo ridefinito non si comporta, sui valori accettati dal metodo che sta ridefinendo, come quest'ultimo (“ensure no less”) (si indebolisce cioè la postcondizione)

4. Si supponga di voler risolvere il problema mediante una soluzione generica, utilizzando una classe `GenSet` che definisce insieme di `Object`, che poi si vuole usare per generare sia gli insiemi di `Integer` e gli insiemi di `OddInteger`. Si sceglie di usare una classe “`Insertor`” (secondo la metodologia del “related subtype”) per effettuare l'inserimento nell'insieme controllando che si tratti di numeri integer o dei valori dispari a seconda del caso. Si realizzi la soluzione.

Risposta:

```
package set; //viene omessa la suddivisione del package in compilation unit

public class GenSet {    //viene omessa l'implementazione della classe
    ...
    void insert(Object o) { ... } //notare che il metodo ha visibilità package: gli insertor devono
                                //pertanto essere implementati in questo package
    ...
}

public class InvalidElementException extends Exception {...}

public interface Insertor {
    void insert(GenSet s, Object o) throws InvalidElementException;
    //inserisce o in s, verificando che o abbia alcune caratteristiche.
    //Se o non possiede tali caratteristiche, lancia ClassCastException.
    //E' una versione molto generica, quindi il secondo parametro ha tipo Object
```

```
//(accettabile anche quella specifica all'esercizio, che accetta parametro integer)
}

public class IntegerInsertor implements Insertor {
    void insert(GenSet s, Object o) throws InvalidElementException {
        if (!(o instanceof Integer)) throw new InvalidElementException();
        s.insert(o);
    }
}

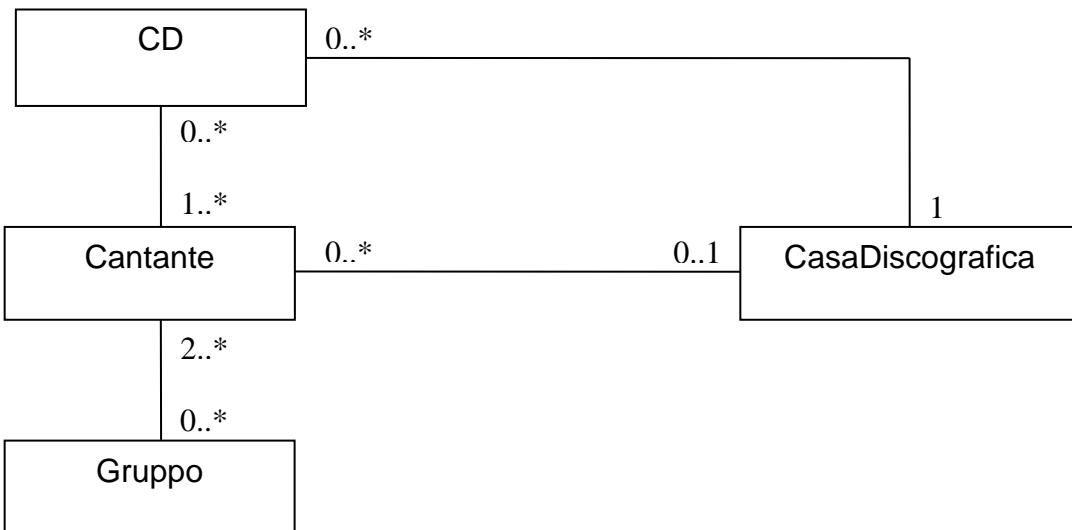
public class OddIntegerInsertor implements Insertor {
    void insert(GenSet s, Object o) throws InvalidElementException {
        Integer i;
        if (!(o instanceof Integer)) throw new InvalidElementException();
        i = (Integer) o;
        if (i.intValue() % 2 == 0) throw new InvalidElementException();
        s.insert(i);
    }
}
```

Esercizio 6

Si vogliono usare i diagrammi UML per esprimere l'associazione tra cantanti e case discografiche. Si vogliono descrivere le seguenti proprietà:

- 1) una casa discografica può avere un numero arbitrario di cantanti e un cantante può incidere musica solo per una casa discografica;
- 2) si esprima il vincolo ulteriore che oltre ai cantanti singoli esistano i gruppi, che sono fatti da più cantanti.
- 3) si introduca ora anche l'entità cd e si esprima in un diagramma UML le seguenti proprietà di cd, case discografiche e cantanti: Un cd viene pubblicato da una e una sola casa discografica e un cantante pubblica un numero arbitrario di cd.
- 4) per il caso 1), si consideri un'implementazione in cui esistono due classi Java Cantante e CasaDiscografica. Come implementereste la relazione che deve sussistere tra gli oggetti delle due classi? Rispondere tratteggiando l'implementazione delle due classi e discutendone le implicazioni.

Risposta:



```
class Cantante {
    private CasaDiscografica laCasa; //è null per cantanti che non hanno una casa discografica
    ...
}

class CasaDiscografica {
    ... //nulla di specifico
}
```

Dal momento che ogni cantante ha al più una casa discografica, posso implementare la relazione con un attributo all'interno della classe Cantante. Lo svantaggio è la difficoltà nel determinare, ad esempio, tutti i cantanti di una casa discografica. Nel caso ciò fosse più importante, si può implementare la relazione inversa aggiungendo alla classe CasaDiscografica un attributo `private Cantante[] cantanti; (o un Vector)` e curare la consistenza dei due tipi di attributi.

Esercizio 7

Con riferimento a Java RMI, spiegare sinteticamente le differenze fondamentali nel passaggio dei parametri a un oggetto remoto, distinguendo tra il caso in cui il parametro viene serializzato e quello in cui si passa un riferimento a un oggetto remoto.

Risposta: se il parametro viene serializzato, viene creato un clone del parametro sulla macchina dove risiede l'oggetto di cui viene invocato il metodo. Se questo a sua volta modifica il parametro ricevuto (invocando un suo mutator), l'effetto sarà sul clone, non sull'oggetto originale. Se viene passato il riferimento ad un oggetto remoto, invece, il metodo può accedere all'oggetto originale e modificarlo.

Esercizio 8

Si consideri il seguente frammento di programma. Quanti casi di test sono necessari per coprire tutte le diramazioni (decisioni) e quanti sono necessari per coprire tutti i cammini? Motivare sinteticamente la risposta.

```
if (condizione_1) {  
    istruz_1;  
}  
else  
    if (condizione_2)  
        istruz_2;  
    else  
        istruz_3;  
if (condizione_2)  
    istruz_4;  
else  
    istruz_5;
```

Risposta: occorre aggiungere ulteriori ipotesi sulle condizioni e sulle istruzioni per poter determinare il numero necessario (ossia minimo) di cammini. Supponiamo allora che le due condizioni logiche siano indipendenti tra di loro (ossia, che possano essere verificate e falsificate indipendentemente l'una dall'altra) e atomiche, che le istruzioni non siano return, e che non modifichino condizione_2. Con queste assunzioni un test con tre casi è sufficiente: ad esempio un test nel quale il primo caso fa si che condizione_1 == true e condizione_2 == true (primo if, ramo then; terzo if, ramo then), il secondo fa si che condizione_1 == false e condizione_2 == true (primo if, ramo else; secondo e terzo if, ramo then), e il terzo fa si che condizione_1 == false e condizione_2 == false (primo if, ramo else; secondo e terzo if, ramo else). I cammini percorribili sono 4 (occorre aggiungere il caso: primo if, ramo then; terzo if, ramo else, gli altri cammini sul grafo di flusso non sono percorribili). Pertanto occorre aggiungere il caso di test con condizione_1 == true e condizione_2 == false.

Appello del 9 Febbraio 2018



Politecnico di Milano
Anno accademico 2017-2018

Ingegneria del Software

Cognome:

LAUREANDO

Nome:

Matricola:

Sezione (segnarne una): Cugola

Mottola

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri un programma Java, rivolto ad appassionati di enigmistica, per la soluzione di semplici Crucipuzzle, di cui viene dato un esempio in figura.

M	J	Y	H	A	E	U	S	O	O	J	E	C	P	E
Q	Z	F	I	N	Z	G	K	G	V	R	V	N	S	T
S	Y	P	O	I	L	G	O	F	O	I	F	M	T	N
O	M	R	O	F	Z	A	G	I	N	D	Y	E	I	E
E	A	A	R	I	G	G	L	O	S	L	C	F	R	M
B	O	Y	S	R	J	O	2	T	X	M	F	A	D	A
N	P	S	O	E	P	L	A	O	I	A	O	G	D	
M	O	G	L	U	A	E	Z	S	B	Q	T	U	O	I
R	S	L	R	C	H	V	Z	T	H	U	V	I	U	M
R	A	C	N	A	M	F	E	O	A	D	G	G	J	I
C	S	J	L	B	T	H	R	V	V	G	N	W	T	T
V	E	M	U	A	R	I	A	I	I	T	V	T	N	
F	Q	G	I	M	M	O	S	R	P	N	I	Z	D	C
Z	N	H	B	O	L	L	A	T	U	N	B	E	I	E
E	P	A	C	D	F	K	P	J	I	I	I	O	R	U

Il gioco consiste nel selezionare parole di senso compiuto all'interno della scacchiera: dall'alto al basso, dal basso all'alto, da sinistra a destra o da destra a sinistra, ma anche in diagonale. Ci sono quindi 8 direzioni possibili della selezione, che si possono individuare con i punti cardinali: N, S, E, O, NO, NE, SO, SE. A tale scopo, si definisce una classe mutabile `Selezione`, i cui oggetti contengono una sequenza di celle selezionate unitamente alla direzione della selezione (a partire dalla prima lettera, detta origine).

Si definisce poi una classe `Crucipuzzle` i cui oggetti includono il contenuto del puzzle (scacchiera) e le selezioni corrette effettuate fino a quel momento. I dettagli si possono evincere dal codice qui sotto riportato, che include solo una specifica delle varie classi.

```
public class Selezione {  
    /* Costruisce una selezione a partire dalle coordinate della prima cella,  
     * seguendo una direzione fissata*/  
    public Selezione(int x, int y, String direzione) ;  
  
    /* Aggiunge una cella in più alla selezione, nella direzione  
     * predefinita, lanciando eccezione se si è raggiunto il bordo  
    public void aggiungiCella() throws OutOfBoundsException;  
  
    /* Ritorna indietro, deselezionando una cella;  
     * lancia eccezione se si cerca di rimuovere l'origine */  
    public void rimuoviCella() throws OriginException ;  
  
    /* Restituisce la stringa corrispondente alle celle selezionate */  
    public /*@ pure @*/ String selezione() ;  
  
    /* Restituisce la direzione della selezione */  
    public /*@ pure @*/ String direzione() ;  
    ....  
}  
  
public class Crucipuzzle {  
    /* Costruisce un crucipuzzle a partire da una matrice rettangolare di  
     * caratteri e da un array che contiene le parole da trovare.  
    public Crucipuzzle(char[][] schema, String[] paroleContenute);  
  
    /* Restituisce il contenuto del puzzle in una matrice rettangolare */  
    public /*@ pure @*/ char[][] scacchiera();  
  
    /* Restituisce un ArrayList con le parole ancora da individuare */  
    public /*@ pure @*/ ArrayList<String> parole();
```

```

/* Aggiunge una nuova selezione al puzzle rimuovendo la parola corrispondente
 * dall'elenco delle parole ancora da individuare; restituisce true sse la selezione
 * completa il puzzle; lancia eccezione se la selezione è relativa ad una parola
 * già trovata o che non è prevista fra le parole contenute nel puzzle. */
public boolean aggiungiSelezione(Selezione s) throws SelezioneErrataException ;
....
```

Domanda a

Si specifichi in JML il metodo aggiungiSelezione

Soluzione

```

//@ensures \old(parole().contains(s.selezione())) &&
//@      parole().size()==\old(parole().size()-1) &&
//@      (\forall String x; \old(parole().contains(x));
//@          x!=s.selezione() ==> parole().contains(x)) &&
//@      \result <==> (parole().size()==0);
//@signals(SelezioneErrataException e) !\old(parole().contains(s.selezione())) &&
//@      parole().size()==\old(parole().size()) &&
//@      (\forall String x; \old(parole().contains(x); parole().contains(x));
public boolean aggiungiSelezione(Selezione s) throws SelezioneErrataException ;
```

Domanda b

E' data la seguente implementazione parziale di Crucipuzzle. L'array di booleani trovata contiene true alla posizione i se e solo se la parola nella stessa posizione dell'ArrayList parole è già stata individuata.

```
import java.util.ArrayList;
public class Crucipuzzle {
//rep:
    private char[][] celle;
    private ArrayList<String> parole;
    private boolean[] trovata;
    private ArrayList<Selezione> selezionate;
    public Crucipuzzle(char[][] schema, String[] paroleContenute) {
        selezionate = new ArrayList<Selezione>();
        ...inizializza celle con lo schema, e parole con paroleContenute....
    }

    public /*@ pure @*/ char[][] scacchiera() {
        .....
    }

    public boolean aggiungiSelezione(Selezione s) throws SelezioneErrataException {
        .....
    }
}
```

Si scriva l'invariante di rappresentazione e si implementino i metodi scacchiera() e aggiungiSelezione.

Soluzione

```
private invariant parole!=null && celle!=null && celle[0]!= null &&
//celle e' una matrice rettangolare:
(\forall int i; 0<i && i<celle.length; celle[i]!=null &&
            celle[i].length==celle[i-1].length) &&
//trovata e parole hanno la stessa lunghezza:
trovata.length==parole.length &&
//l'array selezionate puo' contenere solo stringhe dell'Ar.List parole:
(\forall Selezione s ;selezionate.contains(s);
    parole.contains(s.selezione())) &&
// le parole selezionate corrispondono a quelle trovate:
(\forall int i;0 <= i && i<trovata.length; trovata[i] <=> selezionate.contains(parole.get(i)))

public boolean aggiungiSelezione(Selezione s) throws SelezioneErrataException
    String word = s.selezione();
    int pos = parole.indexOf(word);
    if (pos == -1) throw new SelezioneErrataException();
    selezionate.add(s);
    trovata[pos]=true;
    for (int i=0; i<trovata.length; i++) {
        if (!trovata[i]) return false;
    }
    return true;
}

public char[][] scacchiera() {
    char[][] s = new char[celle.length][celle[0].length];
    for(int i=0; i<celle.length; i++) {
        for(int j=0; j<celle[0].length; j++) {
            s[i][j] = celle[i][j];
```

```
        }
    }
    return s;
}
```

Si osservi che scacchiera() non puo' restituire celle (ma nemmeno un suo clone), in quanto celle e' una parte *mutable* del rep.

Esercizio 2

Si consideri la seguente interfaccia che realizza il tipo `Mutex`. Un solo thread per volta può acquisire il `Mutex`. L'acquisizione è realizzata attraverso il metodo `lock` che sospende il chiamante se il `Mutex` è già acquisito (“locked”). Il metodo `unlock` rilascia il `Mutex` per altri thread.

```
public interface Mutex {  
    /** Acquisisce il mutex, sospendendo il chiamante in attesa che diventi disponibile */  
    public void lock();  
  
    /** Rilascia il mutex */  
    public void unlock();  
}
```

Domanda a

Si fornisca il codice della classe `MyMutex` che implementa l'interfaccia sopra riportata. Si usino solo i meccanismi base della sincronizzazione di Java senza sfruttare classi ed i pacchetti `java.util.concurrent`.

Soluzione

```
public class MyMutex implements Mutex {  
    private boolean locked;  
  
    public MyMutex() { locked = false; }  
  
    public synchronized void lock() {  
        while(locked) {  
            try { wait(); }  
            catch(InterruptedException ex) { ex.printStackTrace(); }  
        }  
        locked = true;  
    }  
  
    public synchronized void unlock() {  
        notifyAll();  
        locked = false;  
    }  
}
```

Domanda b

Si consideri la seguente classe che usa un `Mutex` per evitare conflitti nell'accesso da parte di thread diversi:

```
class Foo {  
    private double x, y;  
    private Mutex m;  
  
    public Foo(double x, double y) {  
        m = new MyMutex();  
        this.x = x; this.y = y;  
    }  
  
    public double getX() {  
        double temp;  
        m.lock();  
        temp = x;  
        m.unlock();  
        return temp;  
    }  
  
    public void setX(double x) {  
        m.lock();  
        this.x = x;  
        m.unlock();  
    }  
  
    public double getY() {  
        return y;  
    }  
}
```

La sincronizzazione è corretta? Perchè?

Soluzione

La soluzione è corretta. L'attributo `x` viene sia letto che scritto (dai metodi `getX` e `setX`, rispettivamente), deve quindi essere “protetto” usando il mutex (l’assegnamento di un `double` è di per sé una operazione la cui atomicità non è garantita in tutte le JVM). Viceversa, l’accesso all’attributo `y` è in sola lettura, senza rischio quindi di conflitti (è scritto solo dal costruttore).

Esercizio 3

Si considerino le seguenti dichiarazioni di classi Java.

```
public abstract class Shape {  
    public abstract String getColor();  
}  
  
public class Circle extends Shape {  
    public Circle(double d, String string, boolean b) {}  
    public String getArea() { return "getArea-Circle"; }  
    public String getColor() { return "getColor-Circle"; }  
}  
  
public class Rectangle extends Shape {  
    public Rectangle(double d, double e, String string, boolean b) {}  
    public String getColor() { return "getColor-Rectangle"; }  
    public String toString() { return "This is a shape with four sides!"; }  
}  
  
public class Square extends Rectangle {  
    public Square(double d) {  
        super(d, d, null, false);  
    }  
    public String getArea() { return "getArea-Square"; }  
    public String getSide() { return "getSide-Square"; }  
}
```

Si consideri poi il seguente codice Java che utilizza le classi sopra.

```
public class Main {  
    public static void main(String[] args) {  
  
1.        Shape s1 = new Circle(5.5, "RED", false);  
2.        System.out.println(s1);  
3.        System.out.println(s1.getArea());  
  
4.        Circle c1 = (Circle) s1;  
5.        System.out.println(c1);  
6.        System.out.println(c1.getArea());  
7.        System.out.println(c1.getColor());  
  
8.        Shape s2 = new Shape();  
9.        Shape s3 = new Rectangle(1.0, 2.0, "RED", false);  
10.       Rectangle r1 = (Rectangle) s3;  
11.       System.out.println(r1);  
12.       System.out.println(r1.getArea());  
  
13.       Shape s4 = new Square(6.6);  
14.       System.out.println(s4);  
15.       System.out.println(s4.getArea());  
16.       System.out.println(s4.getColor());  
  
17.       Rectangle r2 = (Rectangle)s4;  
18.       System.out.println(r2);  
19.       System.out.println(r2.getArea());  
20.       System.out.println(r2.getSide());  
    }  
}
```

Domanda a)

Si indichino dapprima quali righe del metodo `main` generano un errore di compilazione, e perchè.

Soluzione

Le righe 3, 12, 15, 19, e 20 generano un errore di compilazione perchè il metodo che viene invocato non è definito nel tipo statico della variabile. La riga 8 genera un ulteriore errore perchè non è possibile instanziare un oggetto da una classe astratta.

Domanda b)

Supponendo che tutte le righe che provocano errori in fase di compilazione siano state rimosse, illustrare l'output del programma. Se non fosse disponibile il codice del metodo eseguito a run-time, ci si può limitare a spiegare a quale classe appartiene il metodo che viene eseguito.

Soluzione

Un esempio di output è il seguente (fra parentesi il numero di riga):

```
(2) Circle@33909752
(5) Circle@33909752
(6) getArea-Circle
(7) getColor-Circle
(11) This is a shape with four sides!
(14) This is a shape with four sides!
(16) getColor-Rectangle
(18) This is a shape with four sides!
```

Notare come le prime due righe sono il risultato dell'esecuzione del metodo `toString` della classe `Object`.

Esercizio 4

Si consideri il progetto di una interfaccia grafica. I singoli elementi grafici sono denominati “widget” e rappresentano diverse modalità di interazione con l’interfaccia. I widget possono, ad esempio, essere bottoni, menu, e checkbox. Un widget può godere di diverse qualità, come ad esempio essere un widget utilizzabile per il risparmio energetico oppure un widget ad alta definizione. Ciascuna di queste qualità possiede essa stessa delle funzionalità, e può essere associata ad un widget in maniera dinamica (a run-time), se e quando opportuno. Ad esempio, la qualità “risparmio energetico” incorpora gli algoritmi per minimizzare il consumo di energia durante la visualizzazione.

Domanda a)

Disegnare un class diagram che rappresenti un possibile progetto per il sistema descritto sopra. Dire quali pattern è opportuno applicare e perché. Non è necessario dettagliare attributi e metodi di ciascuna classe, è sufficiente rappresentare le classi utilizzando nomi sufficientemente descrittivi e le loro relazioni.

Soluzione

La soluzione più naturale è applicare un pattern decorator, dove le classi che rappresentano le singole qualità sono i “decorator”.

Domanda b)

Tutti i widget istanziati durante una specifica esecuzione devono poter notificare le azioni svolte dall'utente su di essi (ad es., il click su un bottone) ad un insieme di classi fornite dall'utente che implementano la logica applicativa. Tale insieme di classi non è noto a priori e può cambiare dinamicamente.

Dire quali pattern è opportuno applicare in questo caso e perchè. Disegnare un sequence diagram che rappresenta una possibile interazione tra un widget e tale insieme di classi fornite dall'utente, secondo il/i pattern prescelto/i. E' sufficiente indicare classi e oggetti coinvolti, e dare un nome sufficientemente descrittivo ai messaggi che questi si scambiano.

Soluzione

La soluzione più naturale è applicare un pattern observer. Le classi fornite dall'utente si registrano ad un "manager" che viene informato dai widget dei loro cambi di stato e notifica le classi utente di conseguenza.

16 luglio 2019



Politecnico di Milano

Anno accademico 2018-2019

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Cugola <input type="checkbox"/> Margara <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica della classe `BinaryTree` che modella un albero binario di elementi (non nulli) di tipo `T`.

```
public class BinaryTree<T> {
    // Costruisce un albero binario con un solo elemento nella sua radice.
    public BinaryTree(T rootElem) ;

    // Ritorna l'elemento alla radice dell'albero
    public /*@ pure @*/ T getRoot() ;

    // Ritorna un Iterator che itera esattamente una volta su ogni elemento nell'albero.
    public /*@ pure @*/ Iterator<T> allElements() ;

    // Inserisce l'elemento elem come figlio sinistro del nodo che contiene l'elemento
    // father, purchè tale figlio sinistro non esista già, in tal caso solleva
    // l'eccezione NonEmptyChildException. Se father non è parte dell'albero,
    // solleva l'eccezione NoSuchElementException. Se l'elemento elem è già
    // parte dell'albero solleva l'eccezione ExistingElementException.
    public void insertLeft(T father, T elem) throws NonEmptyChildException,
        NoSuchElementException, ExistingElementException ;

    // Inserisce l'elemento elem come figlio destro del nodo che contiene l'elemento
    // father, purchè tale figlio destro non esista già, in tal caso solleva
    // l'eccezione NonEmptyChildException. Se father non è parte dell'albero,
    // solleva l'eccezione NoSuchElementException. Se l'elemento elem è già
    // parte dell'albero solleva l'eccezione ExistingElementException.
    public void insertRight(T father, T elem) throws NonEmptyChildException,
        NoSuchElementException, ExistingElementException ;

    // Ritorna true se l'elemento elem è parte dell'albero, falso altrimenti.
    public /*@ pure @*/ boolean contains(T elem) ;

    // Ritorna l'elemento a sinistra dell'elemento father (null se tale figlio
    // sinistro non esiste). Solleva l'eccezione NoSuchElementException se father
    // non è parte dell'albero.
    public /*@ pure @*/ T left(T father) throws NoSuchElementException ;

    // Ritorna l'elemento a destra dell'elemento father (null se tale figlio
    // destro non esiste). Solleva l'eccezione NoSuchElementException se father
    // non è parte dell'albero.
    public /*@ pure @*/ T right(T father) throws NoSuchElementException ;

    // Ritorna l'elemento padre dell'elemento elem; null se elem è la radice
    // dell'albero. Solleva l'eccezione NoSuchElementException se elem
    // non è parte dell'albero.
    public /*@ pure @*/ T father(T elem) throws NoSuchElementException ;
}
```

Domanda a)

Si specificino in JML i metodi `insertLeft` e `father`.

Soluzione

Definiamo:

```
unchanged() : getRoot().equals(\old(getRoot()) &&
    (\forallall T el; ; contains(el) <=> \old(contains(el))) &&
    (\forallall T el; contains(el) ;
        \old(left(el)==null) ? left(el)==null : left(el).equals(\old(left(el)))) &&
```

```

(\forall T el; contains(el) ;
  \old(right(el)==null) ? right(el)==null : right(el).equals(\old(right(el))) &&
(\forall T el; contains(el) ;
  getRoot().equals(el) ? father(el) == null : father(el).equals(\old(father(el)))))

Usiamo quindi unchanged per la nostra specifica:

//@requires father != null && elem != null
//@ensures \old(contains(father)) && \old(!contains(elem)) && \old(left(father)==null) &&
//@ left(father).equals(elem) && getRoot().equals(\old(getRoot())) &&
//@ (\forall T el; \old(contains(el)) ; contains(el)) &&
//@ (\forall T el; !el.equals(elem) && contains(el) ; \old(contains(el))) &&
//@ (\forall T el; !el.equals(father) && \old(contains(el)) ;
//@   \old(left(el)==null) ? left(el)==null : left(el).equals(\old(left(el))) &&
//@ (\forall T el; contains(el) ;
//@   \old(right(el)==null) ? right(el)==null : right(el).equals(\old(right(el))) &&
//@ (\forall T el; \old(contains(el)) ;
//@   getRoot().equals(el) ? father(el) == null : father(el).equals(\old(father(el))) )
//@ left(elem)==null && right(elem)==null
//@signals (NonEmptyChildException e) \old(left(father)!=null) && unchanged();
//@signals (NoSuchElementException e) \old(!contains(father)) && unchanged();
//@signals (ExistingElementException e) \old(contains(el) && unchanged();
public void insertLeft(T father, T elem) throws NonEmptyChildException,
NoSuchElementException, ExistingElementException ;

//@requires elem != null
//@ensures \old(contains(elem) && (elem.equals(getRoot()) => \result==null) &&
//@ (!elem.equals(getRoot()) => ( left(\result).equals(elem) || right(\result).equals(elem)))
//@signals (NoSuchElementException e) \old(!contains(elem))
public /*@ pure @*/ T father(T elem) throws NoSuchElementException ;

```

Domanda b)

Si consideri una versione `BinaryTreeBis` della classe `BinaryTree` la cui unica differenza è che l'iteratore restituisce gli elementi in ordine di inserzione. La classe `BinaryTreeBis` può essere definita come erede di `BinaryTree` rispettando il principio di sostituzione di Liskov?

Soluzione

Si, visto che l'iteratore di `BinaryTree` non precisa un ordine particolare.

Domanda c)

Si fornisca l'invariante pubblico della classe `BinaryTree`.

Soluzione

```

//@public invariant
//@ getRoot() != null && contains(getRoot()) && !contains(null) &&
//@ (\forall T el; contains(el) && left(el) != null; contains(left(el))) &&
//@ (\forall T el; contains(el) && right(el) != null; contains(right(el))) &&
//@ (\forall T el; contains(el);
//@   getRoot().equals(el) ? father(el) == null :
//@                         father(el) != null && contains(father(el))) &&
//@ (\forall T el; contains(el) && left(el) != null ; father(left(el)).equals(el)) &&
//@ (\forall T el; contains(el) && right(el) != null ; father(right(el)).equals(el)) &&
//@ (\forall T el; contains(el);
//@   !left(el).equals(getRoot()) && !right(el).equals(getRoot()) &&
//@ (\forall T el1; contains(el1); (\forall T el2; contains(el2) && !el1.equals(el2);
//@   !left(el1).equals(left(el2)) && !left(el1).equals(right(el2)) &&
//@   !right(el1).equals(left(el2)) && !right(el1).equals(right(el2))))

```

Esercizio 2

Si consideri la seguente classe `Numbers` che implementa una sequenza di numeri reali di dimensioni fissata al momento della costruzione. Il metodo `average` calcola la media dei due elementi in posizione `pos1` e `pos2` impostando a tale valore medio entrambi gli elementi.

```
public class Numbers {  
    private double data[];  
    private Object lock[];  
  
    public Numbers(int size) {  
        data = new double[size];  
        lock = new Object[size];  
        for (int i=0; i<size; i++) lock[i] = new Object();  
    }  
  
    public void set(int pos, double val) {  
        synchronized(lock[pos]) { data[pos] = val; }  
    }  
  
    public double get(int pos) {  
        synchronized(lock[pos]) { return data[pos]; }  
    }  
  
    public void average(int pos1, int pos2) {  
        synchronized(lock[pos1]) {  
            synchronized(lock[pos2]) {  
                double avg = (data[pos1] + data[pos2]) / 2.0;  
                data[pos1] = data[pos2] = avg;  
            }  
        }  
    }  
}
```

Domanda a)

La classe `Numbers` presenta un problema di sincronizzazione potendo dar luogo a deadlock. Si scriva una sequenza di chiamate da parte di due thread paralleli che può portare a un deadlock e si indichi come modificare il codice della classe `Numbers` per evitare il rischio di deadlock pur mantenendo una sincronizzazione che eviti conflitti nell'accesso ai dati.

Soluzione

Il deadlock deriva dalla doppia sincronizzazione del metodo `average`. È sufficiente che un thread invochi `average(i, j)` e l'altro `average(j, i)` per i medesimi valori di `i` e `j` perché il deadlock possa manifestarsi.

Una possibile soluzione consiste nell'eliminare la sincronizzazione con oggetti espliciti (attributo `lock`) sincronizzando tutti i metodi su `this`.

Domanda b)

Sulla base della versione corretta della classe `Numbers` elaborata al punto precedente, si aggiunga un metodo `waitForPositiveSum()` che sospende il chiamante fino a quando la somma di tutti gli elementi in `Numbers` è negativa. Si devono modificare altri metodi della classe `Numbers`?

Soluzione

Si riporta nel seguito una versione corretta della classe `Numbers` che elimina il rischio di deadlock e aggiunge il metodo `waitForPositiveSum()`. Si noti l'aggiunta dell'istruzione `notifyAll` al termine del metodo `set`.

```
public class NumbersOk {
    private double data[];

    public NumbersOk(int size) {
        data = new double[size];
    }

    public synchronized void set(int pos, double val) {
        data[pos] = val;
        notifyAll();
    }

    public synchronized double get(int pos) {
        return data[pos];
    }

    public synchronized void average(int pos1, int pos2) {
        double avg = (data[pos1] + data[pos2]) / 2.0;
        data[pos1] = data[pos2] = avg;
    }

    public synchronized void waitForPositiveSum() throws InterruptedException {
        double sum;
        do {
            sum = 0.0;
            for (int i=0; i<data.length; i++) sum += data[i];
            if (sum<0.0) wait();
        } while (sum>0.0);
    }
}
```

Esercizio 3

Un framework per lo sviluppo di applicazioni mobili integra un sistema di localizzazione `LocationService` che deve essere istanziato diversamente a seconda dell'hardware installato sullo specifico device. Al momento, esistono tre implementazioni concrete: `GPSLocationService` che sfrutta GPS, `WiFiLocationService` che sfrutta WiFi, e `DummyLocationService` utilizzato in ambiente di test (quando non sono presenti GPS e WiFi). Una classe `Device` ritorna informazioni sulla specifica dotazione hardware del dispositivo in uso. Le definizioni delle classi sono riportate di seguito.

```
public interface LocationService {
    public Location getLocation();
}

public class GPSLocationService implements LocationService {
    @Override
    Location getLocation() { ... }
}

public class WiFiLocationService implements LocationService {
    @Override
    Location getLocation() { ... }
}

public class DummyLocationService implements LocationService {
```

```

@Override
Location getLocation() { ... }

}

public class Device {
    public static boolean hasGPS() { ... }
    public static boolean hasWiFi() { ... }
}

```

Domanda a)

Mediante l'uso di un apposito design pattern, progettare una soluzione che nasconde agli utilizzatori la scelta della specifica implementazione di `LocationService` e semplifichi l'evoluzione del sistema nel caso nuove implementazioni di `LocationService` diventino disponibili in futuro.

Soluzione

Si può ricorrere a un factory method. La logica di creazione è in questo modo localizzata in un solo punto del codice e può essere modificata/estesa facilmente in futuro.

```

public class LocationServiceFactory {
    public static LocationService getLocationService() {
        if (Device.hasGPS()) return new GPSLocationService();
        else if (Device.hasWiFi()) return new WiFiLocationService();
        else return new DummyLocationService();
    }
}

// Utilizzo
LocationService locationService = LocationServiceFactory.getLocationService();

```

Domanda b)

Un'applicazione di chat sfrutta il `LocationService` per gestire richieste di invio posizione ai contatti. La concreta modalità di gestione delle richieste dipende dallo stato di privacy in cui si trova l'applicazione. Attualmente sono disponibili due stati di privacy, `High` e `Low`.

Mediante l'uso di un apposito design pattern, progettare una soluzione che gestisca gli stati di privacy e semplifichi l'evoluzione del sistema nel caso nuovi stati di privacy diventino disponibili in futuro.

Si completi il codice di `ChatApp` (anche aggiungendo opportuni attributi e metodi) e si fornisca la definizione di eventuali classi che si ritiene necessario aggiungere.

```

public class ChatApp {
    ...
    public void handleLocationRequest(LocationService service) { ... }
    ...
}

```

Soluzione

Si può ricorrere a uno state pattern. La logica di gestione delle richieste viene delegata a un oggetto `PrivacyState`. Future estensioni possono avvenire mediante l'implementazione di nuove sotto-classi di `PrivacyState`, senza modificare il codice di `ChatApp`.

```

public interface PrivacyState {
    public void handleRequest(LocationService service);
}

public class HighPrivacy implements PrivacyState {
    @Override

```

```

    public void handleRequest(LocationService service) { ... }

}

public class LowPrivacy implements PrivacyState {
    @Override
    public void handleRequest(LocationService service) { ... }
}

public class ChatApp {
    private PrivacyState privacyState;

    public ChatApp(PrivacyState privacyState) {
        this.privacyState = privacyState;
    }

    public void handleLocationRequest(LocationService service) {
        privacyState.handleRequest(service);
    }

    public void setPrivacyState(PrivacyState privacyState) {
        this.privacyState = privacyState;
    }
}

```

Esercizio 4

Si consideri il seguente metodo Java:

```

public static boolean m(Predicate<String> p, String s) {
    if (p == null) {
        p = x -> true;
    }
    if (s == null) {
        s = "";
    }
    if (s.length() < 2) return false;
    else if (p.test(s)) return true;
    else return false;
}

```

Domanda a)

Si fornisca un insieme di casi di test per la copertura delle istruzioni (statement coverage).

Soluzione

Un singolo caso permette di entrare nei tre rami `if` ed eseguire le istruzioni in essi contenute:

`p = null, s = null.`

Servono poi un caso di test per entrare nel ramo `else if` e un caso di test per entrare nel ramo `else`, rispettivamente:

`p = x -> true, s = "aaa"`

`p = x -> false, s = "aaa"`

Domanda b)

Si fornisca un insieme di casi di test per la copertura delle decisioni (edge coverage).

Soluzione

I casi precedenti sono sufficienti per coprire anche le decisioni.

Appello 09 luglio 2015



Politecnico di Milano
Anno accademico 2014-2015

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Baresi <input type="checkbox"/> Ghezzi <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri un’astrazione sui dati che definisce un conto corrente bancario (classe CCB), che fornisce le operazioni seguenti:

- creazione di un CCB, che crea un conto corrente con saldo 0;
- deposito nel CCB, che ha per parametro un valore float rappresentante il valore depositato;
- prelievo dal CCB, che ha per parametro un valore float che rappresenta il valore da prelevare. L’operazione può avvenire soltanto se il conto corrente non va in rosso;
- calcolo del saldo del CCB;
- calcolo dei movimenti effettuati: viene prodotto l’elenco di tutte le operazioni fatte sul CCB dalla sua creazione in ordine cronologico. Ogni movimento è un valore positivo se si è trattato di un deposito, negativo se di un prelievo.

Si consideri il seguente scheletro di specifica della classe CCB:

```
public class CCB {  
    public CCB(){...}  
    public List<Float> movimenti(){...}  
    public void deposito(float x){...}  
    public void prelievo (float x){...}  
    public float saldo(){...}  
}
```

Domanda a

Precisare a fianco di ciascun metodo della classe se esso è puro. Si fornisca la specifica JML in termini di pre- e post-condizioni delle operazioni CCB, prelievo e saldo, cercando se possibile di definire operazioni **totali**.

SOLUZIONE: Sono `/*@ pure */` i metodi `saldo()`, `movimenti()`. Non è invece necessario (pur se non errato) specificare come puro il costruttore `CCB` (in quanto non ha parametri e non serve richiamarlo nella specifica di altri metodi).

Per quanto non richiesto dal testo, è utile a fini didattici mostrare i contratti anche dei metodi `movimenti()` e `deposito`. Il metodo `movimenti()` è l’unico metodo puro “primitivo”, ossia che deve essere specificato con un commento. Ri fatto restituisce tutte le informazioni necessarie per specificare gli altri metodi.

```
/*@ensures \result \neq null && (*\result e' una lista di  
/*@ tutte le operazioni di deposito o prelievo su this in ordine cronologico*);  
public /*@pure*/ List<Float> movimenti()  
  
/*@ensures movimenti().size()==0;  
public CCB()  
  
/*@ensures \result ==  
/*@ (\sum int i; 0 <=i && i < movimenti().size(); movimenti().get(i));  
public /*@pure*/ float saldo()
```

NB: L’operatore `\sum` restituisce 0 quando il suo range è vuoto. Quindi quando `movimenti().size() == 0` necessariamente è `saldo() == 0`. Non è comunque errato (ma soltanto inutile) aggiungere alla postcondizione di `CCB` la condizione `saldo() == 0`.

```
/*@ensures x>0 && \old(saldo())>=x) &&  
/*@ movimenti().size() == 1+\old(movimenti().size()) &&
```

```

//@  movimenti().get(movimenti().size()-1) == -x &&
//@  movimenti().sublist(0,movimenti().size()-2).equals(\old(movimenti()));
//@signals (NegativeException e) (x<0 || \old(saldo())<x)) &&
//@  movimenti().equals(\old(movimenti()));
public void prelievo (float x) throws NegativeException;

//@ensures x>0 &&
//@  movimenti().size() == 1+\old(movimenti().size()) &&
//@  movimenti().get(movimenti().size()-1) == x &&
//@  movimenti().sublist(0,movimenti().size()-2).equals(\old(movimenti()));
//@signals (NegativeException e) x<0 && movimenti().equals(\old(movimenti()));
public void deposito (float x) throws NegativeException;

```

Domanda b

Specificare l'invariante astratto che il saldo è sempre non negativo. Mostrare che questa proprietà è effettivamente garantita dalla specifica fornita delle operazioni.

SOLUZIONE:

```
public invariant saldo()>=0;
```

Il costruttore verifica l'invariante in quanto definisce un CCB con `saldo=0`. Mostriamo che se l'invariante vale al momento della chiamata dei metodi non puri di CCB allora vale anche all'uscita. Ci sono solo due metodi non puri: deposito: può solo aumentare il saldo (a patto che sia stato specificato correttamente il lancio di un'eccezione se l'argomento è negativo); prelievo: la sua post-condizione prevede `\old(saldo())>=x`, pertanto `\old(saldo()) + movimenti().get(size()-1) >=0`. Il nuovo saldo() al termine sarà uguale alla somma di tutti i nuovi movimenti, ossia la somma di quelli vecchi (uguale a `\old(saldo())`) più il nuovo movimento uguale a `-x`, ossia proprio:

```
\old(saldo())+ movimenti().get(size()-1) == saldo() con x>=0.
```

Domanda c

Si consideri una rep costituita da

1. un `ArrayList<Float>` `lista` in cui sono memorizzate, in ordine cronologico, tutte le operazioni di prelievo o deposito.
 2. un `ArrayList<Long>` `tempo`, nella cui posizione `i`-esima è memorizzato (come un numero `Long` positivo) l'istante di tempo in cui è stata effettuata l'operazione `i`-esima dalla creazione.
- Si scriva l'invariante di rappresentazione della classe e la funzione di astrazione (scegliendo liberamente per quest'ultima la notazione da usare).

SOLUZIONE:

RI:

```

private invariant lista!= null && tempo!=null &&
    !lista.contains(null) && !tempo.contains(null) &&
    lista.size()== tempo.size() &&
    (\forallall int i; 0 <=i && i<tempo.size()-1;
        0<tempo(i) && tempo.get(i) < tempo. get(i+1));

```

AF:

```
private invariant movimenti().equals(lista);
```

Una soluzione equivalente consiste nel reimplementare il metodo `toString()`

- Si mostri un'implementazione scorretta del metodo `movimenti` che esponga la `rep` e si spieghi perché ciò sia assolutamente da evitare.

SOLUZIONE:

```
return lista;
```

Il metodo espone la parte mutabile della `rep`, consentendo al codice cliente di modificare direttamente lo stato concreto di un oggetto, aggirando information hiding, e causando potenziali violazioni del RI tramite effetti collaterali.

Esercizio 2

Si consideri questo frammento di codice Java:

```
class Cane{  
    public void abbaia(){  
        System.out.println("bau");  
    }  
  
    public void abbaia(int num){  
        for(int i=0; i<num; i++)  
            System.out.println("bau");  
    }  
}  
  
class CaneDaCaccia extends Cane{  
    public void fiuta(){  
        System.out.println("ffff");  
    }  
  
    public void abbaia(){  
        System.out.println("woo");  
    }  
}  
  
-----  
  
Cane fido = new Cane();  
CaneDaCaccia pluto = new CaneDaCaccia();  
Cane febo = new CaneDaCaccia();  
CaneDaCaccia pippo = new Cane();  
  
fido.abbaia();  
pluto.abbaia();  
febo.abbaia();  
pippo.abbaia();  
fido.abbaia(3);  
pluto.abbaia(3);  
febo.abbaia(3);  
pippo.abbaia(3);
```

- Marcare con E le istruzioni dopo ---- che generano un errore in compilazione.
- A fianco di ciascuna chiamata di metodo che non generi errore, si scriva quale sarebbe il valore stampato in uscita.

SOLUZIONE:

```
CaneDaCaccia pippo = new Cane(); //ERRATA  
  
fido.abbaia(); //bau  
pluto.abbaia(); //woo  
febo.abbaia(); //woo  
pippo.abbaia(); //pippo non può essere inizializzato  
fido.abbaia(3); //bau bau bau  
pluto.abbaia(3); //bau bau bau  
febo.abbaia(3); //bau bau bau  
pippo.abbaia(3); //pippo non può essere inizializzato
```

Esercizio 3

Si consideri la seguente specifica di un metodo statico:

```
//@ requires true;
//@ ensures \result <==> (\exists int i; 1<=i && i <elements.size();
//@   elements.get(i).equals(elements.get(i-1)) ||
//@   (elements.get(i).equals(elements.get(i-2)) && i>1))
//@ signals (BadArgumentException e) elements==null ||
//@   (\exists int i; 0<=i && i < elements.size(); elements.get(i)== 0)
public static /*@pure */ boolean twoNext(ArrayList<Integer> elements)
throws BadArgumentException
```

Definire un insieme di casi di test secondo una strategia *black-box* e fornire un sintetico commento che giustifichi perché si ritiene significativo questo insieme.

Nel testing black box, la specifica fornisce uno strumento che consente di partizionare il dominio di input. Una volta partizionato il dominio è necessario:

- testare ogni categoria (Metodo delle combinazioni);
- testare i valori ai confini delle categorie (Metodo dei casi limite).

È necessario come prima cosa andare ad identificare le categorie di interesse utilizzando la specifica. Iniziamo con l'analisi della postcondizione normale:

```
//@ (\exists int i; 1<=i && i <elements.size();
//@   elements.get(i).equals(elements.get(i-1)) ||
//@   (elements.get(i).equals(elements.get(i-2)) && i>1))
```

è vera in due condizioni:

- $\exists i : 1 \leq i & i < \text{elements.size}(); \text{elements.get}(i) == \text{elements.get}(i-1)$
esiste un elemento uguale al suo predecessore (condizione A).
- $\exists i : 1 \leq i & i < \text{elements.size}(); (\text{elements.get}(i) == \text{elements.get}(i-2)) \& i > 1$
esiste un elemento uguale al predecessore del suo predecessore (condizione B).

Possiamo a questo punto analizzare la post-condizione eccezionale anch'essa fatta da due clausole:

- $\text{elements} == \text{null}$
elements è nullo. (condizione D)
- $(\exists i : 0 \leq i & i < \text{elements.size}(); \text{elements.get}(i) == 0)$
elements contiene un valore zero (condizione C)

Partiamo con il *test di categoria*. Il primo test associa a elements il valore null rende soddisfatta la condizione D.

Si noti che la Condizione D e' mutuamente esclusiva con A, B e C. C invece non è mutuamente esclusiva con A e B e pertanto occorre considerare tutte le possibilità.

A questo punto, date le condizioni A, B, e C è possibile identificare otto sottocategorie a seconda che A, B e C siano soddisfatte. Partendo da queste sottocategorie il seguente *test di categoria* può essere delineato:

A	B	C	Test
Vera	Vera	Vera	[1, 2, 3, 4, 1, 1, 1, 0, 4]
Vera	Vera	Falsa	[1, 2, 3, 4, 1, 1, 1, 6, 4]
Vera	Falsa	Vera	[1, 2, 3, 4, 1, 1, 9, 0, 4]
Vera	Falsa	Falsa	[1, 2, 3, 4, 1, 1, 9, 6, 4]
Falsa	Vera	Vera	[1, 2, 3, 4, 1, 0, 1, 6, 4]
Falsa	Vera	Falsa	[1, 2, 3, 4, 1, 10, 1, 6, 4]
Falsa	Falsa	Vera	[1, 2, 3, 4, 1, 10, 14, 6, 4]
Falsa	Falsa	Falsa	[1, 2, 3, 4, 1, 10, 16, 6, 4]

Focalizziamoci sui *test al confine*. Occorre testare il metodo almeno per i seguenti casi:

- array vuoto [];
- array con un elemento [1];

Inoltre sarebbe utile testare casi in cui le condizioni che rendono vera/falsa la specifica sono posizionate “agli estremi dell’ArrayList, per esempio [1, 1, 10, 20, 8, 9] e [6, 7, 8, 9, 8, 1 1] sono test di confine per la condizione A, [1, 6, 1, 20, 8, 9] e [2, 6, 4, 1, 8, 1] sono test di confine per la condizione B e [0], [0,2], [1,2,0] sono test di confine per la condizione C.

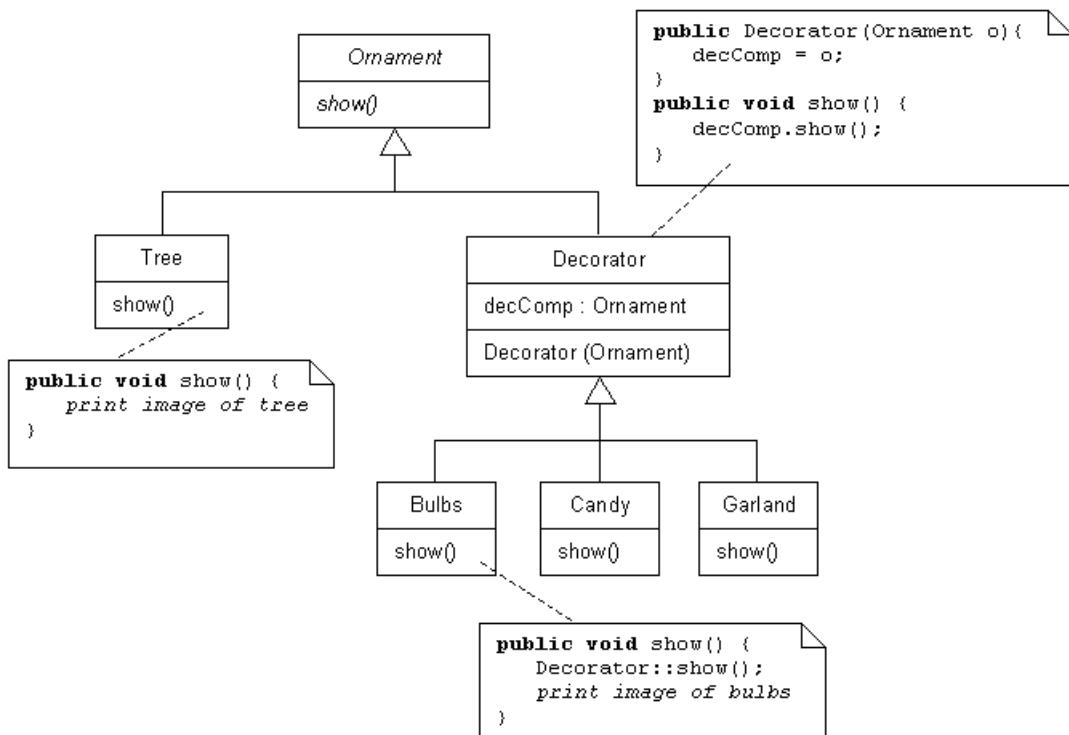
Esercizio 4

Si usi il design pattern *Decorator* per modellare un *AlberoDiNatale*, come componente di base, e le sue solite decorazioni (*Decorazione*) —ad esempio, *Luci*, *Ghirlanda* e *Dolcetto*. Si assuma che ogni tipo di decorazione è aggiunto una sola volta. *AlberoDiNatale* ha un metodo *show()* che disegna l’albero sullo schermo. Ogni decorazione ha pure un metodo *show()* che disegna la decorazione stessa. Attraverso il pattern *Decorator*, l’albero completo viene disegnato attraverso l’invocazione del metodo *show()* della classe *AlberoDiNatale* opportunamente gestito attraverso la gerarchia di classi definita.

Si realizzi il diagramma UML delle classi per la soluzione proposta e lo si esemplifichi con un paio di configurazioni significative. Si mostri in particolare che cosa accade invocando il metodo *show()* su un oggetto di tipo *AlberoDiNatale* che è stato decorato, nell’ordine, con i decoratori *Luci*, *Ghirlanda* e *Dolcetto*; si ipotizzi per semplicità che ciascun metodo *show()* stampi semplicemente il nome della classe in cui è definito.

SOLUZIONE

Il pattern decorator permette di incapsulare un oggetto all’interno di un altro oggetto che ne “decora” il comportamento. Il decorator ha la stessa interfaccia del componente che decora per cui la sua presenza è trasparente a un client. Il decorator chiama i metodi dell’oggetto che incapsula, ma li può appunto decorare eseguendo operazioni addizionali. Il fatto che il decoratore sia trasparente permette di includere il decoratore in un altro decoratore ricorsivamente (una specie di matroska di decoratori) in cui ogni decoratore decora le funzionalità di quello che contiene Nel caso in questione, il componente base è di tipo *AlberoDiNatale*,



sottoclasse della classe astratta *Ornamento*. Questa classe, possiede un metodo *show()*, che stampa le caratteristiche dell’*Ornamento*. È possibile decorare il metodo *show* dell’albero mediante i vari ornamenti che possono essere aggiunti via via all’albero. Nel caso specifico le decorazioni, sottoclassi della classe astratta *Decorazione* (a sua volta sottoclasse di ornamento), includono *Luci*, *Ghirlanda* e *Dolcetto*. Ogni decorazione ha un reference all’*Ornamento* decorato. Inoltre, le decorazioni possono contenere attributi addizionali per esempio le *Luci* potrebbero avere un colore, i dolcetti potrebbero avere un gusto etc. Quando viene invocato il metodo *show* su una decorazione, il metodo *show* del componente contenuto viene chiamato, e il risultato viene decorato con il nuovo componente. Per esempio, nel caso delle *Luci* viene

effettuata una stampa che specifica che l'albero è decorato per mezzo di luci di un determinato colore, mentre nel caso dei Dolcetti la decorazione e' composta da Dolcetti di un dato sapore.

Due oggetti o1 e o2 di tipo Ornamento, di cui il secondo specificato come richiesto (un AlberoDiNatale decorato con Luci, Ghirlanda e Dolcetto), possono essere definiti come segue:

```
Ornamento o1=new (Ghirlanda(new Luci(Color.RED, new Dolcetto("Menta",
    AlberoDiNatale())));
Ornamento o2=new Dolcetto("Limone", new Ghirlanda(new Luci(Color.BLUE, new
    AlberoDiNatale())));
o2.show();
```

Quando viene chiamato il metodo show dell'ornamento ogni componente richiama il metodo show dell'elemento contenuto e lo decora mediante il suo risultato. Nel caso in questione un possibile output dell'invocazione del metodo show sull'oggetto o2 è

```
AlberoDiNatale\\
Luci Blu\\
Ghirlanda\\
Dolcetto al Limone\\
```



Politecnico di Milano

Anno accademico 2012-2013

Ingegneria del Software – Appello del 12 febbraio 2014

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi

Ghezzi

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si scrivano le pre- e post-condizioni in JML per i seguenti metodi statici:

- `public static boolean diagonali(int [][] a)` che riceve come parametro una matrice quadrata di interi e restituisce `true` se le somme dei valori sulle due diagonali sono uguali, `false` altrimenti.
- `public static boolean diagonali(ArrayList<Integer> a, int p)` che riceve come parametri un `arrayList` di interi e una posizione `p`. Il metodo restituisce `true` se `p` non è maggiore della lunghezza di `a` e se tutti gli elementi in posizione minore di `p` sono minori di tutti quelli in posizione maggiore di `p`, mentre restituisce `false` altrimenti.
- `public static boolean diagonali(int [][] a, int p)` che riceve come parametri una matrice quadrata di interi e un intero `p`. Se `p` è minore della dimensione della matrice, la coppia di indici `(p, p)` identifica un elemento `e` sulla diagonale della matrice e divide la diagonale medesima in due parti d_1 e d_2 (`e` escluso). Il metodo restituisce `true` se, e solo se, la somma degli elementi della sotto-matrice la cui diagonale è d_1 è uguale alla somma degli elementi della sotto-matrice la cui diagonale è d_2 .

Esercizio 2

Si considerino le seguenti classi Java:

```
public class ClassA {  
    public void stampa(ClassA p) {  
        System.out.println("AAA");  
    }  
}  
  
public class ClassB extends ClassA {  
    public void stampa(ClassB p) {  
        System.out.println("BBB");  
    }  
  
    public void stampa(ClassA p) {  
        System.out.println("AAA/BBB");  
    }  
}  
  
public class ClassC extends ClassA {  
    public void stampa(ClassC p) {  
        System.out.println("CCC");  
    }  
  
    public void stampa(ClassA p) {  
        System.out.println("AAA/CCC");  
    }  
}
```

- Si spieghi cosa stamperebbe il seguente metodo `main`, motivando brevemente le risposte:

```
public static void main(String[] args) {  
    ClassA a1, a2;  
    ClassB b1;  
    ClassC c1;  
  
    a1 = new ClassB();  
    b1 = new ClassB();  
    c1 = new ClassC();  
    a2 = new ClassC();  
    b1.stampa(b1);  
    a1.stampa(b1);  
    b1.stampa(c1);  
    c1.stampa(c1);  
    c1.stampa(a1);  
    a2.stampa(c1);  
}
```

- Cosa succederebbe se `ClassC` ereditasse da `ClassB` e non da `ClassA`?

Esercizio 3

Si consideri il seguente metodo Java:

```
public static int test(int x, int y) {  
    int z = 7;  
  
    while (z > 0) {  
        if (z - x < 0 && y != 1) z = z % (y-1);  
        else return z;  
        z++;  
    }  
    return z;  
}
```

- Si disegni il diagramma del flusso di controllo;
- Si spieghi il comportamento del metodo per $x = 8$ e $y = -2$;
- Con il caso di test precedente vengono coperte tutte le decisioni? Giustificare la risposta, mostrando, in caso negativo, quali decisioni non sono coperte.

Esercizio 4

Un ristorante deve gestire i clienti, i tavoli, con il relativo numero di posti e la suddivisione in fumatori e non fumatori, e le prenotazioni, effettuate dai clienti per un certo giorno ed ora e un determinato numero di persone. Ad ogni prenotazione viene assegnato uno o più tavoli, nella zona fumatori o non fumatori in funzione della richiesta fatta. Il ristorante vuole anche tener traccia dei camerieri che servono i diversi tavoli, delle pietanze e bevande ordinate e del relativo conto finale (per cliente o per tavolo).

Dei clienti interessa il nome e numero di telefono, mentre dei camerieri interessa il nome e gli anni di servizio. Per quanto riguarda piatti e bevande, interessa il nome, il prezzo unitario ed eventuali richieste particolari. L'ammontare finale del conto è dato dalla somma dei prezzi dei singoli piatti e delle bevande, tenendo conto di eventuali richieste particolari, moltiplicati per il numero di pezzi serviti.

Si progetti un diagramma delle classi “dettagliato” per la gestione del ristorante e un diagramma di sequenza per la prenotazione di un tavolo per quattro persone. Il diagramma di sequenza deve usare un *frame* per rappresentare i due casi in cui il tavolo richiesto sia o non sia disponibile.

Appello 11 febbraio 2015



Politecnico di Milano
Anno accademico 2013-2014

Ingegneria del Software

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi

Ghezzi

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Esercizio 1

Si consideri la classe `Sudoku`, che memorizza tutte le informazioni relative ad una partita al gioco omonimo. La classe fornisce i metodi: `inizializza` per l'inizializzazione della griglia per una nuova partita, `riga` e `colonna`, che restituiscono un `ArrayList<Integer>` di nove elementi con i valori contenuti nella riga (o colonna) passata come parametro, `area`, che restituisce un altro `ArrayList<Integer>` di nove elementi con i valori della sotto-matrice da 3x3 elementi in cui viene divisa la scacchiera del Sudoku (le nove aree sono numerate da 0 a 8 muovendosi riga per riga da sinistra a destra: la prima in alto a sinistra è l'area 0 e l'ultima in basso a destra è l'area 8. Per semplicità si assuma che il valore restituito per le celle vuote sia 0. La classe offre anche un metodo `muovi` che prende tre parametri: `riga`, `colonna` e `valore` per aggiungere `valore` alla posizione specificata da `riga` e `colonna`. Si fornisca:

- la definizione in JML dell'invariante pubblico della classe `Sudoku`.
- la specifica completa del metodo `muovi`, considerando che non è possibile aggiungere un elemento in una posizione già occupata e che la griglia deve sempre rispettare le regole del gioco.

Si ricorda che il Sudoku impone che ogni riga e ogni colonna debbano contenere i numeri da 1 a 9 senza ripetizioni, o un loro sotto-insieme quando la griglia è incompleta. La stessa regola —ovvero numeri da 1 a 9 senza ripetizioni— deve valere anche per ognuna delle nove aree 3x3 all'interno della griglia.

Esercizio 2

Si consideri la classe `Counter` che fornisce questa interfaccia, descritta informalmente:

```
public class Counter
    public Counter(); //Inizializza this a zero
    public int get(); //Restituisce il valore di this
    public void incr(); //Incrementa di 1 il valore di this
}
```

Domanda 1

1. L'operazione `get` è un osservatore puro. Che cosa significa esattamente?
2. Si fornisca una formalizzazione del costruttore e dell'operazione `incr` in termini dell'osservatore puro.
3. Quale invariante astratto è possibile definire per la classe `Counter`?

Domanda 2

Si consideri una sottoclasse di Counter, Counter2, che ridefinisce l'operazione incr in modo che il contatore venga incrementato di 2. La classe Counter2 può essere considerata un sottotipo di Counter? Perché? Perché no?

Domanda 3

Si consideri ora la seguente sottoclasse CounterN della classe Counter:

```
public class CounterN extends Counter
    public CounterN(int n); //Inizializza this a un valore qualunque n
    public int get(); //Restituisce il valore di this
    public void incr(); //Incrementa di 1 il valore di this
}
```

La classe CounterN può essere considerata un sottotipo di Counter? Perché? Perché no?

Domanda 4

Si consideri invece la seguente sottoclass CounterN1 della classe Counter:

```
public class CounterN1 extends Counter
    public CounterN1(); //Inizializza this a zero
    public int get(); //Restituisce il valore di this
    public void incr(int n); //Se n>0 incrementa di n il valore di this
                            //altrimenti lancia eccezione IllegalArgument
}
```

1. L'operazione `incr` può essere una ridefinizione dell'operazione `incr` definita per `Counter`?
2. L'introduzione dell'operazione `incr` in `CounterN1` consente di vedere `CounterN1` come sottotipo di `Counter`? Perché? Perché no?

Esercizio 3

Si vuole realizzare un programma che consente di inserire una serie di valori numerici e di visualizzarli in due modalità: una tabella di valori numerici e un diagramma a barre in cui i valori sono rappresentati da barre proporzionali al valore. Si vuole consentire di modificare i singoli valori numerici in tabella ed effettuare automaticamente la modifica conseguente del diagramma a barre.

Si fornisca un diagramma delle classi UML per rappresentare un programma che realizza questo problema, facendo riferimento a opportuni design pattern che risultino convenienti da adottare.

Esercizio 4

Si supponga di avere a disposizione la classe `Contatti` che contiene i contatti memorizzati dall'utente di un telefono cellulare. La classe `Contatti` offre gli osservatori `getNome`, `getCognome`, e `getNumeroTel`. Si fornisca:

- Un rep per la classe `Contatti`.
- L'implementazione di un Iteratore per scandire i contatti in ordine di memorizzazione.
- L'implementazione di un metodo `boolean sequenzaOrdinata(Iterator<Integer> i)` che, dato l'iteratore al punto precedente, verifica se la sequenza associata all'iteratore è ordinata, cioè gli elementi estratti dall'iteratore appaiono in ordine alfabetico per cognome. La soluzione deve essere **ricorsiva**, senza fare uso di alcuna istruzione di ciclo.

Esercizio 5

Si consideri il seguente frammento di programma:

```
1. while (numProdotti) <= PROD_MAX {  
2.     ----  
3.     if (costoProdotto) <= restaDaSpendere) {  
3.         ----  
4.         if (restaDaSpendere > 0) {  
5.             numProdotti++;  
6.         } else {  
7.             ----  
8.             break;----//qui codice che non cambia n e j  
9.         } else {  
10.            ----  
11.        }  
12.    };
```

1. Si definiscano casi di test che coprano tutti branch (si considerino le parti tralasciate ---- come se fossero una singola macro-istruzione);
2. Se il criterio di copertura fosse quello di copertura delle istruzioni si otterebbe lo stesso risultato? Perché sì? Perché no?.
3. Si calcoli la condizione logica che impone che il ciclo venga eseguito due volte e la seconda volta si esca con l'istruzione break.

Appello 1 Luglio 2016



Politecnico di Milano
Anno accademico 2014-2015

Ingegneria del Software

Cognome:

LAUREANDO

Nome:

Matricola:

Sezione (segnarne una): Ghezzi

Mottola

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la classe `QuasiRetta` che contiene un insieme di punti che approssimano una retta.

Il costruttore della classe riceve come parametri due punti (definiti dalla classe `Punto`) e un numero reale `approx`. Si assuma già definita la classe `Punto`, immutabile e con un metodo `boolean equals(Punto p)` definito opportunamente. Questi due punti definiscono univocamente una retta. Il significato di `approx` viene chiarito nel seguito.

Dopo la creazione, ulteriori punti possono essere aggiunti mediante il metodo pubblico `void aggiungi(Punto p)` fornito dalla classe. Questo metodo aggiunge alla retta il punto `p` purché questo disti dalla retta non più di `approx`. La distanza di un punto dalla retta viene calcolata da un metodo pubblico `float dist(Punto p)` che calcola la distanza del punto `p` dalla retta. Esistono anche un metodo `void elimina(Punto p)` che elimina il punto passato come parametro, se questo esiste in `QuasiRetta` e un metodo `boolean appartiene(Punto p)` che verifica se il parametro appartiene a `QuasiRetta`. Si suppone che la classe esporti un attributo pubblico e finale `APPROX` il cui valore è l'approssimazione che viene passata alla classe con il costruttore. Da ultimo, il metodo `int totBuoni()` restituisce il numero di punti di `QuasiRetta` che possono essere considerati appartenenti alla retta, in quanto la loro distanza è inferiore al valore di `APPROX` diviso per 1000.

Si chiede di specificare formalmente in termini di pre e post condizioni JML, definendo eventualmente altri metodi pubblici puri che ritenete utili, le seguenti operazioni:

1. il costruttore `QuasiRetta(Punto p1, Punto p2, float approx);`
2. il metodo `void aggiungi(Punto p) throws IllegalPointException`, dove l'eccezione viene lanciata se la distanza del punto dalla retta eccede l'approssimazione richiesta;
3. il metodo `void elimina(Punto p);`
4. il metodo `int totBuoni();`.

Si supponga di voler definire anche un metodo `ArrayList<Punto> listaPunti()` che restituisce tutti i punti di una `QuasiRetta` in un `ArrayList`. Pur trattandosi di un metodo osservatore puro, la sua implementazione potrebbe essere sbagliata, nel senso di consentire l'accesso diretto alla rappresentazione interna. Si descriva una implementazione che ha questo inconveniente.

SOLUZIONE:

```
//@requires p1!= null && p2!=null && approx>=0 && !p1.equals(p2);
//@ensures APPROX=approx && appartiene(p1) && appartiene(p2) &&
//@    dist(p1)==0 && dist(p2)==0 &&
//@    (\forallall Punto p; appartiene(p); p1.equals(p) || p2.equals(p));
public QuasiRetta(Punto p1, Punto p2, float approx)
```

Seguono gli altri metodi:

```
//@requires p!= null
//@ensures dist(p)<APPROX && appartiene(p) &&
//@    (\forallall Punto p1; !p.equals(p1); \old(appartiene(p1)) <==> appartiene(p1));
//@signals(IllegalPointException e) dist(p)>=APPROX/1000 &&
//@    (\forallall Punto p1;; \old(appartiene(p1)) <==> appartiene(p1));
public void aggiungi(Punto p) throws IllegalPointException

//@requires p!= null &&
//@    (\num_of Punto p1; appartiene(p1); !p.equals(p1)) >=2;
//@ensures !appartiene(p) &&
//@    (\forallall Punto p1; !p.equals(p1); \old(appartiene(p1)) <==> appartiene(p1));
public void elimina(Punto p)

//@ensures \result == (\num_of Punto p; appartiene(p); dist(p)<APPROX/1000);
public /*@ pure @*/ int totBuoni()
```

Per l'esportazione della rep, si puo' ipotizzare che `QuasiRetta` sia memorizzata come un `arrayList` di punti e che il metodo `listaPunti` esporti il riferimento all'arraylist.

Esercizio 2

Si consideri una classe Roditore che ha due eredi Ratto e Topo. La classe Topo ha come erede la classe Topolino. Si considerino i casi seguenti:

```
Roditore rod;
Ratto rat = new Ratto();
Topo jerry = new Topo();
Topolino mickey = new Topolino();
```

Segnare con una X gli assegnamenti seguenti che generano un errore di compilazione, e spiegare il perche':

- rod = rat;
- rod = jerry;
- mickey = null;
- mickey =rat;

Soluzione: mickey =rat e' scorretta

Quali delle seguenti dichiarazioni di array sono corrette per un array che ci si attende che possa contenere fino a 10 oggetti dei tipi Ratto, Topo e Topolino? Motivare le risposte in caso positivo e negativo.

1. Ratto[] array = new Ratto[10];
2. Roditore[] array = new Ratto[10];
3. Roditore[] array = new Roditore[10];
4. Roditore[10] array;

Soluzione: Ipotizziamo, a titolo di esempio, che nel codice ci siano i seguenti assegnamenti che inseriscono in array oggetti dei tipi richiesti:

```
array[0]= new Topolino();
array[1] = new Ratto();
array[2]= new Topo();
```

Caso (1) KO: Ratto non e' compatibile con Topo e Topolino: si avra' errore di compilazione per l'istruzione array[0]= new Topolino().

Caso (2) KO: Il tipo dinamico dell'array diventa Ratto[]. Quindi, durante la compilazione non si avranno errori, ma eseguendo array[0]= new Topolino() si ottiene un errore a runtime (ArrayStoreException), in quanto la chiamata del costruttore new Ratto[10] segnala che l'array ha tido dinamico Ratto, che non e' compatibile con Topo e Topolino. Si noti, infatti, che gli array, a differenza delle collezioni, sono covarianti.

Caso (3) OK, si costruisce un array di tipo Roditore, sovratipo di Ratto, Topo e Topolino.

Caso (4): Sintatticamente scorretto.

Esercizio 3

Si consideri il seguente programma Java

```
public class Esame implements Runnable
{
    private transient int x;
    private volatile int y;

    public static void main(String [] args)
    {
        Esame that = new Esame();
        (new Thread(that)).start(); /* Linea 8 */
        (new Thread(that)).start(); /* Linea 9 */
    }
    public synchronized void run() /* Linea 11 */
    {
        for (;;) /* Line 13 */
        {
            x++;
            y++;
            System.out.println("x = " + x + " , y = " + y);
        }
    }
}
```

Quali delle seguenti risposte sono corrette? Si fornisca anche una breve spiegazione per la risposta fornita.

1. Si genera un errore di compilazione in corrispondenza di Linea 11;
2. Si genera un errore di compilazione a causa di errori alle Linee 8 e 9;
3. Il programma stampa valori per x e y che potrebbero non essere sempre uguali sulla stessa riga di stampa (per esempio, può stampare $x = 1, y = 2$);
4. Il programma stampa valori per x e y che sono sempre gli stessi sulla stessa riga di stampa (per esempio, stampa $x = 1, y = 1$ su una riga e $x = 2, y = 2$ sulla successiva);
5. Il programma stampa i valori di x in ordine crescente, ovvero
 $x = 1, \dots$
 $x = 2, \dots$
 $x = 3, \dots$

Soluzione: Gli unici casi veri sono (4) e (5). Infatti, il primo thread che entra nel metodo synchronized va in loop infinito e blocca l'accesso all'altro thread: il programma si comporta come se fosse un normale programma sequenziale che non termina. NB: Le dichiarazioni transient e volatile in questo caso non hanno alcun effetto; le variabili d'istanza x e y sono inizializzate a 0 per default.

Esercizio 4

Si consideri il seguente frammento di programma (a, b e c sono variabili booleane):

```
if ((a || b) && c)
{
    << Statements >>
}
else
{
    << Statements >>
}
```

Si definisca il minimo insieme di casi di test per ciascuno dei seguenti casi:

1. vengono coperte tutte le istruzioni;
2. vengono coperti tutti i branch;
3. vengono coperti tutti i branch e tutte le condizioni.

Giustificate le vostre risposte.

1) Bastano due casi:

```
A = true | B = not eval | C = false
A = true | B = not eval | C = true
```

dove “not eval” significa che la condizione non viene valutata e quindi il valore e’ irrilevante.

2) Stessi caso del punto precedente, in quanto vi sono statements per ogni branch.

3) Servono tre casi, a causa della valutazione cortocircuitata di Java.

```
A = true | B = not eval | C = false
A = false | B = true | C = true
A = false | B = false | C = not eval
```

Esercizio 5

Si scriva il risultato dell'esecuzione del seguente frammento di programma:

```
List<Integer> cinqueInteri = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> x = cinqueInteri.stream()
    .map((n) -> {
        return n * 2;
    })
    .collect(Collectors.toList());

System.out.println("x = " + x);

List<Integer> y = cinqueInteri.stream()
    .filter((n) -> n % 2 == 0)
    .collect(Collectors.toList());

System.out.println("y = " + y);

Optional<Integer> z = cinqueInteri.stream()
    .reduce((n, a) -> n + a);

System.out.println("z = " + z.orElse(42));

Optional<Integer> w = new ArrayList<Integer>().stream()
    .reduce((n, a) -> n + a);
System.out.println("w = " + w.orElse(42));
```

Soluzione:

```
x = [2, 4, 6, 8, 10]
y = [2, 4]
z = 15 (ossia calcola la somma)
w = 42
```

NB: Il programma ha bisogno ovviamente che siano presenti le seguenti dichiarazioni di import:

```
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

Ingegneria del Software – a.a. 2005/06

Appello del 16 febbraio 2007

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi Ghezzi Morzenti SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità.
Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Totale

Esercizio 1 (punti 10)

Il tipo di dato astratto (ADT) **Tombola** fornisce un'astrazione per una partita a tombola. Un giocatore estrae un numero usando il metodo **estrai()**. Questo metodo restituisce ogni volta un numero (diverso e a caso) tra 1 e 90. Se l'estrazione di un numero provoca una tombola, il metodo **estrai** deve generare un'eccezione **ExceptionTombola** e chiudere la partita. A ogni partita a tombola è associato un certo insieme di cartelle dei vari giocatori partecipanti; ogni cartella è rappresentata da un esemplare del tipo di dato astratto **Cartella**. L'ADT **Tombola** fornisce i metodi puri **tombola**, **cartelle** e **numeriEstratti**:

```
//@ requires true;
//@ ensures (*restituisce true se i numeri di c sono stati estratti tutti*) ;
public boolean /*@pure@*/ tombola(Cartella c)

//@ requires true;
//@ ensures (*restituisce le cartelle attive*) ;
public arrayList<Cartella> /*@pure@*/ cartelle()

//@ requires true;
//@ ensures (*restituisce la sequenza dei numeri estratti fino a quell' istante, in ordine cronologico di estrazione (quelli estratti per primi occupano le prime posizioni dell'arrayList)*) ;
public arrayList<Integer> /*@pure@*/ numeriEstratti()
```

a- Si scriva la postcondizione del metodo **estrai**, che restituisce il nuovo estratto:

```
//@ requires true;
public int estrai() throws ExceptionTombola
```

evidenziando che:

1. Il numero estratto è un numero non estratto in precedenza, compreso tra 1 e 90.
2. La sequenza dei numeri già estratti resta immutata, con l'unica aggiunta del numero estratto
3. Il numero estratto non causa una tombola.

SOLUZIONE

```
//@ ensures 1<=result && result <=90 && \old(!numeriEstratti().find(result)) &&
numeriEstratti().lastElement() == result &&
numeriEstratti.size() == \old(numeriEstratti.size()+1) &&
\forall(int i; 0<=i && i<numeriEstratti().size()-1; numeriEstratti.get(i) == \old(numeriEstratti.get(i)))
&&
\forall(int i; 0<=i && i< cartelle().size()); !cartelle.get(i).tombola()
```

b- Si scriva la clausola signals relativa al metodo **estrai**. In particolare, il metodo prevede che se il numero estratto provoca la tombola, questo viene comunque inserito nella sequenza dei numeri estratti.

```
//@ signals (ExceptionTombola e)
//alcune condizioni sono identiche a quelle della ensures:
numeriEstratti.size() == \old(numeriEstratti.size())+1) &&
\forall(int i; 0<=i && i< numeriEstratti().size()-1);numeriEstratti.get(i) ==\old(numeriEstratti.get(i))
&&
//ma serve anche indicare che l'ultimo elemento della sequenza dei numeri estratti è nuovo:
!exists( int i; i == numeriEstratti().lastElement(); \old(numeriEstratti().find(i)) &&
// e che una cartella fa tombola
\exists(int i; 0<=i && i<= cartelle().size()-1; cartelle.get(i).tombola()).
```

c- Sapendo che l'ADT **Cartella** fornisce il metodo puro per conoscere i numeri di una cartella:

```
//@ requires true ;
//@ ensures (*restituisce i numeri in this*) ;
public arrayList<Integer> /*@pure@*/ numeri()
```

Si scriva l'invariante pubblico per la classe **Tombola** per imporre che una tombola non può avere due cartelle contenenti gli stessi numeri e che le cartelle di ogni esemplare di **Tombola** devono coprire tutti i numeri da 1 a 90, ovvero ogni numero deve essere presente in almeno una cartella.

```
//@public invariant
//@(\forall int i; 0<=i && i< cartelle().size()-1;
    //(\forall int j; i<j && j<= cartelle().size()-1;
        //(\exists(int k; 1<=k && k<=90;
            //cartelle.get(i).numeri().find(k) && !cartelle.get(j).numeri().find(k)))) &&
//@(\forall int n; 1<= n && n<=90;
    //(\exists int i; 0<=i && i< cartelle().size(); cartelle().get(i).numeri().find(n))));
```

Esercizio 2 (punti 6)

Si scriva in Java un iteratore che permetta di ottenere, in una sequenza casuale, tutti i numeri compresi tra 1 e 90 (si può immaginare che tale iteratore possa essere impiegato nell'implementazione del metodo **estrai** del precedente esercizio).

Nell'implementare l'iteratore si suggerisce di far uso dei seguenti dati

```
private boolean[] tabellone;
private int numEstratti ;
```

Si definisca sia il metodo **nuovaSequenza** che restituisce l'iteratore, sia la classe che definisce l'iteratore (ossia il generatore di numeri interi). L'iteratore deve: (a) fornire ogni numero una sola volta, (b) marcare il numero estratto nell'array di booleani e (c) incrementare il contatore **numEstratti**. Si ipotizzi di poter disporre di un metodo statico **numero** che restituisce un numero casuale tra 1 e 90.

```
import java.util.Iterator;

public class Tabellone {

    public Iterator<Integer> nuovaSequenza() {
        return new SequenzaCasuale();
    }

    public static class SequenzaCasuale implements Iterator<Integer> {
        private boolean[] tabellone;
        private int numEstratti;

        SequenzaCasuale() {
            tabellone = new boolean[90];
            for (int i=0; i<90; i++) tabellone[i]=false;
            numEstratti = 0;
        }

        public boolean hasNext() {
            return numEstratti<90;
        }

        public Integer next() throws NoSuchElementException {
            if (numEstratti>=90) throw new NoSuchElementException("nuovaSequenza");

            numEstratti++;

            int n = numero();

            while (tabellone[n]) n = (n+1)%90;
            tabellone[n]=true;
            return n;
        }
    }
}
```

Esercizio 3 (punti 5)

Si fornisca l'invariante privato per la classe **Cartella** che fornisce un'astrazione della cartella della tombola. Si ricorda che (a) ogni cartella contiene 15 numeri, tutti diversi, fra 1 a 90 (b) i numeri sono disposti su 3 righe, (c) ogni riga contiene 5 numeri, (d) ogni cartella deve contenere almeno un numero per ognuna delle nove decine di cui è costituito l'insieme dei numeri estraibili.

```
public class Cartella {
    private int[][] num;
    ...
}
```

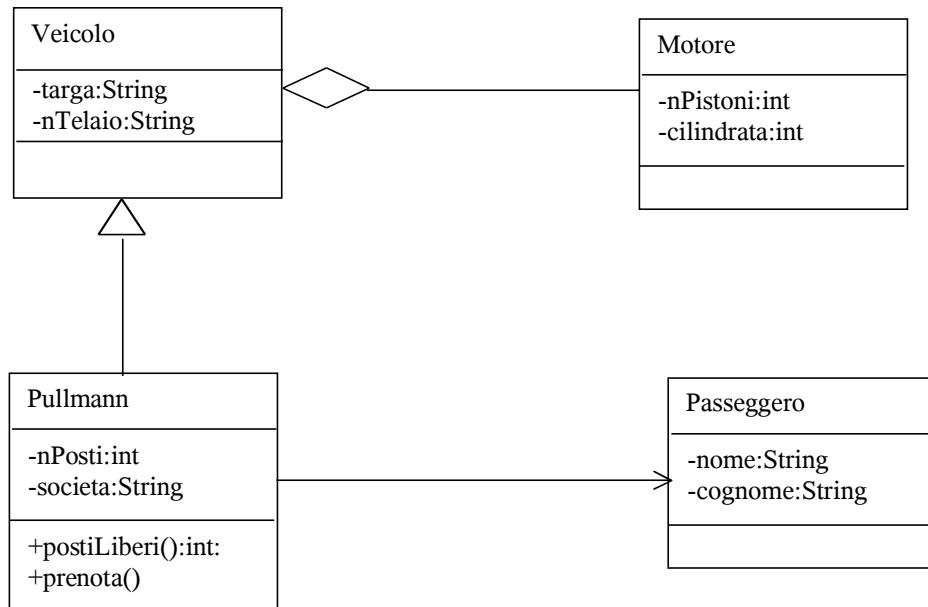
SOLUZIONE :

/* private invariant

```
num.length==3 &&
(\forall int k; 0<=k && k<3 ; num[k].length==5 ) &&
(\forall int i ; 0<=i && i<=2 ;
  (\forall(int j ; 0<=j && j<=4 ; 1<=num[i][j] && num[i][j]<=90 &&
    !(\exists int h ; 0<=h && h<=2;
      (\exists int k; 0<=k && k<=4 && i !=h && j !=k; num[i][j]==num[h][k])) &&
    (\forall int n ; 0<=n && n<=8 ;
      (\exists int h,k ; 0<=h && h<=2 && 0<=k && k<=4 ; num[h][k]/10 == n)) ;
```

Esercizio 4 (punti 4)

Si scrivano le parti dichiarative delle classi descritte dal seguente diagramma delle classi UML.
Si scriva anche la postcondizione del metodo “prenota” della classe Pullman.



```
package esame;

public class Veicolo {
    private String targa;
    private String nTelaio;
    private Motore motore;
}

public class Motore {
    private int nPistoni;
    private int cilindrata;
    private Veicolo veicolo;
}

public class Pullman extends Veicolo{
    private int nPosti;
    private String societa;
    private Passeggero[] passeggeri;

    public int postLiberi(){
        ...
    }

    public void prenota(){
        ...
    }
}

public class Passeggero {
```

```
    private String nome;  
    private String cognome;  
}
```

Parte B

```
//@ ensures postiLiberi() = \old(postiLiberi()) -1;  
public void prenota(){  
    ...  
}
```

Ingegneria del Software – a.a. 2005/06

Appello del 24 luglio 2006

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi ?, Ghezzi ?, Morzenti ?, SanPietro ?

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
1

Esercizio 1 (punti 6)

L'ADT *Matrice* fornisce un'astrazione per una matrice quadrata di interi di ordine n . L'ADT presenta un metodo *leggiMatrice* che restituisce un vettore di interi che corrisponde alla matrice letta colonna per colonna. Il metodo legge il contenuto della matrice quadrata e riempie il vettore partendo dalla prima colonna. Ad esempio, se la matrice avesse dimensione 3, il vettore restituito dovrebbe contenere gli elementi $a[0][0]$, $a[1][0]$, $a[2][0]$, $a[0][1]$, $a[1][1]$, $a[2][1]$, $a[0][2]$, $a[1][2]$ e $a[2][2]$. Si supponga che l'ADT fornisca anche i metodi *lunghezza* ed *elem* specificati sotto:

```
//@ requires true ;
//@ ensures (*restituisce l'ordine della matrice quadrata*) ;
public int /*@pure@*/ lunghezza() ;

//@ requires i >= 0 && i < lunghezza() && j >= 0 && j < lunghezza() ;
//@ ensures (*restituisce l'elemento in posizione i,j dove i rappresenta il numero della riga e j il numero della colonna*)
public int /*@pure@*/ elem(int i, int j) ;
```

- a- Si scriva la specifica formale in JML del metodo *leggiMatrice* ricordando che la postcondizione deve definire il corretto ordine degli elementi contenuti nel vettore di ritorno.

```
//@ requires true;
```

```
//@ ensures
/result != null &&
(result.length = lunghezza() * lunghezza()) &&
(forall int i; i >= 0 && i < lunghezza());
  (forall int j; j >= 0 && j < lunghezza();
    \result[i * lunghezza() + j] == elem(j, i)) ;
```

//@assignable \nothing; --oppure si dice esplicitamente che tutti gli elem(i,j) non cambiano

```
public int[] leggiMatrice ()
```

- b- Supponendo che la rappresentazione (REP) sia la seguente :

```
private int[][] m;
```

si scriva in Java un opportuno iteratore per leggere gli elementi della matrice, definendo sia il metodo di Matrice che restituisce l'iteratore, sia la classe che definisce l'iteratore (ossia il generatore).

L'iteratore deve scandire la matrice colonna per colonna e quindi la sequenza in cui l'iteratore restituisce gli elementi deve ancora essere: primo elemento della prima colonna, secondo elemento della prima colonna, ultimo elemento della prima colonna, primo elemento della seconda colonna, ..., ultimo elemento dell'ultima colonna.

```

Metodo iteratore :
public Iterator<Integer> getIterator() {

    return new MatriceGen(this) ;
}

```

dentro la classe Matrice:

```

private static class MatriceGen implements Iterator<Integer>{
    private int riga;
    private int colonna;
    private Matrice mat;

    public MatriceGen(Matrice x) {
        mat=x ; riga=0 ; colonna=0 ;
    }
    public boolean hasNext() {
        return colonna<mat.lunghezza();
    }

    public Integer next() throws NoSuchElementException {
        if (colonna>=mat.lunghezza()) throw new NoSuchElementException();
        Integer ret = new Integer(mat.m[riga][colonna]);
        riga++; //si posiziona sulla riga successiva
        if (riga == mat.m.length) {//se le righe sono finite
            riga = 0; colonna++; //passa alla prima riga della colonna successiva
        }
        return ret;
    }
}

```

NB : Per l'iteratore c'e' anche una soluzione molto semplice, anche se poco efficiente:
si costruisce un array con la leggiMatrice(), se ne ricava una lista (tramite ad esempio il metodo Arrays.asList(), descritto dalla documentazione Java) e poi si ritorna l'iteratore standard di questa lista.

Basta quindi definire il metodo iterator comse segue, senza definire una classe interna:

```

public Iterator<Integer> getIterator() {
    return Arrays.asList(this.leggiMatrice()).iterator();
}

```

L'iteratore quindi si sposta su una lista che è un nuovo oggetto che contiene i valori della matrice originale.

Esercizio 2 (punti 12)

Si consideri una versione semplificata del Sudoku, noto gioco solitario di logica. La classe **Sudoku** rappresenta la griglia del gioco. Si ricorda che la griglia è una matrice quadrata di 9x9 caselle. Ciascuna casella può essere vuota oppure contenere un numero fra 1 e 9. Inizialmente la griglia contiene da 20 a 35 numeri e il giocatore deve riempire le caselle vuote disponendo i numeri ---sempre compresi tra 1 e 9--- in modo che ogni riga e ogni colonna non contenga numeri ripetuti. Le righe e le colonne sono numerate da 0 a 8, da sinistra a destra e dall'alto verso il basso. NB: In questa versione semplificata, il piano NON viene diviso in regioni rettangolari in cui controllare l'unicità dei numeri, come invece avviene nel Sudoku "tradizionale".

L'utente avrà a disposizione un'opportuna interfaccia per inserire i numeri e visualizzare la griglia, ma a noi interessa concentrarci sulla logica del gioco e quindi la classe **Sudoku** offre i metodi:

- *riga* e *colonna* per ottenere la parte di griglia richiesta come ArrayList di interi. Le celle vuote sono inizializzate a zero.

- *inserisci* per aggiungere un numero n in posizione x,y della griglia. Il metodo scrive il numero n nella posizione indicata anche se questa è già occupata da un altro numero. Se l'inserimento non viola le regole, il metodo ritorna true; altrimenti il numero non viene inserito e il metodo ritorna false.
- *grigliaCompleta* per capire se una griglia è stata completata in modo corretto. Il metodo restituisce true se la griglia è completa e finita; false altrimenti.

a- Scrivere la specifica del metodo *colonna*:

```
//@ requires
x >= 0 && x <= 8;

//@ ensures
\result.size() == 9 && (\forall int i; 0 <= i < 9; \result.get(i) == riga(i).get(x));

public ArrayList<Integer> /*@pure@*/ colonna(int x)
```

b- Completare la specifica del metodo *inserisci*:

```
//@ requires n >= 1 && n <= 9 && x >= 0 && x <= 8 && y >= 0 && y <= 8;
```

NB: questa versione di inserisci ipotizza che $n \geq 1$, e quindi non puo' essere usata per cancellare una casella

```
//@ ensures
(\forall int i; i >= 0 && i <= 8; /*elementi non in (x, y) non cambiano*/
  (\forall int j; j >= 0 && j <= 8;
    (i != x || j != y) ==> riga(i).get(j) == \old(riga(i).get(j))) &&
  (\forall int i; i >= 0 && i <= 8;
    (i != y ==> riga(x).get(i) != n) && /*nessuna ripetizione sulla riga x*/
    (i != x; colonna(y).get(i) != n)) /*nessuna ripetizione sulla colonna y*/
? \result == true && riga(x).get(y) == n /*se nessuna ripet. risultato e' true*/
:\result == false && riga(x).get(y) == \old(riga(x).get(y))) /*altrimenti false*/
```

```
public boolean inserisci(int n, int x, int y)
```

c- Scrivere la specifica del metodo *grigliaCompleta*. Si faccia attenzione al fatto che il metodo *inserisci* non consente l'aggiunta di elementi che violano le regole del gioco.

```
//@ requires true
/* NB: non e' errato supporre come requires che la griglia non violi le regole del gioco, anche se
in realtà sarebbe una condizione da invariante pubblico */
//@ ensures
\result == true <==>
  (\forall int i; int i >= 0 && i <= 8; (\forall int j; j >= 0 && j <= 8; riga(i).get(j) != 0))
//@ assignable \nothing ;
public boolean grigliaCompleta()
```

d- Si scriva l'invariante pubblico della classe **Sudoku** per imporre la presenza di almeno 20 elementi (diversi da zero) nella griglia.

```
//@ public invariant
(\sum_{int i; 0 <= i < 9} (\sum_{int j; 0 <= j < 9} this.riga(i).get(j) > 0)) >= 20;
```


Esercizio 3 (punti 7)

Un computer è composto da una CPU e da alcune periferiche. Le periferiche possono essere tastiere, dischi rigidi, monitor, stampanti (laser o a getto di inchiostro) e altro. Alla CPU devono essere sempre connessi almeno una tastiera, un disco e un monitor.

Di ogni periferica si conosce il tipo di connessione fisica (USB, PS2, ATA, VGA, DVI, ...). Le connessioni tra CPU e periferiche sono caratterizzate da un numero di IRQ per la gestione degli interrupt.

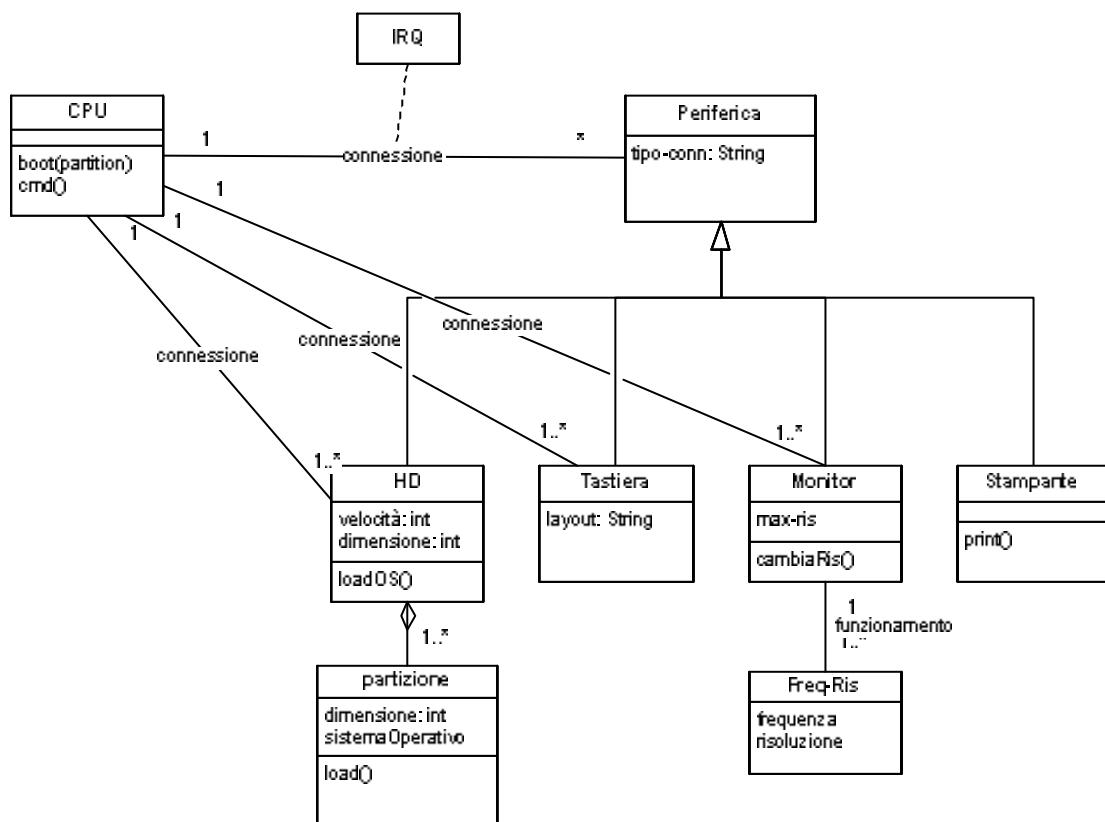
La tastiera è caratterizzata da un layout. Del disco rigido sono note velocità di rotazione e dimensione. I dischi sono divisi in partizioni (almeno una) di dimensioni diverse, ognuna formattata con un File System e con un sistema operativo. Ogni monitor ha una risoluzione massima e offre un elenco di coppie risoluzione-frequenza a cui può funzionare.

La CPU reagisce al comando di boot proveniente dall'esterno caricando il sistema operativo dalla partizione selezionata dall'utente. Avviato il sistema, riceve comandi dalla tastiera e li interpreta, agendo eventualmente sulle altre periferiche.

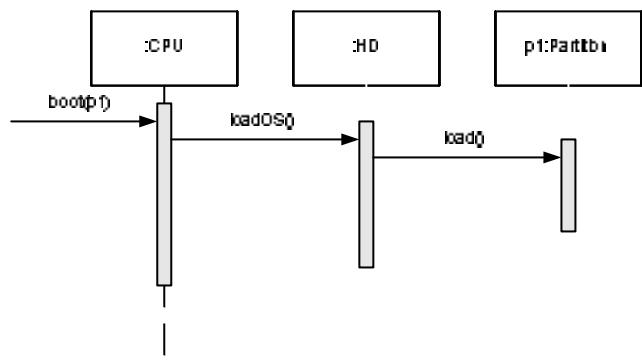
- a- Tracciare il class diagram UML descritto sopra.
- b- Disegnare il sequence diagram che modella l'accensione del sistema
- c- Disegnare il sequence diagram dell'interazione di un utente che – tramite tastiera – invoca comandi sulla CPU. Se il comando è “cambia-risoluzione”, la CPU modifica la risoluzione di un monitor in base ai parametri provenienti sempre dalla tastiera.

SOLUZIONI UML

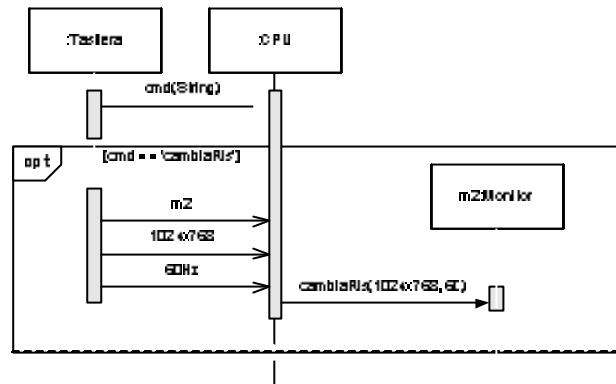
a)



b)



c)





Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. *Carlo Ghezzi*

prof. *Angelo Morzenti*

prof. *Pierluigi San Pietro*

20133 Milano (Italia)

Piazza Leonardo da
Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Appello di Ingegneria del Software

13 Luglio 2004

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi Morzenti SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h15m.
6. Punteggio totale a disposizione: 26/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio 1 (punti 10)

1. Utilizzando opportune formule che definiscono le pre e post-condizioni, si fornisca la specifica di un'astrazione procedurale che in input riceve due array x e y di interi.

a) L'astrazione restituisce true se x è una permutazione di y, false altrimenti, sotto l'ipotesi che ne' x ne' y abbiano elementi duplicati. Se almeno uno dei due array è null, allora viene lanciata l'eccezione unchecked NullPointerException.

Es. x = [1,5, 2], y = [2, 5, 1] risultato = true

x = [1,5, 2], y = [2, 5, 0] risultato = false

x = [1,5, 2], y = [2, 5, 2] risultato = indefinito

```
public static boolean risultato(int[]x, int[]y) {
```

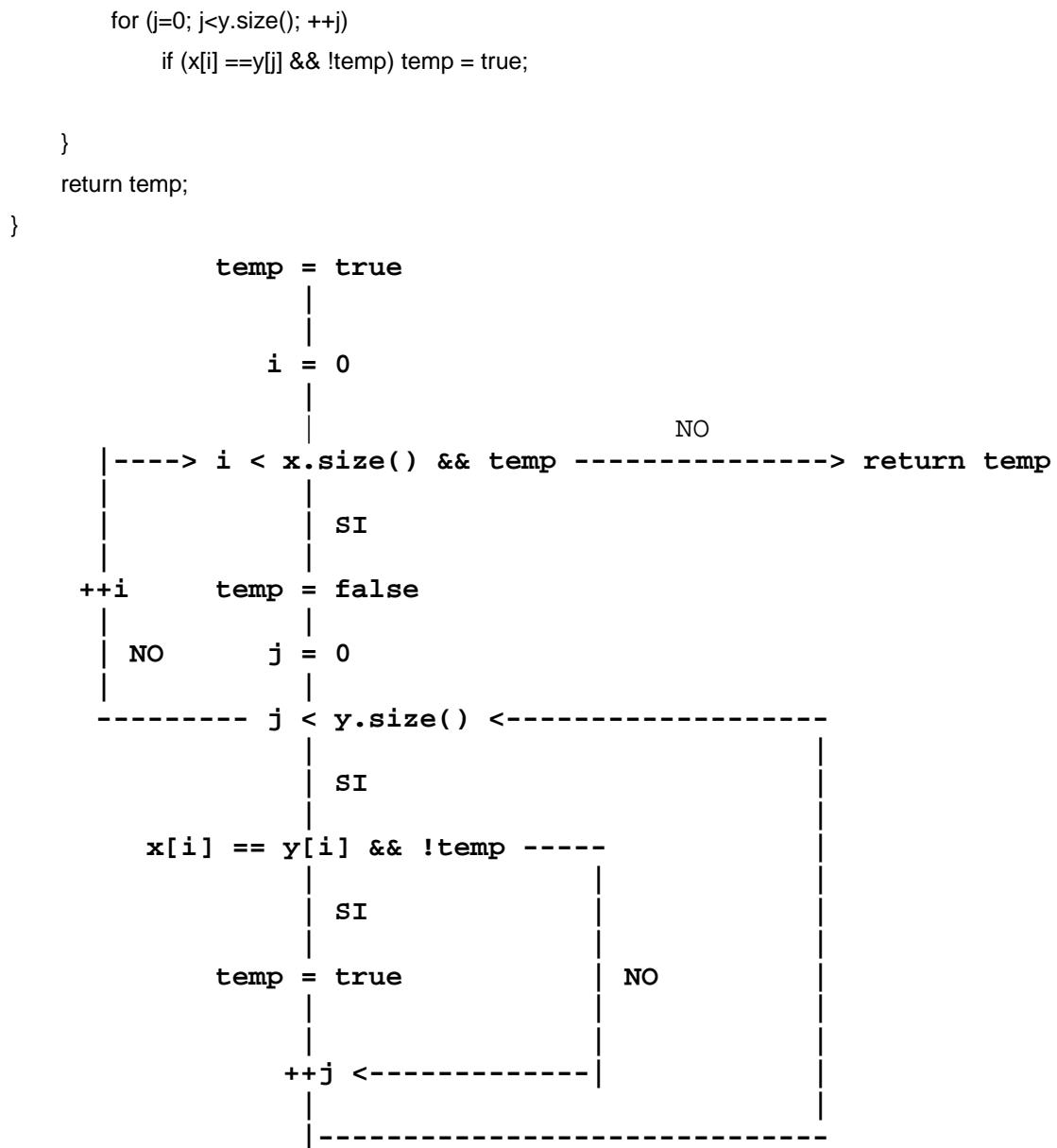
```
//REQUIRES: forall i,j, 0<=i<j<x.size()-1, x[i]!= x[j] && forall i,j, 0<=i<j<y.size()-1, y[i]!= y[j]
//EFFECTS: if (x==null) || (y==null) throw NullPointerException
//           else if (x.size<>y.size) return false
//           else if x.size()==0 return true
//           else return
//                  (forall i, 0<=i<x.size()-1, exists j, 0<=i<x.size()-1, x[i] = y[j]) &&
//                  (forall i, 0<=i<y.size()-1, exists j, 0<=i<y.size()-1, y[i] = x[j])
```

2. In base alla specifica fornita, si definiscano cinque dati numerici significativi a supporto del test funzionale (black-box testing) in base ai criteri di copertura delle combinazioni proposizionali e dei valori limite, specificando inoltre i risultati attesi per ciascun dato.

< x, y >	
<[null], [1, 2]>	solleva NullPointerException
<[1, 2, 3], [4, 5]>	false
<[], []>	true
<[3, 7, 1, 6], [1, 6, 7, 3]>	true
<[2], [2]>	true
<[1, 2, 3], [4, 5, 6]>	false

3. Si consideri la seguente implementazione del caso (a) (*non necessariamente corretta* rispetto alla specifica). Si identifichi il minimo numero di dati di test, specificandone valore numerico e risultato atteso, per garantire la copertura al 100% del codice secondo il criterio delle diramazioni.

```
public static boolean risultato(int[]x, int[]y) {
    boolean temp=true;
    for (int i=0; i < x.size() && temp; ++i) {
        temp= false;
```



Basta un solo caso di test, ad esempio $<[5], [3, 5]>$, per il quale il valore atteso secondo la specifica dovrebbe essere false, ma per il quale la funzione ritorna true.

Esercizio 2 (punti 16)

Si consideri la seguente specifica dell'astrazione sui dati PilaDoppiaTesta.

```
public class PilaDoppiaTesta {  
    /*OVERVIEW: Tipo mutable, che rappresenta una pila con due estremità (sinistra e destra), che consente  
    l'inserimento sia in testa a sinistra che in coda a destra. L'estrazione avviene esclusivamente dalla testa di  
    sinistra, mantenendo l'ordine di inserzione. Gli elementi inseriti sono di tipo Object, ma diversi da null.  
    Un oggetto tipico è [o1, o2, ..., on], dove o1 e' la testa sinistra della pila, on è la testa destra.  
*/  
  
    public PilaDoppiaTesta ()  
        //EFFECTS: costruisce la pila vuota  
    public void pushLeft (Object x)  
        //REQUIRES: x<>null  
        //EFFECTS: posto this = [o1, o2, ..., on], n>=0, this_post = [x, o1, o2, ..., on]  
        //MODIFIES this  
  
    public void pushRight (Object x)  
        //REQUIRES: x<>null  
        //EFFECTS: posto this = [o1, o2, ..., on], n>=0, this_post = [o1, o2, ..., on, x]  
        //MODIFIES this  
  
    public Object pop() throws EmptyException;  
        //EFFECTS: if this è vuota throw EmptyException  
        //           else, posto this = [o1, o2, ..., on], this_post = [o2, ..., on] e o1 è restituito  
        //MODIFIES this  
  
    public Iterator LeftToRight();  
        //EFFECTS: restituisce un generatore che itera su tutti gli elementi di this,  
        //           nell'ordine da sinistra a destra  
        //REQUIRES: this non può essere modificato mentre il generatore è in uso.  
  
    public Iterator RightToLeft();  
        //EFFECTS: restituisce un generatore che itera su tutti gli elementi di this,  
        //           nell'ordine da destra a sinistra  
        //REQUIRES: this non può essere modificato mentre il generatore è in uso.  
  
    public int size(); //EFFECTS: restituisce il numero di elementi nella pila  
}
```

(continua esercizio 2)

- a) Si specifichi, come sottoclasse di PilaDoppiaTesta, una variante in cui il metodo PushLeft lancia un'eccezione checked di tipo InvalidObject qualora sia passato un argomento x che vale null. La variante rispetta il principio di sostituzione?

```
public VariantePilaDoppiaTesta extends PilaDoppiaTesta {  
    public void pushLeft (Object x) throws InvalidObject  
        //EFFECTS: if (x == null) throw new InvalidObject else  
        //posto this = [o1, o2, ..., on], n>=0, this_post = [x, o1, o2, ..., on]  
        //MODIFIES this
```

Non rispetta il principio perche' viola la regola delle signature

(continua esercizio 2)

- b) Si specifichi, come sottoclasse di PilaDoppiaTesta, un ADT DoppiaPila che possiede un'operazione supplementare popRight(), che è come pop ma estraе e restituisce l'elemento più a destra nella pila.

DoppiaPila rispetta il principio di sostituzione?

```
public class DoppiaPila extends PilaDoppiaTesta {  
    //OVERVIEW: è una PilaDoppiaTesta in cui è possibile estrarre oggetti anche dalla testa di  
    destra.
```

```
    public Object popRight() throws EmptyException;  
    //EFFECTS: if this è vuota throw EmptyException  
    // else, posto this = [o1, o2, ..., on], this_post = [o1, ..., on-1] e on è restituito  
    //MODIFIES this
```

NON rispetta il principio di sostituzione perché viola banalmente la proprietà dell'Overview di PilaDoppiaTesta: “L'estrazione avviene sempre dalla testa di sinistra”. Quindi, sorprenderebbe tutti gli utilizzatori che potrebbero non recuperare con un pop un elemento inserito, pur ritrovando con pop() tutti gli elementi da loro inseriti prima di quello.

(continua esercizio 2)

- c) Si completi la seguente implementazione della classe PilaDoppiaTesta, specificando AF e RI.
Note: Il metodo iterator() dei Vector restituisce un generatore agli elementi a partire dalla posizione 0. Non vi è un simile metodo predefinito che restituisce un generatore dall'ultima posizione alla posizione 0: il metodo RightToLeft() va quindi implementato ad hoc, definendo anche la classe del generatore.

L'implementazione è corretta rispetto alla specifica? Giustificare la risposta.

```
public class PilaDoppiaTesta {  
    Vector v; //rep  
    //AF(c) = [v(0), v(1), ... v(v.size()-1)].....  
    //RI(c) = v != null && forall i, 0<=i<v.size(), v(i) != null .....
```

```
    public PilaDoppiaTesta () {v = new Vector();}  
    public void pushLeft (Object x) {  
        v.add(x,0); //aggiunge x nella prima posizione di v  
    }  
    public void pushRight (Object x){  
        v.add(x); //aggiuge x nell'ultima posizione di v  
    }  
    public Object pop() throws EmptyException {  
        if (v.size() ==0) throw new EmptyException;  
        Object o = v.elementAt(0);  
        v.removeElementAt(0);  
        return o;  
    }  
    public int size() {  
        return v.size();  
    }  
    public Iterator LeftToRight() {  
        return v.iterator();  
    }  
    public Iterator RightToLeft(){  
.....return new Iterator{  
..... int pos = v.size()-1;  
..... public boolean hasNext(){ return pos >= 0; }  
.....  
..... public Object next() throws NoSuchElementException {  
..... if (!hasNext()) throw new NoSuchElementException();  
..... Object res = v.elementAt(pos);  
..... pos--;  
..... return res;  
..... }  
.....  
..... public void remove() throws UnsupportedOperationException {  
..... throw new UnsupportedOperationException()
```

```
..... }  
}
```

L'implementazione E' CORRETTA rispetto alla specifica, in quanto:

- 1- l'implementazione conserva il RI, infatti il costruttore lo conserva, e i metodi, se invocati in maniera che la clausola REQUIRES sia soddisfatta, anche lo conservano
- 2- le operazioni sono corrette rispetto all'astrazione
- 3- se i metodi vengono invocati in modo da rispettare le clausole REQUIRES, le proprietà dell'astrazione (in particolar modo il fatto che gli oggetti nella coda siano diversi da null) vengono anche rispettate (le altre proprietà sono banalmente rispettate).

(continua esercizio 2)

- d) Si desidera ora utilizzare la classe PilaDoppiaTesta per implementare una PilaSenzaSimili di Object, che si comporta come una pila qualunque ma in cui non è possibile inserire oggetti “simili” a oggetti già presenti in Pila. Per stabilire se due oggetti sono simili, la Pila utilizza l’approccio related subtype, tramite il metodo simili di un’interfaccia Verificatore. Il Verificatore da utilizzare è stabilito alla chiamata del costruttore di PilaSenzaSimili. Le specifiche di PilaSenzaSimili e del Verificatore sono:

```
public class PilaSenzaSimili {  
    //OVERVIEW: una Pila di Object in cui non si possono inserire elementi "simili" a quelli presenti nella pila.  
    public PilaSenzaSimili (Verificatore v)  
        // REQUIRES: v != null  
        // EFFECTS: costruisce una pila vuota  
        public Object pop()//EFFECTS: restituisce ed elimina da this l'elemento in cima  
        public void push(Object o) throws SimilarException  
            //EFFECTS: if o è simile a uno qualunque degli elementi già presenti nella Pila, lancia SimilarException,  
            //else inserisce o in cima alla pila; v  
}
```

```
public interface Verificatore {  
    public boolean simili(Object o1, Object o2);  
    //EFFECTS: return true se o1 e o2 sono simili, false in tutti gli altri casi.  
    //Devono valere le proprietà riflessiva e simmetrica:  
    // se o1.equals(o2) allora deve valere simili(o1,o2);  
    // se simili(o1,o2) allora simili(o2,o1).  
}
```

Completare l’implementazione seguente della classe PilaSenzaSimili.

```
public class PilaSenzaSimili {  
    //AF(c) = AF(p).....  
    //RI(c) = p<>null && v!= null && forall Object x,y in p, (v.simili(x,y) ==> x==y).....  
    private PilaDoppiaTesta p;  
    private Verificatore v;  
    public PilaSenzaSimili(Verificatore v) (p = new PilaDoppiaTesta(); this.v = v;}  
    public Object pop() {return p.pop();}  
    public void push(Object o) throws SimilarException {  
        for (Iterator i = p.leftToRight(); p.hasNext();) {  
            Object o1 = p.next();  
            if (v.simili(o,o1)) throw new SimilarException();  
        }  
        .....  
        .....  
        .....
```

```
.....  
    p.pushLeft(o);  
}  
}
```

Considerare poi la classe Double, corrispondente al tipo primitivo double. Si codifichi una classe RifiutaDoubleVicini che implementi l’interfaccia Verificatore e permetta di inserire oggetti nella Pila con il seguente criterio di similitudine: due oggetti x1 e x2 sono simili se entrambi x1 e x2 sono Double e inoltre vale $\text{Math.abs}(x1-x2) < 0.00001$, oppure (nel caso in cui non siano entrambi Double) se x1 e x2 sono “equals”. Mostrare un esempio di costruzione di una pila.

```
public class RifiutaDoubleVicini implements Verificatore {  
    public boolean simili(Object o1, Object o2) {  
        //EFFECTS: return true se o1 e o2 sono simili, false in tutti gli altri casi  
        if (!(o1 instanceof Double && o2 instanceof Double)) return o1.equals(o2);  
        double d1 = ((Double) o1).value();  
        double d2 = ((Double) o2).value();  
        if (Math.abs(d1-d2)<0.00001)  
            return true;  
        else return false;  
    }  
....  
PilaSenzaSimili p = new PilaSenzaSimili (new RifiutaDoubleVicini ());  
....
```



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da Vinci,

32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Recupero di Ingegneria del Software –

15 luglio 2002

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Recupero prova: 1 2

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno presi in considerazione.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h.30m se si svolge una sola parte, 3h per entrambe le parti.
6. Punteggio totale a disposizione per ciascuna prova: 13 punti. Una prova può essere sommata al risultato dell'altra prova (già svolta o quella presenbte) solo ottenendo almeno 6 punti.

PROVA 1: Esercizio 1.1 (5 punti)

Parte (a)

Si considerino le seguenti classi Java.

Completare opportunamente le parti con i puntini, aggiungendo, ove necessario, modificatori di visibilità, dichiarazioni e/o implementazioni di metodi, ecc. Si trascuri l'implementazione di altri metodi oltre a quelli strettamente necessari. Non implementare equals, repOk, toString, ecc.

```
package Linker;
public abstract..... class Modulo{
//OVERVIEW: Tipo mutabile che rappresenta una unità di codice scritto in un
//linguaggio specificato.
    private String linguaggio; //memorizza il nome del linguaggio
    public abstract String getCodice()
    //EFFECTS: restituisce una rappresentazione stampabile del codice
}
public Modulo(String linguaggio) {
    //EFFECTS: inizializza this a essere un modulo scritto nel linguaggio dato
    this.linguaggio=linguaggio;
}
public String getLinguaggio() {
    //EFFECTS: restituisce il nome del linguaggio in cui e' scritto this
    return linguaggio;
}

package CoseUtili;
public class CodiceScorrettoException extends Exception {
//da non implementare}

package CoseUtili;
public class MetodiUtili {
    public static String ByteCode2Obj (String codiceByteCode)
        throws CodiceScorrettoException {
        //REQUIRES: codiceByteCode è una stringa contenente il codice corretto di
        //un modulo di codice ByteCode
    /   //EFFECTS: restituisce la traduzione di codiceByteCode in codice oggetto OBJ
    //NB: da non implementare!!!
    }
}

package Linker;
public..... class ModuloByteCode extends Modulo{
//OVERVIEW: Tipo che rappresenta un modulo di codice in
//linguaggio JavaByteCode
//AF: una coppia (this.linguaggio, this.codice)
//rep: una stringa
protected..... String codice; //anche private è corretto
    public ModuloByteCode (String codice) {
        //inizializza this a contenere il codice, scritto in linguaggio "JavaByteCode"
        super("JavaByteCode");
        this.codice=codice;
    }

    public String getCodice() {
        return this.codice.....
    }
    public ModuloObj traduciInObj()
        .... throws CodiceScorrettoException {
        return new ModuloObj(CoseUtili.MetodiUtili.ByteCode2Obj( this.codice);.....
    }
}
```

```

public ..... class ModuloObj extends .....Modulo{
//OVERVIEW: Tipo che rappresenta un modulo di codice in
//linguaggio oggetto (OBJ)
//rep: una stringa
    protected String codice;.....
    public ModuloOBJ (String codice) {
        super("OBJ");
        this.codice=codice;.....
    }
    public String getCodice() {
        return this.codice
    }
}

```

Esercizio 1, Parte (b)

In un file separato di nome LinkerUser.java, contenuto in un altro package, è riportata la seguente classe (NB: si rammenta che import permette di importare una classe e utilizzarne i metodi pubblici senza dot notation):

```

import Linker.*;
public class LinkerUser {
    public static void main(String args[]) {
        Modulo m[] = new Modulo[4];
        String cod1;
        .....//qui cod1 assegnato a codice JavaByteCode corretto;
        try {
            m[0] = new ModuloByteCode(cod1);
            m[1] = m[0].traduciInObj();
            m[2] = new ModuloObj(m[1].getCodice());
            stampa(m);
        }
        catch (Exception e) { System.out.println("Eccezione");}
    }
    public static void stampa(Modulo moduli[]) {
        for (i=0; i<moduli.length; i++)
            System.out.println(moduli[i].getLinguaggio());
    }
}

```

Si ricorda che il metodo statico System.out.println() stampa in output il valore dell'espressione passata come argomento.

Riportare in modo preciso il risultato dell'esecuzione (che cosa viene stampato, eventuali errori o eccezioni run-time): Per ogni chiamata del metodo getLinguaggio() indicare qual è la classe in cui è definito il metodo effettivamente chiamato durante l'esecuzione. Se viene sollevata un'eccezione, dire quale eccezione viene sollevata, da chi viene intercettata e come prosegue, se prosegue, l'esecuzione.).

La riga m[0].traduciInObj(); richiederebbe un cast a ModuloByteCode e quindi così il programma non compila.

Se ci fosse il cast il risultato sarebbe:

stampa:

JavaByteCode, OBJ, OBJ, Eccezione

Genera NullPointerException quando cerca di stampare m[3]: l'eccezione è unchecked e si propaga al chiamante, che la intercetta nel catch.

(NB: entrambe le soluzioni, con o senza cast, sono accettabili).

Cosa succederebbe se si eliminasse il blocco try...catch dal main (lasciando tutte le altre istruzioni all'interno del blocco)?

il programma non compila, perche' la traduciInObj dichiara throws di eccezione checked

PROVA 1: Esercizio 1. 2 (8 punti)

Si consideri la seguente specifica della classe MultiSet:

```
public class MultiSet {  
    // OVERVIEW: multi-insiemi (detti anche bag) di oggetti di tipo Object:  
    //un multiinsieme è un contenitore di elementi in ordine qualunque, di molteplicità qualunque.  
    //Il valore null non fa parte di un multinsieme  
    // Usa equals per confrontare gli oggetti  
    //Un tipico MultiSet è una sequenza s = x1 x2... xn, dove  
    //ciascun valore xi compare in s un numero >=1 di volte.  
    //Per es.: A B A C A C, dove A, B, C sono valori di tipo Object.  
    //costruttori:  
    public MultiSet(){  
        //EFFECTS: inizializza this come insieme vuoto  
    }  
    //metodi:  
    public MultiSet insert(Object x);  
        //REQUIRES: x!= null;  
        //MODIFIES: this  
        //EFFECTS: aggiunge x in una posizione qualunque di this e ritorna this  
    public void remove(Object x);  
        //REQUIRES: x!= null;  
        //MODIFIES: this  
        //EFFECTS: se x è in this, ne elimina una qualunque (e una sola) occorrenza da this  
    public int molt(Object x)  
        //REQUIRES: x!= null;  
        //EFFECTS: restituisce il numero di elementi E di this per i quali x.equals(E) è true  
}
```

a) Sono dati i seguenti metodi:

```
public static boolean condizione1(MultiSet m, Object z) {  
    //REQUIRES: m<> null, z<> null  
    //EFFECTS: restituisce .....?  
    int i = m.molt(z);  
    int j = m.molt(z); m.insert(z); //NB: errore in originale  
    return (i==j-1);  
}  
public static boolean condizione2(MultiSet m, Object z) {  
    //REQUIRES: m<> null, z<> null  
    //EFFECTS: restituisce .....?  
    m.insert(z); //NB: errore in originale  
    int j = m.molt(z);  
    int i = m.molt(z);  
    return (i==j-1);  
}
```

calcolare qual è l'effetto della chiamata dei due metodi (cioè valore ritornato e side effects), per ogni oggetto z di tipo Object e per ogni MultiSet m, sotto l' ipotesi che l'implementazione di MultiSet sia corretta.

Entrambi i metodi inseriscono m in z, ma il primo restituisce sempre true, il secondo sempre false,

b) Completare, con AF e RI e il codice del metodo molt, la successiva implementazione della classe MultiSet, che utilizza la seguente classe Set (di cui è assegnata solo la specifica).

```
public class Set {  
    // OVERVIEW: insiemi di oggetti di tipo Object. Il valore null non fa parte dell'  
    // insieme :per es.: {A,B, C } dove A, B, C sono valori di tipo Object.  
    // Usa equals per confrontare gli oggetti e non introdurre duplicati.  
    //costruttori:  
    public Set(); //EFFECTS: inizializza this come insieme vuoto
```

```

    //metodi:
    public void insert(Object x);
        //MODIFIES: this
        //EFFECTS: this._post = this + {x}
    public void remove(Object x);
        //MODIFIES: this
        //EFFECTS: this._post = this - {x}
    public boolean isIn (Object x)
        //EFFECTS: restituisce true se x.equals elemento in this
    public Iterator elements()
        //EFFECTS: restituisce un iteratore a tutti gli elementi di this, in un ordine qualunque
        //REQUIRES: this non cambia mentre l'iteratore è in uso
    }

public class MultiSet {
    //scrivere qui la funzione di astrazione, definita in base al valore tipico descritto nella specifica, e
    preceduta da eventuali definizioni utili:
    .....
    .....
    .....
/Sia c un generico oggetto concreto, con c.ins={c1, .., cn}, dove ci e' una Coppia
// Allora:
//AF(c)= c1.obj..... c1.obj c2.obj ..... c2.obj ..... cn.obj .... cn.obj
//           |-----| |-----| |-----|
//           c1.i volte      c2.i volte      cn.i volte
//Una definizione più formale è invece la seguente:
//Definizioni: z^n sia z z ...z (z concatenato n volte) per ogni Object z;
// per ogni c oggetto concreto Multiset e x oggetto di tipo Coppia,
// c' = c - x sia definito come: c'.ins=c.ins.remove(x),
// per ogni oggetto concreto c, detto x un qualunque valore in c.ins,
//AF(c)= (x.obj)^{x.i} AF(c - x)
    .....
    //rep:
    protected Set ins;
    //Rl=c.ins<>null &.
        c.ins contiene solo oggetti di tipo Coppia &&
        forall x in c.ins, x.i>0
        forall x!=y in c.ins, if c.ins.isIn(x) && c.ins.isIn(y) ==> x.obj!=y.obj&&
    .....
    private class Coppia {
        //OVERVIEW: Tipo immutabile rappresentante una coppia (Object,int)
        //rep di Coppia:
        Object obj;
        int i;
        Coppia(Object o, int m) {obj=o; i= m;}
        .....
        //qui definire altri metodi se ritenuti necessari
        .....
occorre ridefinire equals:
        public boolean equals(Object z) {
            if (!(z instanceof Coppia)) return false;
            return equals((Coppia) z); }
        }
        public boolean equals(Coppia c) { return (obj== c.obj && i == c.i);}
    .....
}
public MultiSet(){ins = new Set();}
//metodi:
public MultiSet insert(Object x){
    for (Iterator itr =ins.elements(); itr.hasNext ();) {
        Coppia c = (Coppia) itr.next();

```

```

        if (c.obj.equals(x)) {
            ins.remove(c);
            ins.insert(new Coppia(c.obj,c.i+1));
            return this;
        }
    }
    ins.insert(new Coppia(x,1)); return this;
}
public void remove(Object x){
    for (Iterator itr =ins.elements(); itr.hasNext();) {
        Coppia c = (Coppia) itr.next();
        if (c.obj.equals(x)) {
            ins.remove(c);
            if (c.i>1) ins.insert(new Coppia(c.obj,c.i-1));
            return;
        }
    }
}
public int molt(Object x) {
    .....
    for (Iterator itr =ins.elements(); itr.hasNext();) {
        .....
        if (c.obj.equals(x))
            return c.i;
    }
    return 0;
}
.....
}
}

```

Coppia c = (Coppia)

c) Dimostrare che il rep invariant RI trovato è effettivamente un invariante per la classe MultiSet;

Giustifichiamo ciascuna delle quattro clausole del RI

- il costruttore istanzia un oggetto di tipo set
- insert e remove inseriscono in ins solo oggetti di tipo coppia
- le coppie inserite in ins dal metodo insert hanno i = 1, oppure un i piu' alto di una coppia che era già in c.ins; le coppie inserite dal metodo remove hanno un i piu' basso di 1 ma solo se la coppia ha un i > 1, quindi esse hanno un i > 1-1 = 0.
- prima di inserire una coppia, insert e remove eliminano la coppia contenente lo stesso oggetto da ins. Ovviamente non potra' quindi mai esserci piu' di una coppia con lo stesso oggetto.

d) Un programmatore inesperto, pensando di semplificare il codice, implementa la seguente remove

```
public void remove(Object x){  
    for (Iterator itr =ins.elements(); itr.hasNext ();) {  
        Coppia c = (Coppia) itr.next();  
        if (c.obj.equals(x)) {  
            ins.remove(c);  
            ins.insert(new Coppia(c.obj,c.i-1));  
            return;  
        }  
    }  
}
```

Dimostrare che la proprietà RI in questo caso non è più garantita.

Viola la condizione che c.i sia >0

PROVA 2: Esercizio 2.1 (3 punti)

Si consideri il seguente frammento di programma Java:

```
class A .... {  
    public synchronized int f (B b) {  
        b.g (...);  
    }  
    ...  
}  
  
class B .... {  
    public synchronized void g (A a) {  
        a.f (...);  
    }  
    ...  
}
```

Si supponga inoltre che esistano due threads t1 e t2. Il thread t1 contiene la seguente istruzione:

 pippo.f (pluto);

Il thread t2 contiene invece la seguente istruzione:

 pluto.g (pippo);

dove pippo e pluto sono oggetti delle classi A e B rispettivamente.

Il programma puo' generare un comportamento anomalo (chiamato *deadlock*) in cui il thread t1 blocca al thread t2 l'accesso all'oggetto pippo e, al tempo stesso t2 blocca a t1 l'accesso all'oggetto pluto, cosi' che nessuno dei due thread puo' ulteriormente procedere. Spiegare chiaramente perche' mostrando un esempio possibile di sequenze di eventi a runtime che dia luogo a un deadlock.

Si supponga che

- (1) t1 invochi f sull'oggetto riferito da pippo, e immediatamente dopo
- (2) t2 invochi g sull'oggetto riferito da pluto.

Siccome (1) effettua un lock sull'oggetto riferito da pippo, il task t2 non puo' procedere ad eseguire l'invocazione del metodo f; analogamente, t1 non puo' procedere ad eseguire l'invocazione del metodo g perche' l'oggetto sul quale operare e' bloccato da (2) che ha effettuato il lock. Pertanto nessuno dei due thread puo' procedere nella computazione.

PROVA 2: Esercizio 2.2 (3 punti)

Parte 1

Si supponga che la specifica di un metodo sia la seguente:

```
int metodoTest( int x) {  
    //EFFECTS: se x >= 10 restituisce la cifra meno significativa del numero  
    //Se 0<=x<=9 restituisce il valore di x  
    //Se x<0 restituisce -1  
}
```

Si definisca un insieme di dati di test secondo una strategia “black box”, fornendo una sintetica motivazione dei dati proposti.

Si scelgano i valori ai confini dei domini: x=9, x=10, x=0, x=1 e i valori all'interno dei domini: x=17, x=5, x=-9.

Parte 2

Si supponga di voler testare il metodoTest nell'ipotesi che non contenga cicli, ma solo istruzioni collegate in sequenza ed if-then-else. A tale proposito, si consideri una strategia di test (detta di copertura delle diramazioni) in cui ogni uscita vera e falsa degli if risulta eseguita da almeno un dato di test. Si consideri la seguente affermazione:

Se un metodo non contiene cicli ma solo istruzioni collegate in sequenza e mediante if-then-else allora il criterio di copertura delle diramazioni coincide con il criterio di copertura di tutti i cammini.

E' vera l'affermazione? Se SI, dare una sintetica dimostrazione. Se NO, fornire un controesempio di possibile metodo f.

NO: p. es. il seguente programma:

```
if (a==1) then B else C  
if (b==1) then D else E
```

per la copertura delle dimazioni bastano due dati di test: p.es. (1,1) e (0,0).

Per la copertura dei cammini occorrono quattro dati (p.es. (1,1) (0,1) (1,0) (0,0))

PROVA 2: Esercizio 2.3 (7 punti)

Si considerino le seguenti specifiche di una classe astratta Figura e di due classi concrete Triangolo e Rettangolo che la estendono:

```
public abstract class Figura {  
    public abstract void disegna() //EFFECTS: disegna la figura sullo schermo  
}  
public class Triangolo extends Figura{  
    public Triangolo (float angolo1, angolo2, lato)  
        // un triangolo è determinato da due angoli e dalla dimensione del lato compreso.  
        public float angolo1();//EFFECTS: restituisce il valore dell'angolo più a sinistra di this  
        public float angolo3();//EFFECTS: restituisce il valore dell'angolo più a destra di this  
        public float angolo2();//EFFECTS: restituisce il valore dell'angolo intermedio di this  
        public void disegna() //EFFECTS: disegna un triangolo sullo schermo  
}  
  
public class Rettangolo extends Figura {  
    public Rettangolo (float altezza, float base)  
        public float base() //EFFECTS: restituisce il valore della base di this  
        public float altezza() //EFFECTS: restituisce il valore dell'altezza di this  
        public void disegna() //EFFECTS: disegna un rettangolo sullo schermo  
}
```

Si desidera ora definire un contenitore di oggetti di tipo Figura, con il vincolo che il contenitore sia omogeneo, nel senso che contenga solo figure “simili” fra loro. Il concetto di somiglianza è stabilito in modo particolare per ogni Figura. Due oggetti di tipo Triangolo sono simili se gli angoli interni corrispondenti sono uguali. Due rettangoli sono simili se hanno lo stesso rapporto base/altezza.

Si desidera utilizzare un approccio “related subtype”.

```
interface Verificatore {  
    //OVERVIEW: tipo immutabile che confronta la somiglianza di due figure... ... ...  
    public boolean simili(Figura f1, Figura f2) throws ClassCastException, NullPointerException;  
        //EFFECTS: se f1 e f2 non sono di tipi confrontabili, ClassCastException  
        // altrimenti true sse f1 è simile a f2 }  
public class NotSimilarException {...}  
  
public class ContenitoreOmogeneo{  
    // OVERVIEW: Un ContenitoreOmogeneo è un insieme illimitato di oggetti di tipo Figura, tutti  
    // simili fra loro. Usa un Verificatore per determinare la somiglianza geometrica degli elementi.  
    public ContenitoreOmogeneo (Figura f, Verificatore v) // costruttore  
        //EFFECTS: this è inizializzato in modo da contenere la Figura f.  
        // v diventa il verificatore da usare per i test di somiglianza successivi.  
    public void inserisci (Figura x) throws NullPointerException, NotSimilarException {...}  
        // MODIFIES this  
        // EFFECTS se x è null lancia NullPointerException  
        // altrimenti se x è geometricamente simile alle figure già inserite inserisce x in this  
        // altrimenti lancia NotSimilarException  
    public void elimina (Figura x)  
        // MODIFIES this  
        // EFFECTS se x è null lancia NullPointerException  
        // altrimenti elimina la figura equals a x in this  
}
```

- Scrivere la specifica e l’implementazione di classi opportune che consentano l’inserimento e la rimozione di oggetti di tipo Triangolo e di tipo Rettangolo in oggetti di tipo ContenitoreOmogeneo, utilizzando l’approccio Related Subtype.

```

Class TriangVer implements Verificatore {
    public boolean simili(Figura f1, Figura f2) throws ClassCastException, NullPointerException;
        //EFFECTS: se f1 e f2 non sono di tipi confrontabili, ClassCastException
        if (!(f1 instanceof Triangolo) || !(f2 instanceof Triangolo)) throw new
ClassCastException();
        Triangolo t1 = (Triangolo) f1; Triangolo t2 = (Triangolo) f2;
        return (t1.angoloMin() == t2.angoloMin() && t1.angoloMax() == t2.angoloMax());
    }
//basta il costruttore di default
}

Class RettVer implements Verificatore {
    public boolean simili(Figura f1, Figura f2) throws ClassCastException, NullPointerException;
        //EFFECTS: se f1 e f2 non sono di tipi confrontabili, ClassCastException
        if (!(f1 instanceof Rettangolo) || !(f2 instanceof Rettangolo)) throw new
ClassCastException();
        Rettangolo r1 = (Rettangolo) f1; Rettangolo r2 = (Rettangolo) f2;
        return (r1.base/r1.altezza() == r2.base()/r2.altezza());
    }
//basta il costruttore di default
}

```

b) Completare poi il codice seguente (tralasciando le dichiarazioni di import):

```

class Prova {
    void main(String args[])
        Figura f1 = new Triangolo(60,30,5);
        Figura f2 = new Rettangolo(10,8);
        ContenitoreOmogeneo c1, c2;
    //inizializzare qui c1 con f1:
    c1 = new ContenitoreOmogeneo(f1, new TriangVer()); .....
    //inizializzare qui c2 con f2:
    c2 = new ContenitoreOmogeneo(f2, new RettVer());.....
}

```

}

c) Qual è il presumibile effetto dell'esecuzione, come ultima linea del metodo main(), dell'istruzione c1.inserSimile(f2)?

Presumibilmente, il metodo inserSimile di c1 cercherà di chiamare il metodo simili del proprio Verificatore: questo provocherà un ClassCastException. NB: non essendo definita l'implementazione di ContenitoreOmogeneo, altre soluzioni ragionevoli possono essere accettabili, purché motivate.



Politecnico di Milano

Anno accademico 2013-2014

Ingegneria del Software – Appello del 28 giugno 2013

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si considerino le seguenti dichiarazioni:

```
public abstract class Point {  
    public abstract double distance (Point p);  
}  
  
public class Point1D extends Point {  
    private double c1;  
  
    public Point1D (double c1) {  
        this.c1 = c1;  
    }  
  
    public double getC1() {  
        return c1;  
    }  
  
    public double distance (Point p) {  
        return Math.abs(((Point1D) p).getC1()-c1);  
    }  
}  
  
public class Point2D extends Point1D {  
    private double c2;  
  
    public Point2D (double c1, double c2) {  
        super(c1);  
        this.c2 = c2;  
    }  
  
    public double getC2() {  
        return c2;  
    }  
  
    public double distance (Point p) {  
        return Math.sqrt(Math.pow(((Point2D) p).getC1()-this.getC1(), 2) +  
                        Math.pow(((Point2D) p).getC2()-this.getC2(), 2));  
    }  
  
    public double distance (Point2D p) {  
        return Math.sqrt(Math.pow(p.getC1()-this.getC1(), 2) +  
                        Math.pow(p.getC2()-this.getC2(), 2));  
    }  
}
```

Domanda 1

Quali dei due metodi distance della classe Point2D rappresenta un caso di overloading e quale invece rappresenta un caso di overriding? **Soluzione:** Il primo è overriding, il secondo overloading.

Domanda 2

La classe Point2D è un sottotipo della classe Point1D in base al principio di sostituibilità? Motivare brevemente la risposta.
Soluzione: I metodi non sono specificati in JML; tuttavia, essendo presente una implementazione, si può considerare il comportamento previsto dalla implementazione per ragionare sul principio di sostituibilità.

La classe Point2D non rispetta il principio di sostituibilità in quanto il metodo overridden distance ha un comportamento normale (non eccezionale) solo se il parametro p è instanceof Point2D. Se p è instanceof Point1D, ma non Point2D, il metodo

lancia una ClassCastException. Questo comportamento non è compatibile con quello di distance in Point1D, che invece ha un comportamento non eccezionale in entrambi i casi. Viene pertanto violata la regola dei metodi.

Domanda 3

Dopo la seguente sequenza di dichiarazioni:

```
Point p;  
Point1D p1;  
Point2D p2;
```

segnare a fianco di ciascuno dei seguenti assegnamenti se essi sono considerati corretti dal compilatore Java in base alle regole di tipo:

```
p = p1;  
  
p = p2;  
  
p1 = p;  
  
p1 = p2;  
  
p2 = p1;
```

Domanda 4

Si consideri il seguente frammento e si pieghi sinteticamente a parole l'effetto di ciascun assegnamento, indicando quali metodi vengono invocati in ciascun caso e scrivendo il risultato calcolato (valore stampato a video).

```
Point p1 = new Point1D(0.0);  
Point p2 = new Point1D(1.0);  
Point p3 = new Point2D(0.0, 1.0);  
Point p4 = new Point2D(1.0, 0.0);  
double x;  
  
x = p1.distance(p2);  
System.out.println(x);  
  
x = p3.distance(p4);  
System.out.println(x);  
  
x = p1.distance(p3);  
System.out.println(x);  
  
x = p3.distance(p1);  
System.out.println(x);  
  
p1=p3;  
x = p2.distance(p1);  
System.out.println(x);  
  
x = p4.distance(p1);  
System.out.println(x);
```

Soluzione:

```
Point1D.distance  
1.0  
Point2D.distance  
1.4142135623730951  
Point1D.distance  
0.0  
Point2D.distance
```

```
Exception in thread "main" java.lang.ClassCastException: Point1D cannot be cast to Point2D
```

Se poi l'esecuzione proseguisse:
Point1D.distance
1.0
Point2D.distance
1.4142135623730951

Domanda 5

Si supponga di introdurre una classe Point3D sottoclasse di Point2D che definisce i punti in uno spazio a 3 dimensioni. Si esamini questo frammento di codice:

```
Point p1 = new Point3D(1.0, 1.0, 1.0);  
Point2D p2 = new Point2D(0.0, 0.0);  
  
p2.distance(p1);
```

spiegare sinteticamente che cosa succede durante l'esecuzione in corrispondenza della chiamata di *distance* e quale valore viene calcolato. **Soluzione:** chiama Point2D.distance e calcola la distanza fra (1.0, 1.0) e (0.0, 0.0).

Domanda 6

Si supponga di aggiungere la seguente precondizione al metodo distance della classe Point1D:

```
//@ requires p instanceof Point1D;
```

La classe Point2D sarebbe considerabile sottotipo di Point1D in base al principio di sostituibilità? Motivare sinteticamente la risposta. **Soluzione:** Ammettendo che il principio di sostituzione valesse senza la precondizione, la precondizione non violerebbe la regola dei metodi.

Domanda 7

Si supponga che, oltre alla precedente modifica della domanda 6, si aggiunga anche la seguente precondizione al metodo **distance** (**Point p**) della classe Point2D:

```
//@ requires p instanceof Point2D;
```

La classe Point2D sarebbe considerabile sottotipo di Point1D in base al principio di sostituibilità? Motivare sinteticamente la risposta. **Soluzione:** Ammettendo che il principio di sostituzione valesse senza la precondizione, la precondizione violerebbe la regola dei metodi, in quanto sarebbe piu' stringente della precondizione originale di Point1D.

Domanda 8

E se, per assurdo, nel metodo distance di Point1D si introducesse la precondizione:

```
//@ requires p instanceof Point2D;
```

e nel metodo **distance (Point p)** della classe Point2D si introducesse:

```
//@ requires p instanceof Point1D;
```

la classe Point2D sarebbe considerabile sottotipo di Point1D in base al principio di sostituibilità? Motivare sinteticamente la risposta. **Soluzione:** Ammettendo che il principio di sostituzione valesse senza le precondizioni, questo caso non violerebbe la regola dei metodi.

Domanda 9

Si considerino le seguenti dichiarazioni che fanno riferimento alle dichiarazioni di classe fornite inizialmente (e cioè senza precondizioni):

```
ArrayList<Point1D> l1;  
ArrayList<Point2D> l2;
```

Quali dei seguenti assegnamenti corretto secondo il compilatore Java, e perché?

```
11 = 12;
```

```
12 = 11;
```

Soluzione: Entrambi i casi sono considerati scorretti dal compilatore Java.

Esercizio 2

Si supponga di voler specificare una classe `Line`, che offre un metodo osservatore `points` che restituisce un `ArrayList<Point2D>`. Per semplicità, si ipotizzi che la classe rappresenti solo rette oblique.

1. Si specifichi un metodo `lineUp` che restituisce una nuova `Line` i cui punti sono ordinati per ascisse crescenti. La precondizione afferma che i punti dell'`ArrayList` appartengono a una retta; la postcondizione afferma che nell'oggetto restituito i punti memorizzati in posizioni successive dell'`ArrayList` si trovano in posizioni successive della retta. **Soluzione:** La signature del metodo non è specificata. Diverse soluzioni sono possibili; per esempio, un producer che parta da un oggetto `Line`, oppure un creator che parta da un `ArrayList` di punti. Il fatto che i punti siano in linea può essere specificato dicendo che esiste un coeff. angolare m e un termine noto q per cui tutti i punti p sono disposti lungo la retta individuata da m e q :

```
p.getc2() == m*p.getc1() + q
```

```
.
```

La specifica per un producer diventa:

```
/*@ requires points != null && points.size() >= 2 &&
@ (\exists double m; !Double.isNaN(m) && !Double.isInfinite(m));
@ (\exists double q; !Double.isNaN(q) && !Double.isInfinite(q));
@ (\forall Point2D p; p != null && points.contains(p);
@   p.getc2() == m*p.getc1() + q));
@ ensures \result != null &&
@ \result.containsAll(points) && points.containsAll(\result) &&
@ (\forall int i; 0 <= i && i < \result.size() - 1;
@   \result.points().get(i).getc1() < \result.points().get(i+1).getc1());
@*/
public static /*@ pure */ Line lineUp(ArrayList<Point2D> points);
```

dove `isNaN`, `isInfinite` sono utilizzati per evitare valori speciali dei `Double`. Si usa il “ \exists ” per confrontare i valori dell’ascissa in quanto le rette sono per ipotesi (che si può immaginare garantita dal costruttore) oblique.

Se si definisse il metodo con la signature di un creator:

```
public /*@ pure */ Line lineUp();
```

basta riscrivere la specifica precedente con `this.points()` al posto di `points`.

In questo caso, in realtà, la proprietà di essere allineati dovrebbe essere garantita dall’invariante pubblico di `Line`.

2. Si fornisca la specifica del metodo al punto precedente eliminando la precondizione e sollevando invece al suo posto un’eccezione `NoLineException`. **Soluzione:**

```
/*@ requires true;
@ ensures points != null && points.size() >= 2 &&
@ (\exists double m; !Double.isNaN(m) && !Double.isInfinite(m));
@ (\exists double q; !Double.isNaN(q) && !Double.isInfinite(q));
@ (\forall Point2D p; p != null && points.contains(p);
@   p.getc2() == m*p.getc1() + q));
@ && \result != null && //qui riprende la ensures precedente
@ \result.containsAll(points) && points.containsAll(\result) &&
@ (\forall int i; 0 <= i && i < \result.size() - 1;
@   \result.points().get(i).getc1() < \result.points().get(i+1).getc1());
@ signals (NoLineException e) points == null || points.size() < 2 ||
@ !(\exists double m; !Double.isNaN(m) && !Double.isInfinite(m));
@ (\exists double q; !Double.isNaN(q) && !Double.isInfinite(q));
@ (\forall Point2D p; this.points().contains(p);
@   p.getc2() == m*p.getc1() + q))*/
public static Line lineUp(ArrayList<Point2D> points) throws NoLineException;
```

3. Con riferimento al punto precedente, si consideri una sottoclasse di `Line` in cui si voglia ridefinire il metodo `lineUp` nel modo seguente:

- la precondizione, che corrisponde sia alla precondizione che alla postcondizione del caso precedente, richiede che i punti appartengano a una retta e i punti in posizioni successive dell'ArrayList siano consecutivi sulla retta,
- la postcondizione specifica che l'oggetto contenga in successione solo i punti consecutivi della retta che hanno coordinate positive.

Si chiede di definire la precondizione e la postcondizione e di discutere sinteticamente se la ridefinizione soddisfa il principio di sostituibilità.

Soluzione: Un modo per esprimere la condizione è il seguente. Dato un punto positivo nella lista dei punti,

- se il suo successivo è ancora positivo allora i due punti sono ordinati;
- altrimenti se il successivo non è positivo allora anche tutti i seguenti non sono positivi.

```
/*@ requires points.size() >= 2 &&
@ (\exists double m; !Double.isNaN(m) && !Double.isInfinite(m));
@ (\exists double q; !Double.isNaN(q) && !Double.isInfinite(q));
@ (\forall Point2D p; points.contains(p);
@   p.getC2() == m*p.getC1() + q))
@ &&
@ (\forall int i; 0 <= i < points.size() - 1;
@   points.get(i).getC1() <= points.get(i+1).getC1());
@ ensures \result != null &&
@ \result.containsAll(points) && points.containsAll(\result) &&
@ (\forall int i; 0 <= i < \result.size() - 1;
@   (\result.points().get(i)>0) => (
@     (\result.points().get(i+1)>0) &&
@     \result.points().get(i).getC1() < \result.points().get(i+1).getC1())
@   ||
@   ((\result.points().get(i+1)<=0) &&
@   \forall int k; i+1 < k < \result.points().size();
@   (\result.points().get(k)<=0))));
```

```
*/
public Line lineUp(ArrayList<Point2D> points);
```

La nuova precondizione, detta pre_{sub} è più forte di quella del metodo originale, detta pre_{super} , quindi viene violato il principio di sostituzione.

La nuova postcondizione $post_{sub}$ è invece più debole di quella originale $post_{super}$, che imponeva a tutti i punti anche con coordinate non entrambe positive di essere ordinate nella linea. La violazione del principio di sostituzione è comunque ulteriormente evidente, in quanto viene violata la regola completa della postcondizione: non è vero che $pre_{super} \Rightarrow (post_{sub} \Rightarrow post_{super}$, in quanto non solo non vale $post_{sub} \Rightarrow post_{super}$, ma $post_{sub}$ può non essere nemmeno definita quando vale solo pre_{super} .

Esercizio 3

Si consideri il seguente frammento di programma C:

```
for (n=0; n<max_size && (c=getc(yyin)) != EOF && c != '\n'; ++n)
buf[n] = (char) c;
```

1. Supponendo di voler coprire tutti i **branch**, qual'è il numero minimo di test che devono essere eseguiti? Giustificare la risposta. **Soluzione:** Il codice presenta un solo branch in corrispondenza del ciclo for per determinare se si deve uscire dal ciclo o se deve essere eseguita un'altra iterazione.

Basta quindi un solo caso di test per cui le condizioni di ingresso nel ciclo sono verificate almeno una volta. Questo caso di test copre sia l'ingresso sia l'uscita dal ciclo che avverrà al più dopo `max_size` iterazioni.

2. Supponendo di voler anche coprire tutte le **condizioni**, quanti e quali test dovrebbero essere eseguiti? Specificare esattamente i casi di test, con i valori delle variabili

Soluzione: Chiamando C1 la condizione `n < max_size`, C2 la condizione `c != EOF` e C3 la condizione `c != '\n'`, per poter coprire tutte le condizioni servono i seguenti casi di test.

- C1 = False
- C1 = True && C2 = False

- C1 = True && C2 = True && C3 = False
- C1 = True && C2 = True && C3 = True

L'operatore `&&` in C, come in Java, è short circuited, quindi le altre possibili combinazioni delle condizioni non sono necessarie. `getc` una funzione C che permette di leggere un carattere da un file, quindi i casi di test devono indicare il valore della variabile `max_size` e il contenuto del file cui fa riferimento `yyin`.

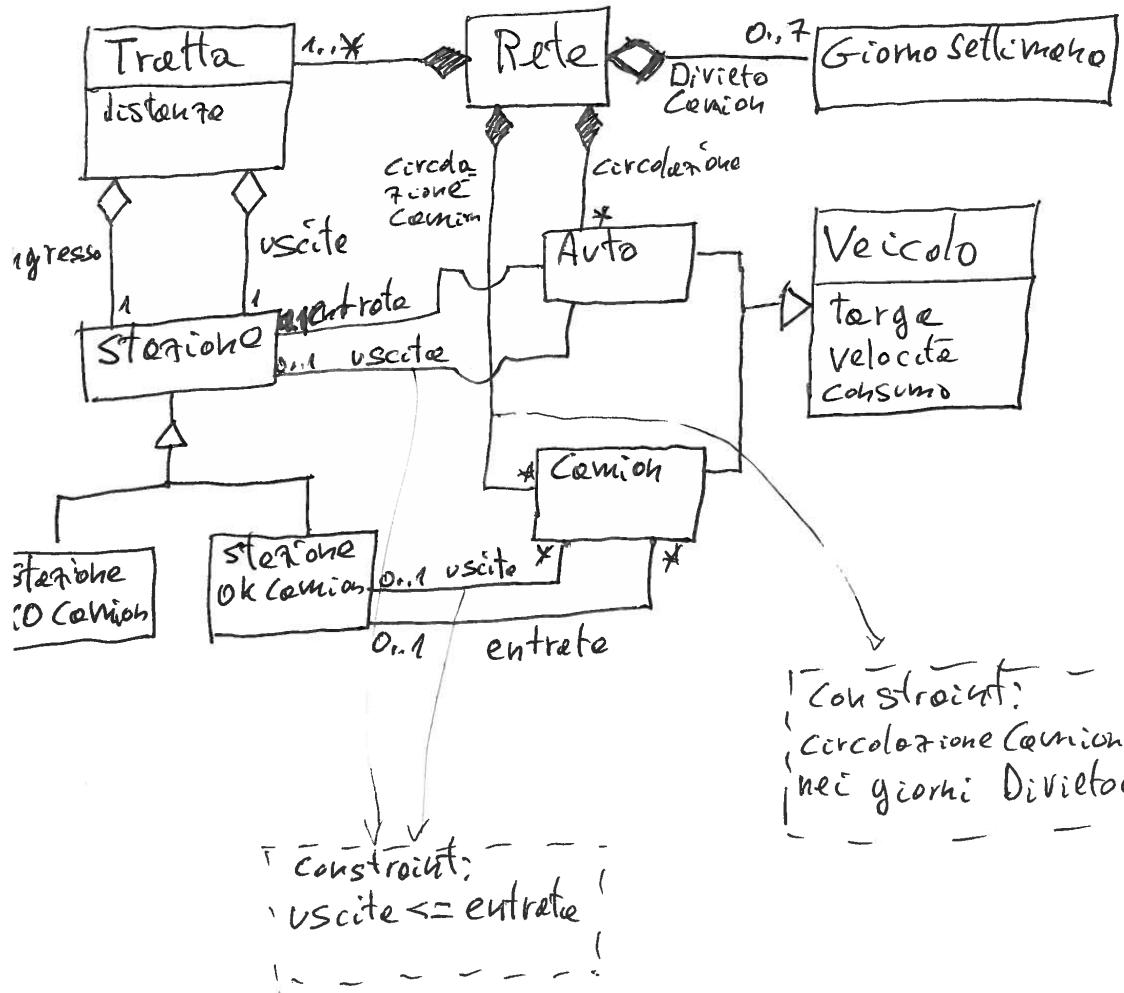
Dei possibili valori delle variabili per i diversi casi sono rispettivamente:

- `max_size = 0`
- `max_size = 1, yyin = riferimento a un file vuoto`
- `max_size = 1, yyin = riferimento a un file non vuoto, il cui prossimo carattere disponibile per la lettura è '\n'`
- `max_size = 1, yyin = riferimento a un file non vuoto, il cui prossimo carattere disponibile per la lettura è 'a'`

Esercizio 4

Si vuole progettare un programma di simulazione di una semplice rete autostradale. La rete è composta da stazioni di ingresso/uscita e tratte, che collegano due stazioni e che hanno una certa distanza. Un veicolo, da un punto di vista della simulazione, è caratterizzato da un tragitto, inteso come coppia stazione di ingresso, stazione di uscita, da una velocità e da un consumo medio di carburante per km. percorso. I veicoli possono essere automobili o camion. Questi ultimi possono avere dei divieti di circolazione in certi giorni della settimana. Alcune stazioni di ingresso/uscita possono essere vietate ai camion.

1. Specificare un diagramma delle classi che descrive questo sistema.
2. Si descriva un diagramma di sequenza che descrive il fatto che un veicolo per entrare in autostrada da una stazione deve ottenere il biglietto di ingresso e che arrivato alla stazione deve sottomettere prima il biglietto di ingresso e poi il pagamento relativo per poter uscire.
3. Si supponga di voler estendere il modello di simulazione per tener conto anche dei rifornimenti di benzina. Si usi uno Statechart per modellare il seguente comportamento. Una pompa di benzina funziona come segue: bisogna inserire la carta di credito e poi scegliere se si vuole prepagare l'importo (per esempio 30 EURO) o se si paga a consuntivo fino al pieno. In entrambi i casi, bisogna staccare la pompa e al termine dell'erogazione riattacarla. Nel caso di importo pagato a consuntivo, l'azione di riempimento del serbatoio è segnalata da un evento esterno (la pompa che rileva il pieno). Nel caso di prepagato invece l'azione ha una durata prefissata che non deve essere modellata.



Soluzione: Mostriamo solo il diagramma UML, le altre risposte essendo ovvie.



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da Vinci,

32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Recupero di Ingegneria del Software – 10 Settembre 2003

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno presi in considerazione.
3. È possibile consultare liberamente libri, manuali o appunti, e scrivere a matita. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h30.
6. Punteggio totale a disposizione: 26 punti. La prova è sufficiente ottenendo almeno 16 punti.

Esercizio 1 (13 punti)

Si vuole progettare un'astrazione sui dati, mediante una classe Java chiamata DataOraria, che corrisponda alla nozione di data (anno, mese e giorno) con in più l'indicazione dell'ora (da 0 a 23). Si sceglie di codificare ogni oggetto della classe DataOraria usando un singolo numero intero, sfruttando il fatto che gli int di Java hanno valore compreso tra -2147483648 e +2147483647: quindi si sfruttano le prime quattro cifre della rappresentazione decimale per l'anno, le successive due per il mese, le due seguenti per il giorno e le ultime due per l'ora. A esempio il 10 settembre 2003, ore 15, viene codificato col numero int 2003091015. Si conviene inoltre che astrattamente un esemplare di DataOraria sia rappresentato dalla stringa “gg/mm/aaaaxx:hh”, dove il due caratteri xx sono dc per le date con anno positivo e ac per le date con anno negativo; nell'esempio precedente la rappresentazione astratta del 10 settembre 2003, ore 15 è quindi “10/09/2003dc:15”.

a) Formalizzare l'invariante di rappresentazione per tale astrazione e codificarlo con il metodo REPOK; nel fare ciò per semplicità si trascurino gli aspetti relativi agli anni bisestili.

b) Codificare la funzione di astrazione mediante il metodo `toString`.

c) Completare la definizione della classe codificando i metodi giornoSucc e spostaAvanti

```
public class DataOraria {  
    // DataOraria è mutabile  
    private int d; // il rep  
    public boolean bisestile() { // da NON implementare}  
    public void giornoSucc() { // modifica this spostandolo in avanti di 1 giorno, senza cambiare l'ora  
  
    }  
    public void spostaAvanti (int nGiorni) { // sposta in avanti this di un certo numero di giorni,  
    // senza cambiare l'ora  
    }  
}
```

d) Dimostrare che l'invariante di rappresentazione viene effettivamente mantenuto nell'applicazione dei metodi della classe.

e) Considerando il metodo booleano **bisestile**, si assuma che la nozione di anno bisestile sia definita informalmente come segue: un anno è bisestile e se solo se è multiplo di 4 ma non di 100, oppure è multiplo di 400. Sulla base di tale definizione si indichi un insieme di dati per il test funzionale del metodo.

f) Si assuma che il metodo **bisestile** sia implementato come segue.

```
public boolean bisestile() {  
    int a = d / 1000000;  
    if (a % 400 == 0)  
        if (a % 100 == 0)  
            return true;  
        else return false;  
    else if (a % 100 == 0)  
        return false;  
    else if (a % 4 == 0)  
        return true;  
    else return false;  
}
```

Indicare se l'implementazione sopra riportata è funzionalmente corretta, se cioè restituisce il risultato giusto in tutti i casi in cui l'oggetto a cui è applicata è accettabile. Motivare adeguatamente la risposta.

g) È possibile definire un insieme di dati di test a supporto del testing strutturale che permetta di coprire tutti i cammini del metodo **bisestile** sopra definito? Se sì, fornire un esempio di tale insieme di dati, altrimenti spiegare il motivo per cui ciò non è possibile e fornire dei dati di test che coprano un insieme massimale di cammini.

Esercizio 2 (5 punti)

In un'applicazione di commercio elettronico, gli utenti (caratterizzati da un proprio user_name e password) hanno un carrello della spesa (shopping cart) che funge da raccoglitore delle merci che vengono via via selezionate durante una sessione di spesa. Ogni merce ha un proprio codice e un prezzo unitario. Le merci possono essere di due tipi: nuova o usata. Diversamente da una merce usata, una merce nuova ha (oltre al prezzo) un valore di IVA.

- 1) descrivere mediante un diagramma delle classi UML le entita' e le associazioni logiche tra di esse che si formalizzano la precedente descrizione a parole;
- 2) produrre un sequence diagram di UML che descrive uno scenario di acquisto on-line. Nello scenario scelto, l'utente innanzitutto comunica a un "login manager" il proprio user_name e la propria password. Ipotizzando la situazione in cui questi sono accettati dal "login manager", l'utente inserisce innanzitutto due copie della merce A nel carrello, poi una copia della merce B, poi ancora elimina una copia della merce A dal carrello e infine si presenta alla cassa per acquistare la merce raccolta nel carrello. A questo la cassa chiede al cliente il numero di carta di credito e la sua scadenza, che vengono controllati nella loro validita' chiedendo a un sottosistema "gestione carte di credito". Ipotizzando nello scenario che il controllo dia un esito favorevole, la procedura di acquisto on-line termina con successo.

Esercizio 3 (8 punti)

È data la seguente specifica di una classe VeicoloAMotore

```
abstract public class VeicoloAMotore {  
    abstract public void accendi()  
        //REQUIRES: !acceso  
        //EFFECTS: acceso_post == true  
    abstract public void spegni()  
        //REQUIRES: acceso  
        //EFFECTS: acceso_post == false  
    abstract public boolean acceso()  
        // EFFECTS: restituisce true se, e solo se, il veicolo è acceso  
}
```

a) È data la seguente specifica di una classe Automobile.

```
public class Automobile extends VeicoloAMotore{  
    public Automobile() //EFFECTS: costruisce un'Automobile non accesa  
    public void accendi() //REQUIRES: true  
        //EFFECTS: acceso_post == !acceso  
}
```

La classe Automobile soddisfa il principio di sostituzione di Liskov rispetto alla classe VeicoloAMotore? La risposta deve essere accuratamente motivata.

b) È data la seguente specifica di una classe AutomobileElettrica.

```
public class AutomobileElettrica extends Automobile{  
    public AutomobileElettrica()//EFFECTS: costruisce un'AutomobileElettrica accesa  
    public void accendi() //REQUIRES: !acceso  
        //EFFECTS: acceso_post == true  
}
```

La classe AutomobileElettrica soddisfa il principio di sostituzione di Liskov rispetto alla classe Automobile? La risposta deve essere accuratamente motivata.

c) Si consideri il seguente frammento di codice, che utilizza le classi sopra specificate, e si risponda alle domande inserite come commenti:

```
VeicoloAMotore a,b,c;  
a = new Automobile(); b = new AutomobileElettrica();  
c = b;  
b.accendi(); //quale metodo accendi() viene chiamato? .....  
c.accendi(); //quale metodo accendi() viene chiamato? .....  
c = a; //l'oggetto c è una copia di a? .....  
c.accendi(); //quale metodo accendi() viene chiamato? .....  
c.spegni(); //quale metodo spegni() viene chiamato? .....
```

SOLUZIONE DEL TEMA: 3 febbraio 2020



Politecnico di Milano

Anno accademico 2018-2019

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>	
Nome:	Matricola:	
Sezione (segnarne una): <input type="checkbox"/> Cugola	<input type="checkbox"/> Margara	<input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica di un database di tratte servite da una compagnia area.

```
public class FlightsDB {
    // \result==true sse esiste una tratta tra from e to
    public /*@ pure */ boolean directlyConnected(String from, String to);

    // \result==true sse esiste una tratta tra from e to oppure
    // una sequenza di tratte con scali intermedi tra from e to
    public /*@ pure */ boolean connected(String from, String to);

    // aggiunge la tratta al database
    // lancia InvalidArgumentException se to e from sono lo stesso aeroporto
    public void addRoute(String from, String to) throws InvalidArgumentException;

    // ritorna l'insieme delle destinazioni che sono raggiungibili da from con due
    // tratte consecutive (un solo scalo) ma non mediante una tratta diretta
    public /*@ pure */ Set<String> reachableWithOneStop(String from);
}
```

Domanda a)

Si specificino in JML i metodi connected, addRoute e reachableWithOneStop.

Soluzione

Definiamo

```
//@unchanged() = (\forall String x; x!=null;
//@           (\forall String y; y!=null && !x.equals(y));
//@           connected(x, y) <==> \old(connected(x, y)) &&
//@           directlyConnected(x, y) <==> \old(directlyConnected(x, y)) )

//@requires from!=null && to!=null
//@ensures \result <==> (directlyConnected(from, to) ||
//@  (\exists String x; x!=null; connected(from, x) && directlyConnected(x, to)) )
public /*@ pure */ boolean connected(String from, String to);

//@requires from!=null && to!=null
//@ensures !from.equals(to) && directlyConnected(from, to) &&
//@  (\forall String x; x!=null;
//@   (\forall String y; y!=null;
//@    directlyConnected(x, y) <==>
//@    (\old(directlyConnected(x, y)) || (x.equals(from) && y.equals(to)))) )
//@signals(InvalidArgumentException) from.equals(to) && unchanged()

public void addRoute(String from, String to) throws InvalidArgumentException;

//@requires from!=null
//@ensures !\result.contains(null) &&
//@  (\forall String x; x!=null && !x.equals(from); \result.contains(x) <==>
//@  (!directlyConnected(from, x) &&
//@   (\exists String y; y!=null && !y.equals(x);
//@    directlyConnected(from, y) && directlyConnected(y, x) ) ) )
public /*@ pure */ Set<String> reachableWithOneStop(String from);
```

Domanda b)

Si consideri la seguente implementazione (parziale) di `FlightsDB` che si basa su una mappa che associa a ogni aeroporto l'insieme degli aeroporti connessi mediante una tratta diretta.

```
public class FlightsDB {  
    private Map<String, Set<String>> connections = new HashMap<>();  
    public boolean directlyConnected(String from, String to) {  
        return connections.containsKey(from) && connections.get(from).contains(to);  
    }  
    public boolean connected(String from, String to) {  
        return directlyConnected(from, to) || (connections.containsKey(from) &&  
            connections.get(from).stream().anyMatch(x -> connected(x, to)));  
    }  
    public void addRoute(String from, String to) throws IllegalArgumentException {  
        if (from.equals(to)) throw new IllegalArgumentException();  
        if (!connections.containsKey(from)) connections.put(from, new HashSet<>());  
        connections.get(from).add(to);  
    }  
    public Set<String> reachableWithOneStop(String from) { ... }  
}
```

Si fornisca un invarianto di rappresentazione per tale implementazione.

Soluzione

```
//@private invariant connections!=null && !connections.containsKey(null) &&  
//@  (\forall String x; connections.containsKey(x);  
//@    connections.get(x)!=null && !connections.get(x).contains(null) &&  
//@    !connections.get(x).contains(x) )
```

Domanda c)

Si consideri una classe `NewFligthsDB` che permette di invocare il metodo `addRoute` con due parametri uguali senza lanciare eccezioni. La classe `NewFligthsDB` può essere definita come erede di `FligthsDB` rispettando il principio di sostituzione di Liskov? Si consideri invece una classe `RemovableFligthsDB` che aggiunge un metodo `remove` per rimuovere una tratta. La classe `RemovableFligthsDB` può essere definita come erede di `FligthsDB` rispettando il principio di sostituzione di Liskov? Motivare le risposte.

Soluzione

La classe `NewFligthsDB` non può essere definita come erede di `FligthsDB`, ad esempio perché non rispetterebbe la post-condizione eccezionale del metodo `addRoute`: un utilizzatore si attenderebbe un'eccezione che non viene sollevata.

Nemmeno la classe `RemovableFligthsDB` può essere definita come erede di `NewFligthsDB` perché viola la proprietà evolutiva che impedisce a una tratta di essere rimossa: un utilizzatore potrebbe stupirsi nel non ritrovare una tratta inserita.

Esercizio 2

Si consideri la seguente interfaccia Java. Il metodo `acquire` può essere invocato al più 10 volte prima di sospendere il chiamante. Il metodo `reset` re-inizializza l'oggetto, permettendo altre 10 invocazioni di `acquire` (incluse quelle già effettuate che erano in attesa al momento dell'invocazione di `reset` e vanno sbloccate).

```

public interface MultiLock {
    /* Sospende il chiamante se vi sono state più di 10 invocazioni precedenti
     * del medesimo metodo (dall'ultima invocazione di reset) */
    public void acquire() ;
    /* Re-inizializza l'oggetto permettendo altre 10 invocazioni di acquire
     * (incluse quelle già effettuate che erano in attesa) */
    public void reset() ;
}

```

Domanda a)

Si fornisca una implementazione per l'interfaccia MultiLock.

Soluzione

```

public class MultiLockImpl implements MultiLock {
    private int numAcquired;
    public MultiLockImpl() {
        numAcquired = 0;
    }
    public synchronized void acquire() {
        while(numAcquired>=10)
            try { wait(); }
            catch(Exception ex) {ex.printStackTrace();}
        numAcquired++;
    }
    public synchronized void reset() {
        numAcquired=0;
        notifyAll();
    }
}

```

Domanda b)

Si consideri il seguente frammento di codice che usa un oggetto di tipo MultiLock:

```

final MultiLock lock = new ...
int numAcquire = ...;
int numReset = ...;
for(int i=0; i<numAcquire; i++) {
    new Thread( () -> lock.acquire() ).start();
}
for(int i=0; i<numReset; i++) {
    new Thread( () -> lock.reset() ).start();
}

```

Quali valori sono ammissibili per le variabili numAcquire e numReset affinchè tutti i thread creati possano prima o poi procedere senza che nessuno rimanga sospeso?

Soluzione

Poichè non vi è garanzia sull'ordine di esecuzione dei thread creati dal frammento di codice, l'unica possibilità affinchè tutti possano procedere è che numAcquire sia minore o uguale a 10, indipendentemente dal valore di numReset (i thread che eseguono il metodo reset potrebbero partire tutti prima dei thread che eseguono acquire).

Esercizio 3

Si consideri il codice del seguente metodo, in cui ogni istruzione è numerata per comodità:

```
1 static int mystery(int x, int y) {
2     if (x>=0 && y>0) {
3         do {
4             if (x >= 4)
5                 y++;
6             else
7                 y--;
8             x--;
9         } while (x != 1);
10    }
11    if (y==0)
12        y=1;
13    return y;
14 }
```

Domanda a)

Qual è il numero minimo di dati di test necessari per coprire tutte le decisioni e le condizioni del programma? Si mostri un esempio, motivandone la minimalità.

Soluzione

Una volta entrati nel ciclo, ossia con $x \geq 0$ e $y > 0$, basta $x \geq 4$ per entrare nel ramo `then` dell'`if`. Il ciclo viene eseguito almeno altre tre volte, per uscire con $x == 1$, coprendo anche il ramo `else` dell'`if`. Ad esempio, basta il caso $(4, 1)$.

Occorre ancora coprire il caso false del primo `if`, rendendo falsa la prima condizione e rendendo falsa la seconda condizione. Serve dunque un caso di test con $x < 0$ e un caso di test con $y \leq 0$: ad esempio $(-1, 1)$, $(0, 0)$. Questi casi di test valutano anche in modo negativo e positivo la condizione dell'`if` alla riga 11.

Non è possibile ridurre ulteriormente il numero di casi, in quanto per coprire le due condizioni all'interno del primo `if` sono comunque necessari tre casi. Dunque tre casi è il numero minimo.

Domanda b)

Si consideri il cammino seguente: 1, 2, 3, 4, 5, 8, 9, 4, 7, 8, 9, 4, 7, 8, 9, 10, 11, 12, 13. Cacolare le condizioni perché sia eseguito (path condition) e si scriva un caso di test per percorrerlo.

Soluzione

Il ciclo è eseguito 3 volte (quindi $x = 4$), incrementando y una volta e decrementandolo due volte, per terminare con $y = 0$ (per eseguire la riga 12). Quindi $y = 1$.

Condizione: $x == 4 \&\& y == 1$, quindi caso di test $(4, 1)$.

Esercizio 4

Si consideri il seguente metodo statico Java, scritto in stile imperativo:

```

public static int deepMystery(final String name, final List<String> names) {
    if (names==null || name==null) return 0;
    int count=0;
    for (String x : names) {
        if (name.equals(x)) count=count+2;
        else count--;
    }
    return count;
}

```

Domanda a)

Scrivere lo stesso programma in stile funzionale, completando il codice seguente nello spazio lasciato bianco:

```

public static int deepMystery(final String name, final List<String> names) {
    if (names==null || name ==null) return 0;

    }

}

```

Soluzione

```

public static int deepMystery(final String name, final List<String> names) {
    if (names==null || name ==null) return 0;
    return names.stream()
        .map(x -> ((x.equals(name)) ? 2 : -1))
        .reduce(0, (subtot, elem) -> subtot + elem);
}

```

Oppure

```

public static int deepMystery(final String name, final List<String> names) {
    if (names==null || name ==null) return 0;
    return names.stream()
        .mapToInt(x -> ((x.equals(name)) ? 2 : -1))
        .sum();
}

```

Ingegneria del Software — SOLUZIONE DEL TEMA 26 Giugno 2020

Esercizio 1

Si consideri la seguente specifica di una classe mutabile SequenzaVettori, che rappresenta una sequenza (senza duplicazioni) di vettori bidimensionali. Un vettore è un elemento della classe Vettore, specificata più avanti.

```
public class SequenzaVettori {
    public SequenzaVettori(); // costruisce una sequenza vuota.

    //@ensures (* \result e' il numero di vettori nella sequenza *)
    public /*@ pure @*/ int size() {...}

    //@ensures (* \result e' il vettore i-esimo della sequenza se 0<=i<size() *).
    //@signals (IndexOutOfBoundsException e) (i<0 || i>=size());
    public /*@ pure @*/ Vettore get(int i) throws IndexOutOfBoundsException;

    //@ensures (* inserisce un nuovo elemento al termine della sequenza *)
    //@signals (DuplicateException e) (* eccezione se v esiste in this *)
    public void put(Vettore v) throws DuplicateException;

    //@ensures (* \result e' il vettore costituito dalla somma dei primi k vettori della sequenza *)
    //@signals (IndexOutOfBoundsException e) (k<2 || k>size());
    public /*@ pure @*/ Vettore sum(int k);
}
```

La classe immutabile Vettore è così specificata:

```
public /*@ pure @*/ class Vettore {
    // costruisce un vettore dati modulo e fase
    public Vettore(double modulo, double fase);

    //@ restituisce il modulo del vettore
    public double modulo();

    //@ restituisce la fase del vettore
    public double fase();

    //@ restituisce il vettore somma di this e v
    public Vettore somma(Vettore v);
    ... altri metodi tipici ...
}
```

Domanda a)

Si scriva la specifica JML del metodo `put`.

Soluzione

```
//@ensures size() == \old(size())+1 &&
//@(\forall int i; 0<=i && i<\old(size()));
//@ \old(!get(i).equals(v)) && get(i).equals(\old(get(i))) &&
//@get(size()-1).equals(v);
//@

//@signals (DuplicateException e) size()==\old(size()) &&
//@(\exists int i; 0<=i && i<size(); \old(get(i)).equals(v)) &&
//@(\forall int j; 0<=j && j<\old(size()); get(j).equals(\old(get(j))));
public void put(Vettore v) throws DuplicateException;
```

Domanda b)

Si scriva la specifica JML del metodo `sum`.

Soluzione

```
// @ensures k>=2 && k<=size() &&
// @ (k==2 ==> \result.equals(get(0).somma(get(1)))) &&
// @ (k>2 ==> \result.equals(sum(k-1).somma(get(k))));
// @signals (IndexOutOfBoundsException e) (k<2 || k>size());
public /*@ pure */ Vettore sum(int k);
```

Domanda c)

Si consideri una variante `SequenzaVettori2` della classe `SequenzaVettori`, che è identica alla classe precedente ma in cui la specifica di `put` è differente:

```
// @ensures (* inserisce il vettore v in una posizione qualunque *)
// @signals (DuplicateException e) (* eccezione se v esiste in this *);
public void put(Vettore v) throws DuplicateException e;
```

È possibile definire questa classe `SequenzaVettori` come derivata da `SequenzaVettori2`, rispettando il principio di sostituzione? Il viceversa? Motivare la risposta.

Soluzione

`SequenzaVettori` può essere sottoclasse di `SequenzaVettori2`, in quanto l'inserimento in coda alla sequenza è compatibile con l'inserimento in posizione qualunque (la postcondizione è più forte). Il viceversa invece viola la regola della postcondizione (la postcondizione è più debole).

Esercizio 2

Si consideri la seguente classe Java:

```
public class ConcTest {  
    private double x, y;  
    public ConcTest(double a) { x = y = a; }  
    public synchronized void setX(double x) { this.x = x; }  
    public synchronized double getX() { return x; }  
    public synchronized double getY() { return y; }  
    public synchronized void waitForEq() throws InterruptedException {  
        while(x!=y) wait();  
    }  
}
```

Domanda a)

La sincronizzazione è corretta? In caso negativo cosa può succedere? E come si deve modificare il codice affinché la situazione critica non si verifichi?

Soluzione

Non possono esserci conflitti nell'accesso allo stato condiviso (gli attributi `x` e `y`), ma potrebbero verificarsi dei deadlock. Il thread chiamante il metodo `waitForEq` potrebbe infatti rimanere per sempre in stato di wait visto che nessun metodo invoca `notify` o `notifyAll`. La soluzione è inserire la chiamata al metodo `notifyAll` al termine del metodo `setX`.

Domanda b)

Si riscriva la classe `ConcTest` eliminando il metodo `waitForEq`, aggiungendo un metodo `setY` che permetta di variare il valore dell'attributo `y` e facendo in modo che due thread possano accedere (leggere o modificare) in parallelo uno l'attributo `x` e l'altro l'attributo `y`. L'implementazione deve comunque evitare conflitti tra diversi thread che accedono allo stesso attributo (`x` oppure `y`).

Soluzione

```
public class ConcTest {  
    private double x, y;  
    private Object lockX, lockY;  
    public ConcTest(double a) {  
        x = y = a;  
        lockX = new Object();  
        lockY = new Object();  
    }  
    public void setX(double x) {  
        synchronized(lockX) { this.x = x; }  
    }  
    public double getX() {  
        synchronized(lockX) { return x; }  
    }  
    public void setY(double y) {  
        synchronized(lockY) { this.y = y; }  
    }  
    public double getY() {  
        synchronized(lockY) { return y; }  
    }  
}
```

Esercizio 3

Si consideri il seguente programma Java.

```
public class Main {
    public static void main(String[] args) {
        1    Person p1 = new Student();
        2    Person p2 = new Person();
        3    Person p3 = new Teacher();
        4    Student p4 = new Student();
        5    Student p5 = new Teacher();
        6    Lecture lec1 = new Lecture();
        7    Lecture lec2 = new OnlineLecture();
        8    OnlineLecture lec3 = new OnlineLecture();

        9    lec1.addAttendant(p1);
       10   lec1.addAttendant(p2);
       11   lec1.addAttendant(p3);
       12   lec1.addAttendant(p4);
       13   lec1.addAttendant(p5);
       14   lec2.addAttendant(p1);
       15   lec2.addAttendant(p2);
       16   lec2.addAttendant(p3);
       17   lec2.addAttendant(p4);
       18   lec2.addAttendant(p5);
       19   lec3.addAttendant(p1);
       20   lec3.addAttendant(p2);
       21   lec3.addAttendant(p3);
       22   lec3.addAttendant(p4);
       23   lec3.addAttendant(p5);
    }
}

abstract class Person {
    public void join(Lecture lec) { System.out.println("Joining " + lec); }
    public void join(OnlineLecture lec) { System.out.println("Joining " + lec); }
}
class Student extends Person {
    public void join(Lecture lec) { System.out.println("Student joining " + lec); }
}
class Teacher extends Person {
    public void join(OnlineLecture lec) { System.out.println("Teacher joining " + lec); }
}
class Lecture {
    public void addAttendant(Person p) { p.join(this); }
    public String toString() { return "a lecture"; }
}
class OnlineLecture extends Lecture {
    public String toString() { return "an online lecture"; }
}
```

Domanda a)

Quali linee del programma non sono valide? Motivare brevemente la risposta.

Soluzione

Linea 2 in quanto Person è una classe astratta e linea 5 in quanto Teacher non è sottoclasse di Student.

Di conseguenza vanno eliminate tutte le righe che contengono p2 o p5, ovvero le linee 10, 13, 15, 18, 20, 23.

Domanda b)

Dopo aver rimosso tutte le linee non valide identificate al punto precedente, indicare cosa stampa il programma, motivando brevemente la risposta.

Soluzione

- (Linea 09) Student joining a lecture
- (Linea 11) Joining a lecture
- (Linea 12) Student joining a lecture
- (Linea 14) Student joining an online lecture
- (Linea 16) Joining an online lecture
- (Linea 17) Student joining an online lecture
- (Linea 19) Student joining an online lecture
- (Linea 21) Joining an online lecture
- (Linea 22) Student joining an online lecture

Ingegneria del Software – a.a. 2006/07

Appello del 23 luglio 2007

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 8)

Il seguente metodo statico

```
public static int rigaSommaMax(int [][] m) {... }
```

accetta come ingresso un array a due dimensioni m di cui si assume che rappresenti una matrice rettangolare (cioè tutte le righe hanno lo stesso numero di elementi; si noti che una matrice quadrata, in cui il numero delle righe è uguale a quello delle colonne, è considerato un caso particolare di matrice rettangolare, quindi accettabile come parametro) avente almeno due righe e due colonne, e produce come risultato l'indice della riga che i cui elementi hanno somma massima (se più righe hanno lo stesso valore massimo il metodo restituisce l'indice di una qualsiasi di queste).

a) Si scriva una specifica, in JML del metodo indicando la pre- e la post-condizione.

Soluzione

```
//@requires m!= null &&
// la matrice ha almeno due righe
//@ m.length>1 &&
// tutte le righe hanno almeno due elementi
//@ (\forall int i; 0<=i<m.length; m[i]!=null && m[i].length>1) &&
//tutte le righe hanno la stessa lunghezza
//@ (\forall int i; 0<=i<m.length; (\forall int j; i<j<m.length; m[i].length == m[j].length) )&&

//@ensures
// non esiste una riga per la quale la somma degli elementi sia maggiore che per la riga di indice \result
//@ !(\exists int i; 0<=i<m.length;
//@ (\sum int j; 0<=j<m[i].length; m[i][j]) > (\sum int j; 0<=j<\result.length; m[\result][j]))
```

b) si trasformi la specifica precedente facendo in modo che il metodo accetti in ingresso una qualsiasi “matrice” con almeno due righe e due colonne, ma se la matrice non è rettangolare (cioè le righe non sono tutte della stessa lunghezza) esso lanci un’eccezione *MatriceNonRettangolareException* (tutti gli altri requisiti sono invariati).

Soluzione

```
//@requires m!= null &&
// la matrice ha almeno due righe
//@ m.length>1 &&
// tutte le righe hanno almeno due elementi
//@ (\forall int i; 0<=i<m.length; m[i].length>1) &&

//@ensures
//tutte le righe hanno la stessa lunghezza
(\forall int i; 0<=i<m.length; (\forall int j; i<j<m.length; m[i].length == m[j].length) ) &&
// non esiste una riga per la quale la somma degli elementi sia maggiore che per la riga di indice \result
!(\exists int i; 0<=i<m.length;
(\sum int j; 0<=j<m[i].length; m[i][j]) > (\sum int j; 0<=j<\result.length; m[\result][j]))&&
//@signals (MatriceNonRettangolareException e)
//@ (! (\forall int i; 0<=i<m.length; (\forall int j; i<j<m.length; m[i].length == m[j].length) ))
```

Esercizio 2 (punti 13)

Si consideri un sistema di gestione del traffico telefonico. Il sistema, fra le altre cose, tiene traccia degli utenti e delle loro chiamate. Una chiamata ha una data, un'ora, una durata e un numero. Le chiamate possono essere vocali, e in tal caso hanno anche uno scatto alla risposta e un costo al secondo, o di tipo dati, che invece hanno un volume di dati scambiati (in Kbyte) e un costo in volume (ossia al Kbyte).

Formalmente, le tre classi sono:

```
abstract public class Chiamata {  
    abstract public Data data();  
    abstract public int ora(); //HHMMSS  
    abstract public int durataInSecondi();  
    abstract public int costoTotale();  
    abstract public String numeroChiamato();  
}  
  
public class ChiamataVocale extends Chiamata {  
    public int costoScatto();  
    public int costoAlSecondo();  
}  
  
public class ChiamataDati extends Chiamata {  
    public int costoKB();  
    public int volumeKByte();  
}
```

La specifica della classe Data è:

```
public classe Data {  
    public int giorno();  
    public int mese();  
    public int anno();  
    //@ ensures (*\result è true sse this precede strettamente d *)  
    public boolean precede(Data d);  
}
```

Per ogni utente, il sistema tiene traccia del nome, del credito residuo e delle chiamate effettuate. E' inoltre possibile effettuare una ricarica di un numero intero di euro.

L'interfaccia della classe è così definita:

```
public class Utente {  
    //@ensures (* \result è il credito in centesimi di euro ancora disponibile *)  
    public int creditoResiduo();  
    //@ensures ....  
    public void ricarica(int euro);  
    //@ensures (*\result è un generatore, in ordine cronologico, delle chiamate effettuate alla data d*)  
    public Iterator<Chiamata> chiamateInData(Data d);  
}
```

a) Scrivere in JML la postcondizione del metodo ricarica.

SOLUZIONE: `this.creditoResiduo() == 100*euro + \old(this.creditoResiduo());`

Sia dato il seguente rep per la classe Utente:

```
private ArrayList<Chiamata> log;
//contiene tutte le chiamate effettuate in ordine cronologico
int spesaTotaleAnnoCorrente; //la spesa complessiva in centesimi di euro dell'utente
nell'anno ) corrente fino alla data attuale
Data dataCorrente; //la data attuale
int totaleRicariche; //il credito complessivo, in euro, acquistato dall'utente
int creditoResiduo; //il credito ancora disponibile
```

b) Scrivere l'invariante di rappresentazione che stabilisce la correttezza e la mutua consistenza dei dati che costituiscono il rep.

SOLUZIONE:

log in ordine crescente && ultima chiamata del log in data <=dataCorrente && totaleRicariche è uguale al totale dei costi delle chiamate sommato al credito residuo && spesaTotaleAnno è pari al costo complessivo delle chiamate per il solo anno corrente.

```
log !=null && dataCorrente!=null && (\forall int j; 0<= j && j<log.size(); log.get(j) !=null) &&
(\forall int j; 0<= j && j<log.size()-1;
 (log.get(j).data().precede(log.get(j+1).data()) ||
 log.get(j).data().equals(log.get(j+1).data()) &&
 log.get(j).ora()<log.get(j+1).ora()) &&
 (log.get(log.size()-1).data().precede(dataCorrente) ||
 log.get(log.size()-1).data().equals(dataCorrente) )
totaleRicariche == 100*(\sum int j; 0<= j && j<log.size(); log.get(j).costoTotale())+creditoResiduo
&&
spesaTotaleAnno ==
(\sum int j; 0<= j && j<log.size()&& log.get(j).data().anno()==dataCorrente.anno();
 log.get(j).costoTotale())
```

c) Con riferimento al rep introdotto al punto precedente, implementare il metodo iteratore chiamateInData(), includendo anche la definizione di tutte le classi appropriate.

SOLUZIONE:

```
public Iterator<Chiamata> chiamateInData(Data d) {return new UtenteGen(this,d);}
private static class UtenteGen implements Iterator<Chiamata> {
    private Utente p; //l'Utente sul cui log si vuole iterare
    private int n; //posizione del prox el. da considerare
    private Data dataAttuale;
    //@requires u!=null;
    UtenteGen(Utente u,Data d) {
        dataAttuale=d;
        p=u;
        for(int i; 0<=i && i<p.log.size() && p.log.get(i).data().precede(dataAttuale);i++);
        n=i;
    }
    public boolean hasNext () {
        return n<p.log.size() && dataAttuale.equals(p.log.get(n).data());
    }
    public int next () throws NoSuchElementException {
        if (!hasNext()) throw new NoSuchElementException();
        return p.log.get(n++);
    }
}
```

Esercizio 3 (punti 4)

Si consideri la seguente variante della classe Utente introdotta nell'esercizio 2, con parte dell'interfaccia definita come segue.

```
public class Utente {  
    ...  
    //@ensures (*\result è la somma del costo industriale di tutte le chiamate effettuate  
    // alla data d*)  
    public int costolIndustrialeChiamateInData(Data d);  
  
    //@ensures \result > costolIndustrialeChiamateInData(d) / 2 &&  
    // \result <= costolIndustrialeChiamateInData(d) * 2  
    public int importoFatturabileInData(Data d);  
}
```

Vengono inoltre definite le due classi UtenteAgevolato e UtentePrivilegiato, possibili classi eredi di Utente, che usufruiscono di tariffe particolarmente favorevoli.

```
public class UtenteAgevolato {  
    ...  
    //@ensures \result == costolIndustrialeChiamateInData(d) * 0.7  
    public int importoFatturabileInData(Data d);  
}  
  
public class UtentePrivilegiato {  
    ...  
    public final int TETTO = ...;  
    //@ensures \result ==  
    //@ costolIndustrialeChiamateInData(d) * 1.5 <= TETTO ?  
    //@ costolIndustrialeChiamateInData(d) * 1.5 : TETTO  
    public int importoFatturabileInData(Data d);  
}
```

Le classi UtenteAgevolato e UtentePrivilegiato sono definibili come classi eredi della classe Utente senza violare il principio di sostituzione? Motivare adeguatamente la risposta, nel caso positivo argomentando che sono soddisfatte le regole di sostituibilità, oppure nel caso negativo mostrando che non lo sono mediante un controesempio.

Soluzione

La classe UtenteAgevolato soddisfa il principio di sostituzione, perchè la postcondizione

$\text{\result} == \text{costolIndustrialeChiamateInData}(d) * 0.7$

Implica logicamente la postcondizione

$\text{\result} > \text{costolIndustrialeChiamateInData}(d) / 2 &&$
 $\text{\result} < \text{costolIndustrialeChiamateInData}(d) * 2$

Invece la classe UtentePrivilegiato non lo soddisfa, perchè l'imposizione del TETTO all'importo fatturabile può far sì che venga violata la condizione di fatturare almeno metà del costo industriale; in particolare, per valori di costolIndustrialeChiamateInData(d) sufficientemente elevati, si possono verificare entrambe le seguenti condizioni

$\text{costolIndustrialeChiamateInData}(d) * 1.5 > \text{TETTO}$

che fa sì che $\text{\result} == \text{TETTO}$, e

TETTO < costoIndustrialeChiamateInData(d)/2

violando quindi la postcondizione del metodo importoFatturabileInData che prescrive che
 $\text{result} \geq \text{costoIndustrialeChiamateInData}(d)/2$

Ingegneria del Software – a. a. 2006/07

Appello del 18 Settembre 2007

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici comporta l'annullamento del compito.
 2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
 3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
 4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
 5. Tempo a disposizione: 1h30.
 6. Punteggio totale a disposizione: 25/30.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totalle

Esercizio 1 (punti 6)

L'astrazione procedurale *scambia* riceve in ingresso due array a e b, che contengono valori interi. Sotto l'ipotesi che i due array abbiano la stessa dimensione, l'astrazione scambia fra loro il contenuto dei due array. Si completi la seguente specifica formale in JML dell'astrazione.

Esercizio 2 (punti 12)

Si consideri la seguente specifica dell'ADT BranoMusicale:

```
public /*@ pure */@class BranoMusicale {  
    //OVERVIEW: tipo immutabile che rappresenta un brano musicale registrabile su supporti digitali  
    //Un tipico BranoMusicale è una quadrupla [info – durata – musica – bitrate], dove "info" è una stringa  
    //di informazioni testuali che descrivono il brano (per es. autore o titolo), "durata" è la durata in secondi  
    //del brano, "musica" è la rappresentazione digitale (in byte) della musica del brano e "bitRate" è  
    //il numero di Kbit al secondo.  
  
    //costruttore:  
    //@ensures(*crea un nuovo brano sulla base dei valori passati come parametri*);  
    public BranoMusicale(String info, int durata, byte[] musica, int bitrate);  
  
    //metodi observer:  
    //@ensures(*restituisce le informazioni testuali associate al brano*);  
    public String getInfo();  
  
    //@ensures(*restituisce la durata in secondi del brano*);  
    public int getDurata();  
  
    //@ensures(*restituisce la rappresentazione digitale della musica del brano*);  
    public byte[] getMusica();  
  
    //@ensures(*restituisce il numero di bit al secondo della codifica del brano*);  
    public int bitRate();  
}
```

a) Completare la parte con i puntini della seguente classe Memoria, che rappresenta una memoria per brani musicali (es. CD, MemoryStick ecc.)

In particolare sono da completare alcune specifiche JML di pre e postcondizioni, l'invariante di rappresentazione, l'invariante pubblico e il codice dei metodi "brani" e "inserisciBrano".

```
public class Memoria {  
    //OVERVIEW: Tipo mutabile. Una Memoria contiene tracce musicali di tipo BranoMusicale  
    //Una tipica Memoria è identificata dalla tripla [capienza, brani, durata], dove;  
    // "capienza" è la capacità in byte della memoria,  
    // "brani" è una sequenza <b1 b2 ... bn> di oggetti della classe BranoMusicale,  
    // "durata" è la durata complessiva in secondi dell'insieme di brani presenti nella Memoria.  
    //@public invariant.....  
    durata()== (\sumof int i; 0<=i && i<brani.size(); brani().get(i).getDurata()) &&  
    capienza()>= (\sumof int i; 0<=i && i<brani.size(); brani().get(i).getMusica().length) ....  
    .....  
    .....  
  
    //rep:  
    private int capienza;  
    private Vector<BranoMusicale> brani;  
    private int durata;  
    //@ private invariant .....
```

capienza() == this.capienza && capienza>=0 && brani!=null &&
brani.length == brani().size() &&
(forall i; 0<=i && i<brani().size(); brani().get(i)==brani[i]) ;
(NB: si suppone una relazione uno a uno fra la posizione nell'array e la posizione nel Vector)

.....
.....

```

//@requires max_capienza>0;
//@ensures (* crea una nuova Memoria di capienza specificata, inizialmente vuota*)
public Memoria(int max_capienza){
    this.capienza=max_capienza;
    this.brani=new Vector();
    this.durata=0;
}

//@ensures (* /result è la durata complessiva dei brani in this *)
public int durata() { return this.durata; }

//@ensures (* /result è la capienza di this*)
public int capienza() { return this.capienza; }

//@ensures (* /result contiene tutti e soli i brani di this *)
public Vector<BranoMusicale> brani() {
//implementazione:
    return brani().clone();...
    .....
    .....
    .....

//@ensures(*/result è true se nella Memoria this c'è ancora abbastanza spazio per il brano b*)
public boolean spazioDisponibile(BranoMusicale b) {
//implementazione:
    int somma =0;
    (for int i; i>0 && i< brani.lengtht; i++)
        somma+=brani.getMusica().length;
    return (b.getMusica().length>=capienza -somma);.

    .....
    .....

}

//    se b supera la capacità ancora disponibile nella Memoria, solleva
//    BranoTroppoLungoException. Altrimenti, inserisce il brano b in this
//@ensures .....
    capienza()>= b.getMusica().length +
        \old(\sumof int i; 0<=i && i<brani().size(); brani().get(i).getMusica().length) &&
        brani().size() == \old(brani.size())+1) &&
        b== brani().lastElement() &&
        (forall int i; 0<=i && i<brani().size()-1; brani().get(i) == \old(brani().get(i)))
        (NB: si suppone che brano sia inserito in ultima posizione)
//@signals( BranoTroppoLungoException e) capienza()<= b.getMusica().length +
    \old(\sumof int i; 0<=i && i<brani().size(); brani().get(i).getMusica().length)
    && brani().equals(\old(brani()));

public void inserisciBrano(BranoMusicale b) throws BranoTroppoLungoException {
//implementazione:
    if (!spazioDisponibile() throw new BranoTroppoLungoException();
    brani.add(b);
}

```

b) Si consideri la classe BranoTroppoLungoException utilizzata dalla classe Memoria. Sulla base dell'informazione disponibile, sarebbe più opportuno definirla come eccezione checked (sottoclasse di Exception) o eccezione unchecked (sottoclasse di RunTimeException)? Motivare brevemente la risposta.

Trattandosi di eccezione rilanciata allo scopo di avvertire il chiamante dell'impossibilità di introdurre il brano in memoria, deve senz'altro essere checked (per costringere il chiamante a "trattare" questo caso).

Esercizio 3 (punti 7)

a) Con riferimento all'esercizio 2, si consideri la seguente specifica:

```
public class BranoMP3-160Kbit extends BranoMusicale {  
    //OVERVIEW: BranoMusicale codificato in MP3 a 160 Kbit/sec.  
    //@also  
    //@public invariant .....  
        this.bitRate() ==160;  
  
        //@ensures(*crea un nuovo brano sulla base dei valori passati come parametri*);  
    public BranoMP3-160Kbit(String info, int durata, byte[] musica);  
}
```

Si completi il public invariant e si indichi, motivando accuratamente ma sinteticamente la risposta, se se questo ADT soddisfa o meno il principio di sostituzione di Liskov rispetto a BranoMusicale.

L'ADT soddisfa il principio di sostituzione: la regola dei metodi non può essere violata da un costruttore (in quanto i costruttori non sono ereditati). Non e' violata nemmeno la regola delle proprietà, perche' un generico Brano può essere comunque da 160Kbit/sec.

b) Sempre con riferimento all'esercizio 2, si consideri la seguente specifica:

```
public class MP3Memory extends Memoria {  
    //OVERVIEW: Una memoria contenente solo brani di tipo BranoMP3-160Kbit.  
    //@also  
    //@public invariant .....  
        (forall int i; 0<=i && i<brani().size(); brani().get(i) instanceof BranoMP3-160Kbit);  
  
        //@requires max_capienza>0;  
        //@ensures (* crea una nuova MP3Memory, di capienza specificata, inizialmente vuota*)  
    public MP3Memory(int max_capienza);  
}
```

Si completi il public invariant e si indichi, motivando accuratamente ma sinteticamente la risposta, se questo ADT soddisfa o meno il principio di sostituzione di Liskov rispetto a Memoria.

L'ADT (un contenitore) non soddisfa il principio di sostituzione, in quanto il suo invarianto impedisce di fatto l'inserimento di Brani che non siano di tipo BranoMP3-160Kbit. Per soddisfare l'overview (e quindi l'invariante) occorrerebbe ridefinire la specifica del metodo inserisciBrano, per impedire l'inserimento di brani con bitrate diverso da 160. Questo violerebbe la regola della postcondizione.

Ingegneria del Software – a.a. 2005/06

Appello del 10 luglio 2006

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
 2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
 3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
 4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
 5. Tempo a disposizione: 2h.
 6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Total

Esercizio 1 (punti 9)

Il metodo statico *trasposta* ha come argomenti due interi $n, m \geq 1$, e due matrici A e T (di interi) costituite, rispettivamente, da n righe e m colonne, e da m righe e n colonne. Il metodo, senza alterare il contenuto di A, memorizza in T la matrice trasposta di A. Si rammenta che una matrice T è la trasposta di una matrice A se, e solo se, T è ottenuta da A scambiando le righe con le colonne. Si ricorda che in Java le matrici sono dichiarate come array di array. Ad esempio, A[0] è l'array che contiene la prima riga della matrice, A[1] la seconda, ecc. A[0][1] è quindi l'elemento A_{12} della matrice A.

a- Si scriva la specifica formale in JML del metodo.

```
//@ requires n>=1 && m>=1 && A.length == n && T.length == m &&
  (forall int i; 0<= i && i<n; A[i].length == m) && (forall int i; 0<= i && i<m; T[i].length == n);
```

```
//@ensures (forall int i; 0<= i && i<n;
  (forall int j; 0<= j && j<m; A[i][j]==T[j][i] && A[i][j] == \old(A[i][j]));
```

Opzionalmente, si poteva aggiungere anche la clausola:

```
\@assignable T[*][*];
```

```
public static void trasposta (int n, int m, int [][] A, int [][] T)
```

b- Si trasformi la specifica JML del punto precedente sostituendo una (o più) a scelta delle precondizioni nel lancio di opportune eccezioni. Si mostri solo la parte di specifica modificata rispetto a quella definita al punto a.

Si toglie ad esempio la condizione $n \geq 1$ & $m \geq 1$ dalla *requires*, aggiungendola in $\&\&$ alla *ensures*. Si aggiunge inoltre la clausola *signals* per lanciare ad esempio una *EmptyException*.

b- Si scelga opportunamente un insieme minimale di casi di test funzionali, che si possa considera sufficiente per verificare il metodo statico, in base al criterio dei casi limite.

Si possono considerare i seguenti casi:

- 1- $n == 1, m == 1$
- 2- $n > 1, m == 1$
- 3- $n == 1, m > 1$
- 4- $n == m > 1$
- 5- $n > 1, m > 1, n != m$

Esercizio 2 (punti 16)

Si consideri la classe **Agenda** che rappresenta un insieme di appuntamenti (detti anche eventi). Ogni evento è un oggetto della classe Evento, specificata informalmente al punto (a). La classe Agenda ha un metodo di inserimento, che aggiunge un evento se questo non esiste. Se l'evento è già presente in Agenda, viene sollevata l'eccezione DuplicateException. Si ha un *conflitto* quando si inserisce un evento sovrapposto anche solo per un minuto con un altro evento in Agenda. In tal caso, l'evento è comunque inserito in Agenda, ma viene segnalata l'eccezione ConflictException.

La classe ha la seguente specifica:

```
public class Agenda {  
    //OVERVIEW: Un'Agenda è un insieme mutabile, anche vuoto e senza duplicazioni, di  
    // Eventi.  
    //@ ensures (* costruisce un'Agenda vuota*);  
    public Agenda()  
  
        //@ requires e!=null;  
        //@ ensures inAgenda(e) && (\forall Evento e1; \old(inAgenda(e1)); inAgenda(e1)) &&  
        //@      (\forall Evento e1; inAgenda(e1); \old(inAgenda(e1)) || e1==e);  
        //@ signals (ConflictException e) inAgenda(e) &&  
        //@      (\exists Evento e1; \old(inAgenda(e1)); e.conflitto(e1));  
        //@ signals (DuplicateException e) \old(inAgenda(e));  
        public void inserisci(Evento e) throws ConflictException, DuplicateException  
  
        //@ requires e!=null;  
        //@ ensures !inAgenda(e) && \old(inAgenda(e));  
        //@ signals (EventDoesNotExist exc) \old(!inAgenda(e));  
        public void rimuovi(Evento e) throws EventDoesNotExistException  
  
        //@ requires e!=null;  
        //@ ensures (\result == \exists Evento e1; this.eventiInData(e.dataEvento()); e1.equals(e));  
        public /*@ pure */ boolean inAgenda(Evento e)  
  
        //@ requires data!=null;  
        //@ ensures (* \result è la lista di tutti e soli gli eventi che accadono in data d *);  
        public /*@ pure */ ArrayList<Evento> eventiInData(Date d)  
}
```

a- Completare in JML, nelle parti con i puntini, la seguente specifica della classe Evento:

```
public /*@ pure */ class Evento {  
    //OVERVIEW: Un Evento è un oggetto immutabile, con alcune informazioni, una data e  
    //un orario di inizio e di fine. L'orario è nel formato hhmm: ad es. 1630 corrisponde alle  
    //ore 16.30. Due eventi sono in conflitto quando sono sovrapposti anche solo per un minuto.  
  
    //@ requires nome!=null && data !null && oraInizio<=oraFine && oraInizio>= 0  
    //@ requires nome!=null && data !null && oraInizio<=2359 && oraFine>= 0 && oraFine<=2359 && 0<=oraInizio%100<60 &&  
    //@ requires nome!=null && data !null && oraInizio<=2359 && oraFine>= 0 && oraFine%100<60;  
    //@ ensures this.nomeEvento() = nome && this.dataEvento().equals(data) &&  
    //@ ensures this.oraInizio() = oraInizio && this.oraFine() = oraFine;  
  
    public Evento(String nome, String note, Date data, int oraInizio, int oraFine)  
  
        //@ requires e!=null;  
        //@ ensures \result ==  
        //@ (e.oraInizio()>=this.oraInizio() && e.oraInizio()<=this.oraFine()) ||  
        //@ (this.oraInizio()>=e.oraInizio() && this.oraInizio()<=e.oraFine())
```

```

public boolean conflitto(Evento e)
    //@ ensures (* \result è l'orario di inizio dell'evento *);
public int orarioInizio()

    //@ ensures (* \result è l'orario di fine dell'evento *);
public int orarioFine()

    //@ ensures (* \result è la data dell'evento *);
public Date dataEvento()
}

```

b- Si considerino le seguenti due estensioni (b1) e (b2) di Agenda e per ciascuna si scriva, *motivando la risposta nel modo più preciso possibile*, se verificano il principio di sostituzione:

b1- La classe AgendaSenzaConflitti estende Agenda, modificando il metodo *inserisci(Evento e)* in modo che nel caso di conflitto di e con altri eventi dell'Agenda, il metodo non inserisca l'evento e, pur lanciando l'eccezione ConflictException.

No, perché si violerebbe la regola delle postcondizioni. Infatti, la postcondizione di *inserisci* prevede l'inserimento dell'elemento anche in caso di conflitto: per non inserire l'elemento dovremo indebolire la postcondizione.

b2) La classe AgendaConDuplicati estende la classe Agenda aggiungendo:

- il metodo osservatore *molteplicità*, allo scopo di verificare quante volte un evento è presente in Agenda.

```

    //@ requires e!=null;
    //@ ensures (\num_of Evento e1; this.eventiInData(e.dataEvento()).contains(e1); e1.equals(e));
public /*@ pure @*/ int molteplicità(Evento e)

```

- il nuovo metodo *inserisciMultipli*, così specificato:

```

    //@ requires e!=null;
    //@ ensures molteplicità(e) == 1 + \old(molteplicità(e)) &&
    //@ (\forall Evento e1; !e.equals(e1); molteplicità(e1)==\old(molteplicità(e1)));
    //@ signals (ConflictException exc) inAgenda(e) &&
    //@ (\forall Evento e1; \old(inAgenda(e1); inAgenda(e1)) &&
    //@ (\exists Evento e1; inAgenda(e1); e.conflitto(e1)));
public void inserisciMultipli (Evento e) throws ConflictException

```

Il metodo *rimuovi* è ridefinito, in modo da aggiornare la corretta molteplicità degli elementi:

```

    //@ also
    //@ ensures (\old(inAgenda(e)) ==> molteplicità(e)==\old(molteplicità(e))-1) &&
    //@ (\forall Evento e1; inAgenda(e1) && !e.equals(e1);
    //@ molteplicità(e1)==\old(molteplicità(e1)));
public void rimuovi(Evento e) throws EventDoesNotExistException

```

Per lo stesso motivo è ridefinito anche il metodo *inserisci*, in modo che la molteplicità di un nuovo evento inserito sia uguale a uno:

```

    //@ also
    //@ ensures \old(!inAgenda(e)) && inAgenda(e) && molteplicità(e)== 1) &&
    //@ (\forall Evento e1; inAgenda(e1) && !e.equals(e1);
    //@ molteplicità(e1)==\old(molteplicità(e1)));
public void inserisci(Evento e) throws ConflictException, DuplicateException

```

NO. La ridefinizione di inserisci e rimuovi, in base alla definizione di JML, rispetta la regola dei metodi. Tuttavia, l'aggiunta del metodo inserisciMultipli comporta la violazione della regola delle proprietà, in quanto l'overview di Agenda prevede che non vi siano eventi duplicati.

c- Un possibile rep della classe Agenda è costituitocostituito da un array di eventi riempito nella parte iniziale e da un intero che indica il numero di eventi effettivamente memorizzati nell'array::

```
private Evento [] evList;  
private int numEv;
```

Per ragioni di efficienza implementativa, nell'array gli eventi con la stessa data sono disposti in ordine crescente rispetto all'orario di inizio (così come definita dal metodo **orarioInizio()**), anche se non necessariamente in posizioni consecutive.

Scrivere in JML l'invariante di rappresentazione. corrispondente a questa soluzione.

```
//@ private invariant evList !=null &&  
    (forall int i; 0<=i && i< numEv; (forall int j; i<j && j<= numEv;  
        evList[i].dataEvento() == evList[j].dataEvento() ==>  
        evList[i].orarioInizio() <=evList[j].orarioInizio()));
```

d- Un possibile invariante per la classe Agenda è il seguente:

Non ci sono eventi duplicati.

d1- Di che tipo di invariante si tratta? **Si tratta tipicamente di un invariante astratto (un public invariant di JML), in quanto è parte della Overview della classe.**

d2- Specificarlo formalmente in JML.

```
//@ public invariant  
    (forall Date d;;  
        (forall int i; 0<=i && i< eventiInData(d).size()-1;  
            (forall int j; i<j && j< eventiInData(d).size();  
                ! eventiInData(d).get(i).equals(eventiInData(d).get(j))));
```

Si noti che qualora l'invariante fosse stato definito come private, non sarebbe in alcun modo possibile garantire che sia verificato solo a partire dalla specifica di Agenda (si potrebbe decidere che per ragioni di efficienza si consente di duplicare un evento in evList anche se poi questa duplicazione è invisibile agli utilizzatori...)

d3- Stabilire, in base alla specifica della classe Agenda, se l'invariante è sempre verificato. Utilizzare eventualmente un ragionamento informale ma chiaro e conciso.

Intuitivamente, l'invariante dovrebbe essere verificato, in quanto:

- 1) **il costruttore inizializza un'Agenda senza eventi (e quindi senza duplicati);**
- 2) **se il metodo inserisci viene chiamato su un'Agenda senza duplicati, la specifica del metodo sembra tale per cui un evento è inserito solo se non è equals a uno esistente. Pertanto, l'Agenda risultante continua a non avere duplicati.**
- 3) **Nessun altro metodo oltre al costruttore e a inserisci è in grado di aggiungere eventi a un'Agenda.**

Si osservi tuttavia che, formalmente, la specifica di inserisci, nel caso di DuplicateException non garantisce nulla... Pertanto, il punto (2) formalmente non può essere garantito e quindi l'invariante non è verificabile in modo formale.



Politecnico di Milano

Anno accademico 2012-2013

Ingegneria del Software – Appello del 17 Luglio 2013

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi

Ghezzi

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1 (punti 15)

Si consideri il seguente ADT Lista-Doppia-Con-Cursore (nel seguito, abbreviata come LDCC). Una LDCC è costituita da una lista sequenziale di elementi (detti *nodi*) e da un *cursore* posizionato “a metà” fra due nodi della lista. Il cursore consente di accedere sia all’eventuale nodo alla sua sinistra che all’eventuale nodo alla sua destra, e può essere mosso sia verso sinistra che verso destra. La lista risulta quindi bidirezionale. Si considerino le seguenti dichiarazioni:

```
import java.util.*;
public class LDCC<T> {
    //Overview Una LDCC e' costituita da una lista sequenziale di nodi di tipo T,
    //e da un cursore posizionato fra due nodi. Null non pu\`o essere parte della lista.
    //Il cursore pu\`o essere mosso di una posizione in entrambe le direzioni, mentre un
    //elemento pu\`o essere aggiunto o rimosso sia alla sua sinistra che alla sua destra.

    //@ensures (*Costruisce un LDCC vuoto*)
    public LDCC() { }

    //observers:
    //@ensures (*\result e' true sse a sinistra del cursore non ci sono nodi*)
    public boolean endLeft() {return true;}

    //@ensures (*\result e' true sse a destra del cursore non ci sono nodi*)
    public boolean endRight() {return true;}

    //@ensure (*\result e' l'elemento a sx del cursore*)
    public T leftItem() throws NoSuchElementException {}

    //@ensure (*\result e' l'elemento a dx del cursore*)
    public T rightItem() throws NoSuchElementException {}

    //@ensures (* \result e' costituito da tutti gli elementi di this,
    //nell'ordine da sx a dx corrispondente a posizioni crescenti, con il cursore;
    //segnalato da un elemento speciale, di valore null*)
    public ArrayList<T> elements() {}

    //modifiers
    //muove il cursore a sx se possibile
    public void moveLeft() throws NoSuchElementException {}

    //muove il cursore a dx se possibile
    public void moveRight() throws NoSuchElementException {}

    //inserisce t a sx del cursore
    public void insertLeft(T t) {}

    //inserisce t a dx del cursore
    public void insertRight(T t) {}

    //cancella, se possibile, l'elemento a sx del cursore
    public void deleteLeft() throws NoSuchElementException {}

    //cancella, se possibile, l'elemento a dx del cursore
    public void deleteRight() throws NoSuchElementException {}

}
```

Domanda 1

Scrivere la specifica JML del metodo insertRight, aggiungendo, se necessario, opportune eccezioni. Si consiglia di scrivere la specifica usando anche il metodo observer elements() e il metodo indexOf(Object x) di ArrayList che

restituisce la posizione a cui si trova la prima occorrenza dell'elemento x.

Soluzione: Si aggiunge NullPointerException per evitare di inserire null. Poiché l'inserimento avviene a destra, il cursore e gli elementi alla sua sinistra non cambiano posizione. A destra si inserisce l'elemento, mentre le posizioni successive sono shiftate di 1 verso dx.

```
//@ensures t!=null && elements.size() == \old(elements.size()+1) &&
//@ elements().indexOf(null) == \old(elements().indexOf(null)) &&
//@ (\forall int i; 0<= i && i <elements().size());
//@     (i<=elements().indexOf(null) ==> (elements().get(i) == \old(elements().get(i))) &&
//@     (i==1+elements().indexOf(null) ==> (elements().get(i) == t)) &&
//@     (i>1+elements().indexOf(null)) ==> (elements().get(i) == \old(elements().get(i-1))));
//signals (NullPointerException e) t==null;
```

```
public void insertRight(T t) {}
```

Domanda 2

Si consideri la seguente implementazione parziale della classe LDCC

```
public class LDCC<T> {
    private ArrayList<T> nodo;
    private int cursore;
    public LDCC() {
        nodo = new ArrayList<T>();
        cursore = -1;
    }
    public boolean endLeft() {return (cursore == -1);}
    public boolean endRight() {return (cursore == nodo.size()-1);}
    public void insertRight(T t) {
        if (t == null) throw new NullPointerException();
        nodo.add(cursore+1,t);
    }
    public void insertLeft(T t) {
        if (t == null) throw new NullPointerException();
        nodo.add(cursore+1,t);
        cursore++;
    }
    public void moveRight() throws NoSuchElementException {
        if (endRight()) throw new NoSuchElementException();
        cursore++;
    }
}
```

dove si intende che *nodo* include tutti gli elementi della lista, mentre il valore di *cursore* indica l'indice del nodo immediatamente a sinistra del cursore.

Scrivere l'invariante di rappresentazione della classe LDCC in JML.

```
private invariant ...
```

Soluzione:

```
private invariant
nodo!=null && cursore>=-1 && cursore < nodo().size() && !nodo.contains(null);
```

Domanda 3

Scrivere la funzione di rappresentazione della classe LDCC, usando la notazione preferita.

Soluzione: Si può usare ad esempio ancora un private invariant:

```

private invariant
elements().indexof(null) == cursore +1 &&
(\forall int i; 0<= i && i<nodo.size();
 (i<=cursore ==> (elements().get(i) == nodo.get(i))) &&
 (i>cursore ==> (elements().get(i+1) == nodo.get(i))));
```

Domanda 4

Implementare il metodo elements().

Soluzione:

```

public ArrayList<T> elements() {
    ArrayList<T> temp = (ArrayList<T>) nodo.clone();
    nodo.add(cursore+1,null);
    return temp;
}
```

Domanda 5 (facoltativa)

Mostrare che l'implementazione di insertRight ne soddisfa la specifica.

Soluzione: Il caso $t = null$ è ovvio. Se t non è null, la add inserisce un elemento in $nodo$ alla destra della posizione $cursore+1$, lasciando nelle loro posizioni gli elementi precedenti e shiftando di 1 a destra quelli successivi. In base alla funzione di astrazione, si osserva che la posizione $cursore+1$ di $nodo$ corrisponde alla posizione di $elements()$ avente valore null, e che, visto che la posizione del cursore non cambia, lo stesso elemento inserito in $nodo$ si ritrova inserito a destra della posizione null nel nuovo valore di $elements()$ (con gli elementi a sx immutati e a dx shiftati di 1). Ma questa è proprio la specifica di $insertRight$.

Esercizio 2 (punti 6)

Si consideri la classe LDCC dell'esercizio 1. La seguente classe MultiLDCC rappresenta anch'essa una LDCC, ma consente di utilizzare più cursori. La lista è associata infatti ad una pila (mai vuota) di cursori (c_1, \dots, c_n) , con $n > 0$, di cui quello in cima alla pila (c_n) è il cursore corrente.

In particolare, una lista MultiLDCC ha inizialmente un solo cursore (che è anche il corrente), ma può anche "impilare" un nuovo cursore, come copia del cursore corrente, tramite un metodo $pushCursor()$. Il metodo $popCursor()$ consente di eliminare il cursore in cima alla pila. Infine, $islastCursor()$ consente di verificare se il cursore corrente è anche l'unico sulla pila. Tutti gli altri metodi di MultiLDCC sono specificati esattamente come quelli di LDCC, considerando che ciascuno di questi metodi si riferisce implicitamente al cursore corrente. Ad es. $elements()$ restituisce sempre un $ArrayList$ che contiene tutti e soli gli elementi della lista, con un elemento=null nella posizione del cursore corrente; $moveLeft()$ muove a sinistra di una posizione il cursore corrente, ecc.

Domanda 1

È possibile definire MultiLDCC come erede di LDCC, rispettando il principio di sostituzione? E' necessario motivare accuratamente la risposta.

Soluzione: Poiché la specifica dei metodi ereditati non cambia, sia la regola della segnatura che la regola dei metodi sono verificate. Occorre quindi stabilire se vale la regola delle proprietà. Nessuna delle proprietà elencate nella Overview della classe cambia avendo a disposizione più cursori (visto che se ne sua sempre uno solo alla volta), quindi il principio risulta rispettato. Il tutto dipende dall'ipotesi (del testo) che la pila dei cursori non sia mai vuota, e quindi la lista abbia sempre un cursore corrente.

Domanda 2

Come fareste a specificare in JML il metodo $pushCursor$, supponendo di avere a disposizione il tipo di dato astratto $Pila<T>$? Si supponga che la specifica di Pila sia quella tradizionale, con $push$, pop , top , $size()$, ecc., ma con un metodo osservatore $content()$ che restituisce in un $ArrayList$ tutti gli elementi della pila, dal fondo alla cima. Si noti che le informazioni disponibili non sono complete poter specificare il metodo $pushCursor$: è necessario aggiungere ulteriori elementi alla specifica.

Soluzione: Non vi è alcun tipo di dati corrispondente ai cursori, mentre per specificare il comportamento a pila occorre conoscere il tipo degli elementi (es. Pila<Integer>, ecc.).. Una soluzione è quella di aggiungere un metodo osservatore pilaCursori() che restituisce una Pila, in cui ogni elemento della Pila non è un cursore, ma una LDCC che contiene gli stessi elementi della MultiLDCC (che si suppone essere definita come erede di LDCC), ma ciascuno con il suo cursore.

```
//@ensures (* tutti gli elementi() contenuti in \result sono identici a this.elementi(),  
//@          a parte il cursore.*);  
public Pila<LDCC<T>> pilaCursori()
```

E' a questo punto facile definire:

```
//@ensures pilaCursori().top().equals(this) &&  
//@ pilaCursori().size() == \old(pilaCursori().size())+1) &&  
//@ (\forall int i; 0\le i && i<content().size()-1; content().get(i)==\old(content().get(i)));  
public void pushCursor()
```

Esercizio 3 (punti 5)

Il seguente metodo Java prende in ingresso un array di al più 5 interi e restituisce l'elemento più grande in valore assoluto, oppure 0 se l'array fosse vuoto oppure -1 se troppo grande.

```
public int max(int[] numbers) {  
    int i, max_value = 0;  
    if (numbers.length > 5) return -1;  
    for (i = 0; i<numbers.length; i++) {  
        if (numbers[i] < 0)  
            max_value = Math.max(max_value, Math.abs(numbers[i]));  
        else max_value = Math.max(max_value, numbers[i]);  
    }  
    return max_value;  
}
```

- Disegnare il diagramma del flusso di controllo.

Soluzione:

- Il metodo è collaudato con i casi di test seguenti (input; valore restituito):

- T1: {0,0,0,0,0}; 0;
- T2: {1,2,3,4,5}; 5;
- T3: {-1,-2,-3,-4,-5}; 5;
- T4: {1,2,3,4,5,6}; -1;
- T5: {-10,10,3,5,-6}; 10;
- T6: {}; -1;

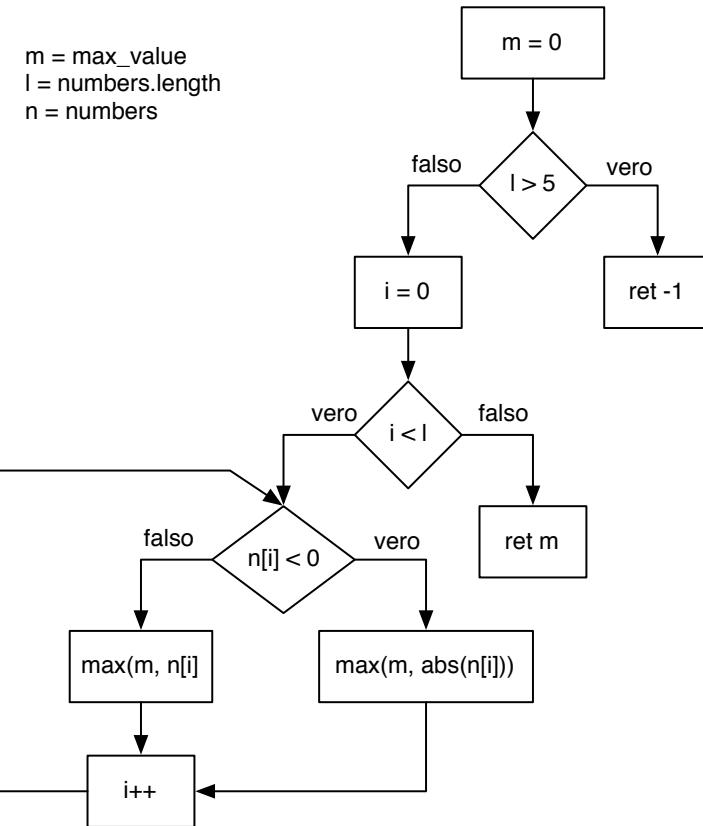
1. Definire le percentuali di copertura delle istruzioni (*statement*) e delle decisioni (*branch*) ottenute eseguendo ogni test separatamente.

Soluzione:

Come misura del numero dei branch consideriamo tutti gli archi del diagramma di flusso di controllo.

T	statement (10)	branch/archi (12)
T1	8	8
T2	8	8
T3	8	8
T4	3	2
T5	9	10
T6	5	4

2. Quale test darebbe errore? **Soluzione:** T6: restituisce 0 e non -1;



Esercizio 4 (punti 7)

Diversi corridori partecipano a una corsa ciclistica a tappe. Ogni corridore appartiene a una squadra e ogni squadra deve schierare 9 corridori al via della manifestazione. Ogni tappa può essere una prova in linea o a cronometro. Ogni corridore può iniziare una tappa e ritirarsi durante il suo svolgimento, o alla fine della medesima. Se una squadra resta con meno di 4 corridori, l'intera squadra è costretta al ritiro. Se completa il percorso, ogni corridore acquisisce un tempo di tappa che verrà utilizzato per stilare la classifica. Questa ovviamente deve elencare solamente i concorrenti ancora in gara, ovvero coloro che non si sono ritirati. La classifica deve tener conto degli abboni assegnati al termine di ogni tappa: 20' al primo classificato, 15' al secondo e 10' al terzo.

- Definire il diagramma delle classi per realizzare il sistema di gestione della corsa. È richiesta anche la definizione dei metodi e degli attributi principali delle classi identificate.
- Realizzare un diagramma di sequenza per spiegare come funziona il metodo che calcola la classifica generale alla fine di ogni tappa.
- Scrivere l'invariante della classe Squadra per imporre il numero minimo di atleti in gara.

Soluzione:

Appello del 29 Giugno 2018



Politecnico di Milano
Anno accademico 2017-2018

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Cugola <input type="checkbox"/> Mottola <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica della classe CodaConRitardo, che implementa l'interfaccia Collection (ed ha quindi anche i metodi `contains`, `containsAll`, `isEmpty`, `removeAll`, `toString`, `size`, ...). La classe definisce una coda la cui testa è l'elemento più in ritardo rispetto alla propria scadenza, ossia quello scaduto da più tempo. Se nessun elemento è ancora scaduto, la *testa non esiste* e il metodo `poll()` restituisce null. Si suppone che la coda non ammetta elementi duplicati, ossia lo stesso oggetto non può essere inserito più volte nella coda.

```
public class CodaConScadenze<E> implements Collection<E> {

    /* costruisce una coda vuota */
    public CodaConScadenze()

    /* inserisce x nella coda se non e' gia' presente (la scadenza e' indicata
       in ms dal tempo attuale);
       * restituisce false sse x e' gia' nella coda */
    public boolean add(E x, long scade)

    /* rimuove tutti gli elementi: */
    public void clear()

    /* ritorna null se la coda e' vuota, altrimenti restituisce, senza
       eliminarla, a testa della coda;
       * se la testa non esiste, restituisce l'elemento con scadenza piu' prossima */
    public E peek()

    /* restituisce ed elimina la testa della coda;
       * ritorna null (senza eliminare nulla) se la testa non esiste
       * (secondo la definizione). */
    public E poll()

    /* restituisce un iteratore su tutti gli elementi (scaduti o no) della coda,
       * in ordine di scadenza.
       * La coda non puo' essere modificata mentre un iteratore e' in uso */
    public Iterator<E> iterator()

    /* restituisce un arrayList che contiene tutti gli elementi della coda, *
       * nell'ordine di scadenza */
    ArrayList<E> toArrayList()

    /* restituisce il valore della scadenza dell'elemento x al momento della
       * chiamata. Se l'elemento non e' presente nella coda, lancia un'eccezione */
    public long scadenza(E x) throws NotFoundException

}
```

Domanda a

Si stabilisca preliminarmente quali metodi sono osservatori. Specificare quindi in JML un invarianto pubblico per la classe. Vi sono proprietà interessanti della classe che non si possono esprimere con un invarianto?

Si specificino i metodi `add()`, `peek()`, `poll()` in funzione (di alcuni) degli osservatori.

Soluzione

Gli osservatori sono `toArrayList()`, `scadenza()` e `peek()`, oltre naturalmente ai metodi di `Collection`: `contains`, `containsAll`, ecc.

L'invariante pubblico stabilisce che non vi sono duplicati:

```
public invariant
  (\forall int i; 0 <= i && i < this.size() - 1;
   (\forall int j; i < j && j < this.size() x;
    toArrayList.get(i) != toArrayList.get(j));
```

La proprieta' che l'ArrayList restituito `toArrayList` e' in ordine di scadenza e' garantita nella postcondizione del metodo, ma non e' errato richiamarla nell'invariante.

Le altre proprieta' tipiche di questa coda sono di tipo evolutivo (ad esempio, tutte le scadenze procedono con lo stesso ritmo) e non possono quindi essere espresse con un invariante.

```
//@ensures
//@\result == (toArrayList().size() == 0 ? null
//@           : toArrayList().get(0));
public /*@ pure */ E peek()

//@ensures
//@\result == !this.contains(e) &&
//@ this.contains(e) && size() == \old(size() + 1) &&
//@ this.containsAll(\old(toArrayList()))
//@ scadenza(x) == scade;
public boolean add(E x, long scade)

//@ensures
//@ toArrayList().size() == 0 || scadenza(toArrayList().get(0)) >= 0
//@ ? (\result == null && toArrayList().equals(\old(ArrayList())))
//@ : (\result == toArrayList().get(0) &&
//@ !this.contains(\result) && size() == \old(size() - 1) &&
//@ \old(toArrayList()).equals(toArrayList().subList(1, toArrayList.size() - 1)));
public E poll()
```

Domanda b

Si consideri la seguente implementazione (parziale) della classe `CodaConScadenze`, che non è ottimizzata dal punto di vista dell'efficienza.

Gli elementi della coda sono memorizzati in un `ArrayList<E>` `elems` in un ordine qualunque.

Un `ArrayList<Long>` `timeStamps` contiene in posizione `i` la scadenza attuale per l'elemento `elems.get(i)`, ovvero il tempo in ms che deve passare prima che l'elemento sia considerato scaduto.

I valori dei tempi sono aggiornati ogni volta che si svolge un'operazione che modifica la coda, come `add` e `poll`, attraverso la chiamata di l'opportuno metodo privato `updateTimeStamps()`. Per implementare questo metodo si è supposta, per semplicità, l'esistenza di una classe di sistema `Time`, il cui metodo `elapsed()` restituisce il tempo in ms passato dall'ultimo azzeramento, ottenuto chiamando il metodo `reset()`.

Un frammento del codice e' mostrato di seguito.

```
public class CodaConScadenze<E> implements Collection<E> {
  private ArrayList<E> elems;
  private Time lastUpdate;
  private ArrayList<Long> timeStamps;

  public CodaConScadenze() {
    elems = new ArrayList<E>;
```

```

        timestamps = new ArrayList<Long>;
        Time lastUpdate = new Time();
        lastUpdate.reset();
    }
    private void updateTimeStamps() {
        long z = lastUpdate.elapsed();
        timestamps.forEach(t -> t-z);
        lastUpdate.reset();
    }
    public boolean add(E x, long scadenza) {
        if elems.contains(x) return false;
        updateTimeStamps();
        elems.add(x);
        timestamps.add(scadenza);
        return true;
    }
    ...
}

```

Si scriva l'invariante di rappresentazione della classe e la funzione di astrazione. Si implementi, eventualmente aiutandosi con commenti, il metodo `poll()`.

Soluzione

RI:

```
private invariant elems!= null && lastUpdate && timestamps!= null &&
!elems.contains(null) && !timestamps.contains(null) &&
elems.size()== timestamps.size();
```

AF:

```
private invariant elems!= null && lastUpdate && timestamps!= null &&
(\forall int i; 0<= i && i<elems.size(); timestamps.get(i)==elems.get(i).scadenza())
toArrayList().containsAll(elems) && toArrayList().size()==elems.size() &&
(\forall int j; 0<= i && j<elems.size()-1;
 toArrayList().get(j).scadenza()<=toArrayList().get(j+1).scadenza());
```

Implementazione di `poll`: Sia `min(ArrayList<Long> x)` un metodo che restituisce l'indice del minimo elemento contenuto in `x`.

```
public E poll() {
    if (elems.size()==0) return null;
    long z = min(timestamps);
    if (z>0) return null;
    x = elems.get(z); //salva in x l'elemento da eliminare;
    /*rimuove elemento in posizione z, sia da elems che da timestamps: */
    elems.set(z,elems.get(elems.size()-1));
    timestamps.set(z,timestamps.get(elems.size()-1));
    elems.remove(elems.size()-1);
    timestamps.remove(elems.size()-1);
    updateTimeStamps();
}
```

Esercizio 2

Si consideri la seguente classe Java che realizza il tipo `Buffer` capace di contenere un massimo di 10 elementi interi positivi. Il valore convenzionale -1 è usato internamente per identificare un elemento vuoto. Il metodo `get(pos)` restituisce il contenuto dell'elemento in posizione `pos`, se presente, altrimenti solleva l'eccezione `EmptyElementException`. Il metodo `set(pos, val)` inserisce il valore `val` in posizione `pos`, sovrascrivendo un eventuale valore già presente.

```
public class Buffer {
    private int[] data;

    public Buffer() {
        data = new int[10];
        for(int i=0; i<10; i++) data[i]=-1;
    }

    public int get(int pos) throws EmptyElementException {
        if(data[pos]==-1) throw new EmptyElementException();
        int x = data[pos];
        data[pos] = -1;
        return x;
    }

    public void set(int pos, int val) {
        data[pos] = val;
    }
}
```

Domanda a

Si modifichi la classe `Buffer` in maniera che il metodo `get(pos)` sospenda il chiamante se l'elemento in posizione `pos` è vuoto, risvegliandolo quando l'elemento richiesto viene inserito. Analogamente, il metodo `set(pos, val)` deve sospendere il chiamante se l'elemento in posizione `pos` è già pieno, in attesa che si svuoti. Si usino solo i meccanismi base della sincronizzazione di Java senza sfruttare le classi dei pacchetti `java.util.concurrent.XXX`.

Soluzione

```
public class SyncBuffer {
    private int[] data;

    public SyncBuffer() {
        data = new int[10];
        for(int i=0; i<10; i++) data[i]=-1;
    }

    public synchronized int get(int pos) throws InterruptedException {
        while(data[pos]==-1) wait();
        int x = data[pos];
        data[pos] = -1;
        notifyAll();
        return x;
    }

    public synchronized void set(int pos, int val) throws InterruptedException {
        while(data[pos] != -1) wait();
        data[pos] = val;
        notifyAll();
    }
}
```

Domanda b

Si aggiunga alla classe sviluppata al punto precedente un metodo `public void setAll(int val)` che imposta tutti gli elementi del buffer al valore `val` (anche se già pieni). Il metodo deve eseguire in maniera asincrona rispetto al chiamante, ovvero il chiamante deve ritornare immediatamente mentre il metodo esegue in un thread separato.

Soluzione

Si noti l'uso della classe anonima per creare il thread che esegue in maniera asincrona il corpo del metodo `setAll`, nonché l'uso della notazione `SyncBuffer.this` per riferirsi all'istanza della classe `SyncBuffer` onde sincronizzarsi correttamente.

```
public class SyncBuffer {  
    ...  
    public void setAll(int val) {  
        new Thread() {  
            public void run() {  
                synchronized(SyncBuffer.this) {  
                    for(int i=0; i<10; i++) data[i] = val;  
                    SyncBuffer.this.notifyAll();  
                }  
            }  
        }.start();  
    }  
}
```

Esercizio 3

Si considerino le seguenti dichiarazioni di classi Java.

```
abstract class Operatore {  
    public abstract void calcola(double a, double b);  
}  
  
class Prodotto extends Operatore {  
    public void calcola(double a, double b) { System.out.println("Prodotto: "+ (a*b)); }  
}  
  
class Somma extends Operatore {  
    public void calcola(double a, double b) { System.out.println("Somma: "+ (a+b)); }  
    public void calcola(int a, int b) { System.out.println("Somma int: "+ (a+b)); }  
}  
  
class SommaDoppia extends Somma {  
    public void calcola(int a, int b) { System.out.println("Somma doppia int: "+ (a+b)*2); }  
}
```

Si consideri poi il seguente frammento di codice Java che utilizza le classi sopra.

1. `Operatore op1, op2;`
2. `SommaDoppia op3, op4;`

3. `op1 = new Somma();`
4. `op2 = new Prodotto();`
5. `op3 = new SommaDoppia();`
6. `op4 = new Somma();`

7. `op1.calcola(3.0,5.0);`

```

8.      op2.calcola(3.0,5.0);
9.      op1.calcola(3,5);
10.     op3.calcola(3,5);
11.     op4.calcola(3,5);

12.    op4 = (SommaDoppia) op3;
13.    op4.calcola(3.0,5.0);
14.    op4 = (SommaDoppia) op1;
15.    op4.calcola(3,5);
16.    op1 = op2;
17.    op1.calcola(3.0,5.0);
18.    op1.calcola(3,5);

```

Domanda a

Si indichino dapprima quali righe del metodo `main` generano un errore di compilazione e quali un errore a tempo di esecuzione, motivando la propria risposta.

Soluzione

La riga 6 non compila essendo il tipo dinamico della variabile `op4` (`Somma`) una sopraclasse (e non una sottoclasse) del tipo statico (`SommaDoppia`)

La riga 11 va eliminata come conseguenza del punto precedente (errore: “variable `op4` might not have been initialized”).

Infine, la riga 14 compila correttamente (il compilatore “si fida” del nostro cast) ma viene restituita un’eccezione a tempo di esecuzione (“`java.lang.ClassCastException: Somma cannot be cast to SommaDoppia`”) perchè l’oggetto il cui riferimento è contenuto in `op1` non è di tipo `SommaDoppia` né di una sottoclasse (è di tipo `Somma`).

Domanda b

Supponendo che tutte le righe che provocano errori in fase di compilazione e/o di esecuzione siano state rimosse, illustrare l’output del programma, motivando brevemente la propria risposta.

Soluzione

```

7.  Somma: 8.0
8.  Prodotto: 15.0
9.  Somma: 8.0
10. Somma doppia int: 16
13. Somma: 8.0
15. Somma doppia int: 16
17. Prodotto: 15.0
18. Prodotto: 15.0

```

Le prime due stampe sono ovvie.

La riga 9 invoca il metodo `calcola(double, double)` e non il metodo `calcola(int, int)` essendo la scelta del metodo da invocare (signature) effettuata a tempo di compilazione, guardando solo al tipo statico e il tipo statico di `op1` è `Operatore`, che ha il solo metodo `calcola(double, double)`. Il codice effettivamente eseguito è poi determinato a tempo di esecuzione e in questo caso viene eseguito il codice del metodo `calcola(double, double)` di `Somma`,

La stampa relativa alla riga 10 è ovvia (tipo statico e dinamico di `op3` sono `SommaDoppia`).

Alla riga 13 il tipo statico e quello dinamico della variabile `op4` sono entrambi `SommaDoppia`, classe che a sua volta possiede entrambi i metodi `calcola`. In questo caso il compilatore (sulla base del tipo dei parametri) sceglie la versione `double` ereditata dalla classe `Somma` e non ridefinita.

La riga 15, visto che la riga 14 non deve essere considerata (si veda la risposta precedente), vede ancora una volta tipo statico e dinamico `SommaDoppia`. In questo caso il compilatore sceglie la versione `int`, ridefinita in `SommaDoppia`.

Le stampa delle righe 17 e 18 sono ovvie: tipo statico Operatore, tipo dinamico Prodotto, unico metodo calcola(double, double), quindi conversione automatica del tipo dei parametri da int a double alla riga 18.

Esercizio 4

Si consideri il seguente frammento di codice, espresso in pseudocodice:

```
READ foo
READ boo
IF boo > 12 OR foo < 70 THEN
    PRINT "pippo "
END IF
PRINT "pluto "
IF foo > 75 THEN
    PRINT "topolino!"
END IF
```

e si indichi quale tra copertura delle istruzioni, copertura dei cammini, copertura delle diramazioni (branch), e copertura delle condizioni viene massimizzata da ciascuna delle seguenti test suite. La risposta potrebbe non essere univoca.

1. boo = 13, foo = 77
2. boo = 14, foo = 74
3. boo = 10, foo = 80; boo = 13, foo = 80
4. boo = 10, foo = 80; boo = 13, foo = 80; boo = 13, foo = 73
5. boo = 10, foo = 80; boo = 13, foo = 80; boo = 9, foo = 60

Si indichi inoltre se si ritiene necessario modificare una test suite o aggiungerne una ulteriore per migliorare la copertura totale ottenuta dalle cinque suite sopra. Se sì, indicate quale suite si ritiene di voler modificare o aggiungere, e perché.

Soluzione

Per ciascuna delle test suite proposte, vale il seguente:

1. Ottiene copertura 100% per le istruzioni.
2. Nessuna copertura viene massimizzata.
3. Ottiene copertura 100% per le istruzioni; notare che la copertura delle diramazioni rimane incompleta.
4. Ottiene copertura 100% per le istruzioni e per le diramazioni.
5. Ottiene copertura 100% per le istruzioni e per le condizioni; quest'ultima include le diramazioni.

Considerate queste test suite, l'unico criterio che non è mai massimizzato è quello dei cammini. Per avere 100% di copertura in quel caso, è sufficiente ad esempio aggiungere un test case che copra entrambi i rami false alla test suite 4., come boo=10, foo = 72.

16 gennaio 2020



Politecnico di Milano
Anno accademico 2019-2020

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>	
Nome:	Matricola:	
Sezione (segnarne una): <input type="checkbox"/> Cugola	<input type="checkbox"/> Margara	<input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica di un dizionario: una struttura dati adatta a memorizzare un insieme finito di parole costruite a partire da un alfabeto.

```
public interface Dictionary {
    // \result==true sse word è in this
    public /*@ pure */ boolean contains(String word);

    // aggiunge word al dizionario
    public void add(String word) throws NullPointerException;

    // \result==true se è presente nel dizionario una parola x della stessa
    // lunghezza di word che differisce al più di un carattere da word stessa
    public /*@ pure */ boolean similar(String word);

    // \result è una lista di tutte le parole nel dizionario che iniziano per prefix;
    // se non ve n'è nessuna, \result è la lista vuota.
    // La stringa prefix deve essere lunga almeno tre caratteri
    public /*@ pure */ List<String> complete(String prefix);
}
```

Si supponga che tutte le parole siano costituite esclusivamente dai 26 caratteri minuscoli (a, b, ..., x, y, z).

NB: Per estrarre il carattere (di tipo `char`) in posizione *i* di una stringa `word`, si può usare la notazione `word.charAt(i)`, con *i* nell'intervallo `0...word.length()-1`.

Domanda a)

Si specifichino in JML i metodi `add`, `similar`, e `complete`.

Soluzione

```
//@ensures word!=null && contains(word) &&
//@ (\forall String x; \old(contains(x)); contains(x)) &&
//@ (\forall String x; contains(x); x.equals(word) || \old(contains(x)));
//@signals(NullPointerException e) word == null &&
//@ (\forall String x;; \old(contains(x)) <==> contains(x));

public void add(String word) throws NullPointerException;

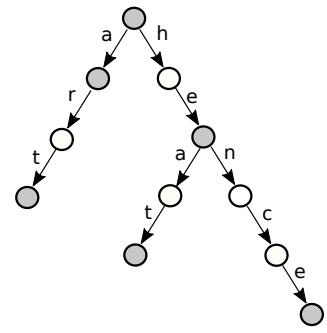
//@requires word!=null
//@ensures \result <==>
//@      (\exists String x; contains(x) && x.length()==word.length() &&
//@          (\numof int i; 0<=i & i<x.length();
//@                  x.charAt(i)!=word.charAt(i))<=1);

public /*@ pure */ boolean similar(String word);

//@requires prefix!=null && prefix.length>=3;
//@ensures (forall String x;;
//@           \result.contains(prefix+x) <==> this.contains(prefix+x));
public /*@ pure */ List<String> complete(String prefix);
```

Domanda b)

Si consideri la seguente implementazione (parziale) di un dizionario, tramite una struttura dati comunemente nota come Trie. Un trie memorizza ogni parola del dizionario in un albero in modo da consentire una ricerca veloce. Ogni nodo dell'albero può avere fino a 26 figli, uno per ciascuna lettera. Ogni figlio di un nodo corrisponde a un carattere. Le parole sono lette partendo dalla radice e arrivando fino a un nodo marcato come “end-of-word”. Tutte le foglie sono “end-of-word”, ma anche alcuni nodi intermedi possono esserlo. Un esempio di Trie è mostrato in figura. La radice corrisponde alla parola vuota, che si assume faccia sempre parte del dizionario. In grigio sono raffigurati i nodi “end-of-word”. Le parole (non vuote) presenti sono: a, art, he, heat, hence.



```

class Node {
    Node next[] = new Node[26];
    boolean isEndOfWord = false;
}

class Trie implements Dictionary {
    private final Node root;

    // Costruisce un trie con la sola radice (corrispondente alla parola vuota).
    public Trie() {
        root = new Node(); root.isEndOfWord = true;
    }

    // Metodo helper che trova il nodo finale della sequenza che corrisponde alla
    // stringa prefix, se esiste. Il nodo restituito non è necessariamente endOfWord.
    // Restituisce null se prefix non è presente nel trie.
    private /* @ pure helper */ Node terminal(String prefix) {
        if (prefix.equals("")) return root;
        Node current = root;
        for (char x : prefix.toLowerCase().toCharArray()) {
            int index = x - 97;
            if (current.next[index] == null) return null;
            current = current.next[index];
        }
        return current;
    }

    public boolean contains(String word) {
        final Node n = terminal(word);
        if (n == null || !n.isEndOfWord) return false;
        return true;
    }

    public void add(String word) throws NullPointerException {
        if (word == null) throw new NullPointerException();
        if (word.equals("")) return;
        Node current = root;
        for (final char x : word.toLowerCase().toCharArray()) {
            int index = x - 97;
            if (current.next[index] == null) current.next[index] = new Node();
            current = current.next[index];
        }
        current.isEndOfWord = true;
    }
}

```

Il metodo privato `terminal` permette di trovare il nodo che corrisponde al prefisso di una parola presente nel dizionario. È utile per implementare il metodo `contains` e gli altri metodi di `Dictionary`, che per semplicità non riportiamo.

Scrivere un invarianto di rappresentazione per la classe `Trie`. Si suggerisce di sfruttare il metodo privato (puro) `terminal`.

Soluzione

L'invariante deve stabilire che la struttura dati è un albero, ossia che non vi sono stringhe differenti che terminano nello stesso nodo e che `isEndOfWord` è vero per tutte le foglie.

```
private invariant
    root != null && root.isEndOfWord &&
    //non ci sono stringhe diverse che terminano nello stesso nodo
    (\forall String x; terminal(x)!=null;
        (\forall String y, terminal(y)!=null;
            !x.equals(y) ==> !terminal(x).equals(terminal(y)))) &&
    // isEndOfWord è vero per tutte le foglie:
    (\forall String x; terminal(x)!=null;
        (\forall int i; 0<=i && i<=25; terminal(x).next(i)==null) ==>
        terminal(x).isEndOfWord);
```

Esercizio 2

Si consideri la seguente classe Java:

```
public class Account {
    private static List<Double> log = new ArrayList<Double>();
    private double balance;

    public Account() { balance = 0.0; }

    public synchronized void deposit(double val) {
        balance = balance+val;
        log.add(val);
    }

    public synchronized void withdraw(double val) {
        balance = balance-val;
        log.add(-val);
    }
}
```

Domanda a)

La sincronizzazione è corretta? In caso affermativo motivare la risposta, in caso negativo indicare come andrebbe modificato il codice per avere una sincronizzazione corretta.

Soluzione

Avere sincronizzato i due metodi `deposit` e `withdraw` non è sufficiente perchè l'attributo `log` è statico, quindi condiviso da tutte le istanze della classe. Occorre sincronizzare l'accesso all'attributo `log` modificando la linea:

```
log.add(val)
in:
synchronized(log)    log.add(val);
e la linea:
log.add(-val)
in:
synchronized(log)    log.add(-val);
```

Domanda b)

Modificare la classe affinchè il chiamante del metodo withdraw venga sospeso se mancano fondi per il prelievo.

Soluzione

```
public class Account {  
    private static List<Double> log = new ArrayList<Double>();  
    private double balance;  
  
    public Account() { balance = 0.0; }  
  
    public synchronized void deposit(double val) {  
        balance = balance+val;  
        synchronized(log) { log.add(val); }  
        notifyAll();  
    }  
  
    public synchronized void withdraw(double val) {  
        while(balance<val) try { wait(); } catch(Exception ex) { ex.printStackTrace(); }  
        balance = balance-val;  
        synchronized(log) { log.add(-val); }  
    }  
}
```

Domanda c)

Si aggiunga alla classe un metodo `void printLog()` che stampa l'intero `log` in maniera asincrona rispetto al chiamante.

Soluzione

```
public void printLog() {  
    List<Double> logCopy;  
    synchronized(log) {  
        logCopy = new ArrayList<Double>(log);  
    }  
    new Thread(() -> logCopy.forEach(System.out::println)).start();  
}
```

Esercizio 3

Si considerino le seguenti dichiarazioni di classi Java.

```
public class Customer { ... }  
public class RegisteredCustomer extends Customer { ... }  
  
public class Shop {  
    public int computeDiscount(Customer c) { return 0; }  
}  
public class OnlineShop extends Shop {  
    public int computeDiscount(Customer c) { return 5; }  
    public int computeDiscount(RegisteredCustomer c) { return 15; }  
}  
public class OnlinePremiumShop extends OnlineShop {
```

```

public int computeDiscount(RegisteredCustomer c) {
    return 20 + super.computeDiscount(c);
}
}

```

Si consideri poi il seguente frammento di codice Java che utilizza le classi dichiarate sopra.

```

01. Customer c1 = new Customer();
02. Customer c2 = new RegisteredCustomer();
03. RegisteredCustomer c3 = new RegisteredCustomer();
04. Shop s1 = new Shop();
05. Shop s2 = new OnlinePremiumShop();
06. OnlineShop s3 = new OnlinePremiumShop();
07. OnlinePremiumShop s4 = new OnlineShop();
08. OnlineShop s5 = new OnlineShop();
09. System.out.println(s1.computeDiscount(c1));
10. System.out.println(s1.computeDiscount(c2));
11. System.out.println(s2.computeDiscount(c2));
12. System.out.println(s2.computeDiscount(c3));
13. System.out.println(s3.computeDiscount(c2));
14. System.out.println(s3.computeDiscount(c3));
15. System.out.println(s4.computeDiscount(c1));
16. System.out.println(s4.computeDiscount(c2));
17. System.out.println(s5.computeDiscount(c1));
18. System.out.println(s5.computeDiscount(c3));
19. s2 = s1;
20. s3 = s1;
21. s5 = s2;
22. s3 = s5;

```

Domanda a)

Si indichino quali righe del codice sopra riportato generano un errore in fase di compilazione, motivando brevemente la propria risposta.

Soluzione

- La riga 7 non compila in quanto `OnlineShop` non è sottotipo di `OnlinePremiumShop`. Di conseguenza non compilano le righe che utilizzano `s4`, ovvero le righe 15 e 16.
- La riga 20 non compila in quanto il tipo statico di `s1` non è sottotipo del tipo statico di `s3`.
- La riga 21 non compila in quanto il tipo statico di `s2` non è sottotipo del tipo statico di `s5`.

Domanda b)

Supponendo che tutte le righe che provocano errori siano state rimosse, scrivere l'output prodotto dal programma, motivando brevemente la risposta.

Soluzione

```

0 // Chiama il metodo con parametro Customer in Shop
0 // Chiama il metodo con parametro Customer in Shop
5 // Chiama il metodo con parametro Customer in OnlineShop
5 // Chiama il metodo con parametro Customer in OnlineShop
5 // Chiama il metodo con parametro Customer in OnlineShop

```

```

35 // Chiama il metodo con parametro RegisteredCustomer in OnlinePremiumShop
5 // Chiama il metodo con parametro Customer in OnlineShop
15 // Chiama il metodo con parametro RegisteredCustomer in OnlineShop

```

Esercizio 4

Si consideri il seguente frammento di codice:

```

int f(int a, int b) {
    if (a > b) {
        System.out.println("A");
    } else {
        System.out.println("B");
    }
    for (int i=0; i<1000; i++) {
        if (a > 0) break;
        else a = 1;
    }
}

```

Domanda a)

Definire un insieme minimo di casi di test per coprire tutte le istruzioni (statement coverage)

Soluzione

Servono due casi di test: [a=-1, b=1], [a=2, b=1]

Domanda b)

Definire un insieme minimo di casi di test per coprire tutte le decisioni (edge coverage).

Soluzione

Non è possibile coprire tutte le decisioni, in quanto la condizione `i<1000` non viene mai valutata come `false`. Per valutare tutte le decisioni che possono essere coperte, i casi di test precedenti sono sufficienti.

Domanda c)

Definire un insieme minimo di casi di test per coprire tutti i cammini (path coverage).

Soluzione

L'esecuzione passa necessariamente dal ramo `if` oppure dal ramo `else`, ed entra nel ciclo `for` una oppure due volte. I possibili cammini sono dunque 4: uno che entra nel ramo `if` ed esegue il ciclo una volta, uno che entra nel ramo `if` ed esegue il ciclo due volte, uno che entra nel ramo `else` ed esegue il ciclo una volta, uno che entra nel ramo `else` ed esegue il ciclo due volte.

Servono quindi 4 casi di test: [a=1, b=0], [a=1, b=2], [a=-1, b=0], [a=-1, b=-2]

Ingegneria del Software – a.a. 2006/07

Appello del 23 luglio 2007

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1 ora e 50 minuti.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 8)

Il seguente metodo statico

```
public static int rigaSommaMax(int [] m) {... }
```

accetta come ingresso un array a due dimensioni m di cui si assume che rappresenti una matrice rettangolare (cioè tutte le righe hanno lo stesso numero di elementi; si noti che una matrice quadrata, in cui il numero delle righe è uguale a quello delle colonne, è considerato un caso particolare di matrice rettangolare, quindi accettabile come parametro) avente almeno due righe e due colonne, e produce come risultato l'indice della riga che i cui elementi hanno somma massima (se più righe hanno lo stesso valore massimo il metodo restituisce l'indice di una qualsiasi di queste).

a) Si scriva una specifica, in JML del metodo indicando la pre- e la post-condizione.

b) si trasformi la specifica precedente facendo in modo che il metodo accetti in ingresso una qualsiasi “matrice” con almeno due righe e due colonne, ma se la matrice non è rettangolare (cioè le righe non sono tutte della stessa lunghezza) esso lanci un’eccezione *MatriceNonRettangolareException* (tutti gli altri requisiti sono invariati).

Esercizio 2 (punti 13)

Si consideri un sistema di gestione del traffico telefonico. Il sistema, fra le altre cose, tiene traccia degli utenti e delle loro chiamate. Una chiamata ha una data, un'ora, una durata e un numero. Le chiamate possono essere vocali, e in tal caso hanno anche uno scatto alla risposta e un costo al secondo, o di tipo dati, che invece hanno un volume di dati scambiati (in Kbyte) e un costo in volume (ossia al Kbyte).

Formalmente, le tre classi sono:

```
abstract public class Chiamata {  
    abstract public Data data();  
    abstract public int ora(); //HHSS  
    abstract public int durataInSecondi();  
    abstract public int costoTotale();  
    abstract public String numeroChiamato();  
}  
  
public class ChiamataVocale extends Chiamata {  
    public int costoScatto();  
    public int costoAlSecondo();  
}  
  
public class ChiamataDati extends Chiamata {  
    public int costoKB();  
    public int volumeKByte();  
}
```

La specifica della classe Data è:

```
public classe Data {  
    public int giorno();  
    public int mese();  
    public int anno();  
    //@ ensures (*\result è true sse this precede strettamente d *)  
    public boolean precede(Data d);  
}
```

Per ogni utente, il sistema tiene traccia del nome, del credito residuo e delle chiamate effettuate. E' inoltre possibile effettuare una ricarica di un numero intero di euro.

L'interfaccia della classe è così definita:

```
public class Utente {  
    //@ensures (* \result è il credito in centesimi di euro ancora disponibile *)  
    public int creditoResiduo();  
    //@ensures ....  
    public void ricarica(int euro);  
    //@ensures (*\result è un generatore, in ordine cronologico, alle chiamate effettuate alla data d*)  
    public Iterator<Chiamata> chiamateInData(Data d);  
}
```

a) Scrivere in JML la postcondizione del metodo ricarica.

b) Sia ora dato il seguente rep per la classe Utente:

```
private ArrayList<Chiamata> log;  
//contiene tutte le chiamate effettuate in ordine cronologico  
int spesaTotaleAnnoCorrente; //la spesa complessiva in centesimi di euro dell'utente  
nell'anno ) corrente fino alla data attuale  
Data dataCorrente; //la data attuale  
int totaleRicariche; //il credito complessivo, in euro, acquistato dall'utente  
int creditoResiduo; //il credito ancora disponibile
```

Scrivere l'invariante di rappresentazione che stabilisce la correttezza e la mutua consistenza dei dati che costituiscono il rep.

c) Con riferimento al rep introdotto al punto precedente, *implementare il metodo iteratore chiamateInData()*, includendo anche la definizione di tutte le classi appropriate.

Esercizio 3 (punti 4)

Si consideri la seguente variante della classe Utente introdotta nell'esercizio 2, con parte dell'interfaccia definita come segue.

```
public class Utente {  
    ....  
    //@ensures (*\result è la somma del costo industriale di tutte le chiamate effettuate  
    // @ alla data d*)  
    public int costoIndustrialeChiamateInData(Data d);  
  
    //@ensures \result > costoIndustrialeChiamateInData(d) / 2 &&  
    // @ \result < costoIndustrialeChiamateInData(d) * 2  
    public int importoFatturabileInData(Data d);  
}
```

Vengono inoltre definite le due classi UtenteAgevolato e UtentePrivilegiato, possibili classi eredi di Utente, che usufruiscono di tariffe particolarmente favorevoli.

```
public class UtenteAgevolato {  
    ....  
    //@ensures \result == costoIndustrialeChiamateInData(d) * 0.7  
    public int importoFatturabileInData(Data d);  
}  
  
public class UtentePrivilegiato {  
    ....  
    public final int TETTO = ...;  
    //@ensures \result ==  
    // @ costoIndustrialeChiamateInData(d) * 1.5 <= TETTO ?  
    // @ costoIndustrialeChiamateInData(d) * 1.5 : TETTO  
    public int importoFatturabileInData(Data d);  
}
```

Le classi UtenteAgevolato e UtentePrivilegiato sono definibili come classi eredi della classe Utente senza violare il principio di sostituzione? Motivare adeguatamente la risposta, nel caso positivo argomentando che sono soddisfatte le regole di sostituibilità, oppure nel caso negativo mostrando che non lo sono mediante un controesempio. Risposte non motivate, anche se corrette, non sono accettabili.

Appello del 25 Gennaio 2019



Politecnico di Milano
Anno accademico 2018-2019

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>	
Nome:	Matricola:	
Sezione (segnarne una): <input type="checkbox"/> Cugola	<input type="checkbox"/> Margara	<input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri il seguente tipo di dato astratto che definisce una `MultiList<T>`. Una `MultiList<T>` è un atomo (ovvero un elemento di un generico tipo `T`) oppure una lista di `MultiList<T>`.

Per esempio, `(1 (2) 3 4 ((5 6) 7))` è una rappresentazione testuale di una `MultiList<Integer>`, il cui primo elemento (`first`) è l'atomo `1` e il cui resto (`rest`) è la `MultiList` `((2) 3 4 ((5 6) 7))`, ovvero la `MultiList` privata del suo primo elemento.

Altri esempi di `MultiList<Integer>` sono l'atomo `3` e la `MultiList` `(3)`. Si noti che queste due `MultiList<Integer>` sono diverse: la prima è una `MultiList` “atomica”, mentre la seconda è una “vera” `MultiList` che contiene al suo interno l’atomo `3`.

```
public class MultiList<T> {
    // Crea una MultiList vuota
    public MultiList() {
        ...
    }

    // Crea una MultiList composta dal solo atomo x (non nullo)
    public MultiList(T x) {
        ...
    }

    // Ritorna true se this e' vuota
    public /*@ pure @*/ boolean isEmpty()

    // Ritorna true se this e' un atomo
    public /*@ pure @*/ boolean isAtom()

    // Ritorna il valore dell'atomo se this e' un atomo
    // Lancia un'eccezione opportuna in ogni altro caso
    public /*@ pure @*/ T getAtom() throws EmptyException, MalformedListException;

    // Ritorna il primo elemento di this, se this non e' un atomo e non e' vuota.
    // Lancia un'eccezione opportuna in ogni altro caso
    public /*@ pure @*/ MultiList<T> first() throws EmptyException, MalformedListException;

    // Ritorna la MultiList<T> che contiene tutti gli elementi
    // di this tranne il primo.
    // Lancia un'eccezione opportuna se this e' vuota o se e' un atomo
    public /*@ pure @*/ MultiList<T> rest() throws EmptyException, MalformedListException;

    // Aggiunge l'elemento x (una MultiList<T> non nulla) come primo elemento di this.
    // Lancia una MalformedListException se this e' un atomo
    public void addFirst(MultiList<T> x) throws MalformedListException;

    // Rimuove il primo elemento (una MultiList<T>) da this.
    // Lancia un'eccezione opportuna se this e' vuota o se e' un atomo
    public void removeFirst() throws EmptyException, MalformedListException;

    // Ritorna true se this contiene la MultiList<T> x come elemento
    // a qualsiasi livello di annidamento.
    // Lancia una MalformedListException se this e' un atomo
    // (mentre ritorna false se this e' empty)
    public /*@ pure @*/ boolean contains(MultiList<T> x) throws MalformedListException;

    // Ritorna true se this e that sono entrambi atomi e hanno
    // lo stesso valore (this.atom.equals(that.atom))
    // oppure se sono entrambe MultiList "vere" che contengono
    // gli stessi elementi nello stesso ordine
    @Override
    public /*@ pure @*/ boolean equals(Object that) {
        ...
    }

    // Ritorna una copia di this, tale per cui this.equals(copy())
    public /*@ pure @*/ MultiList<T> copy();
}
```

Domanda a)

Si specificino in JML i metodi addFirst, removeFirst e contains.

Soluzione

```
//@requires x!=null;
//@ensures !isAtom() && first().equals(x) && rest().equals(\old(copy()));
//@signals(MalformedListException e) \old(isAtom()) && copy().equals(\old(copy()))
public void addFirst(MultiList<T> x) throws MalformedListException;

//@ensures !isAtom() && !isEmpty() && this.equals(\old(rest()));
//@signals(EmptyException e) \old(isEmpty()) && isEmpty();
//@signals(MalformedListException e) \old(isAtom()) && copy().equals(\old(copy()))
public void removeFirst() throws EmptyException, MalformedListException ;

//@requires x!=null;
//@ensures !this.isAtom() && (\result==true <=> (!this.isEmpty() &&
//@          (first().equals(x) ||
//@          (!first().isAtom() && first().contains(x)) ||
//@          (rest()!=null && rest().contains(x)))) )
//@signals(MalformedListException e) \old(isAtom()) && copy().equals(\old(copy()))
public /*@ pure */ boolean contains(MultiList<T> x) MalformedListException {
```

Domanda b)

Si consideri la seguente implementazione di una MultiList<T>.

```
public class MultiList<T> {
    private T atom;
    private List<MultiList<T>> list;

    public MultiList() {
        atom = null; list = new ArrayList<MultiList<T>>();
    }
    public MultiList(T x) { // richiede che x non sia null
        atom = x; list = null;
    }
    public /*@ pure */ boolean isEmpty() {
        return atom==null && list.isEmpty();
    }
    public /*@ pure */ boolean isAtom() {
        return atom != null;
    }
    public /*@ pure */ T getAtom() throws EmptyException, MalformedListException {
        if(isEmpty()) throw new EmptyException();
        if(!isAtom()) throw new MalformedListException();
        return atom;
    }
    public /*@ pure */ MultiList<T> first() throws EmptyException, MalformedListException {
        if(isEmpty()) throw new EmptyException();
        if(isAtom()) throw new MalformedListException();
        return list.get(0);
    }
    public /*@ pure */ MultiList<T> rest() throws EmptyException, MalformedListException {
        if(isEmpty()) throw new EmptyException();
        if(isAtom()) throw new MalformedListException();
        if(list.size()==1) return null;
        MultiList<T> res = new MultiList<T>();
        res.list.addAll(list.sublist(1,list.size()));
        return res;
    }
}
```

```

public void addFirst(MultiList<T> x) throws MalformedListException {
    if(isAtom()) throw new MalformedListException();
    list.add(0, x);
}
public void removeFirst() throws EmptyException, MalformedListException {
    ifisEmpty() throw new EmptyException();
    if(isAtom()) throw new MalformedListException();
    list.remove(0);
}
public /*@ pure */ boolean contains(MultiList<T> x) MalformedListException {
    if(isAtom()) throw new MalformedListException();
    ifisEmpty() return false;
    if(first().equals(x)) return true;
    if(!first().isAtom() && first().contains(x)) return true;
    return !rest().isEmpty() && rest().contains(x);
}
@Override
public /*@ pure */ boolean equals(Object that) {
    // ....
}
public /*@ pure */ MultiList<T> copy() {
    // ....
}
}

```

Scrivere in JML l'invariante di rappresentazione (Rep Invariant) per MultiList<T>.

Soluzione

```

//@private invariant (atom==null <==> list!=null)  &&
//@                      (list!=null ==>
//@                          (\forall int i; 0<=i && i<list.size();
//@                            list.get(i)!=null));

```

Servirebbe anche una condizione di non circolarità della MultiList. Questa si puo' ottenere ad esempio definendo un nuovo predicato boolean /*@ pure */ containsEqual(MultiList<T> lista, MultiList<T> elem), identico a contains() ma usando equals:

```

//@ensures !this.isAtom() && (\result==true <==> (!this.isEmpty() &&
//@          (first() == x) ||
//@          (!first().isAtom() && first().containsEqual(x)) ||
//@          (!rest().isEmpty() && rest().containsEqual(x))))
//@boolean /*@ pure */ containsEqual(MultiList<T> lista, MultiList<T> elem)

```

e aggiungendo inoltre un'opportuna condizione nel RI:

```

(\forall int i; 0<=i && i<list.size();
 ! (list.get(i).containsEqual(list.get(i))))

```

Tuttavia, l'implementazione di addFirst puo' inserire un oggetto anche se questo gia' esistente nella lista, pertanto violando questa parte dell'invariante.

Domanda c)

Si consideri la classe NullMultiList<T> che modifica MultiList<T> rimuovendo il vincolo che una MultiList<T> non possa contenere atomi nulli. La classe NullMultiList<T> è un'estensione valida di MultiList<T> secondo il principio di sostituzione di Liskov? Perché? È vero il contrario?

Soluzione

NullMultiList<T> indebolisce la pre-condizione dei metodi in MultiList<T> (in particolare, il costruttore di un atomo e il metodo addFirst), ma indebolisce anche la post-condizione del metodo first, dato che questo potrebbe ritornare un valore nullo. Di conseguenza NullMultiList<T> non è un'estensione valida di MultiList<T> e nemmeno il contrario.

Esercizio 2

Si consideri la seguente interfaccia TenIntArray che rappresenta un array di interi che può contenere al massimo 10 elementi interi. Un chiamante può accedere agli elementi dell'array mediante il loro indice (metodo get()), aggiungere un elemento in ultima posizione (metodo add()) e rimuovere l'elemento in ultima posizione (metodo remove()). Se il chiamante prova ad aggiungere un elemento in un TenIntArray pieno, viene sospeso fino a quando un altro thread rimuove un elemento.

```
public interface TenIntArray {  
  
    /**  
     * Ritorna il valore contenuto in posizione pos, se presente,  
     * altrimenti lancia una InvalidIndexException.  
     */  
    public int get(int pos) throws InvalidIndexException;  
  
    /**  
     * Rimuove l'ultimo elemento dall'array.  
     * Se l'array e' vuoto non fa nulla.  
     */  
    public void remove();  
  
    /**  
     * Aggiunge val nella prima posizione libera dell'array.  
     * Se l'array e' pieno, blocca il chiamante fino a quando una  
     * posizione si libera.  
     */  
    public void add(int val);  
}
```

Domanda a)

Si completi il codice della classe TenIntArrayImpl che implementa TenIntArray. Si usino solo i meccanismi base della sincronizzazione di Java senza sfruttare classi e pacchetti di java.util.concurrent.XXX.

```
public class TenIntArrayImpl implements TenIntArray {  
    private final int array[];  
    private int size; // Indica l'indice della prima posizione libera nell'array  
  
    public TenIntArrayImpl() {  
        array = new int[10];  
        size = 0;  
    }  
  
    ...  
}
```

Soluzione

```
public class TenIntArrayImpl implements TenIntArray {  
    private final int array[];  
    private int size; // Indica l'indice della prima posizione libera nell'array
```

```

public TenIntArrayImpl() {
    array = new int[10];
    size = 0;
}

@Override
public synchronized int get(int pos) throws InvalidIndexException {
    if (pos < 0 || pos >= size) {
        throw new InvalidIndexException();
    } else {
        return array[pos];
    }
}

@Override
public synchronized void remove() {
    if (size > 0) {
        size--;
        notifyAll();
    }
}

@Override
public synchronized void add(int val) {
    while (size == 10) {
        try {
            wait();
        } catch (final InterruptedException e) {
            e.printStackTrace();
        }
    }
    array[size] = val;
    size++;
}
}

```

Domanda b)

Si consideri la seguente classe Test che utilizza TenIntArray. Indicare per quali valori di k si può garantire che tutti i thread lanciati nel main() terminino.

```

public class Test implements Runnable {
    private final TenIntArray t;
    private final int id;

    public Test(TenIntArray t, int id) {
        this.t = t;
        this.id = id;
    }

    @Override
    public void run() {
        if (id%2!=0) {
            t.add(id);
        } else {
            t.remove();
        }
    }
}

public static void main(String[] args) {

```

```

        final int k = ???;
        final TenIntArray arr = new TenIntArrayImpl();
        for (int i = 1; i <= k; i++) {
            final Thread thread = new Thread(new Test(arr, i));
            thread.start();
        }
    }
}

```

Soluzione

Alcuni thread potrebbero non terminare per $k > 20$. In questo caso, potrebbero essere eseguiti per primi tutti i thread con indice i pari, che lasciano t vuoto. Successivamente 10 thread con i dispari potrebbero riempire t , bloccando ogni successivo thread con i dispari che invoca $t.add(i)$.

Esercizio 3

Si considerino le seguenti dichiarazioni di tipi Java.

```

interface Product {
    public double getPrice();
}

class Drink implements Product {
    private double price;
    public Drink(double price) { this.price = price; }
    public double getPrice() { return price; }
    public boolean moreExpensive(Drink d) { return this.getPrice() > d.getPrice(); }
}

class Coca extends Drink {
    public Coca(double price) { super(price); }
    public int getCalories() { return 200; }
}

class CocaZero extends Coca {
    public CocaZero(double price) { super(price); }
    public int getCalories() { return 0; }
    public boolean moreExpensive(CocaZero c) { return false; }
}

```

Si consideri poi il seguente codice Java che utilizza le classi sopra.

```

public class Main {
    public static void main(String[] args) {
1.    Product p;
2.    Drink d1, d2;
3.    Coca c1, c2;

4.    p = new Coca(2.0);
5.    d1 = p;
6.    d2 = new Drink(1.5);
7.    c1 = new Coca(2.1);
8.    c2 = new CocaZero(2.5);

9.    System.out.println(p.getPrice());
10.   System.out.println(d1.getPrice());
11.   System.out.println(d2.getPrice());
12.   System.out.println(c1.getPrice());
13.   System.out.println(c2.getPrice());
}
}

```

```

14. System.out.println(p.getCalories());
15. System.out.println(c1.getCalories());
16. System.out.println(c2.getCalories());

17. System.out.println(c1.moreExpensive(d2));
18. System.out.println(c1.moreExpensive(c2));

19. CocaZero cz = new CocaZero(2.0);
20. System.out.println(c2.moreExpensive(cz));
}
}

```

Domanda a)

Si indichino quali righe del metodo `main` generano un errore di compilazione e perchè.

Soluzione

Linea 5. non è possibile assegnare ad un `Drink` un `Product` (`Product` è sopra-classe, non sotto-classe di `Drink`).

Linea 10. eliminata la linea 5 anche la 10 deve essere eliminata.

Linea 14. La classe `Product` non ha il metodo `getCalories`.

Domanda b)

Supponendo che tutte le righe che provocano errori in fase di compilazione siano state rimosse, illustrare l'output del programma, spiegando quali metodi vengono invocati e perchè.

Solution

Output:

```

2.0
1.5
2.1
2.5
200
0
true
false
true

```

Si noti in particolare l'ultimo risultato: viene invocato il metodo `moreExpensive` di `Drink` e non quello di `CocaZero` visto che `CocaZero` non ridefinisce il metodo `moreExpensive` di `Drink` ma ne aggiunge un altro con diversa signature (non si applica quindi il binding dinamico).

Esercizio 4

Si consideri la seguente funzione:

```

public static int fun(int a, int b) {
    int temp = 0;
    if (a > b) {
        temp++;
    }
    if (a + b > 10) {
        temp += a;
    } else {
        temp += b;
    }
}

```

```
    }
    return temp;
}
```

Si definisca un insieme di casi di test per ciascuna delle seguenti richieste:

- a) L'insieme copre tutte le istruzioni (statement coverage);
- b) L'insieme copre tutti i branch/decisioni (branch/edge coverage);
- c) L'insieme copre tutti i cammini (path coverage).

Soluzione

- a) T1(a=10, b=5) T2(a=1, b=1)
- b) T1(a=10, b=5) T2(a=1, b=1)
- c) T1(a=10, b=5) T2(a=1, b=1) T3(a=2, b=1) T4(a=10, b=20)

Il caso T1 copre le istruzioni dentro i due rami `if`. Il caso T2 copre le istruzioni nel ramo `else` (senza entrare in nessuno dei due rami `if`). T1 e T2 sono quindi sufficienti sia per coprire tutte le istruzioni che tutte le decisioni.

Per coprire tutti i cammini occorre anche considerare il caso in cui venga eseguito il primo ramo `if` e il secondo ramo `else` (cammino coperto da T3) e il caso in cui non venga eseguito il primo ramo `if` e venga invece eseguito il secondo (cammino coperto da T4).



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da

Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

I Prova di Ingegneria del Software

24 Aprile 2003

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità.
Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 13/30.

Esercizio 1

Le seguenti classi EsprBinPre ed EsprBinInf definiscono rispettivamente delle espressioni binarie prefisse e infisse dell'aritmetica: l'attributo oprtr è una stringa che rappresenta l'operatore (a esempio "*" per la moltiplicazione, "+" per la somma etc.), mentre i due attributi oprnd1 e oprnd2 corrispondono alle due sottoespressioni, che per semplicità si assume possano essere due generici oggetti.

<pre>public class EsprBinPre { private String oprtr; private Object oprnd1; private Object oprnd2; public EsprBinPre(Object o1, String s, Object o2) { oprnd1=o1; oprnd2=o2; oprtr=s } public String toString() { return "(" + oprtr + oprnd1.toString() + oprnd2.toString() + ")"; } }</pre>	<pre>public class EsprBinInf { private String oprtr; private Object oprnd1; private Object oprnd2; public EsprBinInf(Object o1, String s, Object o2) { oprnd1=o1; oprnd2=o2; oprtr=s } public String toString() { return "(" + oprnd1.toString() + oprtr + oprnd2.toString() + ")"; } }</pre>
--	--

Domanda (a): in riferimento al codice

```
EsprBinPre e1 = new EsprBinPre( new Integer(3), "*", new Integer(2));  
EsprBinInf e2 = new EsprBinInf( new Integer(2), "+", new Integer(5));  
EsprBinPre e3 = new EsprBinPre( e2, "/", new Integer(7));  
EsprBinInf e4 = new EsprBinInf( e1, "-", e2 );  
  
System.out.println(e2.toString());  
System.out.println(e3.toString());  
System.out.println(e4.toString());
```

cosa viene stampato con le tre println?

Risp. (2 + 5) (/ (2 + 5) 7) ((* 3 2) – (2 + 5))

Domanda (b):

Si aggiunge alla classe EsprBinPre il metodo stampa definito nell'ipotesi che negli oggetti, ai quali viene inviato il messaggio stampa, i due attributi che rappresentano le sottoespressioni (gli operandi oprnd1 e oprnd2) siano sempre esclusivamente oggetti (non null) di classe EsprBinPre o Integer. Il metodo funziona in modo ricorsivo: per ogni sottoespressione, se questa è un oggetto di classe EsprBinPre effettua una chiamata ricorsiva, se è di classe Integer la stampa usando toString().

```
public void stampa() {  
    System.out.println("(");  
    System.out.print(oprtr);  
    if ( oprnd1 instanceof EsprBinPre)  
        oprnd1.stampa();  
    else System.out.print(oprnd1.toString());  
    if ( oprnd2 instanceof EsprBinPre)  
        oprnd2.stampa();  
    else System.out.print(oprnd2.toString());  
    System.out.print(")");  
}
```

Cosa c'è che non va nel metodo `stampa`, che verrebbe segnalato dal compilatore? correggere il metodo `stampa` in modo che esso venga compilato correttamente.

Risp. La classe Object non possiede alcun metodo stampa: occorre un casting.

```
public void stampa() throws BadExprException {
    System.out.println("(");
    System.out.print(oprtr);
    if ( oprnd1 instanceof EsprBinPre)
        ((EsprBinPre)oprnd1).stampa();
    else
        System.out.print(oprnd1.toString());
    if ( oprnd2 instanceof EsprBinPre)
        ((EsprBinPre)oprnd2).stampa();
    else
        System.out.print(oprnd1.toString());
    System.out.print(")");
}
```

Esercizio 2

Domanda (a): Con riferimento all'esercizio 1, definire un'eccezione `BadEsprException` di tipo *checked*, che possieda solo due costruttori, uno senza argomenti e uno con un argomento di tipo stringa. Modificare il codice del metodo `stampa` in modo che esso lanci un'eccezione `BadEsprException` nel caso in cui una delle due sottoespressioni non sia né di classe `EsprBinPre` né di classe `Integer`.

Risp.

```
class BadExprException extends Exception {
    public BadExprException() { super(); }
    public BadExprException(String s) { super(s); }
}

public void stampa() throws BadExprException {
```

```

System.out.println("");
System.out.print(oprtr);
if ( oprnd1 instanceof EsprBinPre)
    ((EsprBinPre)oprnd1).stampa();
else if ( oprnd1 instanceof Integer )
    System.out.print(oprnd1.toString());
else throw new BadExprException("EsprBinPre: stampa");
if ( oprnd2 instanceof EsprBinPre)
    ((EsprBinPre)oprnd2).stampa();
else if ( oprnd2 instanceof Integer )
    System.out.print(oprnd2.toString());
else throw new BadExprException("EsprBinPre: stampa");
System.out.print(")");
}

```

Domanda (b):

Si consideri il seguente metodo `stampaArrayEsprPre` per stampare un array di oggetti di classe `EsprBinPre`.

```

public static void stampaArrayEsprPre ( EsprBinPre [] a) {
    for (int i = 0; i < a.length; i++)
        a[i].stampa();
}

```

Il metodo `stampaArrayEsprPre` è stato scritto con riferimento alla versione del metodo `stampa` che non sollevava l'eccezione `BadExprException`. Modificarlo in modo che, nel caso in cui una chiamata del metodo `stampa` lanci un'eccezione `BadExprException`, esso la catturi e la gestisca stampando su `System.out` la stringa `a[i].toString()`.

Risp.

```

public static void stampaArrayEsprPre ( EsprBinPre [] a) {
    for (int i = 0; i < a.length; i++)
        try {
            a[i].stampa();
        } catch (BadExprException e) {
            System.out.print(a[i].toString()); }
}

```

Domanda (c)

Se non si volesse modificare il corpo della prima versione del metodo `stampaArrayEsprPre`, quale sarebbe la modifica minimale da apportare alla sua intestazione, in modo che esso possa essere compilato correttamente? Dopo tale modifica quale sarebbe il comportamento del metodo `stampaArrayEsprPre` nel caso venga lanciata da una chiamata di `stampa()` un'eccezione `BadExprException`?

Risp. Occorrerebbe aggiungere la clausola `throws BadExprException` all'intestazione del metodo; in tal caso l'eccezione non verrebbe gestita ma semplicemente propagata.

Domanda (d)

Per ottenere esattamente lo stesso comportamento a seguito del lancio di un'eccezione `BadExprException` senza dover minimamente modificare il codice della prima versione di `stampaArrayEsprPre`, come dovrebbe essere diversamente definita l'eccezione `BadExprException`?

Risp. `BadExprException` dovrebbe essere definita come eccezione unchecked.

```
class BadExprException extends RuntimeException {  
    public BadExprException() { super(); }  
    public BadExprException(String s) { super(s); }  
}
```

Esercizio 3

Si consideri il seguente frammento di specifica della classe Messaggio (da non completare ne' da implementare), che descrive un generico messaggio (SMS) di un telefono cellulare "Java-enabled".

```
public class Messaggio {  
    ...  
    public String toString()//restituisce una rappresentazione testuale del messaggio  
    public String mittente()  
        //restituisce il numero di telefono o, se disponibile, il nome della persona che ha inviato il messaggio  
}
```

a) Si completi negli spazi con i puntini la seguente classe Cursore, di cui e' data la specifica e una implementazione parziale. Si supponga che le classi NotFoundException, OutOfMemoryException, Messaggio e Cursore risiedano nello stesso package. Si noti che non tutte le parti con i puntini sono necessariamente da completare. L'implementazione riportata non è necessariamente la migliore o la più efficiente.

```
public class Cursore {  
    //Overview: un cursore per spostarsi in avanti o indietro sui messaggi contenuti in memoria.  
    //Vi e' un'unica memoria per tutto il telefono, mentre ci possono essere piu' cursori attivi.  
    //Da un punto di vista astratto, un cursore tipico può essere considerato come coincidente con  
    // il testo del Messaggio su cui e' posizionato il cursore.  
    //Tipo mutable.
```

```
public static final DIM_MAX= 128;  
//rep:  
private static Object memoria= new Object[DIM_MAX]; //memoria globale per i messaggi  
private static int primaPosizioneLibera=0; //indice globale alla prima posizione libera in memoria  
private int posizioneCorrente; //la posizione del cursore this
```

//costruttori:

```
public Cursore(Messaggio m) throws OutOfMemoryException, NullPointerException {  
    //EFFECTS: if m e' null then throw NullPointerException  
    // else if non c'e' memoria disponibile then throw OutOfMemoryException  
    //else inserisce m in memoria e costruisce un cursore posizionato sul messaggio;  
    .....if (m==null) throw new NullPointerException();  
    if (primaPosizioneLibera>=DIM_MAX) memoria[primaPosizioneLibera] = m;  
    primaPosizioneLibera++; }
```

```
public Cursore (String mittente) throws NotFoundException {  
    //EFFECTS: if c'e' un messaggio memorizzato il cui mittente e' mittente then costruisce un Cursore posizionato  
    // su quel messaggio  
    //else throw NotFoundException  
    for (int i = 0; i<primaPosizioneLibera;i++ ) {  
        if (.....((Messaggio)memoria[i]).mittente().equals(mittente)) { //cast necessario!  
            ..... posizioneCorrente=i;  
        }  
    }  
    ..... throw new NotFoundException();  
}
```

//Metodi:

```
public void scorriAvanti() throws NotFoundException {  
    //EFFECTS: if ci sono messaggi in posizione successiva, sposta this nella posizione successiva  
    // else throw NotFoundException  
    //MODIFIES: this  
    if (posizioneCorrente + 1 < primaPosizioneLibera) posizioneCorrente++;  
    else throw new NotFoundException();  
}  
public void scorrilIndietro() throws NotFoundException {
```

```

//EFFECTS: se ci sono messaggi in posizione precedente, sposta this nella posizione precedente
// else throw NotFoundException
//MODIFIES: this
//NB: non e' necessario scriverne l'implementazione (del tutto analoga a quella di scorriAvanti).
}

public Messaggio corrente(){
//EFFECTS: restituisce il messaggio nella posizione del cursore
    return (Messaggio) memoria[posizioneCorrente]; //cast necessario!
.....
.....
}

```

b) Descrivere con opportuni commenti RI, e implementare AF.
Si tenga presente che RI deve includere **anche** le eventuali ipotesi sugli attributi **statici** del rep.

```

public String toString() {
    return memoria[posizioneCorrente].toString(); //non serve cast!
}
.....
.....
.....
//RI(c) = .....
.....
.....
.....
memoria è un array di Object di dimensione DIM_MAX &&
0 <primaPosizioneLibera<=DIM_MAX &&
forall i, 0<=i< primaPosizioneLibera ==> memoria[i] è un Messaggio != null &&
0<=posizioneCorrente <primaPosizioneLibera
}

```

c) Discutere brevemente l'operazione di manutenzione: "aggiungere alla classe Cursore un nuovo metodo elimina() che cancella il messaggio nella posizione del cursore this e sposta il cursore nella prima posizione successiva".

Il problema più serio riguarda la specifica del metodo: la cancellazione di un messaggio che sia puntato da più cursori genererebbe dei cursori "dangling".

Una soluzione potrebbe essere ad esempio di modificare la specifica di corrente() in modo che possa lanciare un'eccezione NotFound quando un cursore non corrisponde più a un elemento. Questa soluzione ha anche il pregio di risolvere un ulteriore, meno grave, problema di specifica: che cosa fare quando il cursore è posizionato sull'ultimo messaggio rimasto in memoria (non vi sono altri elementi su cui farlo spostare...)?

Tuttavia, la modifica della specifica di un metodo pubblico è molto pericolosa, perché potrebbe sorprendere gli utilizzatori della classe Cursore.

Anche risolvendo i problemi di specifica, c'è un'ulteriore difficoltà dal punto di vista dell'*implementazione*: poiché un metodo elimina() può introdurre dei "buchi" nell'array memoria, per mantenere valido RI si dovrebbe "ricompattare" la memoria eliminando le parti null. Tuttavia, l'array memoria è statico, e quindi la sua ricompattazione "spiazzerebbe" tutti gli oggetti di tipo Cursore. Occorre quindi riprogettare un'implementazione differente, a parità di specifica della classe (ad esempio, lasciando i "buchi", ma inserendo sempre un messaggio nella prima posizione libera).

d) Dimostrare sinteticamente che RI è verificato, e che scorriAvanti() implementa correttamente la propria specifica.

Rammentiamo RI, numerandone le varie parti:

I) memoria è un array di Object di dimensione DIM_MAX &&

II) $0 \leq \text{primaPosizioneLibera} \leq \text{DIM_MAX}$ &&

III) forall i, $0 \leq i \leq \text{primaPosizioneLibera} \Rightarrow \text{memoria}[i]$ è un Messaggio != null &&

IV) $0 \leq \text{posizioneCorrente} \leq \text{primaPosizioneLibera}$

La (I) vale sempre. Gli attributi static fanno parte di RI sono inizializzati solo dal costruttore
`public Cursore(Messaggio m) throws OutOfMemoryException, NullPointerException`

la *prima volta* che questo viene chiamato. Questo definisce gli attributi static in modo che verifichino RI: `primaPosizioneLibera = 1`; la (II) vale banalmente, la (III) diventa `memoria[0] = m` che è `!= null`; la (IV) vale perché si inizializza correttamente l'unico attributo non static `posizioneCorrente` a `0 < primaPosizioneLibera = 1`.

Si noti che non vi sono altri modi di inizializzare la parte static per la prima volta se non chiamando questo costruttore.

Per le chiamate successive alla prima, occorre ipotizzare che RI sia valido per gli attributi statici al momento della chiamata del costruttore `Cursore(Messaggio m)`. Infatti se ad esempio il costruttore fosse chiamato con un RI falso (es. `memoria[k] = null` per un $k < \text{primaPosizioneLibera}$), si potrebbe inizializzare un oggetto (senza ritornare eccezioni) con un RI che resta falso.

Il costruttore `Cursore(Messaggio m)` si comporta quindi come un modificatore: se RI è verificato quando è chiamato, allora mantiene l'invariante. Infatti: (II): `primaPosizioneLibera` è incrementato di uno; (IV): `posizioneCorrente` diventa `primaPosizioneLibera - 1`; (III): `memoria[posizioneCorrente] =`

memoria[primaPosizioneLibera-1] che diventa m!=null: se (IV) valeva alla chiamata, l'aggiunta del nuovo elemento m mantiene valida la condizione (IV).

Anche per il costruttore public Cursore (String mittente) throws NotFoundException

occorre ipotizzare che RI sia valido per gli attributi statici quando il costruttore e'chiamato (in questo caso *anche* per la prima chiamata).

Il costruttore mantiene la parte di RI (I,II, III) relativa agli attributi statici (poiche' non li modifica) nel caso in cui l'oggetto sia già stato inizializzato. Se esiste un messaggio con quel nome nella porzione 0..primaPosizione-1 allora il nuovo valore di posizioneCorrente verifica la (IV) di RI.

Per il metodo scorriAvanti(), ipotizzando che RI sia valido al momento della chiamata: se posizioneCorrente = primaPosizioneLibera-1, allora si lancia un'eccezione e non si modifica oggetto; altrimenti RI e' mantenuto, perche' (I,II, III) non sono modificate e posizioneCorrente resta nella porzione 0..primaPosizioneLibera-1.

Il metodo corrente() e'un observer e quindi non modifica il rep.

Prova che scorriAvanti() verifica specifica: se RI e'valido e posizione corrente non e' l'ultima, allora posizioneCorrente viene incrementato alla posizone successiva: questa posizione contiene il messaggio successivo, poiche', *in base alla condizione (III) di RI*, non puo contenere null o, per la (I), un oggetto che non sia un messaggio.



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Appello di Ingegneria del Software

8 Settembre 2004

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h15m.
6. Punteggio totale a disposizione: 26/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 1 (punti 10)

La classe `PilaDilInteriFiniti` realizza una pila di capacità limitata i cui elementi sono numeri interi compresi tra 1 e 9 (estremi inclusi). Nell'implementazione della classe si decide di rappresentare la pila come un intero positivo in base 10: ad esempio, il numero 1325 rappresenta la pila in cui sono stati inseriti, nell'ordine, i numeri 5, 2, 3 e 1. Di conseguenza, l'operazione che esegue `top` produrrebbe 1, l'operazione `pop` produrrebbe 325, mentre `push(7)` produrrebbe 71325.

1. Si completi innanzitutto la **specifica** della classe, rispondendo anche alle domande che in essa appaiono. Si deleghi ai clienti del modulo il rispetto del vincolo che i valori da inserire nella pila debbano essere compresi tra 1 e 9. Il tentativo di inserimento in una pila al limite della capacità provoca il lancio dell'eccezione `PilaPienaException`.

```
public class PilaDilInteriFiniti {  
    // OVERVIEW: una PilaDilInteriFiniti tipica e' mutabile o immutabile ? mutable  
    //Una PilaDilInteriFiniti tipica e' <ln ln-1 ...l1> dove ln rappresenta l'intero inserito piu' recentemente;  
    // <> e' pertanto la pila vuota  
    // La pila contiene solo numeri tra 1 e 9  
    public PilaDilInteriFiniti()  
        //EFFECTS: costruisce una pila vuota  
  
    public void push (int n) throws PilaPienaException;  
    //REQUIRES: n >0 && n<=9  
    //EFFECTS: if this è piena throw PilaPienaException,  
    //else, posto this = <n1 n2 ... nk>, k>=0, this_post = <n n1 n2 ... nk>  
    //MODIFIES this  
  
    public int top() throws EmptyException;  
    //EFFECTS: if this è vuota throw EmptyException  
    //else, posto this = <n1 n2 ... nk>, k>=1, n1 è restituito  
  
    public int pop() throws EmptyException;  
    //EFFECTS: if this è vuota throw EmptyException  
    //else, posto this = <n1 n2 ... nk>, k>=1, n1 è restituito e this_post = <n2 ... nk>  
    //MODIFIES this  
  
    public int length()  
        //EFFECTS: restituisce il numero di elementi in this  
}
```

2. Facendo riferimento all'implementazione che abbiamo sopra tratteggiato e alla PilaDlnteriFiniti tipica definita nella specifica, utilizzando le seguenti dichiarazioni in Java che definiscono la rappresentazione (rep) di PilaDlnteriFiniti:

```
private long codifica; // contiene l'intero che rappresenta la pila
```

```
private int numeroCifre; // contiene il numero corrente di cifre della codifica
```

NB: il valore massimo per un valore di tipo long è 9,223,372,036,854,775,807

- a) si definisca la funzione di astrazione.

```
// introduciamo una funzione ausiliaria: cifra(n,i) restituisce l'i-esima cifra decimale  
// del numero n, n>=0, i>=1.  
//AF(c) =if c.numeroCifre >0 return <cifra(c.codifica, c.numeroCifra)  
//cifra(c.codifica, c.numeroCifra -1), ..., cifra(c.codifica,1)> else return <>
```

- b) si definisca e si implementi un Rep Invariant RI per la classe PilaDlnteriFiniti.

```
//RI(c) = c.numeroCifre>=0 && c.codifica>=0 &&  
// if c.numeroCifre>0 then (c.codifica ha esattamente c.numeroCifre cifre decimali  
// && tutte le cifre di c.codifica sono comprese tra 1 e 9) else c.codifica == 0  
// (si noti che abbiamo scelto di rappresentare la pila vuota ponendo sia c.numeroCifre  
// che c.codifica a zero; altre soluzioni sono possibili, ad esempio ponendo solo  
// c.numeroCifre a zero. Ovviamente l'implementazione dei metodi deve essere  
// consistente con la rappresentazione scelta)
```

```
public boolean repOk() {  
    int k, val;  
  
    if (numeroCifre < 0 || codifica < 0) return false;  
    if (numeroCifre == 0) return (codifica == 0);  
    //da adesso in poi controlli per numeroCifre > 0  
    //Verifichiamo che codifica abbia esattamente numeroCifre cifre decimali  
    if (codifica < 10^(numeroCifre - 1) ||  
        codifica > (10^numeroCifre) - 1) return false;  
    //Verifichiamo che ogni cifra decimale sia compresa tra 1 e 9  
    val = codifica;  
    for (k = 1; k <= numeroCifre; k++) {  
        if (val % 10 == 0) return false; //ovviamente occorre solo controllare che sia != 0  
        val = val / 10;  
    }  
    return true;  
}
```

3. Fornire l'implementazione del costruttore e dei metodi push, pop e length e dimostrare che per essi RI è effettivamente invariante.

```
public class PilaDiInteriFiniti {  
    private long codifica; //contiene l'intero che rappresenta la pila  
    private int numeroCifre; //contiene il numero di cifre della codifica  
  
    public PilaDiInteriFiniti () {  
        numeroCifre=0; codifica = 0;  
    }  
  
    public void push (int n) throws PilaPienaException {  
        if (codifica>= 99,999,999,999,999,999) throw new PilaPienaException();  
        codifica = codifica + n* (10^numeroCifre); numeroCifre++;  
    }  
  
    public int pop()throws EmptyException {  
        if (codifica ==0) throw new EmptyException();  
        numeroCifre--;  
        int ris = codifica / (10^numeroCifre)  
        codifica =codifica % (10^numeroCifre);  
        return ris;  
    }  
  
    public int length() {  
        return numeroCifre;  
    }  
}
```

4. Si vuole definire nella classe PilaDiInteriFiniti un metodo appendStack tale per cui date, ad esempio, le due stack s1: <1 7 4> e s2: <2 1 5 6> il risultato è <1 7 4 2 1 5 6>. Spiegare quali sono le differenze fra i due modi seguenti di realizzazione del metodo, precisandone anche la dichiarazione:

a) La funzione viene invocata scrivendo s1.appendStack(s2). Ad esempio:

s1=s1.appendStack(s2);

b) La funzione viene invocata scrivendo appendStack (s1, s2). Ad esempio:

s1=appendStack (s1, s2);

//in class PilaDiInteriFiniti:

//caso a:

```
public PilaDiInteriFiniti appendStack(PilaDiInteriFiniti s) throws PilaPienaException{ ... }
```

//caso b:

```
public static PilaDiInteriFiniti appendStack(PilaDiInteriFiniti s1, PilaDiInteriFiniti s2)  
throws PilaPienaException { ... }
```

Il secondo caso differisce dal primo per il fatto che il metodo viene dichiarato static e che necessita di un parametro in più (il primo metodo opera implicitamente su this). Abbiamo ipotizzato che il metodo possa sollevare una eccezione di tipo PilaPienaException, che viene sollevata nel caso in cui la concatenazione delle due pile dia una pila di dimensioni più grandi rispetto alle dimensioni massime della pila.

5. Si definiscano casi di test per la funzione appendStack (nel caso (b)) utilizzando un metodo black box.

Dal momento che l'esercizio non fornisce una specifica completa per il metodo appendStack, sulla quale basare il nostro test black box, risolviamo l'ambiguità del problema ipotizzando che il metodo abbia REQUIRES: s1 != null and s2!= null, e che la sua specifica sia strutturata come precedentemente detto al punto 4. Un test black box, allora, potrebbe contenere i seguenti casi:

- a) s1 e s2 sono entrambi vuoti;
- b) s1 è vuoto, s2 no;
- c) s2 è vuoto, s1 no;
- d) s1 e s2 entrambi non vuoti;
- e) i casi a) e d), ma con s1 e s2 alias.

La specifica del metodo appendStack dice che il metodo solleva l'eccezione PilaPienaException, ma non è specificato quale sia la dimensione massima della pila, quindi non ci è possibile definire casi di test che coprano il caso di pila risultante troppo grossa.

Esercizio 2 (punti 6)

Si consideri una classe X in cui appare il metodo m avente la seguente interfaccia

int m(float k) throws C;

Una sottoclasse SC ridefinisce il metodo nei modi qui di seguito elencati. Per ciascuno, si dica se il principio di sostituzione è rispettato, giustificando accuratamente la risposta (non basta scrivere SI/NO). Nel caso non lo fosse, mostrare un esempio in cui si dimostra la non sostituibilità, spiegando perché ciò fa sì che la ridefinizione non sia accettabile. Si ipotizzi che la specifica sia la stessa per tutti i metodi laddove questi non lanciano eccezioni.

1. int m(float k);

La ridefinizione rispetta la regola delle segnature, che consente di eliminare una o più eccezioni. Tuttavia, una tale modifica di solito viola la regola dei metodi, modificando la postcondizione del metodo. Il principio di sostituzione potrebbe tanto essere violato quanto non esserlo, a seconda di quali sono le postcondizioni di X.m e di SC.m. Se la specifica di X.m prevedesse, ad esempio, di lanciare un'eccezione di tipo C ogni volta che l'argomento k è 1, allora *nessuna* specifica di SC.m potrebbe in alcun modo verificare il principio di sostituzione. Se, invece, la specifica di X.m è nondeterministica (ad esempio, se affermasse che il metodo m può sollevare l'eccezione oppure no, ma non specifica in quali casi la solleva), allora “eliminando” il sollevamento dell'eccezione nella postcondizione di SC.m viene rafforzata la postcondizione, e il principio di sostituzione non viene violato.

2. int m(float k) throws C, K;

NO, in quanto K non era prevista nel metodo originario: viola la regola delle segnature,

3. int m(float k) throws C1; //C1 sottoclasse di C

SI, in quanto l'eccezione di tipo C1 è anche di tipo C (rispetta la regola delle signature), quindi la m ridefinita lancerà ancora eccezioni di tipo C, sia pure più specifiche (restringendo il numero di eccezioni che posso sollevare, rinforzo la postcondizione di m e quindi rispetto la regola dei metodi).

4. int m(float k) throws C1; //C1 superclasse di C

NO; in quanto il chiamante potrebbe vedersi restituire un'eccezione di tipo diverso da quella prevista.

Esempio: il metodo:

```
statici void prova(X x) {  
    try {
```

```
        x.m(1)
```

```
    catch(C c)
```

se chiamato su un oggetto di tipo SC potrebb, eseguendo x.m(1) ritrovarsi un'eccezione di tipo C1, che non può essere catturata dal catch (C c).

Esercizio 3 (punti 10)

Nella redazione di una testata giornalistica ci sono tre tipi di giornalisti: gli editori, i reporter, ed i fotografi. Ogni dipendente è caratterizzato da un nome e da un salario e ha diritto ad almeno un benefit (cioè un oggetto che viene concesso in uso al dipendente dall'azienda, ma che è di proprietà dell'azienda). Ci possono essere vari tipi di benefit: telefono cellulare, macchina fotografica, computer (che può essere o un portatile, o un palmare). Tra i benefit ci possono anche essere degli apparecchi che hanno funzionalità sia di telefono cellulare che di macchina fotografica.

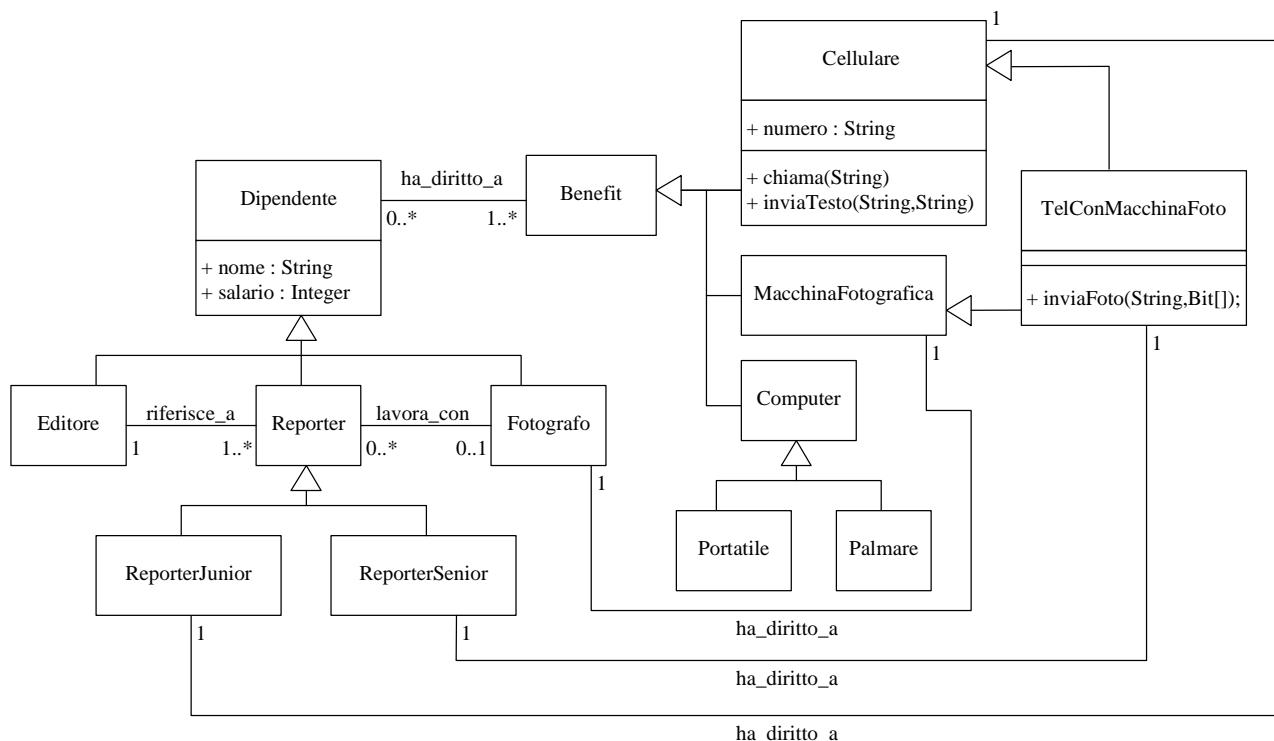
Un telefono cellulare è caratterizzato da un numero di telefono, e offre la funzionalità di chiamata di un altro numero, e di spedizione di un testo ad un altro telefono. Se il telefono ha anche funzionalità di macchina fotografica, permette anche di inviare immagini (che si possono immaginare come sequenze di bit).

I fotografi hanno diritto, come benefit, ad esattamente una macchina fotografica.

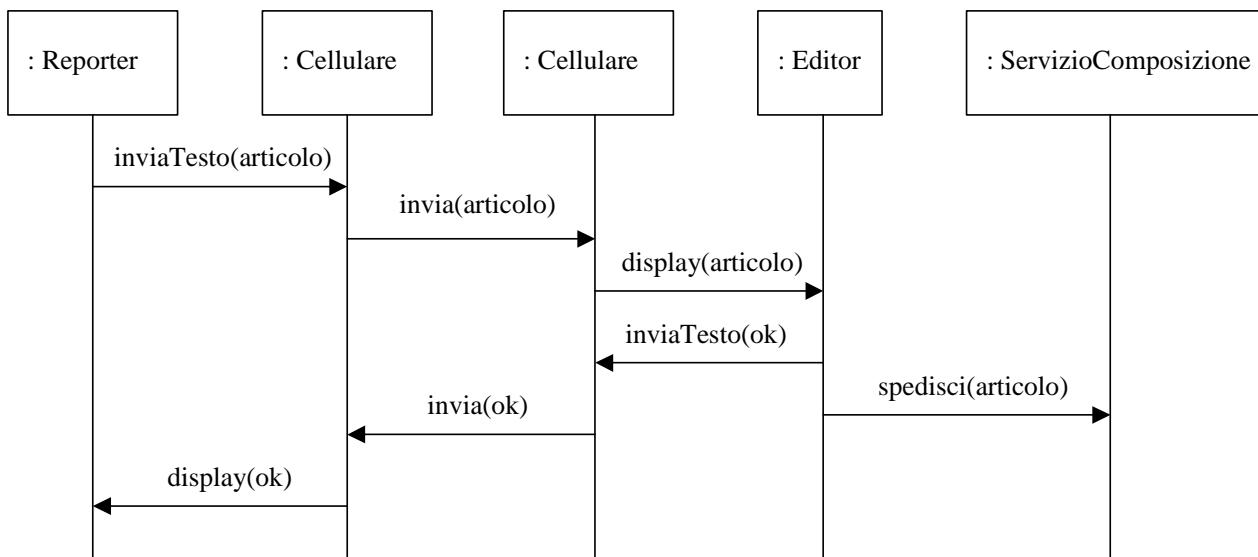
Ci sono 2 tipi di reporter: i reporter junior e quelli senior. I reporter junior hanno diritto ad esattamente un telefono cellulare; i reporter senior hanno invece diritto, come benefit, ad un apparecchio con doppia funzionalità cellulare/macchina fotografica.

Un reporter può lavorare in coppia con un fotografo, e fa riferimento ad un editor.

1. Scrivere un diagramma delle classi UML che rappresenti gli elementi della redazione descritti sopra.



2. Scrivere un sequence diagram UML che descriva il seguente svolgimento di eventi: un reporter spedisce, mediante telefono cellulare, un testo al suo editor, il quale lo controlla e manda al reporter la conferma dell'accettazione dell'articolo. L'editor, dopo aver confermato l'accettazione dell'articolo al reporter, manda l'articolo al servizio di composizione per l'inclusione nel giornale.



3. Si supponga di avere le seguenti interfacce **CellulareI** e **MacchinaFotografical**:

```

interface CellulareI {
    void chiama(String num);
    void inviaTesto(String num, String testo);
}
  
```

```

interface MacchinaFotografical {
    void scatta();
}
  
```

Si completino le seguenti dichiarazioni:

```

interface TelefonoConMacchinaFotografical extends CellulareI, MacchinaFotografical { }

class TelefonoCellulare implements CellulareI {
    public final numero String;
    void chiama(String num) /* codice del metodo non mostrato */;
    void inviaTesto(String num, String testo) /* codice del metodo non mostrato */;
}

class MacchinaFotografica implements MacchinaFotografical {
    void scatta() /* codice del metodo non mostrato */;
}

class TelefonoConMacchinaFotografica extends TelefonoCellulare, implements MacchinaFotografical {
    void scatta() /* codice del metodo non mostrato */;
}
  
```



Politecnico di Milano

Anno accademico 2012-2013

Ingegneria del Software – Appello del 29 Giugno 2012

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione.
È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

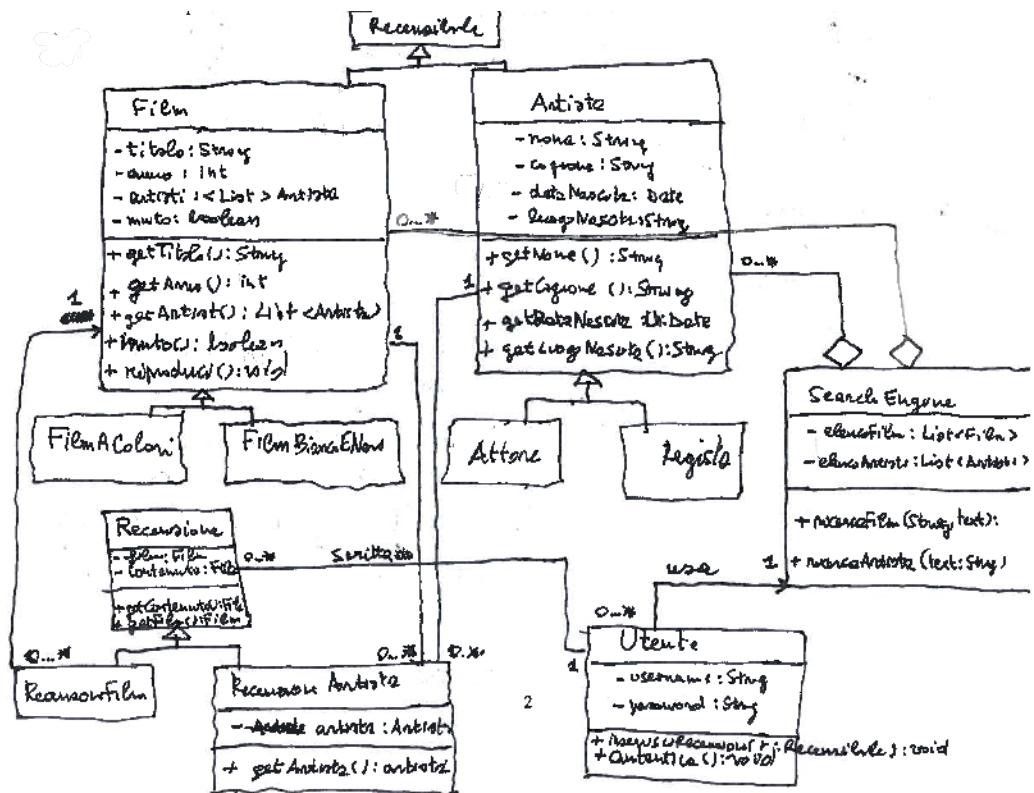
Si consideri un'applicazione che gestisce archivi di film. Un film può essere a colori o in bianco e nero, e può essere muto. In ogni caso, ha un titolo, un anno di pubblicazione, e un elenco di artisti, fra cui il regista e un cast di attori.

Il sistema deve permettere ai soli utenti registrati di inserire proprie recensioni, relative a un film o a un artista per un certo film. L'utente quindi seleziona innanzitutto il film e quindi inserisce direttamente la recensione (se si tratta di recensione del film) oppure cerca anche l'artista e quindi ne inserisce la recensione.

Si chiede di definire:

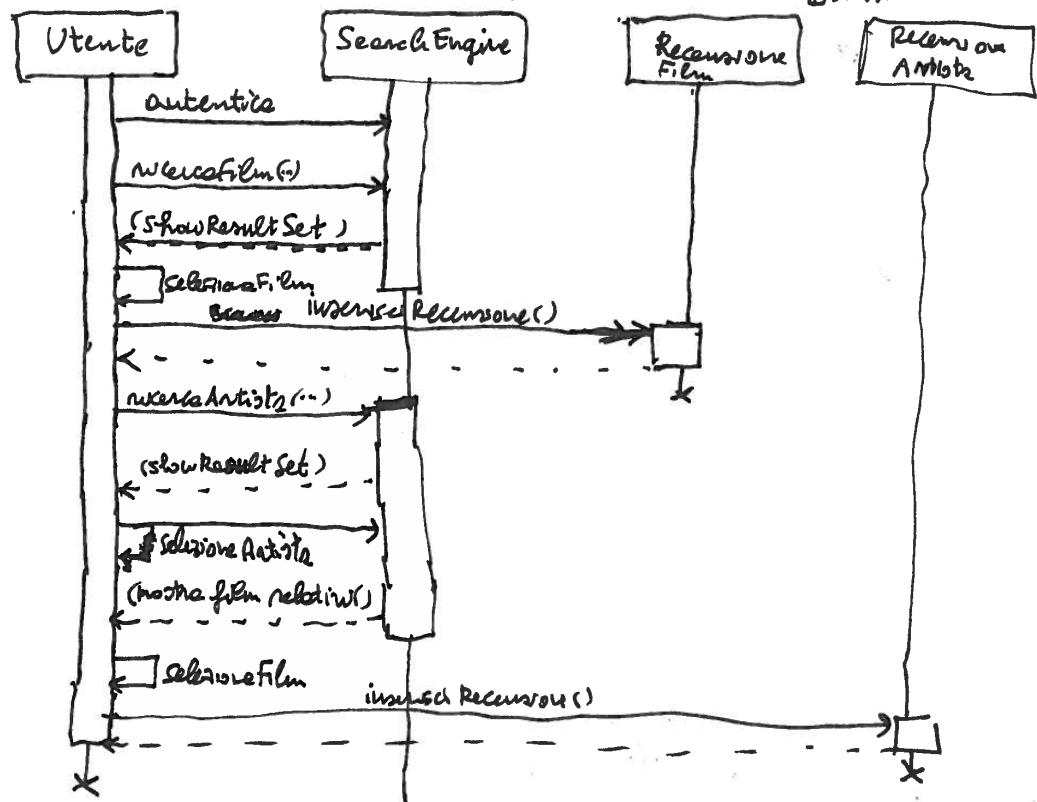
1. il diagramma UML delle classi che rappresenta questa specifica, arricchendo eventualmente la descrizione con dettagli che qui non sono specificati (ma senza esagerare!);
2. la specifica dell'aggiunta di una recensione, con relativa autenticazione, mediante un sequence diagram di UML;
3. la definizione dell'utilizzo di un film mediante un diagramma a stati gerarchico. Per poter guardare il film, occorre innanzitutto caricarlo (operazione LOAD). Il film caricato è predisposto per la visione a partire dall'inizio. Per vedere il film, occorre fornire il comando START. Il comando PAUSE ferma la visione a un certo punto e la ripresa avviene mediante il comando START. Il comando STOP ferma la visione, riposizionando il sistema all'inizio del film. Il comando RESTART consente di riprendere la visione dall'inizio: esso corrisponde dunque all'abbreviazione della sequenza: STOP START (purché ovviamente il film sia stato caricato). Infine, l'operazione END scarica il film dalla memoria. Si ipotizzi anche che un macrostato PLAYABLE indichi che il film è stato caricato in memoria e quindi può essere visionato.

Soluzione

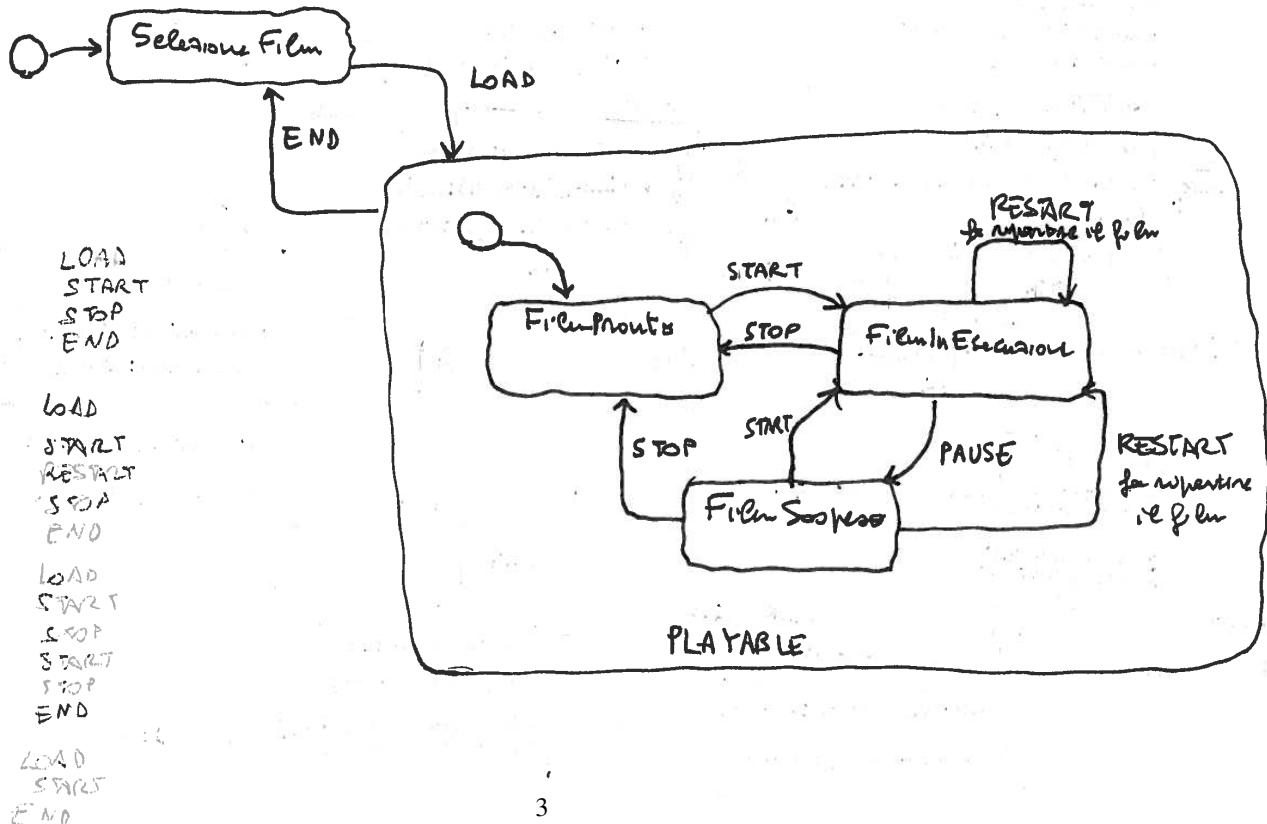


2)

Caso d'utente che si autentica, sceglie un film e poi sceglie gli recensori in artiste



3)



Esercizio 2

Si consideri una variante del problema dei film di cui all'Esercizio 1, in cui esistono solo i film e i film muti. Si considerino in Java le classi **Film** e **FilmMuto** e si discuta se **Film** debba essere sottoclasse di **FilmMuto** o viceversa nei seguenti casi alternativi:

1. **Film** ha due metodi **startPlayingSound()** e **startPlayingVideo()** mentre **FilmMuto** ha solo il metodo **startPlayingVideo()**.
2. entrambe le classi hanno un solo metodo **startPlaying()** e inoltre esistono due metodi osservatori **hasVideo()** e **hasAudio()**. Il metodo **startPlaying()** ha precondizione TRUE per entrambe le classi. Per la classe **Film** ha post-condizione **hasVideo() && hasAudio()**, mentre per la classe **FilmMuto** ha post-condizione **hasVideo()**.
3. come sopra, ma la post-condizione di **startPlaying()** di **FilmMuto**, è **hasVideo() && !hasAudio()**.

Soluzione

1. **Film** può ereditare da **FilmMuto**, mentre il viceversa violerebbe la regola della segnatura.
2. **Film** può ereditare da **FilmMuto**, mentre il viceversa violerebbe la regola dei metodi, in quanto la postcondizione di **Film.startPlaying()** implica la postcondizione di **FilmMuto.startPlaying()** (ossia è più forte), ma non viceversa.
3. Nessuna possibilità di fare ereditare una dall'altra, in quanto la postcondizione di **Film.startPlaying()** e quella di **FilmMuto.startPlaying()** sono in contraddizione: nessuna delle due implica l'altra.

Esercizio 3

Si supponga di voler realizzare un'applicazione per gestire i film di una videoteca personale. Sono date le classi **Film**, e **Artista**, che contengono le ovvie informazioni: titolo, anno, regista, lista di attori per **Film**, nome cognome, etc per **Artista**. La classe **Videoteca** è una collezione di **Film**.

Si definisca e si implementi in Java per **Videoteca** un iteratore **direttoDa(Artista a)** che consente di iterare su tutti i film della collezione diretti da un certo regista.

Il metodo **bestOf(Artista a)** restituisce il film con la migliore recensione di un certo regista passato come parametro. Fornirne la specifica JML. La specifica può basarsi su opportuni metodi osservatori di Videoteca e sulla disponibilità di un metodo osservatore **getValutazione()** per avere la valutazione di un certo film. La valutazione è un intero da 1 (min) a 5 (max), ipotizzando che una mancata recensione di un film di un regista valga 0.

Si considerino i seguenti due casi per gestire la situazione di “regista non presente nella collezione”:

1. lo esclude la precondizione;
2. non lo richiede la precondizione, ma, nel caso, viene generata un'opportuna eccezione.

Soluzione All'interno della classe Videoteca si definiscono il metodo e la classe privata per l'iteratore:

```
public class Videoteca {  
    private ArrayList<Film> film;  
    /*@ensures (*\result e' true se f e' nella collezione *);  
    public /*@ pure @*/ boolean contains (Film f);  
...  
    public direttoDa(Artista a) {  
        return new VideoIter(this, a);  
    }  
    private static class VideoIter implements Iterator<Film> {  
        private Videoteca vid;  
        private Artista reg;  
        private int n;  
        private /*@ helper @*/ searchNext () {  
            for(; n<vid.size() && !vid.film.get(n).regista().equals(reg); n++);  
        }  
        VideoIter(Videoteca v, Artista a) {vid=v; reg=a; n=0; searchNext();}  
        public boolean hasNext{return n<vid.size();}  
        public Film next() throws NoSuchElementException {  
            if (n>=vid.size()) throw new NoSuchElementException();  
            Film f = film.get(n);  
            searchNext();  
            return f;  
        }  
    }  
}
```

Specifica del metodo **bestOf(Artista a)**: Si usa il metodo puro contains definito sopra per Videoteca. Versione con precondizione:

```
/*@ requires (\exists Film f; this.contains(f); f.regista().equals(a));  
/*@ ensures this.contains(\result) && \result.regista().equals(a) &&  
// il film restituito ha valutazione massima:  
/*@ (\forall Film f; contains(f) && f.regista().equals(a);  
/*@     \result.getValut(a)>= f.getValut(a));  
public Film bestOf(Artista a)
```

Versione con eccezione:

```
/*@ ensures this.contains(\result) && \result.regista().equals(a) &&  
/*@ (\forall Film f; contains(f) && f.regista().equals(a);  
/*@     \result.getValut(a)>= f.getValut(a));
```

```
//@ signals(NotFoundException e)
//@      !(\exists Film f; contains(f); f.regista().equals(a));
public Film bestOf(Artista a) throws NotFoundException;
```

Esercizio 4

- Si consideri la specifica fornita nell'Esercizio 1 (parte 3) per la classe **Film**, che descrive come poter vedere un film. In base a questa descrizione, quali sequenze di invocazioni di metodi della classe potrebbero essere definite per testare la classe stessa secondo un metodo black-box? Per ciascuna sequenza di invocazioni, si dica quale è lo scopo atteso del test.
- Si consideri il seguente metodo in Java che effettua la ricerca binaria in un array:

```
static int binarySearch(int[] search, int find) {  
    int start, end, midPt;  
  
    start = 0;  
    end = search.length - 1;  
    while (start <= end) {  
        midPt = (start + end) / 2;  
        if (search[midPt] == find) {  
            return midPt;  
        }  
        else if (search[midPt] < find) {  
            start = midPt + 1;  
        } else {  
            end = midPt - 1;  
        }  
    }  
    return -1;  
}
```

Si fornisca un insieme di casi di test minimo che consenta la copertura di tutte le istruzioni del programma.

Soluzione

- Alcune sequenze significative:
LOAD START STOP END uscita corretta in caso di stop
LOAD START STOP START il film riparte dall'inizio
LOAD START PAUSE START il film riprende dal punto di interruzione (pausa)
LOAD START RESTART il film riparte dall'inizio
LOAD START PAUSE RESTART il film riparte dall'inizio dopo una pausa

- Bastano due casi di test.

- search = [1], find = 0;
- search = [0,1,2,3,4,5,6], find = 1

Il caso (a) copre l'istruzione return -1. Il caso (b) fa percorrere tutte le altre istruzioni.



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da
Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

II Prova di Ingegneria del Software

28 Giugno 2004

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi Morzenti SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h30m.
6. Punteggio totale a disposizione: 13/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 1 (punti 4)

Si consideri la seguente specifica informale sotto riportata che definisce le regole riguardanti il superamento dell'esame di Ingegneria del Software.

“Verranno svolte due prove intermedie. Ogni prova intermedia assegna un massimo di 13 punti ed è considerata valida se lo studente ottiene almeno un punteggio minimo di 6 punti per la I prova e di 6 punti per la II prova; chi ottiene un punteggio inferiore a quello minimo in una prova è obbligato a ripeterla nelle prove di recupero. La ripetizione riguarda comunque entrambe le prove; pertanto il mancato raggiungimento della soglia minima in una delle due prove richiede la partecipazione a una prova di recupero che riguarda l'intero programma del corso.

L'attività svolta in laboratorio permette di ottenere un massimo di 4 punti, ed è considerata sufficiente se lo studente ottiene almeno 2 punti. Non è previsto il recupero del laboratorio. Pertanto in caso di valutazione insufficiente lo studente dovrà ripetere il corso nell'anno accademico successivo: saranno annullati gli eventuali risultati ottenuti durante le prove in itinere e non sarà possibile partecipare agli appelli.

Per superare l'esame è inoltre necessario che la somma dei punteggi delle due prove in itinere sia almeno di 16 punti sui 26 disponibili e che il risultato complessivo (che comprende anche il voto relativo all'attività di laboratorio) sia almeno 18; lo studente che non soddisfa le precedenti condizioni dovrà recuperare entrambe le prove in un appello a propria scelta.”

1. Utilizzando opportune formule che definiscono le pre e post-condizioni, si fornisca la specifica di un'astrazione procedurale che in input riceve tre valori: P1, P2, L, che rappresentano i voti ottenuti nella I e nella II prova intermedia e nel laboratorio, rispettivamente. In output viene fornito un valore intero che rappresenta il voto conseguito (se sufficiente), oppure 0, se l'allievo deve sostenere il recupero, o -1 se deve ripetere;

```
public static int risultato(int P1, P2, L)
//REQUIRES: 0 ≤ P1 ≤ 13 && 0 ≤ P2 ≤ 13 && 0 ≤ L ≤ 4
//EFFECTS: if (L < 2) return -1
//else if (P1 < 6 || P2 < 6 || P1 + P2 < 16) -> return 0
//else return P1+P2+L
```

2. In base alla specifica fornita, si definiscano dati a supporto del test funzionale (black-box testing) in base al criterio di copertura delle combinazioni proposizionali e i risultati attesi per ciascun dato.

```
< P1, P2, L >
< 7, 7, 0 > (per L<2), valore atteso: -1
< 3, 4, 3 > (sia per P1<6 che per P2<6), valore atteso: 0
< 3, 12, 3 > (per P1<6 ma P2>=6, P1+P2>=16), valore atteso: 0
< 7, 7, 3 > (per P1 + P2 < 16) valore atteso: 0
< 10, 9, 3 > valore atteso: 22
```

3. Si identifichino almeno 5 dati di test corrispondenti a *valori limite* dei parametri dell'astrazione.

```
< P1, P2, L >
< 7, 7, 1 > (valore limite di L<2), valore atteso: -1
< 5, 5, 2 > valore atteso: 0
< 6, 6, 2 > valore atteso: 0
< 7, 8, 2 > valore atteso: 0
< 8, 8, 2 > valore atteso: 18
< 13, 13, 4 > valore atteso: 30
```

Esercizio 2 (punti 6)

La classe astratta Coda specifica le operazioni per la manipolazioni di una coda FIFO di Object: insert (inserisce in coda un Object), remove (estrae un Object dalla coda, eliminandolo, e lo restituisce come risultato), length (lunghezza della coda). L'iteratore Elements restituisce un generatore che consente di iterare su tutti gli oggetti in coda. Le operazioni ritornano un'eccezione (NullPointerException) nel caso in cui l'oggetto a cui si applicano sia null. L'operazione di inserimento può, in alcuni casi che non vengono dettagliati, sollevare un'eccezione di tipo OggettoNonValido anziché inserire l'oggetto. Non vengono generate altre eccezioni dalle operazioni (tranne l'eccezione CodaVuota nel caso si cerchi di estrarre un oggetto da una coda vuota); i metodi che implementano le operazioni definiscono inoltre funzioni totali.

1. Considerando la seguente implementazione parziale di Coda, si completino le operazioni con la loro specifica (OVERVIEW, REQUIRES, EFFECTS, MODIFIES)

```
public abstract class Coda {  
    .....  
    protected int tot;  
    public Coda() {tot = 0};  
    .....  
    public abstract void insert (Object x) throws OggettoNonValido;  
    .....  
    public abstract Object remove() throws CodaVuota;  
    .....  
    public abstract Iterator Elements();  
    .....  
    public int length() {return tot;}  
    .....  
}  
  
public abstract class Coda {  
    //OVERVIEW: Coda è un contenitore di Object. Tipo Mutable. L'inserimento di un oggetto nel  
    //contenitore potrebbe non essere sempre possibile. L'estrazione rimuove gli oggetti nello stesso  
    //ordine del loro avvenuto inserimento (seguendo cioe' una politica FIFO).  
    //Un oggetto tipico è [o1, o2, ..., on], dove o1 e' l'elemento inserito da piu' tempo.  
    protected int tot;  
    public Coda() {tot = 0};  
    //REQUIRES: true  
    //EFFECTS: inizializza this alla coda vuota  
  
    public abstract void insert (Object x) throws OggettoNonValido;  
    //REQUIRES: true  
    //EFFECTS: // posto this = [o1, o2, ..., on],  
    //if x != null /then (this_post = [o1, o2, ..., on, x] || solleva OggettoNonValido),  
    //else solleva NullPointerException  
  
    public abstract Object remove () throws CodaVuota;  
    //REQUIRES: true  
    //EFFECTS: // posto this = [o1, o2, ..., on],  
    //se n>0, this_post = [o2, ..., on] e ret_val = o1, altrimenti solleva CodaVuota  
  
    public abstract Iterator Elements();  
    //REQUIRES: true  
    //EFFECTS: restituisce un generatore che restituisce gli elementi di this nell'ordine o1, .., on.  
  
    public int length() {return tot;}  
    //REQUIRES: true  
    //EFFECTS: restituisce la lunghezza della coda. Formalmente, se this = [o1, o2, ..., on], allora  
    //ret_val = n, e this_post = this_pre
```

}

(continua esercizio 2)

2. Si vogliono definire due sottoclassi concrete di Coda, chiamate rispettivamente Codallimitata e CodaFinita, che rappresentano, rispettivamente, le code che possono crescere arbitrariamente di lunghezza e le code di lunghezza finita e predeterminata. Nel secondo caso, si assume che la lunghezza massima delle code sia definita da una costante statica privata e che il tentativo di inserimento di un Object in una coda piena generi un'eccezione checked CodaPiena. Codallimitata e CodaFinita rispettano il principio di sostituibilità? Motivare la risposta facendo riferimento a una precisa descrizione della specifica di CodaFinita e di Codallimitata.

```
public class CodaFinita extends Coda {  
    //OVERVIEW: CodaFinita è una Coda di dimensione massima MAX uguale per tutte le code  
    //Al raggiungimento del numero massimo di elementi, l'inserimento restituisce eccez. CodaPiena  
    public abstract void insert (Object x) throws OggettoNonValido, CodaPiena  
    //EFFECTS: if la dimensione massima di this è maggiore di MAX, throw CodaPiena else....  
}  
public class Codallimitata extends Coda {  
    //OVERVIEW: Codallimitata è una Coda la cui dimensione può essere qualunque  
}
```

La classe CodaIllimitata rispetta il principio di sostituzione, la classe CodaFinita no. CodaIllimitata non modifica la specifica dei metodi classe Coda, quindi le regole delle segnature e dei metodi sono banalmente verificate. L'overview di CodaIllimitata impone solo la proprietà aggiuntiva che le dimensioni della coda siano qualunque, ma non nega nessuna delle proprietà specificate nell'Overview della sopraclass (come la politica FIFO, la mutabilità, ecc.). La classe CodaFinita invece aggiunge un'eccezione (checked) alla signature del metodo insert, in violazione della regola delle signature.

3. Si consideri la classe AltraCodaFinita, che ha la stessa definizione della classe CodaFinita ma con la seguente modifica: quando la coda è piena, il metodo insert genera un'eccezione OggettoNonValido (anziché l'eccezione CodaPiena). La classe AltraCodaFinita soddisfa il principio di sostituibilità? Anche in questo caso giustificare la risposta, aiutandosi con una precisa descrizione della specifica di CodaFinita.

Se il metodo solleva un’eccezione di tipo OggettoNonValido, allora la regola delle signature viene rispettata. Anche la regola dei metodi viene rispettata, dal momento che il metodo resta totale (precondizioni invariate) e la postcondizione di Coda.insert consente di sollevare ad arbitrio l’eccezione: AltraCodaFinita “raffina” questa postcondizione specificando più in dettaglio un caso in cui viene sollevata l’eccezione, OggettoNonValido ed è quindi più forte (ossia, implica) la postcondizione di Coda.insert. Per quanto riguarda, infine, la regola delle proprietà, la classe AltraCodaFinita ha più proprietà (la limitatezza della dimensione della coda) della classe Coda, ma rispetta ancora le proprietà dell’overview di Coda (politica FIFO e mutabilità). Pertanto, anche la regola delle proprietà è soddisfatta, e la classe AltraCodaFinita rispetta il principio di sostituibilità.

4. Si consideri la classe TerzaCodaFinita, la cui specifica è identica a quella di CodaFinita, con la seguente modifica: quando la coda è piena, il metodo insert non fa nulla. Dire, anche in questo caso, se TerzaCodaFinita rispetta o non rispetta il principio di sostituibilità, giustificando come al solito la risposta.

```
public class TerzaCodaFinita extends Coda {  
    .....  
    .....  
    .....  
    .....  
    .....  
    .....  
}
```

Stavolta il metodo TerzaCodaFinita.insert ha una postcondizione che non rispetta quella specificata nella classe Coda: quest’ultima dice che insert può solo inserire un oggetto nella coda oppure non inserirlo e sollevare un’eccezione. Dal momento che TerzaCodaFinita.insert in certi casi non inserisce un oggetto in coda, ma in tali casi non solleva l’eccezione corrispondente, la condizione ensure no less sulle postcondizioni del metodo insert viene violata.

(continua esercizio 2)

5. Si tratteggi l'implementazione della classe Codalllimitata volendo che essa sia utilizzabile per definire un'astrazione polimorfa, usabile per definire code non limitate in dimensioni di elementi di un qualunque tipo. Si vuole verificare che gli elementi inseriti in una coda illimitata siano tutti dello stesso tipo. A tale scopo si adotti l'approccio *element subtype*, definendo un'interfaccia chiamata Accodabile con un metodo confrontaClasse; il metodo insert, se la coda in cui avviene l'inserzione non è vuota, invoca il metodo confrontaClasse su uno degli elementi già presenti in coda, passando come parametro l'oggetto di tipo Accodabile che si sta tentando di inserire. Nel caso che l'oggetto abbia tipo scorretto, questo non viene inserito nella coda, e viene sollevata un'eccezione di tipo OggettoNonValido. Definire una classe StringaAccodabile, che contiene un dato di tipo String e implementa Accodabile, codificando in dettaglio il suo metodo confrontaClasse.

```

public class Codalllimitata extends Coda {
    .....
    public void insert .....
    .....
    .....
    .....
}

public interface Accodabile {
    public boolean confrontaClasse(Object x);
}

public class StringaAccodabile implements Accodabile {
    private String s;
    .....
    .....
    .....
    .....
    .....
}

public class Codalllimitata extends Coda {
    //OVERVIEW: Codalllimitata di elementi omogenei
    //Rep:
    private Vector coda;
    //AF: AF(c) = [c.coda.get(0) ... c.coda.get(coda.size()-1)]
    //RI(c) = c.coda != null
    public void insert(Object x) throws OggettoNonValido {
        if ( !(x instanceof Accodabile) ||
            (!c.isEmpty() && !((Accodabile) x).confrontaClasse(coda.firstElement())))
            throw new OggettoNonValido();

        coda.add(x); //...a questo punto può inserire x nel vettore coda
    }
}

public class StringaAccodabile implements Accodabile {
    private String s;
    public StringAccodabile(String s) {this.s = s;}
    public String contenuto() {return s;}
    public boolean confrontaClasse (Object x) {
        if !(x instanceof StringAccodabile) return false;
        return true;
    }
}

NB: L'implementazione di Codalllimitata qui riportata è più completa di quanto strettamente richiesto dall'esercizio.

```

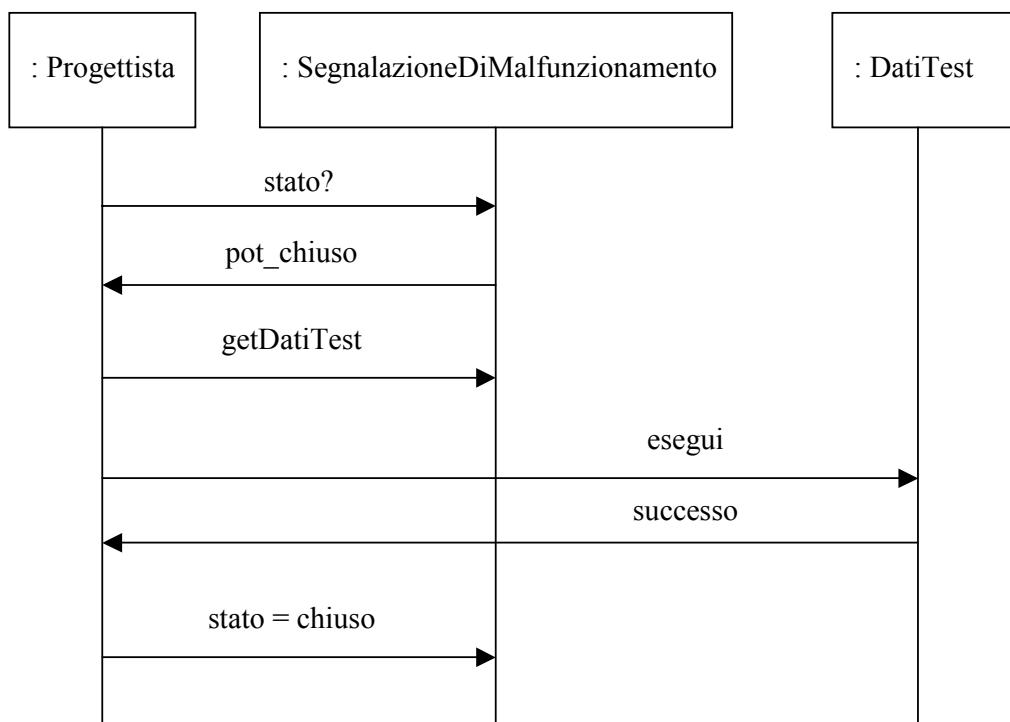
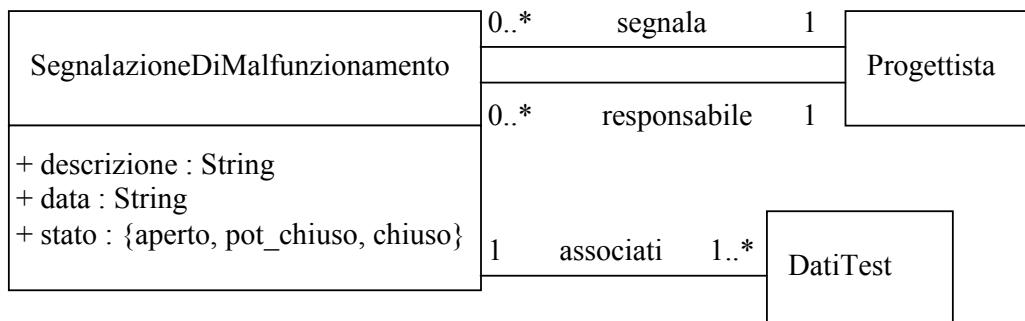
}

Esercizio 3 (punti 3)

Si descriva un diagramma delle classi UML per la seguente situazione. In una società che sviluppa software, quando si scopre un errore in un modulo software, viene generata una SegnalazioneDiMalfuncionamento, che contiene la descrizione del malfuncionamento e la data in cui esso si è manifestato. Ogni progettista può segnalare malfuncionamenti e ogni malfuncionamento ha associato il progettista che l'ha segnalato. Un malfuncionamento ha un attributo che ne indica lo stato. Quando viene segnalato, il suo stato viene considerato "aperto". Per eliminare un malfuncionamento, gli viene assegnato un progettista responsabile della sua correzione. Una volta corretto, lo stato del malfuncionamento diventa "potenzialmente chiuso". Al malfuncionamento sono associati uno o più dati di test. Se questi vengono eseguiti con successo, lo stato del malfuncionamento diventa "chiuso".

1. Si descriva un Class Diagram che illustra le entità in gioco e le relative associazioni.
2. Si descriva mediante un Sequence Diagram uno scenario in cui un particolare progettista esamina un malfuncionamento che si trova nello stato "potenzialmente chiuso", esegue i test associati al malfuncionamento e, nel caso questi vengano eseguiti con successo, dichiari il malfuncionamento "chiuso".

Soluzione:



Appello del 16 Gennaio 2018



Politecnico di Milano
Anno accademico 2017-2018

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Cugola <input type="checkbox"/> Mottola <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica della classe Bunch

```
public class Bunch<T> {
    // OVERVIEW: Una collezione di oggetti di tipo T.
    // Gli elementi sono disposti in ordine qualunque e sono di molteplicità qualunque.
    // Il valore null non fa parte della collezione
    // Usa equals per confrontare gli oggetti

    // @ensures (*costruisce il bunch vuoto*)
    public Bunch();

    // Aggiunge x a this
    // Lancia eccezione se x è null
    public void ins(T x) throws NullPointerException;

    // Elimina una comparsa di x
    // Restituisce il numero di elementi rimanenti uguali a x.
    // Lancia eccezione se x è null
    public int del(T x) throws NullPointerException;

    // Restituisce il numero di elementi uguali a x
    public /*@ pure */ int num(T x)
}
```

Domanda a

Scrivere la specifica JML del metodo `del(T x)` e un invarianto pubblico per la classe.

Soluzione

```
public invariant (\forall T t;; num(t) \geq 0);

// @ensures x!=null &&
// @ (\forall T t;!t.equals(x); num(t) == \old(num(t)) &&
// @ \old(num(x)==0) ? num(x) ==0 && \result==0
// @ : num(x) ==\old(num(x)-1) && \result==num(x);
// @signals(NullPointerException e) x == null;
public int del(T x) throws NullPointerException
```

Domanda b

Complefare, con AF e RI (esprimibili con opportuni inviarianti) e con il codice del metodo `del(T x)`, la seguente implementazione della classe Bunch. Il rep è costituito da una lista di coppie, ciascuna contenente un elemento, se presente, e la sua molteplicità. I dettagli possono essere ricavati dall'esame del codice.

```
public class Bunch<T> {
    //rep:
    private List<Pair> elem;
    private static class Pair {
        // OVERVIEW: Tipo immutabile rappresentante una coppia (T,int)
        T t;
        int i;
        Pair(T y, int j) {
            t = y;
            i = j;
```

```

        }
        boolean contains(T x) {
            return (x.equals(t));
        }
    }
    public Bunch() {
        elem = new ArrayList<T>;
    }
    private int find(T x) {
        for (int j=0; j<elem.size(); j++) {
            Pair p = elem.get(j);
            if (p.contains(x)) return p.i;
        }
        return -1;
    }
    public /*@ pure @*/ int num(T x) {
        int j = find(x);
        if (j===-1) return 0;
        return elem.get(j).i;
    }
    public void ins(T x) throws NullPointerException {
        if (x== null) throw new NullPointerException();
        i = find(x);
        if (i>-1) elem.set(i,new Pair(x,1+p.i));
        else elem.add(new Pair(x,1));
        return this;
    }
    public int del(T x) throws NullPointerException {
        if (x== null) throw new NullPointerException();
        int j = find(x);
        if (j===-1) return 0;
        .....
        .....
        .....
        .....
        .....
        .....
        .....
        .....
    }
}

```

Soluzione

```

public int del(T x) throws NullPointerException {
    if (x== null) throw new NullPointerException();
    int j =find(x);
    if (j===-1) return 0;
    int temp=elem.get(j).i-1;
    if (temp==0) {
        elem.remove(j);
        return 0;
    }
    elem.get(j).i=temp;
    return temp;
}

```

RI:

Soluzione

```
private invariant elem != null &&
(\forall int j; 0<=j && j<elem.size();
 elem.get(j).i>0);
```

AF:

Soluzione

```
private invariant
(\forall int j; 0<=j && j<elem.size();
 num(elem.get(j).t)== elem.get(j).i) &&
(\forall T x;;
 ! (\exists int j; 0<=j && j<elem.size(); x.equals(elem.get(j).t)
 ==> num(x)==0;
```

Esercizio 2

Si consideri il seguente scheletro di classe Java:

```
public class DoubleBuffer {
    private String val1, val2;

    public DoubleBuffer() { val1 = val2 = null; }

    // Sospende il chiamante se val1==null e rimuove il valore prima di ritornarlo
    public String get1() { ... }

    // Sospende il chiamante se val1!=null
    public void put1(String v) { ... }

    // Sospende il chiamante se val2==null e rimuove il valore prima di ritornarlo
    public String get2() { ... }

    // Sospende il chiamante se val1!=null
    public void put2(String v) { ... }
}
```

Domanda a

Si implementino i metodi get1, put1, get2 e put2. Si massimizzi il parallelismo tra i thread chiamanti.

Soluzione

```
public class DoubleBuffer {
    private String val1, val2;
    private Object lock1, lock2;
    public DoubleBuffer() {
        val1 = val2 = null;
        lock1 = new Object(); lock2 = new Object();
    }
    public String get1() throws InterruptedException {
        String v=null;
        synchronized(lock1) {
            while(val1==null) lock1.wait();
            v = val1;
            val1=null;
            lock1.notifyAll();
        }
        return v;
    }
    public void put1(String v) throws InterruptedException {
        synchronized(lock1) {
            while(val1!=null) lock1.wait();
            val1 = v;
            lock1.notifyAll();
        }
    }
    public String get2() throws InterruptedException {
        String v=null;
        synchronized(lock2) {
            while(val2==null) lock2.wait();
            v = val2;
            val2=null;
            lock2.notifyAll();
        }
    }
}
```

```

        return v;
    }
    public void put2(String v) throws InterruptedException {
        synchronized(lock2) {
            while(val2!=null) lock2.wait();
            val2 = v;
            lock2.notifyAll();
        }
    }
}

```

Domanda b

Si aggiunga alla classe precedente il metodo `String readLongest()` che sospende il chiamante se `val1` oppure `val2` sono nulli per poi ritornare il più lungo dei due valori (si usi `String.length()`).

Soluzione

```

public String readLongest() throws InterruptedException {
    synchronized(lock1) {
        while(val1!=null) lock1.wait();
        synchronized(lock2) {
            while(val2!=null) lock2.wait();
            if(val1.length()>val2.length()) return val1;
            else return val2;
        }
    }
}

```

Esercizio 3

Si consideri la classe `Couple` così definita:

```
public class Couple {  
    public int val1;  
    public int val2;  
}
```

e una lista data di istanze di tale classe, opportunamente inizializzata:

```
List<Couple> data = new ArrayList<Couple>();  
data.add(...); ...; data.add(...);
```

Si scriva un frammento di programma Java 8 che usa lo stile funzionale per restituire la somma di tutti i secondi elementi (`val2`) delle coppie nella lista `data` il cui primo elemento sia maggiore o uguale a zero.

Soluzione

```
int result = data.stream().filter(v -> v.val1>=0)  
    .map(v -> v.val2)  
    .reduce(0, (tot, val) -> tot+val);
```

Esercizio 4

Si consideri il seguente codice. Operando su un array di esattamente tre elementi, il codice deve riorganizzarli in ordine decrescente.

```
public int[] order(int v[]) {  
    int tmp;  
    if (v[0]<v[1]) {  
        tmp = v[0];  
        v[1] = v[1];  
        v[1] = tmp;  
    }  
    if (v[1]<v[2]) {  
        tmp = v[1];  
        v[1] = v[2];  
        v[2] = tmp;  
    }  
    return v;  
}
```

Si identifichi il numero minimo di casi di test in maniera da soddisfare (separatamente) ciascuno dei seguenti criteri e insieme trovare tutti gli eventuali errori presenti nel codice.

1. copertura di tutte le istruzioni;
2. copertura di tutte le diramazioni;
3. copertura di tutte le condizioni.

Giustificate le vostre risposte.

Soluzione

Ci sono evidentemente due errori nel codice. Nel ramo `true` del primo `if`, i due valori non vengono scambiati correttamente. Invece, `v[0]` viene duplicato in `v[1]` che scompare. Inoltre, il codice non copre ad esempio il caso in cui l'array in ingresso sia completamente ordinato in maniera crescente. Il terzo elemento non ha maniera di diventare il primo.

Per coprire tutte le istruzioni (domanda 1) e insieme trovare entrambi gli errori, è sufficiente un singolo caso di test del tipo `[2, 3, 4]`. Questo produce in output `[2, 4, 2]`. Il numero 3 è scomparso, l'array non è ordinato in maniera decrescente, ma tutte le istruzioni sono coperte.

Per coprire tutte le diramazioni (domanda 2) e insieme trovare entrambi gli errori, partiamo ancora da `[2, 3, 4]` che copre le diramazioni `true` dei due `if` e trova tutti gli errori. Manca un caso di test che copra le diramazioni `false` dei due `if`. Per questo basta aggiungere un caso di test con un array già ordinato in maniera decrescente, ad esempio `[2, 3, 4]`.

Lo stesso insieme di casi di test è ovviamente sufficiente a coprire tutte le condizioni (domanda 3).

Ingegneria del Software – a.a. 2004/05

Appello del 15 luglio 2005

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi ?, Ghezzi ?, Morzenti ?, SanPietro ?

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 28/30.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 10)

Il metodo statico `highLowNum` riceve in ingresso due array `nums` e `highs` e un intero `n` positivo. L'array `nums`, di lunghezza non precisata, contiene dei numeri interi, per ipotesi tutti diversi tra di loro; l'array `highs` è lungo esattamente `n`. Il metodo trova gli `n` numeri interi più grandi di `nums` e li inserisce in ordine decrescente nell'array `highs`; l'array `nums` non viene modificato.

- a) Si scriva la specifica formale in JML del metodo.

- b) Si trasformi la specifica JML del punto precedente sostituendo una a scelta delle precondizioni nel lancio di opportune eccezioni.

Esercizio 2 (punti 10)

Si consideri la classe **Numerobinario** che rappresenta numeri interi, positivi e senza segno in binario (ad esempio, 1110_2 corrisponde al numero 14_{10}). Si supponga che la classe fornisca:

- ? il metodo **int dimensione()**, per conoscere il numero di cifre del **Numerobinario** considerato;
- ? il costruttore **Numerobinario(int n)**, per creare un **Numerobinario** di n bit inizializzati a 0;
- ? il metodo **void cambiaBit(int pos, int val)**, per assegnare al bit di posizione pos il valore val;
- ? due iteratori per scorrere le cifre binarie dalla meno significativa alla più significativa (**Iterator destraSinistra()**) e dalla più significativa alla meno significativa (**Iterator sinistraDestra()**).

Utilizzando solamente i metodi elencati sopra, si realizzino i seguenti metodi:

2 – **simmetrico** per sapere se la sequenza di bit è simmetrica (palindroma):

```
public boolean simmetrico(){
```

```
}
```

3- **specchio** per invertire l'ordine di memorizzazione della sequenza di bit:

```
public Numerobinario specchio(){
```

```
}
```

4 - Supponendo che la classe usi un Vector per memorizzare la sequenza di cifre binarie e un int per memorizzare il numero (per esempio se il Vector è 1110 il valore è 14), dichiarate come segue,

```
Vector bits;  
int val;
```

si implementi il costruttore che prende in ingresso un numero intero e memorizza le cifre binarie nel Vector a partire da quella meno significativa:

```
public NumeroBinario(Integer n){  
}  
}
```

5 - Si scriva l'invariante di rappresentazione come private invariant in JML:

6 - Si realizzi anche l'iteratore per poter implementare il metodo ***Iterator destraSinistra()***

Esercizio 3 (punti 8)

Disegnare un diagramma delle classi UML che rappresenti una rete di computer. Questa si compone di nodi, i quali possono essere di 2 tipi: host e router. Gli host sono connessi ad esattamente un router, mentre i router possono essere connessi ad un numero qualunque di host e ad almeno un router. I nodi di una rete sono connessi tra loro mediante link fisici. Un link fisico può collegare più host e più router tra loro. Ogni connessione tra nodi della rete è link fisici è caratterizzata da un indirizzo di IP. Un host nella rete può offrire dei servizi. Ogni servizio, su un certo host, è caratterizzato da una porta. Inoltre ogni servizio si caratterizza per il tipo di protocollo su cui è trasportato, che può essere o TCP, o UDP.

Soluzioni

Esercizio 1

Parte A

```
//@ requires n>0 && nums != null && highs != null && highs.length==n && nums.length>=n &&
//@ (\forall int i; 0<=i && i<nums.length-1;
//@   (\forall int j; i<j && j<nums.length; nums[i] != nums[j]));
//@ ensures (\forall int i; 0<=i && i<nums.length; nums[i]==\old(nums[i])) &&
//@ (\forall int i; 0<=i && i<highs.length;
//@   (\exists int j; 0<j && j<nums.length; highs[i]==nums[j] )) &&
//@ (\forall int i; 0<=i && i<highs.length-1; highs[i]>highs[i+1]) &&
//@ (\forall int i; 0<=i && i<highs.length;
//@   (\numof int j; 0<=j && j<nums.length; highs[i]<=nums[j]) <=n);
```

Parte B

```
//@ requires n>0 && highs.length==n && nums.length>=n &&
//@ (\forall int i; 0<=i && i<nums.length-1;
//@   (\forall int j; i<j && j<nums.length; nums[i] != nums[j]));
//@ ensures nums!=null && highs!=null && ...poi come sopra
//@ signals (NullPointerException e) (nums == null || highs == null);
```

```
public static void highLowNums (int [] nums, int [] highs, int n) throws NullPointerException
```

Esercizio 2

Parte 1

```
public boolean simmetrico(){
    Iterator iSD = this.sinistraDestra();
    Iterator iDS = this.destraSinistra();
    while (iSD.hasNext() && iDS.hasNext())
        if (((Integer) iSD.next()).intValue()
            != ((Integer) iDS.next()).intValue()) return false;
    return true;
}
```

NB: in questo modo la verifica della simmetria è effettuata due volte per ciascuna coppia di bit.

Parte 2

```
public NumeroBinario specchio(){
    NumeroBinario tmp = new NumeroBinario(this.dimensione());
    int n = 0;

    for (Iterator itr = this.destraSinistra(); itr.hasNext();
         tmp.cambiaBit(n++, ((Integer) itr.next()).intValue()));

    return tmp;
}
```

Parte 3

```
public NumeroBinario(Integer n){
    int tmp;

    val = n.intValue();
    bits = new Vector();

    tmp = val;
    while(tmp/2 != 0){
        bits.add(new Integer(tmp%2));
        tmp=tmp/2;
```

```

        }
        bits.add(new Integer(tmp%2));
    }
}

```

NB: si intende qui che `bits` contenga nella prima posizione la cifra piu' significativa (cosi' come specificato anche dall'invariante di rappresentazione). Per partire invece dalla cifra meno significativa si puo' chiamare opportunamente il metodo `specchio()` (modificando anche l'invariante).

Parte 4

```

//@ private invariant bits != null && bits.size()>0 &&
//@ (\forall int i; i>=0 && i<bits.size(); ((Integer) bits.get(i)).intValue() == 0
//@ || ((Integer) bits.get(i)).intValue() == 1) &&
//@ val == (\sum int i; i>=0 && i<bits.size();
//@           Math.pow(2, i)*((Integer) bits.get(i)).intValue());

```

Parte 5

```

private static class ItrDS implements Iterator {
    private NumeroBinario num;
    private int cp;

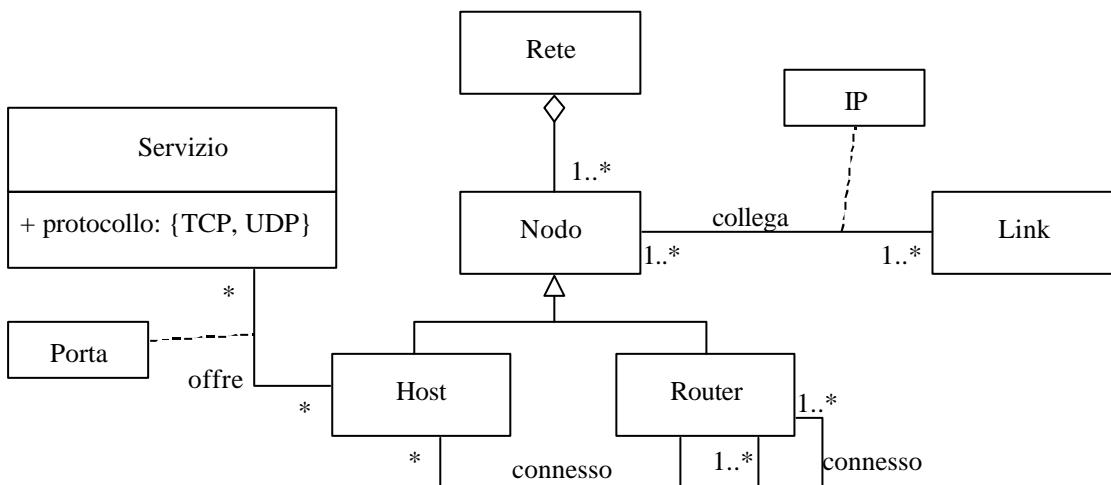
    ItrDS(Numerobinario n){
        num = n;
        cp = n.dimensione();
    }

    public boolean hasNext(){
        return cp>0;
    }

    public Object next() throw NoSuchElementException {
        if (cp<=0) throw new NoSuchElementException();
        cp--;
        return num.bits.get(cp);
    }
}

```

Esercizio 3



Ingegneria del Software — Soluzione del Tema 18/06/2021

Esercizio 1

Si consideri la seguente classe Java SensorsData per salvare e analizzare valori di temperatura provenienti da sensori ambientali. Ogni sensore è rappresentato da un identificativo univoco di tipo `String` e ogni valore di temperatura è rappresentata da un `float`. Per ogni sensore sono salvati solo i 100 valori di temperatura più recenti.

```
public class SensorsData {
    // Ritorna gli identificativi di tutti i sensori per cui e' salvato
    // almeno un valore di temperatura.
    public /*@ pure */ Set<String> sensors();

    // Ritorna il numero di valori salvati per il sensore con identificativo id.
    public /*@ pure */ int numValuesFor(String id);

    // Ritorna l'i-esimo valore di temperatura salvato per il sensore con identificativo id.
    // I valori sono in ordine di inserimento, dal meno recente (posizione 0) al
    // piu' recente (posizione 99).
    public /*@ pure */ float getValue(String id, int i);

    // Aggiunge il valore di temperatura temp per il sensore con identificativo id (aggiungendolo
    // se non presente). Se sono gia' salvati 100 valori, il meno recente viene cancellato.
    public void add(String id, float temp);

    // Ritorna l'insieme di identificativi di tutti i sensori per cui la somma
    // dei valori registrati supera il valore t.
    // Lancia una NegativeThresholdException se il valore di t e' negativo.
    public /*@ pure */ Set<String> sensorsExceeding(float t) throws NegativeThresholdException;
}
```

Domanda a)

Si specifichi in JML il metodo `add()`.

Soluzione

Definiamo un predicato JML `unchanged(String s)` come segue:

```
//@ unchanged(String s) = sensors().contains(s) &&
//@     numValuesFor(s)==\old(numValuesFor(s)) &&
//@     (\forall int i; i>=0 && i<numValuesFor(s);
//@      getValue(s, i)==\old(getValue(s, i)))

//@ ensures sensors().contains(id) &&
//@ (\forall String s; \old(sensors()).contains(s) && !s.equals(id); unchanged(s)) &&
//@ (\forall String s; sensors().contains(s) && !s.equals(id); \old(sensors()).contains(s)) &&
//@
//@ !\old(sensors()).contains(id) ==> (
//@     numValuesFor(id)==1 && getValue(id, 0)==temp) &&
//@
//@ \old(sensors()).contains(id) ==> ( getValue(id, numValuesFor(id)-1)==temp &&
//@     (\old(numValuesFor(id)) < 100 ==>
//@         numValuesFor(id)==\old(numValuesFor(id))+1 &&
//@         (\forall int i; i>=0 && i<\old(numValuesFor(id))-1;
//@          getValue(id, i)==\old(getValue(id, i))) ) &&
//@     (\old(numValuesFor(id)) == 100 ==>
//@         numValuesFor(id)==\old(numValuesFor(id)) &&
//@         (\forall int i; i>=1 && i<\old(numValuesFor(id));
//@          getValue(id, i-1)==\old(getValue(id, i))) ) ) )
public void add(String id, float temp);
```

Domanda b)

Si specifichi in JML il metodo sensorsExceeding().

Soluzione

```
//@ensures t>=0 && (\forall String s; ;  
//@ \result.contains(s) <=> ( sensors().contains(s) &&  
//@ (\sum int i; i>=0 && i<numValuesFor(id); getValue(s, i)) > t ) )  
//@signals NegativeThresholdException && t<0  
public /*@ pure */ Set<String> sensorsExceeding(@float t) throws NegativeThresholdException;
```

Domanda c)

Si consideri un'implementazione che utilizza una mappa per contenere i valori di temperatura, come mostrato di seguito.

```
public class SensorsData {  
    private final Map<String, List<Float>> values;  
    ... }
```

Per tale implementazione, si definiscano l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

Nell'invariante di rappresentazione escludiamo che la mappa sia nulla o vuota, che contenga liste nulle, liste vuote o liste con più di 100 elementi.

```
//@(* Representation invariant RI *)  
//@private invariant values != null && values.size() > 0 &&  
//@ (\forall String s; values.containsKey(s);  
//@ values.get(s)!=null && !values.get(s).contains(null) &&  
//@ values.get(s).size()>0 && values.get(s).size()<=100 )
```

La funzione di astrazione definisce sensors(), numValuesFor() e getValue() in funzione della mappa usata nella rappresentazione. Gli altri metodi pubblici sono infatti definiti a partire da essi.

```
//@(* Abstraction function AF *)  
//@private invariant  
//@ (\forall String s; ; sensors().contains(s) <=> values.containsKey(s)) &&  
//@ (\forall String s; values.containsKey(s);  
//@ numValuesFor(s) == values.get(s).size() &&  
//@ (\forall int i; i>=0 && i<numValuesFor(s); getValue(s, i)==values.get(s).get(i))
```

Domanda d)

Si consideri ora una classe SensorsData2 che aggiunge un metodo addSensor(String id) che aggiunge un sensore (senza alcun valore di temperatura associato). È possibile definire SensorData2 come sottoclasse di SensorData in accordo con il principio di sostituzione?

Soluzione

Non è possibile, in quanto l'aggiunta del metodo porterebbe a violare una proprietà (invariante pubblico) della classe, ovvero che essa include solo sensori per cui è salvato almeno un valore di temperatura.

Esercizio 2

Si consideri la seguente classe Java:

```

public class GPSTrack {
    private double latitude[], longitude[];
    private int numPoints;
    public GPSTrack() {
        latitude = new double[1000];
        longitude = new double[1000];
        numPoints = 0;
    }
    public void addPoint(double latitude, double longitude) {
        synchronized(this.latitude) {
            this.latitude[numPoints] = latitude;
            this.longitude[numPoints] = longitude;
            numPoints++;
        }
    }
    public double getLatitude(int point) {
        synchronized(this.latitude) {
            return this.latitude[point];
        }
    }
    public double getLongitude(int point) {
        synchronized(this.longitude) {
            return this.longitude[point];
        }
    }
}

```

Domanda a)

La classe è correttamente sincronizzata? In caso affermativo spiegare brevemente perché. In caso negativo spiegare dove sia il problema e mostrare come vada cambiato il codice per ottenere una corretta sincronizzazione.

Soluzione

La classe è correttamente sincronizzata. Tutti i metodi che modificano lo stato dell'oggetto acquisiscono il lock sull'oggetto `this.latitude` il cui riferimento non cambia per tutta la vita dell'istanza.

Domanda b)

Si aggiunga alla classe GPSTrack un metodo `public void wait100()` che sospende il chiamante fin tanto che non siano stati aggiunti almeno 100 punti alla traccia. Si chiarisca se e come vadano cambiati gli altri metodi.

Soluzione

```

public void wait100() throws InterruptedException {
    synchronized(this.latitude) {
        while(numPoints < 100) this.latitude.wait();
        return;
    }
}

```

Bisogna anche aggiungere l'istruzione `this.latitude.notifyAll()` all'interno della parte sincronizzata del metodo `addPoint`.

Domanda c)

Supponendo l'esistenza di una classe `Printer` così fatta:

```

class Printer {
    static void plot(double lat, double lon) { ... }
}

```

il cui metodo `plot` stampa su una mappa il punto di coordinate indicate, si aggiunga alla classe `GPSTrack` un metodo: `public void plot()` che, **in parallelo rispetto al chiamante**, stampa tutti i punti della traccia.

Soluzione

```
public void plot() {
    double lat[], lon[];
    int np;
    synchronized(this.latitude) {
        lat = new double[1000];
        lon = new double[1000];
        np = numPoints;
        for(int i=0; i<numPoints; i++) {
            lat[i] = this.latitude[i];
            lon[i] = this.longitude[i];
        }
    }
    new Thread( () -> {
        for(int i=0; i<np; i++) Printer.plot(lat[i], lon[i]);
    }).start();
}
```

Esercizio 3

Si consideri questo frammento di codice Java:

```
public class Libro{
    public void leggi(){
        System.out.println("leggo_tutto");
    }

    public void leggi(int pag){
        System.out.println("leggo_pagina"+ pag);
    }
}

public class LibroConAudio extends Libro{
    public void ascolta(){
        System.out.println("ascolto");
    }

    public void leggi(){
        System.out.println("leggo_e_ascolto");
    }
}

1 Libro book = new Libro();
2 LibroConAudio book2 = new LibroConAudio();
3 Libro sound = new LibroConAudio();
4 LibroConAudio sound2 = new Libro();
5 book.leggi();
6 sound.leggi();
7 book2.leggi();
8 sound2.leggi();
9 sound.leggi();
10 book.leggi(2);
11 book2.leggi(2);
12 sound.leggi(2);
13 sound2.leggi(2);
```

Domanda a)

Scrivere il numero di riga delle istruzioni (se ne esistono) che generano un errore in compilazione.

Soluzione

Riga 4, in quanto `Libro` non è sottoclasse di `LibroConAudio`. Di conseguenza anche le righe 8 e 13.

Domanda b)

Supponendo di eliminare le eventuali righe che generano errori, si scriva, per ogni riga numerata, il valore stampato in uscita.

Soluzione

5: leggo tutto
6: leggo e ascolto
7: leggo e ascolto
9: leggo e ascolto
10: leggo pagina 2
11: leggo pagina 2
12: leggo pagina 2

Esercizio 4

Si consideri il seguente metodo Java:

```
public static int test(int x, int[] a) {  
    int count;  
    if (x <= 0 || a == null) return 0;  
    count = x;  
    while (count > 1) {  
        if (count > 0) count--;  
        else return count;  
    }  
    return count;  
}
```

Domanda:

Per ciascuno dei seguenti criteri di copertura, determinare un insieme minimale di casi di test che massimizzi la copertura:

1. Copertura delle istruzioni;
2. Copertura delle decisioni (branch);
3. Copertura delle decisioni e delle condizioni.

Soluzione

Copertura delle istruzioni. Non si può ottenere una copertura totale, in quanto non si può mai entrare nel ramo `else` del secondo `if` (quello all'interno del ciclo), che risulta vero in tutti i casi in cui si entra nel ciclo medesimo.

- (`x=2, a=null`) copre il ramo `if`.
- (`x=2, a=[1]`) entra nel `while` una volta (eseguendo il ramo `if` e decrementando `count`) e infine esegue l'ultima istruzione `return`

Copertura delle decisioni (branch). I casi di test riportati sopra massimizzano anche il criterio di copertura delle decisioni.

- Il primo caso valuta positivamente il primo `if`.

- Il secondo caso valuta negativamente il primo `if`, positivamente e negativamente il `while`, positivamente il secondo `if`.

Copertura delle decisioni e delle condizioni. Per massimizzare la copertura, oltre ai casi precedenti occorre aggiungere un caso che renda falsa la prima condizione della prima decisione (primo `if`).

- ($x=0, a=qualsiasi$) per valutare positivamente la prima condizione del primo `if`.

Appello del 20 Luglio 2018



Politecnico di Milano
Anno accademico 2017-2018

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Cugola <input type="checkbox"/> Mottola <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica di una classe Labirinto.

```
public class Labirinto {  
    /*OVERVIEW: Un labirinto costituito da celle, con una sola  
    entrata e una sola uscita. L'uscita e' sempre raggiungibile  
    dall'entrata. La struttura del labirinto non cambia dopo la  
    creazione.  
    Ogni cella puo' essere occupata da un numero arbitrario  
    di giocatori.  
    Un giocatore si puo' muovere da una cella ad un'altra  
    adiacente, muovendosi in una delle 4 direzioni cardinali  
    N,S,E,O.  
*/  
  
    //genera in modo casuale un labirinto di n>=2 celle (contando in e out),  
    //in cui esiste almeno un cammino da in a out (celle distinte);  
    //solleva l'eccezione se i parametri non soddisfano i vincoli previsti  
    public Labirinto(int n, Cella in, Cella out) throws InvalidException;  
  
    //sotto l'ipotesi che la direzione d sia una fra N,S,O,E (precondizione)  
    //restituisce la cella adiacente nella direzione data oppure la  
    //cella x stessa se non ci sono celle adiacenti nella direzione data  
    public Cella adiacente(Cella x, char d);  
  
    //restituisce un elenco di tutte le celle del labirinto,  
    public ArrayList<Cella> celle();  
  
    //restituisce un elenco di tutti i giocatori presenti.  
    public ArrayList<Giocatore> giocatori();  
  
    //restituisce un cammino costituito da celle adiacenti  
    //a partire dall'ingresso fino all'uscita del labirinto.  
    public ArrayList<Cella> soluzione();  
  
    //pone il giocatore g nella cella di ingresso del labirinto  
    //solleva l'eccezione se g e' gia' presente nel labirinto  
    public void entrata(Giocatore g) throws GiocatorePresenteException;  
  
    //restituisce true sse g si trova nella cella di uscita del labirinto  
    public boolean uscita(Giocatore g)  
  
        //restituisce la Cella in cui si trova g  
        //solleva l'eccezione se g non e' presente nel labirinto  
        public Cella posizione(Giocatore g) throws GiocatoreNonPresenteException;  
  
        //sotto l'ipotesi che il giocatore sia presente nel labirinto (precondizione)  
        //muove il giocatore g di una cella nella direzione indicata  
        //restituisce true se la mossa e' possibile, false altrimenti (ovvero se non ci  
        //sono celle adiacenti nella direzione indicata).  
        //lancia l'eccezione se la direzione non e' ammissibile (non e' una tra N,S,O,E)  
        public boolean muovi(Giocatore g, char d) throws DirezioneErrataException;  
}
```

Le classi Giocatore e Cella non vengono specificate non essendo rilevanti per le domande che seguono.

Domanda a)

Indicare quali metodi della classe `Labirinto` sono osservatori puri. Si scriva poi in JML un invariante pubblico per la classe `Labirinto`, individuando inoltre eventuali proprietà che non possono essere espresse con un invariante. Si dia poi la specifica JML dei metodi `Labirinto`, `uscita`, `muovi`.

Soluzione

Osservatori: `adiacente`, `celle`, `soluzione`, `posizione`, `uscita`, `giocatori`
L'invariante può esprimere le seguenti condizioni: la soluzione ha almeno dimensione 2, la soluzione è contenuta nel labirinto, la posizione di ogni giocatore è parte del labirinto, il labirinto non contiene celle o giocatori ripetuti, la soluzione è formata da una sequenza di celle adiacenti.

Per semplificare le formule, introduciamo un predicato che afferma che un `ArrayList` *x* non contiene due elementi con lo stesso reference.

```
noDuplicate(x)=  
(\forall int i; 0<=i && i<x.size()-1;  
 !(\exists int j;i<i && j<x.size(); x.get(i)==x.get(j))
```

e un altro che afferma che gli elementi dell'`ArrayList` non cambiano (anche l'ordine resta uguale).

```
fixed(x)=  
x.size()==\old(x.size()) &&  
(\forall int i; 0<=i && i<x.size();  
 x.get(i)==\old(x.get(i))  
  
//@public invariant soluzione().size()>=2 &&  
//@celle().containsAll(soluzione()) &&  
//@\forall Giocatore g; giocatori().contains(g);  
//@ celles().contains(posizione(g)) &&  
//@noDuplicate(celle()) && noDuplicate(giocatori()) &&  
//@\forall int i; 0<=i && i<soluzione().size()-1;  
//@\ \exists char d; (d== 'N' || d== 'S' || d== 'E' || d== 'O');  
//@ soluzione().get(i+1)==adiacente(soluzione().get(i),d);
```

Un invariante non è in grado di esprimere la proprietà evolutiva che la struttura del labirinto non cambia dopo la creazione (se non per l'inserimento o lo spostamento dei giocatori nelle celle).

Definiamo quindi come abbreviazione la proprietà che le celle non cambiano, la soluzione non cambia e le adiacenze non cambiano:

```
nochange = fixed(celle()) && fixed(soluzione()) &&  
 (\forall Cella c1; celle().contains(c1) &&  
 (\forall Cella c2; celle().contains(c2);  
 (\forall char d; d=="N" || d=="S" || d=="E" || d=="O";  
 c1.equals(adiacente(c2,d)) <=> c1.equals(\old(adiacente(c2,d))))
```

che dovrà essere inserita come postcondizione di tutti i metodi modificatori.

```
//@ensures(n>1 && in!=null && out!=null && !(in==out) &&  
//@celle().size()==n && soluzione().size()>=2 &&  
//@celle().containsAll(soluzione()) && giocatori().size()==0 &&  
//@soluzione().get(0)==in && soluzione().get(soluzione().size()-1)==out;  
//@signals(InvalidOperationException e) n<2 || in==null || out == null || in==out;  
public Labirinto(int n, Cella in, Cella out) throws InvalidOperationException;  
  
//@ensures \result==(soluzione().get(soluzione().size()-1).contains(g));  
public boolean /*@ pure @*/ uscita(Giocatore g)
```

```

//@requires giocatori().contains(g);
//@ensures (d=='N' || d=='S' || d=='E' || d=='O') &&
//@nochange && posizione(g)==\old(adiacente(posizione(g),d)) &&
//@(\forall Giocatore g1; giocatori().contains(g1) && g!=g1;
//@    posizione(g1)==\old(posizione(g1)));
//@signals (DirezioneErrataException e) nochange &&
//@ fixed(giocatori()) &&
//@ (\forall Giocatore g; giocatori().contains(g));
//@    posizione(g)==\old(posizione(g))) &&
//@ (d!='N' && d!='S' && d!='E' && d!='O');
public boolean muovi(Giocatore g, char d) throws DirezioneErrataException;

```

Domanda b)

Si consideri una variante della classe in cui il movimento può avvenire in 8 direzioni (ossia anche diagonalmente: NO, NE, SE, SO). Che cosa dovrebbe cambiare nella classe `Labirinto` (senza aggiungere nuovi metodi)?

Sarebbe possibile definire una siffatta classe come estensione di `Labirinto`? Oppure definire `Labirinto` come un'estensione di questa? Giustificare accuratamente la risposta.

Soluzione

Occorre modificare la specifica dei metodi `adiacente` e `muovi` per consentire le nuove direzioni.

Nel metodo `adiacente`, il parametro `d` ha una precondizione che richiede che sia una fra N,S,O,E: aggiungere le altre direzioni indebolisce la precondizione ed è quindi accettabile secondo la regola dei metodi.

Per il metodo `muovi`, invece, la postcondizione prevede il vincolo N,S,O,E: il rilassamento del vincolo la indebolisce, violando la regola dei metodi.

Quindi la nuova classe non puo' ereditare dalla precedente, a meno che non si definisca un secondo metodo per il movimento nelle altre 4 direzioni (p.es. `muoviInDiagonale`).

Anche il viceversa non e' possibile (in quanto stavolta la precondizione di `adiacente` sarebbe rafforzata).

Esercizio 2

Si consideri la seguente classe Java che realizza il tipo `Coppia`.

```

public class Coppia {
1.     private double elem1, elem2;
2.     private Object lock;
3.
4.     public Coppia() { elem1 = elem2 = 0; }
5.
6.     public synchronized double get1() {
7.         return elem1;
8.     }
9.
10.    public synchronized void set1(double val) {
11.        elem1 = val;
12.    }
13.    notifyAll();
14.}
15.

```

```

16.     public void waitForNegative() throws InterruptedException {
17.         synchronized(lock) {
18.             while(elem2>=0) wait();
19.         }
20.     }
}

```

Domanda a)

Si indichi se le seguenti affermazioni sono vere o false, motivando brevemente la propria risposta:

1. I metodi `get1` e `set1` non possono eseguire in parallelo.

2. I metodi `set2` e `waitForNegative` non possono eseguire in parallelo.

3. I metodi `set1` e `set2` non possono eseguire in parallelo.

4. Il codice compila ed esegue senza errori. In caso negativo evidenziare. Sfruttando i numeri di linea indicati nel testo, quali linee vadano modificate (e come) per evitare gli errori.

Soluzione

1. Vero, i due metodi sono sincronizzati sullo stesso oggetto (`this`).
2. Vero, i due metodi sono sincronizzati sullo stesso oggetto (`this.lock`).
3. Falso, i due metodi sono sincronizzati su oggetti diversi (`this` e `this.lock`).
4. Il codice compila correttamente ma ha un errore run-time eseguendo i metodi `set2` e `waitForNegative`. L'errore deriva dall'invocare `wait` e `notifyAll` su `this` quando si ha il lock sull'oggetto `this.lock` ma non su `this`. La soluzione prevede di sostituire la linea 13 come segue:
 13. `lock.notifyAll()`
 Analogamente la linea 18 diventa:
 18. `while(elem2>=0) lock.wait();`

Domanda b)

Si aggiunga, alla classe sviluppata al punto precedente, un metodo `public void set12()` che imposta il valore dell'attributo `elem2` al doppio del valore dell'attributo `elem1`. Nel fare ciò, se necessario, si modifichi il codice di sincronizzazione in maniera da evitare conflitti nell'accesso agli attributi condivisi da parte dei vari metodi (incluso il nuovo).

Soluzione

Il nuovo metodo `set12` deve accedere a entrambi gli attributi `elem1` ed `elem2`, occorre quindi sincronizzarlo rispetto a tutti gli altri metodi. Ciò può essere ottenuto cambiando il codice di sincronizzazione in maniera da sincronizzare tutti i metodi su `this` come segue:

```

public class Coppia {
    ...
    public synchronized double get1() {
        return elem1;
    }

    public synchronized void set1(double val) {
        elem1 = val;
    }

    public synchronized void set2(double val) {
        elem2 = val;
        notifyAll();
    }

    public synchronized void waitForNegative() throws InterruptedException {
        while(elem2>=0) wait();
    }

    public synchronized void set12() {
        elem2 = 2*elem1;
        notifyAll();
    }
}

```

In alternativa di può tenere la soluzione originale, con sincronizzazione separata, avendo l'accortezza, nel nuovo metodo `set12`, di acquisire il lock tanto su `this` quanto su `this.lock`, come segue:

```

public synchronized void set12() {
    synchronized(lock) {
        elem2 = 2*elem1;
        lock.notifyAll();
    }
}

```

Esercizio 3

Il diagramma UML in Figura 1 modella un calcolatore moderno e le parti di cui è tradizionalmente composto. Un calcolatore include un processore, una memoria, ed eventualmente card addizionali, ad esempio, una o più sound card. Un processore può essere una CPU oppure una accelerator card che aggiunge un determinato comportamento ad una CPU. Un processore deve essere inizializzato e può leggere/scrivere dalla memoria per caricare il programma ed eseguirlo. Una memoria può essere una RAM o una cache associata ad una RAM. Un calcolatore può gestire al massimo tre card addizionali. Per gestire una card addizionale, il calcolatore deve avere una extension board. Esistono diverse sound card con diverse funzioni.

Domanda a)

Indicare se e quali pattern UML sono stati utilizzati nel diagramma. Per ciascun pattern eventualmente individuato, indicare quali classi del diagramma sono coinvolte, quale è il ruolo del pattern, e secondo voi per quale motivo è stato applicato.

Soluzione

Si possono trovare almeno tre pattern applicati nel diagramma:

1. Proxy pattern: nel modellare l'accesso del processore alla memoria. Il proxy viene utilizzato per mascherare la differenza tra accesso in RAM e in cache.

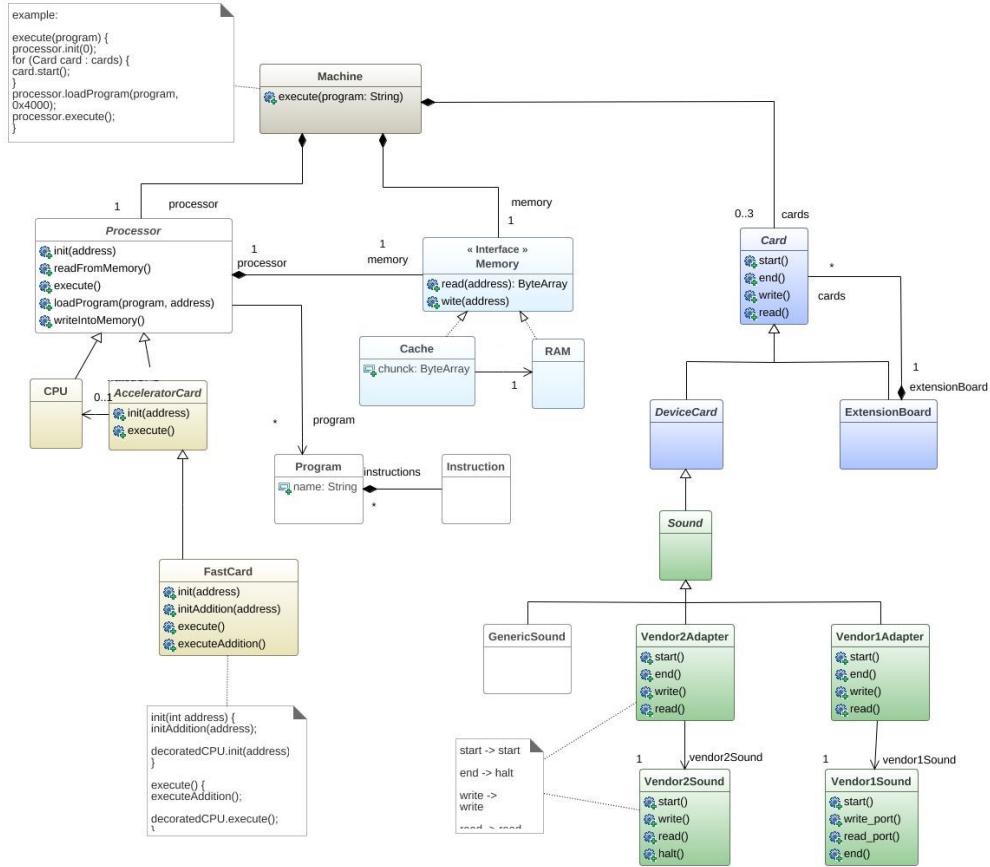


Figura 1: Diagramma UML di un moderno calcolatore.

2. Adapter pattern: nel modellare l'uso di sound card di diversi produttori. L'adapter converte comandi di alto livello nei comandi di basso livello per accedere alla sound card, che possono essere diversi da produttore a produttore.
3. Decorator pattern: nel modellare l'uso di accelerator card. Il decorator viene usato per aggiungere i comportamenti della accelerator card alla CPU esistente.

Domanda b) Estendere o modificare il diagramma in Figura 1 per modellare calcolatori con diverse *application-specific CPUs* invece che una sola CPU general purpose, ed uno *scheduler* che decide dinamicamente quale CPU deve eseguire l'istruzione corrente. Una application-specific CPU è una particolare CPU ottimizzata per eseguire solo un sottoinsieme delle istruzioni che possono comparire in un programma. Il resto del comportamento del calcolare resta quello descritto sopra. Indicare le modifiche al diagramma e se e quali pattern possono essere applicati in questo caso.

Soluzione

Si può utilizzare il pattern Strategy. Ciascuna application-specific CPU rappresenta una strategia ottimizzata per eseguire una determinata istruzione. Il comportamento dello scheduler può essere incorporato nella classe che fornisce l'interfaccia per il pattern strategy, o inserito come un componente separato ed associato allo Strategy.

Esercizio 4

Si consideri il seguente codice Java per la classe `Item`, ed una test suite composta da due test, `testOne` e `testTwo`, descritti sotto.

```
public class Item {
    private String item;
    private int size;
    public void setItem(String text)
    {
        item = text;
        size = item.length();
    }
    public void reduce()
    {
        if (size > 0)
        {
            item = item.substring(1,size);
            size = size - 1;
        }
    }
    public int getSize() { return size; }
    public String getItem() { return item; }
}

public void testOne() {
    Item item1 = new Item();
    item1.setItem("grip");
    assertEquals(4, item1.getSize());
    assertEquals("grip", item1.getItem());
    item1.reduce();
    assertEquals(3, item1.getSize());
    assertEquals("rip", item1.getItem());
}

public void testTwo() {
    Item item1 = new Item();
    item1.setItem("");
    assertEquals(0, item1.getSize());
    assertEquals("", item1.getItem());
    item1.reduce();
    assertEquals(0, item1.getSize());
    assertEquals("", item1.getItem());
}
```

Si risponda alle seguenti domande (a) e (b).

Domanda a)

Si calcoli la copertura della suite secondo il criterio delle istruzioni, cammini, e diramazioni (branch). Motivare la risposta.

Soluzione

La test suite che viene proposta raggiunge copertura totale secondo tutti i criteri proposti.

Domanda b)

Esiste qualche potenziale problema nel codice della classe `Item` che la suite non rileva? Quale test andrebbe aggiunto per rilevare il problema? Come andrebbe cambiata l'implementazione di `Item` per risolvere il problema?

Soluzione

Il fatto che la stringa venga inserita con un metodo `set`, affidandosi al costruttore di default per l'inizializzazione degli attributi, rende l'utilizzo della classe vulnerabile ad un eventuale `NullPointerException`, che verrebbe rivelata da un test come:

```
public void testThree() {  
    Item item1 = new Item();  
    assertEquals(0, item1.getSize());  
    assertEquals("", item1.getItem());  
}
```

Il fix è banale, e richiede di modificare la classe aggiungendo un costruttore che prenda la stringa di test come parametro.

Politecnico di Milano

Ingegneria del Software – a.a. 2008/09

Appello del 17 Luglio 2009

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totalle 1

Esercizio 1 (punti 6)

parte a

Si consideri il seguente metodo statico:

public static String maxSottostringa (String[] S)

Il metodo riceve come parametro un array di stringhe, che contiene almeno una stringa, e in cui tutte le stringhe non sono vuote.

Il metodo restituisce la sottostringa massimale comune, ossia la più grande sottostringa consecutiva comune a tutti gli elementi dell'array. Qualora non vi sia nessuna stringa comune (nemmeno di un singolo carattere) il metodo restituisce la stringa vuota "". Se vi sono più sottostringhe massimali della stessa lunghezza, il metodo ne restituisce una qualunque.

Ad esempio, se $S = [\text{acca accbb ccaba}]$, il metodo restituisce la stringa cc.

Specificare **maxSottostringa(...)** in JML. Non è necessario che la specifica sia eseguibile. Per comodità si riportano alcuni metodi pubblici della classe String, con i relativi contratti estratti dalla libreria java:

int length(): Restituisce la lunghezza della stringa this.

public String substring(int beginIndex, int endIndex) Restituisce una nuova stringa che è una sottostringa della stringa, e che inizia al beginIndex specificato e si estende fino al carattere all'indice endIndex - 1. NB: Gli indici della stringa vanno da 0 a $\text{this.length()}-1$, come per gli array.

Throws: IndexOutOfBoundsException – se $\text{beginIndex} < 0$ oppure $\text{endIndex} > \text{length()}$ oppure $\text{beginIndex} > \text{endIndex}$.

Una semplice soluzione non eseguibile è:

```
//@ requires S != null && S.length>0 &&
//@ ensures \result != null &&
// \result è una sottostringa di ogni elemento di S:
//forall int h; 0<= i && i< S.length;
//exists int i; 0<= i && i<S[h].length();
//exists int j; i<j && j<=S[h].length; \result.equals(S[h].substring(i,j))
// ) ) &&
//forall String y; y != null;
//ogni altra sottostringa di tutti gli elementi ha lunghezza non superiore a quella di \result:
//forall int h; 0<= i && i< S.length;
//forall int i; 0<= i && i<S[h].length;
//forall int j; i<j && j<=S[h].length; y.equals(S[h].substring(i,j))
```

))) => $y.length \leq x.length$)

parte b

Il metodo definito nella parte (a) definisce una funzione parziale o totale? Perché? Se la funzione non è totale, come si potrebbe modificare la specifica rendendo la funzione totale?

```
//@ requires true;  
//@ensures S.length>0 && (\forall int i; 0<= i && i< S.length; S != null &&  
S[i].length>0)  
&& \result != null &&  
// \result è una sottocatena di ogni elemento di S:  
(\forall int h; 0<= i && i< S.length();  
(\exists int i; 0<= i && i< S[h].length();  
(\exists int j; i< j && j<= S[h].length; \result.equals(S[h].substring(i,j)))  
) ) &&  
(\forall String y; y != null;  
//ogni altra sottocatena di tutti gli elementi ha lunghezza non superiore a quella di \result:  
(\forall int h; 0<= i && i< S.length;  
(\forall int i; 0<= i && i< S[h].length;  
(\forall int j; i< j && j<= S[h].length; y.equals(S[h].substring(i,j))  
) ) ) => y.length <= \result.length )  
// signals (NullPointerException e) S == null ||  
(S.length>0 && (\exists int i; 0<= i && i< S.length; S[i] != null);  
// signals (IllegalArgumentException e) S != null && (S.length == 0 ||  
(\exists int i; 0 <= i <= S.length; S[i] != null && S[i].length == 0));
```

Esercizio 2 (punti 14)

La classe **Corso** descrive un corso universitario. Ha tre metodi osservatori puri:

- **public int getCodice()**, che restituisce il codice del corso
- **public Docente getDocente()**, che restituisce il riferimento al docente
- **public boolean precedenza(Corso c)**, che restituisce un valore vero se il corso passato come parametro costituisce una precedenza obbligatoria per il corso in questione (e cioè il suo esame deve essere superato prima).

Il costruttore di **Corso** riceve come parametri un codice intero, un riferimento a **Persona** e un **ArrayList<Corso>** che contiene le precedenze obbligatorie.

La classe **Corso** è immutabile, ed in particolare non è possibile modificare l'insieme delle precedenze una volta che l'istanza del corso è stata creata.

Si consideri il seguente vincolo:

Un corso C non può essere precedenza di se stesso né può avere come precedenza un corso K che ha come precedenza il corso C.

(a) Come si potrebbe esprimere questo vincolo come invariante pubblico di classe?

```
//@ public invariant (@forall Corso k; k != null;  
                      this.precedenza(k) => !k.precedenza(this));
```

(b) L'invariante è soddisfatto dalla specifica della classe? Se sì, perché? Se no, che cosa bisogna fare perché sia soddisfatto?

Normalmente le specifiche dei metodi della classe devono supportare l'invariante di classe, e quindi bisogna gestire accuratamente i casi che potrebbero portare alla sua violazione. Per esempio, considerando un insieme di numeri senza duplicati, la specifica del metodo add(int num) deve escludere la possibilità di inserire un valore duplicato nell'insieme.

Tuttavia, in questo caso l'invariante di classe viene garantito per costruzione, in quanto l'elenco delle dipendenze viene richiesto nel costruttore del Corso. Per questo

- *un corso non potrà mai avere come precedenza se stesso (dovrei passare nel costruttore un'istanza dell'oggetto che sto creando)*
- *dato un corso C che ha come precedenza il corso K, K non può avere come precedenza il corso C. Nella costruzione di C, infatti, devo passare come parametro il riferimento al corso K, affinché C abbia come precedenza K. Tut-*

tavia, affinché K abbia come dipendenza C , al momento della costruzione di K dovrei passare un riferimento al corso C , che non è ancora stato costruito.

Per questi motivi, non è strutturalmente possibile violare l'invariante pubblico.

La classe **Curriculum** descrive l'insieme degli esami superati da uno studente. Fornisce il seguente metodo osservatore puro:

- **public ArrayList<Corso> getCV()**, che restituisce gli esami superati e i seguenti ulteriori metodi:
- **public Curriculum()**, il costruttore che inizializza il curriculum con zero esami sostenuti
- **public void addEsame(Corso c)**, che viene chiamato quando l'esame è stato superato, e quindi va aggiunto al curriculum. Il metodo deve lanciare l'eccezione **EsameIllegaleException** se il curriculum contiene già un esame superato con lo stesso codice dell'esame che si vuole inserire nel curriculum
- **public int getVoto(Corso c)**, che restituisce il voto ottenuto dallo studente nel corso specificato.

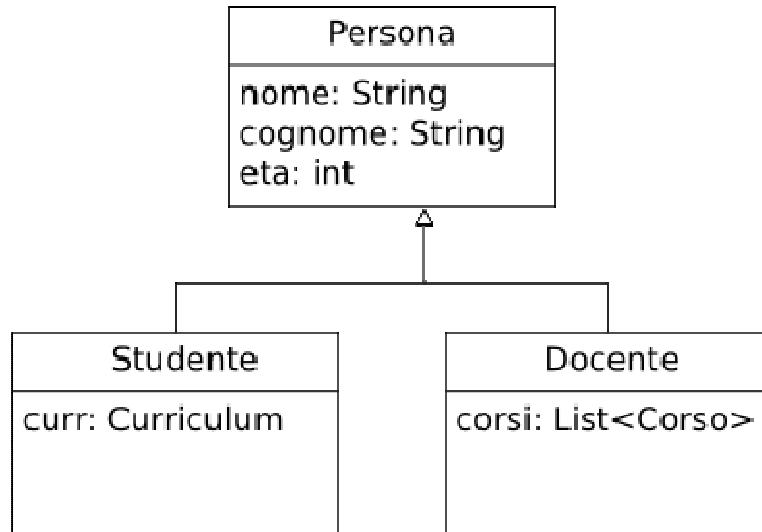
(c) Fornire la specifica JML del costruttore **Curriculum** e del metodo **addEsame()**.

```
//@ ensures getCV().size == 0;
public Curriculum();

//@requires c != null;
//@ensures (\forall int i; 0 <= i && i < \old(getCV().size());
//          \old(getCV().get(i).getCodice()) != c.getCodice()) &&
//          (\forall int i; 0 <= i && i < getCV().size();
//            getCV().get(i) == c ||
//            (\exists int j; 0 <= j && j < \old(getCV().size());
//              \old(getCV().get(j)) == getCV().get(i) ));
// signals (EsameIllegaleException e) (\exists int i; 0 <= i && i < \old(getCV().size());
//          \old(getCV().get(i).getCodice()) == c.getCodice());
```

Si ipotizzi di voler definire le classi **Studente** e **Docente**. Esse hanno attributi comuni (per esempio, nome, cognome, età), ma anche attributi aggiuntivi diversi: **Studente** ha un **Curriculum** e **Docente** ha un insieme di corsi.

(d) Come pensate di definire queste classi? Definite la struttura, senza scendere in dettagli implementativi.



Si supponga che una classe **CorsoDiLaurea** abbia due osservatori puri che consentono di estrarre gli studenti e i docenti del corso di laurea. I metodi sono rispettivamente:

- **public ArrayList <Docente> getDocenti();**
- **public ArrayList <Studente> getStudenti();**

È richiesto che i due osservatori rispettino il vincolo che i docenti degli esami superati dagli studenti del corso di laurea siano docenti dello stesso corso di laurea.

(e) Specificare i due osservatori in JML.

E' sufficiente che il vincolo sia imposto da uno solo dei due metodi:

```
//@ ensures \result != null  
public ArrayList <Docente> getDocenti();  
  
//@ ensures \result != null && (\forall int i ; 0 <= i && i < \result.size();  
    (\forall int j; 0 <= j && j < \result.get(i).getCurr().getCV();  
     (\exists int z; 0 <= z && z < getDocenti().size();  
      \result.get(i).getCurr().getCV().get(j).getDocente() ==  
      getDocenti().get(z))));  
public ArrayList <Studente> getStudenti();
```

Si supponga che la classe **Curriculum**, invece di esportare il metodo **getCV**, definisca un iteratore che consente di ottenere uno alla volta i corsi superati da uno studente.

(f) Definire in modo schematico la classe Curriculum in questa nuova versione.

```
class Curriculum implements Iterable<Corso> {  
    private static class CurrIter implements Iterator<Corso> {  
        public boolean hasNext() { ... }  
        public Corso next() { ... }  
        public void remove () { ... }  
    }  
  
    public void addEsame(Corso c) { ... }  
    public Iterator<Corso> iterator() {  
        return new CurrIter ();  
    }  
}
```

(g) Definire un metodo che calcola la media degli esami di un certo curriculum, usando l'iteratore.

//intestazione del metodo...

Curriculum cur // parametro del metodo

```
int tot = 0, num = 0;  
for(Corso c : cur) {  
    num++;  
    tot += cur.getVoto(c);  
}  
return ( (float) tot ) / num;
```

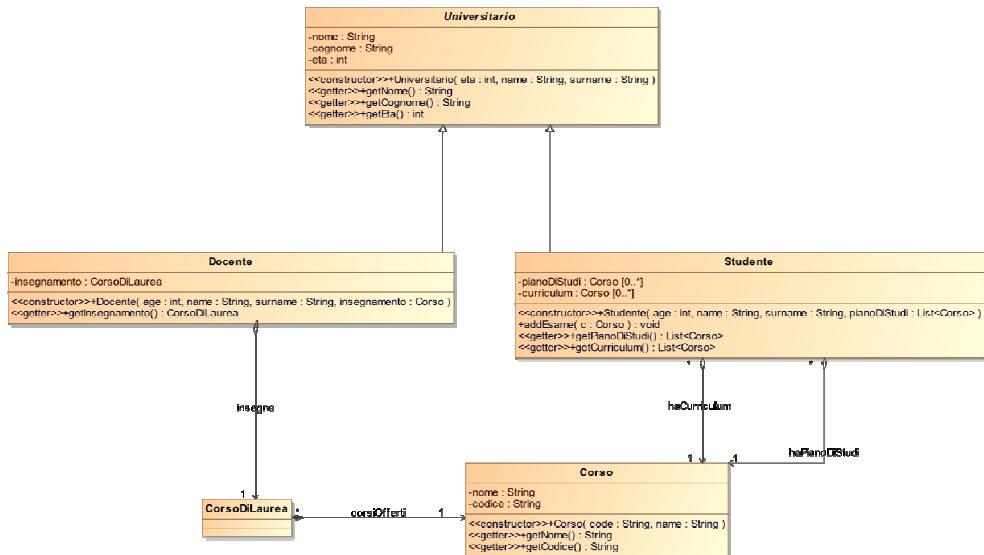
Esercizio 3 (punti 5)

Si formalizzi con un diagramma delle classi UML la situazione di cui all'esercizio precedente. Più precisamente, un'università è caratterizzata da corsi laurea, nei quali operano docenti e studenti. Ogni studente ha un curriculum di esami superati e ogni docente ha associati i corsi che insegna. Oltre al curriculum, ogni studente ha anche un piano di studi, e cioè un insieme di corsi ancora da superare.

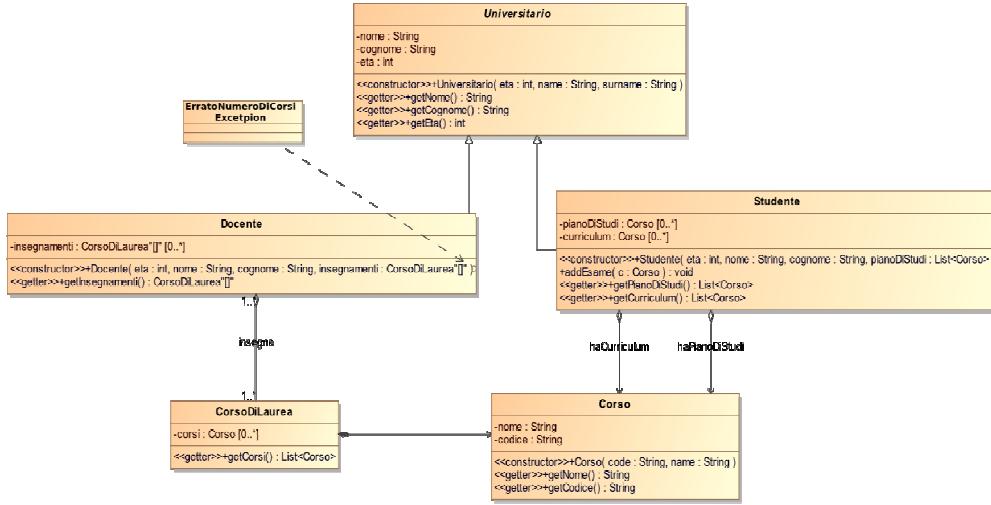
- Si dica esplicitamente che cosa cambia nel diagramma nelle seguenti ipotesi:

(a) ogni docente insegna in uno e uno solo corso di laurea, piuttosto che (b) un docente può insegnare in più corsi di laurea, ma almeno in uno e in non più di 3, piuttosto
che (c) non esistono vincoli al numero di corsi laurea in cui un docente può insegnare.

(a)

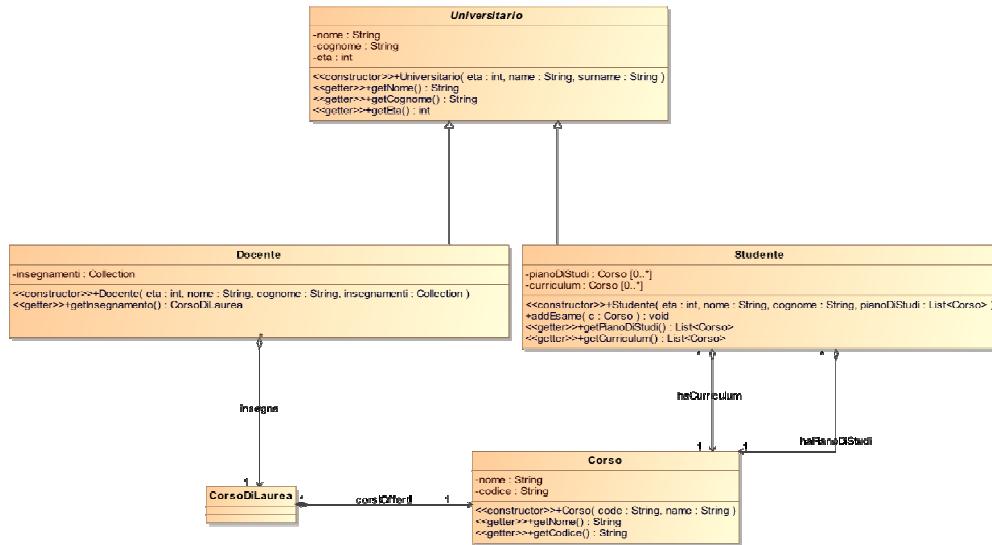


(b)



Si noti che UML non consente di specificare la molteplicità 1..3 per una relazione, per questo bisogna specificare comunque la molteplicità 1.. e poi inserire nel costruttore un controllo che l'array di CorsoDiLaurea passato come parametro non contenga mai più di 3 elementi e che non sia vuoto. Nel caso questo accada viene lanciata l'eccezione **ErratoNumeroDiCorsiException**.*

(c)



- Usando JML si esprima il vincolo che per ogni studente la somma degli esami del curriculum e degli esami del piano di studi deve essere 30.

```
Public class Studente extends Universitario {  
//@public invariant getPianoDiStudi.size() + getCurriculum.size() == 30;  
//corpo della classe  
}
```



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da

Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

I Prova di Ingegneria del Software

27 Aprile 2004

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h30m.
6. Punteggio totale a disposizione: 13/30.

Esercizio 1 (punti 3)

Specificare la seguente astrazione procedurale *getInterval*, tramite pre e post condizioni:

- Sia dato un array *times*, che si ipotizza non nullo e che contiene una sequenza di valori float ordinati per valori strettamente crescenti (che rappresentano punti dell'asse dei tempi); sia dato inoltre un valore *timePoint* di tipo float. Fornire come risultato della chiamata un oggetto della seguente classe *Interval*, corrispondente a un intervallo temporale e avente come estremi due punti contigui di *times*, tale che *timePoint* sia maggiore o uguale all'estremo inferiore e minore dell'estremo superiore. Se il valore di *timePoint* è minore del primo valore di *times* o maggiore o uguale dell'ultimo viene lanciata l'eccezione *OutOfBoundsException*.

Si utilizzi la seguente breve specifica della classe *Interval*.

```
class Interval {  
    //OVERVIEW...  
    //AF(c) = [c.min(), c.max() )  
    ...costruttori e metodi vari...  
    float min()  
    //EFFECTS: restituisce l'estremo inferiore di this  
    float max()  
    //EFFECTS: restituisce l'estremo superiore di this
```

public static *Interval* getInterval (float[] times, float timePoint)
.....

```
public static Interval getInterval (float[] times, float timePoint) throws OutOfBoundsException  
//REQUIRES: times<>null && times.length>1 &&  
//forall i, 0<=i<times.length-1, times[i]< times[i+1]  
//EFFECTS: if (timePoint<times[0] || timePoint>=times[times.length-1]) throw  
//OutOfBoundsException  
//else return Interval x tale che exists int i: times[i]=x.min() <=timePoint < x.max()=times[i+1];
```

- Dare una versione della specifica in cui nel caso il *times* sia nullo viene generata un'opportuna eccezione.

public static *Interval* getInterval (float[] times, float timePoint)
.....

```
public static Interval getInterval (float[] times, float timePoint) throws OutOfBoundsException,  
NullPointerException
```

poi si elimina dal REQUIRES la condizione times<>null e si aggiunge all'inizio della clausola EFFECTS:

if (times ==null) throw NullPointerException

- Quali sono le differenze principali tra l'assumere che la condizione di vettore non nullo sia una precondizione piuttosto che la sua violazione generi un'eccezione?

Nel primo caso, si ipotizza che sia parte del “contratto” fra chiamante e chiamato che il chiamante rispetti la precondizione. In caso di mancato rispetto, il chiamato è “autorizzato” a qualunque tipo di comportamento. Non deve quindi mai accadere che il chiamante non rispetti la precondizione.

Se la violazione genera un’eccezione, invece, si ipotizza che il chiamante possa evitare di verificare la precondizione (e quindi anche non rispettarla), ma che sia in tal caso capace di gestire l’eccezione corrispondente alla violazione, dichiarata nella segnatura del metodo.

Esercizio 2 (punti 5)

Consideriamo la seguente specifica di un tipo di dato astratto modificabile (mutable) che rappresenta una coda di messaggi con priorità. Ogni messaggio ha associata una priorità, che è un valore nell'intervallo 0-9 (0 = bassa, 9 = alta). Le operazioni possibili sono l'inserimento di un certo messaggio con una certa priorità nella coda (*insert*) e l'estrazione (*extract*) dalla coda del messaggio da più tempo inserito fra quelli a priorità più alta presenti nella coda. *Extract* restituisce tale messaggio e lo elimina dalla coda. Un'operazione *length* restituisce la lunghezza della coda (numero di messaggi presenti in coda) e un'altra operazione *merge* riceve come parametro una coda che deve essere fusa con la coda corrente, facendo in modo che nella nuova coda corrente tutti i messaggi contenuti nella coda-parametro siano posti, a pari priorità, "dietro" a quelli della coda corrente.

1. Si completi la seguente specifica della classe CodaPriorità (supponendo nota la classe Messaggio), a partire dalla clausola OVERVIEW proposta, specificando per ciascun metodo, le clausole REQUIRES, MODIFIES e EFFECTS (se necessarie). Si specifichi anche quali operazioni siano costruttori, mutators e observers. Per ogni metodo si completi il prototipo, segnalando eventuali eccezioni.

(NB: ovviamente l'ammontare di righe vuote o di puntini non è correlato con la lunghezza della soluzione...)

```
public class CodaConPriorita {  
    /*OVERVIEW: Una coda con priorità è una sequenza di coppie del tipo (x,n) dove x è un Messaggio e n la sua priorità, 0<=n<=9.  
    La sequenza è composta da una successione di 10 "segmenti" (anche vuoti), ognuno dei quali contiene messaggi con la stessa priorità, disposti, all'interno del segmento, in ordine cronologico di inserzione nella coda (a sinistra, cioè con posizioni di indice più basso, i messaggi inseriti prima); il primo segmento contiene i messaggi a priorità 0, il secondo quelli a priorità 1, e così via.  
    Esempio: (m1, 3) (m4, 3) (m2,4) (m3,4) (m6,4) (m5,6) */
```

```
    public CodaConPriorita () { COSTRUTTORE  
        //EFFECTS: costruisce una coda vuota (senza elementi)  
    }  
    //Metodi:  
  
    public void insert(Messaggio m, int priorita) throws WrongPriorityException ..... { MUTATOR  
        //EFFECTS: if (0<= priorita <=9) inserisce (m,priorità) alla destra di tutti gli elementi con minore o  
        //uguale priorità, e a sinistra di tutti quelli con priorità maggiore .....  
        //else throw WrongPriorityException .....  
        //MODIFIES: this .....  
    }  
    public Messaggio extract() throws EmptyException ..... { MUTATOR+OBSERVER  
        //EFFECTS: if (this non è vuota), then, se max è la massima priorità presente nella coda, restituisce il  
        //Messaggio m corrispondente alla coppia (m,max) più a sinistra nella coda, al contempo eliminandolo  
        //dalla coda.....  
        //else throw EmptyException .....  
        //MODIFIES: this .....  
    }  
  
    public void merge(CodaConPriorita c) throw NullPointerException ..... { MUTATOR  
        .....  
        //EFFECTS: if (c == null) then throw NullPointerException  
        // this_post è una sequenza in cui sono presenti nello stesso ordine relativo, tutti gli elementi di  
        //this_pre, e tutti gli elementi di c di priorità i compaiono, nell'ordine originale di c subito dopo tutti gli  
        //elementi di this a priorità i. ....  
        //MODIFIES: this .....  
    }  
  
    public int length() ..... { observer  
        .....  
        //EFFECTS: return il numero di elementi di this .....  
    }
```

}

NB: La insert potrebbe anche verificare se il Messaggio m è vuoto, e in tal caso non inserire l'elemento ma lanciare un'opportuna eccezione

2. Si consideri un'implementazione per la quale la rep scelta sia un array di 10 riferimenti a vector che rappresentano i messaggi accodati per le diverse priorità, in modo che il messaggio ricevuto più recentemente per una certa priorità sia il LastElement del vector relativo. Inoltre un attributo privato intero maxPriorita memorizza la massima priorità presente in quel momento nella coda. Quando la coda è vuota maxPriorita valga -1. Data la rep, si definiscano anche l'AF e l'RI. Completare l'implementazione seguente, indicando brevemente, per ciascun metodo, (a) perché il rep invariant viene effettivamente mantenuto e (b) perché la specifica fornita viene rispettata.

//rep:

```
private Vector[] coda;  
private int maxPriorita;...
```

//AF(c):

// una sequenza C = C0...C9 formata dalla concatenazione di 10 segmenti Ci, ognuno dei quali è vuoto se e solo se coda[i] è vuota, altrimenti Ci = (coda[i].firstElement(),i)...(coda[i].lastElement(), i)

//RI(c) =

// coda è un array di 10 elementi di tipo Vector (!= null) &&

//tutti gli elementi nei Vector della coda sono di tipo Messaggio &&

se tutti i Vector sono vuoti, maxPriorita == -1, altrimenti maxPriorita è il massimo indice i per cui il Vector coda[i] contiene almeno un elemento

```
public CodaConPriorita ()..... {  
    coda=new Vector[10];  
    for (int i =0; i<10; ++i)  
        coda[i]=new Vector();  
    maxPriorita = -1;
```

}

Mantiene RI perché:

Per costruzione, maxPriorita è -1.....

.....

Rispetta specifica perché

Per la AF, se tutti i Vector coda[i] sono vuoti l'insieme di elementi è pure vuoto.

```
public void insert(Messaggio m, int priorita) throws WrongPriorityException ..... {  
    if (0<= priorita && priorita <=9) {  
        coda[priorita].add(m);  
        if (maxPriorita < priorita) maxPriorita = priorita;  
    }  
    else throw new WrongPriorityException();.....  
}
```

Mantiene RI perché:

.....

Inserisce un elemento di tipo Messaggio; aggiorna correttamente maxPriorita nel caso il nuovo messaggio abbia priorità maggiore di tutti quelli presenti

Rispetta specifica perché

Aggiunge m alla destra (come LastElement) di tutti gli elementi con la stessa priorità. La tesi segue perche' AF mantiene l'ordinamento degli elementi in coda con la stessa priorità.

```
public Messaggio extract() throws EmptyException ..... {  
    if (maxPriorita == -1) throw new EmtpyException();.....  
    Messaggio m = (Messaggio) coda[maxPriorita].get(0);  
    coda[maxPriorita].remove(0);  
    while (maxPriorita >= 0 && coda[maxPriorita].size() == 0)  
        maxPriorita--;  
    return m;  
}
```

Mantiene RI perché:

aggiorna correttamente maxPriorita nel caso in cui non rimanga alcun messaggio con la stessa priorità` di quello estratto

Rispetta specifica perché

trova l'i massimo corrispondente al Vector non vuoto e poi ne prende il primo elemento: per AF, questo è l'elemento a priorità massima e piu' a sinistra.

```
public void merge(CodaConPriorita c) throws NullPointerException ..... {  
    for ( int i =10; i>=0; --i)  
        for (int j=0; j<c.coda[i].size(); j++)  
            this.insert(c.coda[i].get(j), i);  
}
```

Mantiene RI perché:

Non modifica struttura; non aggiunge ne' toglie Vector; tutti gli elementi aggiunti sono di tipo Messaggio (perche' RI vale per c). L'inserimento mantiene RI (perche' aggiorna maxPriorita)

Rispetta specifica perché

Il metodo inserisce tutti gli elementi di c nella corrispondente coda, rispettandone la sequenza di inserimento e mettendoli dopo tutti quelli di this. Poiché AF rispetta l'ordine, la specifica è verificata.

```
public int length().....{  
    int len =0;  
    for (int i = 0;i<=maxPriorita; ++i) len = len + coda[i].size();  
    return len;.....  
}
```

Mantiene RI perché:

è un observer.....

Rispetta specifica perché

Restituisce la somma del numero di elementi in ogni coda: per la AF tutti e soli questi elementi sono in coda.

3. Si fornisca un'implementazione di AF ed RI, in modo che sia utile ai fini del debugging del programma.

```
public String toString() {  
    String s = " ";  
    for ( int i = 0; i < 10; i++)  
        for (int j=0; j<coda[i].size(); j++)  
            s = s + (" " + this.coda[i].get(j).toString() + " " + i + " ");  
    return s;  
}  
  
public Boolean repOK() {  
    boolean vuota =true;  
    if (coda.length!= 10) return false;  
    for ( int i = 9; i >= 0; --i)  
        if (coda[i].size()>0) {  
            vuota = false;  
            if (maxPriorita <i) return false;  
            for (int j=0; j<coda[i].size(); j++)  
                if (!coda[i].get(j) instanceof Messaggio) return false;  
        }  
    if (vuota) return maxPriorita== -1;  
    return true;  
}
```

Esercizio 3 (punti 2)

Si consideri il seguente frammento incompleto di programma Java:

```
class MammaMia {...};  
class Mortacci {...};  
class CheSo ...{  
    void f (...) throws MammaMia, Mortacci {  
        ...  
        throw new MammaMia();  
        ...  
    }  
    void g(...) throws MammaMia, Mortacci {  
        ...  
        try {  
            ...  
            f();  
        } catch (MammaMia m) { System.out.println('1');  
        } finally { throw new Mortacci();}  
    }  
}
```

Si ipotizzi che l'eccezione sollevata all'interno del metodo f() non venga gestita da f(), ma venga propagata.

Si consideri il seguente frammento di codice che invoca il metodo g:

```
CheSo oggetto = new CheSo ();  
try {  
    oggetto.g(...);  
    ...  
} catch (MammaMia m) { System.out.println('2');}  
} catch (Mortacci m) { System.out.println('3'); }
```

Supponendo che la chiamata di f() comporti il lancio dell'eccezione MammaMia, si indichi che cosa stampa il programma in seguito alla chiamata del metodo g(..) sulla variabile *oggetto*.

1
3

Esercizio 4 (punti 3)

Rispondere alle seguenti domande relative a Java:

(A) Si consideri la seguente classe:

```
class StranaCosa {  
    static int i = 0;  
    int j = 0;  
    void boh( ) {  
        i++;  
        j++;  
        System.out.println(i);  
        System.out.println(j);  
    }  
}
```

Si supponga che altrove appaia il seguente frammento:

```
StranaCosa x = new StranaCosa( );  
StranaCosa y = new StranaCosa( );  
x.boh( );  
y.boh( );
```

Che cosa viene stampato dal programma?

- 1**
- 1**
- 2**
- 1**

(B) Si suppongano date le classi X e Y, dove Y è erede di X. Si supponga che X possieda un metodo:

```
void a() {System.out.println( "a di X");}
```

mentre la classe Y ridefinisce il metodo come segue:

```
void a() {System.out.println( "a di Y");}
```

In presenza del seguente codice:

```
X x = new Y();  
x.a();
```

si indichi qual è il tipo statico di x e quale il suo tipo dinamico, e che cosa viene stampato.

Tipo statico: X, tipo dinamico Y. "a di Y"

(C) Sono date le classi X, Y, W e Z e con le seguenti relazioni di ereditarietà:

W eredita da Z; Y e Z ereditano da X. Le variabili x,y,z,w sono dichiarate come X x; Y y; W w; Z z;

In base alle dichiarazioni, quali tra le seguenti assegnazioni sono legali in Java? (Scrivere SI o NO a fianco di ciascuna.)

1. y = x; **NO**
2. x = y; **SI**
3. z = x; **NO**
4. x = w; **SI**
5. w = x; **NO**
6. y = z; **NO**



Dipartimento di Elettronica e Informazione

Politecnico di Milano – sede di Cremona

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da

Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Appello di Ingegneria del Software

13 Settembre 2004

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità.
Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h45m
6. Punteggio totale a disposizione: 26/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio (punti 18)

La classe `PilaFinitaDiInteriFiniti` realizza una pila i cui elementi sono numeri interi compresi tra 1 e 9 (estremi inclusi). Il contenuto della pila è codificato mediante un singolo numero intero non negativo, e le operazioni di push e di pop sono realizzate con opportune operazioni aritmetiche nella seguente maniera:

- la pila vuota è rappresentata dal numero 0;
- se s è il numero che rappresenta la pila e n è il numero che viene messo sulla pila, dopo una push di n su s il nuovo contenuto della pila s' è $s' = (s + n) * 10$;
- se s è il numero che rappresenta la pila (supposta non vuota), il numero che si trova in cima alla pila è $(s / 10) \% 10$;
- se s è il numero che rappresenta la pila (supposta non vuota), dopo una pop il contenuto della pila s' è dato da $s' = s / 10 - (s / 10) \% 10$.

Per esempio, se sulla pila vengono inseriti i numeri 7, 3 e 9 (in questo ordine), il numero che rappresenta la pila dopo questi inserimenti è 7390. Se su questa pila eseguo una pop (che mi ritorna il valore 9, che è uguale a $(7390 / 10) \% 10$), dopo l'operazione il contenuto della pila è dato dal numero 730 (cioè $7390 / 10 - (7390 / 10) \% 10$), e così via.

1. *Si completi la seguente specifica:*

```
public class PilaFinitaDiInteriFiniti {  
    .....  
    .....  
    .....  
    .....  
    .....  
    public PilaFinitaDiInteriFiniti ();  
    .....  
    .....  
    .....  
    public void push (int n) throws FullStackException;  
    .....  
    .....  
    .....  
    public int topEPop() throws EmptyStackException;  
    .....  
    .....  
    .....  
    public int length();  
    .....  
    .....  
    .....  
}
```

2. Si definisca o si implementi un Rep Invariant per la classe PilaFinitaDiInteriFiniti, usando come rep la seguente dichiarazione:

```
private int pila;
```

3. Definire ed implementare una funzione di astrazione per PilaFinitaDiInteriFiniti.

4. Completare la seguente implementazione parziale della classe PilaFinitaDiInteriFiniti e dimostrare che essa mantiene il RI.

```
public class PilaFinitaDiInteriFiniti {  
    private int pila;  
    public PilaFinitaDiInteriFiniti (){ pila = 0; }  
  
    public void push (int n) throws FullStackException {  
        int new_pila = .....  
        if (new_pila < 0) {  
            .....  
        }  
        pila = new_pila;  
    }  
  
    public int top() throws EmptyStackException {  
        .....  
        return (pila/10)%10;  
    }  
  
    public int pop() throws EmptyStackException {  
        if (length() == 0) .....  
        .....  
    }  
  
    public int length(){  
        if pila == 0 return 0;  
        return (new Integer(pila/10)).toString.length;  
    }  
}
```

5. Si consideri la specifica seguente del metodo statico mergeStack specificato di seguito:

```
public static PilaFinitaDiInteriFiniti mergeStack(PilaFinitaDiInteriFiniti p1, PilaFinitaDiInteriFiniti p2)
    throws StackMergeFailedException;
// EFFECTS: ritorna una nuova pila p_merge che è la fusione delle pile p1 e p2, cioè tale che se un numero
//           N compare in p1 un numero i1 di volte ed in p2 un numero i2 di volte, N appare in
//           p_merge i1+i2 volte.
//           Se la fusione delle due pile non è possibile perchè la p_merge eccede la dimensione
//           massima di una pila, il metodo solleva un'eccezione StackMergeFailedException.
```

Generare casi di test black-box per il metodo mergeStack.

6. *Specificare* un erede di PilaFinitaDiInteriFiniti, chiamata PilaDiInteriFinita, che non ha limiti rispetto al numero di elementi inseribili nella pila.

La classe così definita rispetta il principio di sostituzione?

Esercizio 2 (punti 8)

Si vuole progettare un dispositivo di distribuzione di bevande calde (caffè, the, cappuccino...). Il dispositivo è azionato inserendo una moneta o selezionando un prodotto dal pannello frontale (costituito da una serie di bottoni). Questo porta la macchina dallo stato di stand by allo stato di lavoro.

- La macchina deve verificare che le monete immesse siano sufficienti per poter acquistare il prodotto selezionato. Se le monete introdotte superassero l'importo del prodotto, la macchina deve rendere all'utente il resto in eccesso.
- La macchina deve preparare le bevande miscelando l'acqua e gli ingredienti e deve restituire all'utente il prodotto finito tramite uno sportello.

È presente un display che indica la disponibilità delle bevande (in base alla disponibilità dei relativi ingredienti) e il denaro richiesto o mancante per il prodotto selezionato.

Definire un class diagram per specificare il sistema dato, specificando le classi, le loro relazioni, gli attributi e i metodi.

Mostrare tramite sequence diagram i seguenti scenari tipici di funzionamento del sistema:

- a) L'utente inserisce della moneta, poi seleziona un prodotto; la moneta inserita è sufficiente e il prodotto è disponibile: la macchina restituisce all'utente il prodotto selezionato e l'eventuale resto.
- b) L'utente seleziona un prodotto prima di aver inserito la moneta; la macchina restituisce a display il costo del prodotto selezionato, l'utente inserisce la moneta e preleva la bevanda e l'eventuale resto.

Ingegneria del Software – a.a. 2006/07

Appello del 10 luglio 2007

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 6)

Il seguente metodo statico:

```
public static int[] fusioneOrdinata (int[ ] a, int[ ] b) {... }
```

effettua la fusione ordinata dei due array a e b, ossia fa in modo che, dopo la chiamata, l'array restituito al chiamante contenga tutti e soli gli elementi contenuti, prima della chiamata, nei due array a e b, e inoltre risulti ordinato.

Si scriva una specifica, in JML del metodo fusioneOrdinata nei due seguenti casi:

a- si assume che i due array a e b non abbiano elementi ripetuti, e siano disgiunti (nessun valore int può essere memorizzato in entrambi)

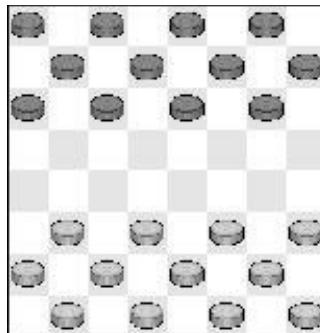
```
//@assignable \nothing
//@requires a!= null && b!= null &&
// ogni valore int memorizzato in a compare una sola volta in a e non compare in b
//@  (\forall int i; 0<=i&&i<a.length; (\num_of int j; 0<=j<a.length; a[j]==a[i])==1 && (\num_of int k;
0<=k<b.length; b[k]==a[i])==0)
// ogni valore int memorizzato in b compare una sola volta in b e non compare in a
//@  (\forall int i; 0<=i&&i<b.length; (\num_of int j; 0<=j<b.length; b[j]==b[i])==1 && (\num_of int k;
0<=k<a.length; a[k]==b[i])==0)
//@ensures \result != null &&
//\result è ordinato
//@  (\forall int i; 0<=i<\result.length-1; \result [i]<= \result [i+1]) &&
// tutti gli elementi di a stanno in \result
//@      (\forall int i; 0<=i<a.length; (\exists int j; 0<=j<\result.length; a[i]== \result[j])) &&
// tutti gli elementi di b stanno in \result
//@      (\forall int i; 0<=i<b.length; (\exists int j; 0<=j<\result.length; b[i]== \result[j])) &&
// \result non contiene elementi spuri
//@      \result.length = a.length + b.length
```

b- si ammettono anche i casi in cui i due array a e b abbiano elementi ripetuti, e non siano disgiunti

```
//@assignable \nothing
//@requires a!= null && b!= null &&
//@ensures \result != null &&
//\result è ordinato
//@  (\forall int i; 0<=i<\result.length-1; \result[i]<= \result[i+1]) &&
//il numero di volte in cui un qualsiasi numero compare in \result è pari alla somma
// del numero di volte incui compare in a e del numero in cui compare in b
//@  (\forall int i; 0<=i && i<\result.length;
//      (\num_of int j; 0<=j<\result.length; \result[j]==\result[i])) ==
//          (\num_of int j; 0<=j<a.length; a[j]== \result[i])+ (\num_of int j; 0<=j<b.length; b[j]== \result[i])
// \result non contiene elementi spuri: ora è diventato pleonastico
//@      \result.length = a.length + b.length
```

Esercizio 2 (punti 12)

Si consideri una versione semplificata del gioco della dama (italiana) in cui le pedine possono solo muovere in avanti. **Scacchiera** e **Pedina** sono le classi principali della soluzione presentata. Il gioco, al quale prendono parte due giocatori, si svolge su una scacchiera formata da 64 caselle: 32 bianche e 32 nere. Le pedine sono 24: 12 bianche e 12 nere e si dispongono sulle caselle scure come indicato in figura.



Le righe e le colonne della scacchiera sono numerate da 1 a 8 dall'alto in basso e da sinistra a destra. La prima pedina nera è quindi in posizione 1:1 (riga:colonna), mentre l'ultima pedina bianca è in posizione 8:8.

La classe **Scacchiera** fornisce i seguenti metodi:

- **public int numPezzi(int colore)** restituisce il numero di pedine sulla scacchiera di un certo colore (0 bianco e 1 nero).
- **public ArrayList<Pedina> pedine(int colore)** restituisce i pezzi sulla scacchiera come una lista di elementi.
- **public ArrayList<Pedina> diagonale(Pedina p, char direzione)** restituisce la sequenza di pedine sulla diagonale sinistra (direzione = s, dal basso a dx all'alto a sx) oppure destra (direzione = d, dal basso a sx all'alto a dx) viste dalla pedina p.

La classe **Pedina** fornisce i seguenti metodi :

- **public int colore()** restituisce il colore della pedina.
- **public int riga()** restituisce la riga in cui la pendina si trova nel momento della chiamata del metodo.
- **public int colonna()** restituisce la colonna in cui la pedina si trova nel momento della chiamata del metodo.

- a. Si scriva la post-condizione del metodo pubblico **nuovaPartita()** della classe **Scacchiera**

il metodo numPezzi è ridondante rispetto al metodo size() applicato al risultato di pedine().

//@ requires true;

```
//@ ensures
// presi i due colori
(\forallall int i; 0 <= i <= 1;
 // il numero di pezzi è uguale a 12
 numPezzi(i) == 12 &&
 // tutte le pedine
 (\forallall int j; 0 <= j < pedine(i).size());
 // stanno sulle caselle nere
 pedine(i).get(j).riga() + pedine(i).get(j).colonna() % 2 == 0 &&
 // e stanno sulle righe previste
 pedine(i).get(j).colore() == 0 ? 6 <= pedine(i).get(j).riga() <= 8 : 1 <= pedine(i).get(j).riga() + <= 3))
```

- b. Si scriva la pre- e post-condizione del metodo **public bool mossaPossibile(Pedina p, char direzione)** della classe **Scacchiera**. Ogni pedina ha due sole possibilità di movimento di una posizione in diagonale: le pedine bianche si possono muovere di una casella verso l'alto a sinistra (direzione vale s) oppure a destra (direzione vale d), quelle nere verso il basso a sinistra o a destra. La casella di destinazione deve essere libera da pedine. Ad esempio, se una pedina bianca in posizione 6:6 volesse muoversi in direzione s (sinistra), questo significherebbe spostarla in posizione 5:5 a patto che la casella sia libera. Il metodo ritorna true se la mossa può essere effettuata correttamente, false altrimenti.

```
// se p non esiste o se non ci sono pedine dell'altro colore, non ha senso porsi il problema
//@ requires p!=null && numPedine(1-p.colore()) > 0 &&
```

```
1<=p.riga()<=8 && 1<=p.colonna()<=8 && 0<=p.colore()<=1 &&
(direzione == 's' || direzione == 'd')
```

```
// ipotizzando che il metodo diagonale() restituisca sempre la diagonale,
// al più con elementi null se la cella è libera
// e che che il primo elemento restituito da diagonale sia
// la pedina stessa
```

```
//@ ensures diagonale(p, direzione).size() > 1 &&
diagonale(p, direzione).get(1)==null ? \result == true: \result == false
```

- c. Si scriva la pre- e post-condizione del metodo **void muovi(Pedina p, char direzione)** della classe **Scacchiera**, sapendo la mossa deve essere possibile e l'invocazione del metodo modifica la posizione di p che è anche l'unica pedina che cambia posizione.

```
//@ requires mossaPossibile(p, direzione);
```

```
//@ ensures p.colore()==0? (p.riga()==\old(p.riga))-1 &&
direzione=='s'? p.colonna()==\old(p.colonna)-1:
p.colonna()==\old(p.colonna)+1:
(p.riga()==\old(p.riga))+1 &&
direzione=='s'? p.colonna()==\old(p.colonna)+1:
p.colonna()==\old(p.colonna)-1) &&
```

```
// p è l'unica pedina che si muove
```

```
(forall int i; 0<=i<=1;
(forall int j; 0<=j<pedine(i).size(); pedine(i).get(j) != p ->
pedine(i).get(j).riga() == \old(pedine(i).get(j).riga()) &&
pedine(i).get(j).colonna() == \old(pedine(i).get(j).colonna())))
```

- d. Si vuole specificare la proprietà che il numero complessivo delle pedine sulla scacchiera non è crescente. Di che tipo di proprietà si tratta? Come sarebbe possibile specificarla in JML?

La proprietà è una tipica proprietà evolutiva. JML non offre un costrutto specifico. Occorre quindi aggiungere alla postcondizione di ogni metodo che modifica lo stato della scacchiera una clausola `numPezzi(0)<=\old(numPezzi(0)) && numPezzi(1)<=\old(numPezzi(1))`.

e. Si scriva l'invariante privato della classe **Scacchiera**, ipotizzando il REP indicato qui sotto e ricordando che le pedine possono stare solo sulle caselle nere e che le pedine di ogni colore non possono essere più di 12:

```
private Pedina[8][8] griglia;

//@ private invariant (\forall int k, 0<=k<=1,
//  (\sum int i; 0<=i<8;
//   (\text{num\_of int j; } 0<=j<8; griglia[i][j].colore() == k) <= 12) &&
// qui avremmo anche potuto usare anche il metodo pubblico numPezzi()

//  (\forall int i; 0<=i<8;
//   (\forall int j; 0<=j<8; griglia[i][j] != null -> (i+j)%2 == 1))
```

Esercizio 3 (punti 7)

Si consideri la classe **Poly**, definita a lezione. Si supponga che in essa siano definiti anche i metodi **valuta()** e **soluzioneMinima()**. Il primo metodo calcola il valore float del polinomio corrispondente al parametro. Ad esempio, **valuta(3.0)**, quando è chiamato sul Poly $-2x^2+3x+2$, restituisce -7. Il secondo metodo restituisce la radice reale minima del Poly, se essa esiste, ossia il più piccolo valore reale x' per cui $\text{valuta}(x') == 0$. Per il Poly dell'esempio, **soluzioneMinima()** restituisce $x' = -0.5$ (l'altra soluzione è 2.0). La soluzione calcolata è sempre esatta (nel senso che è calcolata fino alla massima precisione possibile con il tipo float). Per il teorema di Ruffini, è noto che se il grado del polinomio supera quattro, la soluzione non può essere calcolata in modo esatto tramite operazioni razionali ed estrazioni di radice. Pertanto, il metodo è definito solo se il grado del polinomio non supera 4.

```
public /*@ pure */ class Poly {  
    ...  
    //@ensures (*\result è la valutazione del polinomio quando la variabile vale var *);  
    public float valuta(float var);  
  
    //@requires this.deg() <=4;  
    //@ensures valuta(\result) ==0 && !(exists float f; f<\result; valuta(f) ==0);  
    //@signals (NessunaSoluzioneRealeException e) !(exists float f; ; valuta(f) ==0);  
    public float soluzioneMinima() throws NessunaSoluzioneRealeException  
}
```

Un'estensione di Poly calcola la soluzioneMinima anche quando il grado è superiore a 4. In tal caso, dovrà ovviamente ricorrere a tecniche di calcolo numerico. Pertanto, la soluzione sarà approssimata. In particolare, si richiede che la valutazione del risultato differisca dallo zero al più di 0.001.

```
public /*@ pure */ class NuovoPoly extends Poly {  
    //@also  
    //@requires true;  
    //@ensures -0.001<=valuta(\result) <=0.001 &&  
    //@ !(exists float f; f < \result - 0.001; -0.001<=valuta(f) <=0.001);  
    public float soluzioneMinima() throws NessunaSoluzioneRealeException;  
}
```

La classe NuovoPoly verifica il principio di sostituzione? Mostrare in dettaglio come mai ciò avviene, costruendo le pre e post condizioni complete per il metodo **soluzioneMinima()** in base alle regole del linguaggio JML. La risposta sarà considerata valida solo se correttamente motivata.

La regola delle proprietà è ovviamente verificata (in quanto il metodo **soluzioneMinima()** è solo un osservatore, e non può quindi influenzare proprietà dello stato astratto).

Apparentemente, la postcondizione di **NuovoPoly.soluzioneMinima()** è più debole della postcondizione di **Poly.soluzioneMinima()**. Tuttavia, la regola dei metodi è verificata, perché la nuova postcondizione si applica solo ai casi non previsti dalla precondizione originale (cioè solo per $this.deg() > 4$). Per il caso con $this.deg() \leq 4$, è sufficiente che la nuova postcondizione sia non contraddittoria con la postcondizione originale. In dettaglio, la regola JML delle specifiche ereditate costruisce la seguente specifica complessiva per il metodo **NuovoPoly.soluzioneMinima()**

```
//@requires this.deg()<=4 || true  
//@ensures (this.deg()<=4 ==> valuta(\result) ==0 && !(exists float f; f<\result; valuta(f) ==0))  
//@     && (true ==> -0.001<=valuta(\result) <=0.001 &&  
//@             !(exists float f; f<\result-0.001; -0.001<=valuta(f) <=0.001);
```

che può essere riscritta come:

```
//@requires true  
//@ensures (this.deg()<=4 ==> valuta(\result) ==0 && !(exists float f; f<\result; valuta(f) ==0)) &&  
//@     -0.001<=valuta(\result) <=0.001 && !(exists float f; f<\result-0.001; -0.001<=valuta(f
```

Pertanto, se il grado è minore o uguale a 4, la postcondizione garantisce che la soluzione (se esiste) sia esatta (e quindi a fortiori verifichi anche il vincolo sull'"approssimazione"); negli altri casi, la soluzione sarà solo approssimata. La regola dei metodi è pertanto verificata.

Ingegneria del Software – a.a. 2009/10

Appello del 10 Settembre 2010

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h10m.
6. Punteggio totale a disposizione: 100 punti nominali. La sufficienza si raggiunge indicativamente con 60 punti.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
.....

Esercizio 1 (punti 20)

Si consideri l'astrazione procedurale **raddoppiaArray**. L'astrazione ha come parametro un array x tale che: gli elementi di x sono Integer, x contiene almeno un elemento, e gli elementi di x sono tutti diversi da null. L'astrazione restituisce un array di Integer, di dimensioni raddoppiate rispetto a quelle di x, così costituito:

- 1) nella metà inferiore dell'array ci sono gli elementi dell'array x (nello stesso ordine);
- 2) nella metà superiore ci sono solo dei null.

Ad es., se x è [1 2 3 4 5], e si cerca di inserire un nuovo elemento, allora l'invocazione **raddoppiaArray(x)** restituisce un nuovo array di dimensione 10, così fatto:

[1 2 3 4 5 null null null null null]

a- Si scrivano la pre- e post-condizione del metodo, evidenziando tutti gli elementi significativi

```
public static Integer[] raddoppiaArray(Integer[] x)
```

b- Si identifichino le eccezioni necessarie per rendere totale il metodo e si scrivano le opportune modifiche alla specifica di cui al punto (a).

Esercizio 2 (punti 60)

Si consideri la seguente specifica di una classe CodaDoppia.

Una CodaDoppia è una coda in cui gli elementi possono essere inseriti o eliminati in due posizioni, la testa e la coda.

```
public class CodaDoppiaInt{
    /*OVERVIEW: Collezione mutabile di oggetti, di tipo Integer, organizzati in una sequenza.
     * Un oggetto tipico di dimensione n è x1 x2 ... xn, in cui x1 è la testa e xn è la coda.
     * L'estrazione e l'inserimento di elementi può avvenire sia in testa che in coda.*/
    //@ensures (*inizializza this alla CodaDoppiaInt vuota*).
    public CodaDoppiaInt(){}
    //osservatori puri:
    //@ensures (* \result è il numero di elementi di this *)
    public /*@ pure */ int size()
    //@ensures (* \result è un ArrayList contenente tutti e soli gli elementi di this *);
    public /*@ pure */ ArrayList<Integer> elementi()
    //@requires size()>0;
    //@ensures elementi().get(0).equals(\result);
    public /*@ pure */ Integer peekFirst()
    //@requires size()>0;
    //@ensures elementi().lastElement().equals(\result);
    public /*@ pure */ Integer peekLast()
    //mutators:
    //@ensures (* aggiunge x in prima posizione *);
    public void insertFirst(Integer x)
    //@requires size()>0;
    //@ ensures (* elimina e restituisce l'elemento in prima posizione in this *)
    public Integer extractFirst()
    }
    //@ensures (* aggiunge x in ultima posizione *);
    public void insertLast(Integer x)
    //@requires size()>0;
    //@ ensures (* elimina e restituisce l'elemento in ultima posizione in this *)
    public Integer extractLast()
}
```

Domanda A) Specificare in JML le postcondizioni dei metodi insertFirst e extractLast

Si consideri ora la seguente classe CodaInt, che ha gli stessi metodi di CodaDoppiaInt, con la stessa specifica, *ad esclusione* di: peekLast, insertFirst e extractLast, che non vi compaiono.

Per semplicità, non riportiamo la specifica dei metodi di CodaInt, in quanto non modificata rispetto a CodaDoppiaInt.

```
public class CodaInt{  
    /* OVERVIEW: Collezione mutabile di oggetti, di tipo Integer, organizzati in una sequenza.  
     * Un oggetto tipico di dimensione n è x1 x2 ... xn, in cui x1 è la testa e xn è la coda. L'estrazione di  
     * elementi avviene in testa, mentre l'inserimento avviene in fondo alla coda.*/  
    public CodaInt ()  
    public /*@ pure @*/ int size()  
    public /*@ pure @*/ ArrayList<Integer> elementi()  
    public /*@ pure @*/ Integer peekFirst()  
    public void insertLast(Integer x)  
    public Integer extractFirst()  
}
```

Domanda B1): E' possibile definire CodaInt come erede di CodaDoppiaInt, rispettando il principio di sostituzione? Giustificare la risposta.

Domanda B2): E' possibile definire CodaDoppiaInt come erede di CodaInt, rispettando il principio di sostituzione? Giustificare la risposta.

Si consideri la classe **CodaInt** della Domanda B. È data un'implementazione realizzata nel modo seguente. Gli elementi sono contenuti in un array *elems*, di dimensione minima 5. Un indice intero *primo* indica la testa e un indice intero *ultimo* indica il fondo della coda memorizzata in *elems*. Il rep quindi è:

```
private Integer[] elems;  
private int primo;  
private int ultimo;
```

CodaInt è implementata inserendo i nuovi elementi nella posizione *ultimo*, ed estraendo gli elementi dalla posizione *primo+1*, aggiornando i valori di *primo* e di *ultimo*.

Quando l'array *elems* non è più sufficiente a contenere gli elementi, si invoca il metodo statico *raddoppiaArray* dell'esercizio 1, che raddoppia la dimensione di *elems*, copiando inoltre gli elementi esistenti nella metà inferiore dell'array e riempendo il resto con null.

E' data la seguente implementazione del costruttore e dei metodi *insertLast* e *extractFirst*:

```
public CodaInt() {  
    elems = new Integer[5];  
    primo = -1;  
    ultimo=0;  
}  
public void insertLast (Integer x) {  
    if (ultimo >= elems.length)  
        elems = raddoppiaArray(elems);  
    elems[ultimo]= x;  
    ultimo++;  
}  
public Integer extractFirst() {  
    if (size()>0) {  
        primo++;  
        return elems[primo];  
    }  
}
```

Domanda C) Alla luce della descrizione informale e del codice presentati, scrivere l'invariante di rappresentazione (ossia, privato) di CodaInt.

Domanda D) Descrivere, nel modo che si ritiene più opportuno, la funzione di astrazione di Codalnt.

Domanda E) Scrivere l'implementazione del metodo `size()`. Argomentare che tale implementazione è corretta rispetto alla sua specifica.

Domanda F) (Facoltativa): l'implementazione qui riportata per Codalnt, pur se tecnicamente corretta, soffre di un serio problema, che la rende inadatta a essere utilizzata in pratica. Quale?

Esercizio 3 (punti 20)

Implementare il seguente metodo iteratore `els()` della classe `Codalnt` dell'Es. 2. Si suggerisce di descrivere la struttura delle eventuali classi aggiuntive rispetto a `Codalnt`.

//@ensures (restituisce un iteratore agli elementi di this, rispettandone l'ordine di inserimento*);*
public Iterator<Integer> els()

Ingegneria del Software – a.a. 2004/05

Appello dell'8 febbraio 2006

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 26/30.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 10)

Il metodo statico `controlla()` riceve in ingresso un array `nums` e un intero `n` positivo e restituisce un valore booleano. L'array `nums`, di lunghezza non precisata, contiene dei numeri interi, per ipotesi tutti diversi tra loro. Il metodo restituisce true se ognuno dei primi `n` numeri dell'array è più piccolo di ognuno dei numeri che occupano le posizioni da `n` alla dimensione dell'array. L'array non viene modificato.

b - Si fornisca la specifica JML del metodo. Si rammenta che la specifica deve considerare i casi in cui l'array sia nullo o di lunghezza minore di `n` e quello in cui gli elementi non siano tutti distinti fra loro.

Soluzione

```
//@assignable nothing;
//@ requires nums!=null && n<nums.length &&
    (\forall int i; 0<=i< nums.length;
     (\forall int j; i<j<nums.length; nums[i] != nums[j]));
//@ ensures \result ==
    (\forall int i; 0<=i<n;
     (\forall int j; n<=j<nums.length; nums[i]<nums[j]));
```

```
public static boolean controlla(int [] nums, int n)
```

b - Si trasformi la specifica JML del punto precedente sostituendo una a scelta delle precondizioni nel lancio di opportune eccezioni.

Soluzione

Data una opportuna classe eccezione `IndiceFuoriDaiLimitiException`:

```
//@assignable nothing;
//@ requires nums!=null&&
    (\forall int i; 0<=i< nums.length;
     (\forall int j; i<j<nums.length; nums[i] != nums[j]));
//@ ensures \result ==
    (\forall int i; 0<=i<n;
     (\forall int j; n<=j<nums.length; nums[i]<nums[j]));
//@signals (IndiceFuoriDaiLimitiException e) n>=nums.length;
public static boolean controlla(int [] nums, int n) throws IndiceFuoriDaiLimitiException
```

Esercizio 2 (punti 9)

Si consideri la classe **InsPunti**, classe immutabile che rappresenta un insieme di punti (senza duplicazioni) in un piano. Ogni punto dell'insieme è univocamente individuato da un indice intero, compreso fra 0 e la cardinalità dell'insieme (meno uno). La classe fornisce, oltre a costruttori e produttori che non sono qui riportati, le primitive seguenti:

```
class InsPunti {  
    ....  
    //restituisce la cardinalità dell'insieme  
    int numPunti() {}  
  
    //restituisce la distanza tra due punti dell'insieme, individuati dai loro indici i e j,  
    //con  $0 \leq i, j < \text{numPunti}()$   
    float distanza(int i, int j){}  
  
    //restituisce il diametro dell'insieme, cioè la distanza massima tra coppie di punti dell'insieme  
    float diametro(){}  
    ....  
}
```

a – Si fornisca la specifica JML (pre- e post-condizione) del metodo diametro()

Soluzione

```
//@ assignable \nothing  
//@ requires this.numPunti() >= 2;  
//@ ensures \exists int i; 0<=i<this.numPunti();  
    (\exists int j; 0<j<this.numPunti();  
        \result == this.distanza(i,j) &&  
        (\forallall int x; 0<=x<this.numPunti();  
            (\forallall int y; 0<y<this.numPunti(); \result >= this.distanza(x, y)  
        ))))  
float diametro(){}  
)) )
```

b – Il REP per la classe viene scelto come segue:

Un array di float `coord`, con $2 \cdot n$ componenti (l'insieme contiene n punti), in cui viene messa la rappresentazione polare di ogni punto (modulo, angolo). I primi due elementi di `coord` memorizzano rispettivamente modulo e angolo del primo punto, il terzo e il quarto modulo e angolo del secondo punto, etc.

Scrivere in JML l'invariante di rappresentazione, considerando che l'insieme può essere vuoto ma non può contenere punti duplicati.

Soluzione

```
//@ private invariant coord != null &&
//@ (\exists int n ; n>=0 ; 2*n == coord.length && coord ha un numero pari di elementi
//@   (\forall int i; 0<=i<n; coord[2*i]>=0) && condizione sul modulo di tutti i punti
//@   (\forall int i; 0<=i<n; 0<coord[2*i+1]<=6.28 )&& condizione sull'angolo di tutti i
//@   punti
//@   (\forall int x; 0<=x<n;
//@     (\forall int y; 0<=y<n;
//@       x != y ==> coord[2*x] != coord[2*y] || coord[2*x+1] != coord[2*y+1] ) ));
```

c – Si fornisca la specifica JML del metodo boolean triangolo(int i, int j, int k){} per sapere se i tre punti, identificati dai loro indici e ipotizzati essere fra loro distinti, possono essere i vertici di un triangolo non degenere. A tal fine si ipotizzi che un triangolo sia non degenere se e solo se la lunghezza di ognuno dei suoi lati è minore della somma delle lunghezze degli altri due lati.

Soluzione

Per verificare che i tre punti distinti non siano allineati, basta imporre le tre disegualanze triangolari.

```
//@ assignable \nothing
//@ requires this.numPunti() >= 3 && 0<=i,j,k <numPunti() && i != j && j != k && k != i;
//@ ensures \result == (distanza(i,j) < distanza(i, k) + distanza(k, j) &&
//@               distanza(j,k) < distanza(j, i) + distanza(i, k) &&
//@               distanza(k,i) < distanza(k, j) + distanza(j, i) );
boolean triangolo(int i, int j, int k){}
```

Esercizio 3 (punti 8)

Disegnare un diagramma delle classi che rappresenti un *grafo gerarchico*.

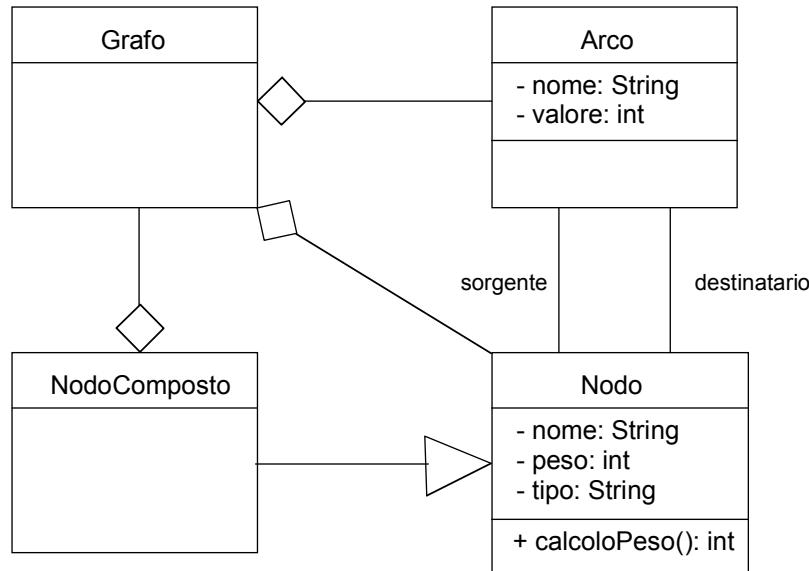
Un grafo gerarchico è un insieme di nodi e archi opportunamente connessi, in cui ogni arco connette esattamente due nodi, e ogni nodo può essere “semplice” o “composto”. Un nodo semplice non è ulteriormente decomposto, mentre un nodo composto contiene al suo interno ancora un grafo gerarchico.

Pertanto, la relazione di contenimento è iterata gerarchicamente: un nodo composto, parte di un grafo, contiene un altro grafo, i cui nodi composti contengono altri grafi e così via.

Ogni arco è orientato e collega un nodo, detto *sorgente* dell'arco, a un nodo detto *destinazione*. Un arco è caratterizzato inoltre da un nome e un colore.

Ogni nodo è identificato da un nome (univoco al suo livello gerarchico) e da un peso. Se il nodo è composto, il suo peso è dato dalla somma dei pesi dei nodi in esso contenuti, ad ogni livello. I nodi semplici si dividono ulteriormente in: nodi pozzo, nodi sorgente e nodi normali. I primi possono avere solo archi entranti, ma non uscenti; i secondi possono avere solo archi uscenti, ma non entranti, e i terzi non pongono restrizioni.

Soluzione (per brevità, non sono riportate le cardinalità). E' corretto, ma un po' macchinoso, anche definire tre sottoclassi di Nodo (sorgente, pozzo, normale) per rappresentare i vari tipi di nodo, a patto poi di indicare le associazioni corrette con la classe Arco.



Ingegneria del Software – a.a. 2009/10

Appello del 10 Febbraio 2011

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 100 punti nominali. La sufficienza si raggiunge indicativamente con 60 punti.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
.....

Esercizio 1 (punti 20)

Si consideri l'astrazione procedurale **sottraiArray**. L'astrazione ha come parametro due array **x** e **y** di elementi **int**. L'array **x** non è vuoto, e sia **x** che **y** non hanno elementi ripetuti.

L'astrazione restituisce un array di int, contenente gli elementi di **x** (se esistono) che NON sono presenti in **y**, in ordine qualunque.

Ad es., se **x** è [8 2 3 5 1], e **y** = [2 5 8 7], un array restituito è [1 3].

a- Si scrivano la pre- e post-condizione del metodo, evidenziando tutti gli elementi significativi

SOL:

```
/*@pre x!=null && y!=null && x.length>0 &&
    (\forall int i; 0<= i && i< x.length-1; (\forall int j; i<j && j< x.length; x[i]!= x[j])) &&
    (\forall int i; 0<= i && i< y.length-1; (\forall int j; i<j && j< y.length; y[i]!= y[j]));
*/
/*@post (\forall int i; 0\le i && i< \result.length;
    (\exists j; 0<= j && j< x.length; x[j]==\result[i]) &&
    !(\exists j; 0<= j && j< y.length; y[j]==\result[i])) &&
    (\forall int i; 0<= i && i< x.length;
    !(\exists j; 0<= j && j< y.length; y[j]==x[i])) ==>
    (\exists j; 0<= j && j< x.length; x[i]==\result[j])) &&
    (\forall int i; 0<= i && i< \result.length-1; (\forall int j; i<j && j< \result.length;
    \result[i]! = \result[j]))
*/
//@\assignable nothing;
```

```
public static int[] sottraiArray(int[] x, int[]y)
```

b- Si identifichino le eccezioni necessarie per rendere totale il metodo e si descrivano le opportune modifiche alla specifica di cui al punto (a).

SOL

Le eccezioni da lanciare sono **NullPointerException**, **EmptyException** e **DuplicateException**.

La specifica diventa (considerando che **\old(pre) == pre**, in quanto c'e' **\assignable nothing**):

```
//@pre true;
//@post ...post già vista && ...pre già' vista...

//@signals (NullPointerException e) x==null || y ==null;
//@signals (EmptyException e) x.length==0;

/*@signals (DuplicateException e)
    (\exists int i; 0<= i && i< x.length-1; (\exists int j; i<j && j< x.length; x[i]==x[j])) ||
    (\exists int i; 0<= i && i< y.length-1; (\exists int j; i<j && j< y.length; y[i]==y[j]));
*/
//@\assignable nothing;
```

Esercizio 2 (punti 60)

Le classi **Punto2D**, **Segmento** e **Triangolo**, rappresentano le note entità geometriche in uno spazio euclideo bidimensionale.

Le specifiche di **Punto2D** e di **Segmento** sono le seguenti:

```
/*@ pure @*/ /* public class Punto2D {  
    //@ensures (* costruisce un punto di coordinate cartesiane x,y*);  
    public Punto(double x, double y);  
  
    //@ensures (*restituisce la distanza euclidea fra this e p*);  
    public double distanza (Punto2D p);  
  
    //@ensures (*restituisce la coord. x di this *);  
    public double getX();  
  
    //@ensures (*restituisce la coord. x di this *);  
    public double getY();  
}  
  
/*@ pure @*/ public class Segmento {  
    //@ requires f != null && o!=null;  
    //@ensures fine()== f && origine()== o;  
    public Segmento(Punto2D o, Punto2D f);  
  
    //@ensures (* restituisce l'estremo iniziale di this *)  
    public Punto2D origine();  
  
    //@ensures (* restituisce l'estremo finale di this *)  
    public Punto2D fine()  
}
```

- a) Un triangolo è individuato da tre punti non allineati, ossia i vertici. Se i tre punti sono allineati, il triangolo è degenere e non può essere costruito. Si ipotizzi che sia definito un metodo statico (da NON specificare né implementare):

```
public static boolean allineati (Punto2d p1, Punto2D p2, Punto2D p3);
```

che restituisce true se, e solo se, i 3 punti p1,p2 e p3 sono allineati.

La base del **triangolo** è definita, ai nostri scopi, convenzionalmente come il lato *maggior o uguale* agli altri due, il latoMinore è il lato minore o uguale agli altri due, mentre il latoMedio è il rimanente. Il metodo **traslaOrizzontale** incrementa o decrementa le coordinate dei vertici di un valore DeltaX.

Completare negli spazi vuoti la specifica JML della seguente classe **Triangolo**.

```
public class Triangolo {  
//Tipo mutabile  
    //@pre v1!=null && v2!= null && v3!=null  
    //@post !allineati(v1,v2,v3)&&  
    (* this è un triangolo | {base().origine(), base().fine() ,latoMinore().fine()} == {v1,v2, v3}* ) &&  
    base().origine == latoMinore().origine() && base().fine()== latoMedio().origine() &&  
    latoMinore().fine()==latoMedio().fine();  
    @*/  
  
    //@signals (TriangoloDegenerException e) allineati(v1,v2,v3);  
    public Triangolo (Punto2D v1, Punto2D v2, Punto2D v3) throws TriangoloDegenerException;  
  
    //@post \result.lunghezza() >= latoMedio().lunghezza()  
    public /*@ pure @*/ Segmento base ();
```

```

@@post \result.lunghezza() <= latoMedio().lunghezza() && ! \result.equals(base());
public /*@ pure @*/ Segmento latoMinore();

@@post !\result.equals(base()) && !\result.equals(latoMinore());
public /*@ pure @*/ Segmento latoMedio();

/*@ base().origine().getX()==\old(base().origine().getX())+DeltaX &&
base().fine().getX()==\old(base().fine().getX())+DeltaX &&
latoMinore().fine().getX()==\old(latoMinore().fine().getX())+DeltaX &&
base().origine().getY()==\old(base().origine().getY())&&
base().fine().getY()==\old(base().fine().getY())&&
latoMinore().fine().getY()==\old(latoMinore().fine().getY()) &&
latoMinore().fine()==latoMedio().fine() && latoMedio().origine()==base().fine() &&
LatoMinore().origine()==base().origine();
*/
public traslaOrizzontale(double DeltaX);
}

```

NB: Nella postcondizione del costruttore, si ipotizza per semplicità che l'origine della base coincida con l'origine del latoMinore e che la fine della base coincida con l'origine del lato medio (e che quindi la fine del latoMinore coincide con la fine del latoMedio). Questa ipotesi non viola la specifica data informalmente, in quanto, dati i due punti di cui è costituito un segmento, la scelta di quale sia l'origine e quale sia la fine è arbitraria.

Nelle postcondizioni di latoMinore e di latoMedio,asseriamo anche che i segmenti sono fra loro tutti diversi (altrimenti in caso di triangolo isoscele o equilatero potrebbe essere restituito lo stesso segmento ad esempio come base e latoMedio). In traslaOrizzontale, la postcondizione assicura che tutti i tre vertici del triangolo sono traslati correttamente, essendo certi che i tre vertici del triangolo sono base.origine, base.fine e latoMinore.fine.

- b) Scrivere la proprietà che la base ha lunghezza maggiore o uguale a quella degli altri lati usando un *invariante pubblico*.

```
public invariant base().lunghezza()>=latoMinore().lunghezza() &&
base().lunghezza()>=latoMedio().lunghezza();
```

- c) Si consideri un'estensione mutabile di Triangolo, detta *TriangoloBaseVariabile*, che rispetto a a Triangolo definisce un nuovo metodo *cambiaBase()*.

Il metodo cambiaBase riceve come parametro un segmento *nuovaBase* e aggiorna il triangolo adottando come nuova base *nuovaBase*, purché questa sia maggiore o uguale degli altri lati (e il triangolo risultante non sia degenere).

Specificare il metodo cambiaBase.

```

@@pre nuovaBase.lunghezza()>=latoMedio().lunghezza();
/*@post (!allineati(nuovaBase.origine(); nuovaBase.fine(); \old(latoMinore().fine())) &&
( base().origine() ==nuovaBase.origine() && base().fine()== nuovaBase.fine() ||
base().fine() ==nuovaBase.origine() && base().origine()== nuovaBase.fine()) &&
latoMinore().fine() == \old(latoMinore().fine()) &&
base().origine() == latoMinore().origine() && base().fine()== latoMedio().origine() &&
latoMinore().fine()==latoMedio().fine();
*/
/*@signals (TriangoloDegenereException e)
allineati(nuovaBase.origine(); nuovaBase.fine(); \old(latoMinore().fine())));
*/
public void cambiaBase(Segmento nuovaBase) throws TriangoloDegenereException;
```

- d) **TriangoloBaseVariabile verifica il principio di sostituzione di Liskov rispetto a Triangolo?**
Giustificare la risposta.

SI. L'unico modo in cui l'aggiunta di un metodo può invalidare il principio di sostituzione è tramite la violazione regola delle proprietà. Le proprietà di un oggetto Triangolo sono: 1) rappresenta un triangolo non degenere; 2) la base è maggiore o uguale agli altri due lati; 3) il latoMinore è minore o uguale agli altri due lati.

Le proprietà 1, 2 e 3 sono ancora garantite dal metodo `cambiaBase()`. Si osservi però che una specifica meno "attenta" violerebbe facilmente la proprietà 3 (è indispensabile che origine fine della base possano essere scambiate, se necessario, in modo da garantire la proprietà che l'origine della base sia anche l'origine del lato minore).

- e) Si consideri una classe `TriangoloEquilatero`, definita come un `Triangolo` in cui però tutti i lati sono sempre uguali.

Si dica se il principio di sostituzione di Liskov può essere verificato costruendo `TriangoloEquilatero` come estensione di `Triangolo`.

Si, è possibile in quanto l'unico metodo di `TriangoloEquilatero` che ha una specifica differente da quella corrispondente di `Triangolo` è il costruttore, che tuttavia non è ereditato. Quindi la regola dei metodi è verificata. La regola delle proprietà è pure verificata, in quanto anche un `TriangoloEquilatero` è un `Triangolo` non degenere, la cui base è non minore degli altri lati, e il latoMinore è non maggiore degli altri lati.

Cosa succederebbe se `TriangoloEquilatero` fosse definita invece come estensione di `TriangoloBaseVariabile`? Giustificare la risposta.

Non si tratta di violazione del principio di sostituzione, ma di violazione delle proprietà fondamentali di una classe.
infatti, il principio di sostituzione resterebbe verificato, ma una chiamata del metodo `cambiaBase` potrebbe invalidare la proprietà di `TriangoloEquilatero` che tutti i lati sono uguali. Pertanto non è possibile definire `TriangoloEquilatero` come estensione di `TriangoloBaseVariabile`, in quanto l'oggetto risultante potrebbe evolvere violando la proprietà di essere equilatero.

Esercizio 3 (punti 20)

Sia dato il seguente rep per la classe TriangoloBaseVariabile

private Segmento[3] lati;

Il rep è definito in modo che *lato[0]*, *lato[1]* e *lato[2]* memorizzano, in un ordine qualunque, come *segmenti* i tre lati del triangolo, indipendentemente dall'essere questi la base, il latoMinore o il latoMedio.

Scrivere l'invariante di rappresentazione della classe.

private invariant (forall int i; 0<=i && i<=2; lato[i]!=null) &&
(\exists int i; 0<=i && i<=2;
\(\exists\) int j; && 0<=j && j<=2;
\(\exists\) int k; 0<=i && i<=2;
i!=j && i!=k && j!=k &&
lato[i].origine() == lato[j].origine() &&
lato[i].fine() == lato[k].origine() &&
lato[j].fine() == lato[k].fine()) &&
!allineati(lato[i].origine(), lato[i].fine(), lato[j].fine()) &&
lato[i].lunghezza()>=lato[k].lunghezza() &&
lato[k].lunghezza()>= lato[j].lunghezza);

NB: i,j,k rappresentano rispettivamente gli indici della base, del lato minore e del lato medio.

In base all'invariante e alle specifiche date, implementare il metodo cambiaBase().

```
int indBase;
if (lato[0].lunghezza() >= lato[1].lunghezza())
    if (lato[0].lunghezza()>= lato[2].lunghezza())
        indBase = 0;
    else indBase=2;
else    if (lato[1].lunghezza()>=lato[2].lunghezza())
        indBase=1;
    else indBase=2;
lato[indBase]= nuovaBase;
lato[(indBase+1) % 3].origine()=nuovaBase.origine();
lato[(indBase+2) % 3].origine()=nuovaBase.fine();
```

26 giugno 2019



Politecnico di Milano
Anno accademico 2018-2019

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>	
Nome:	Matricola:	
Sezione (segnarne una): <input type="checkbox"/> Cugola	<input type="checkbox"/> Margara	<input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica della classe `RecentContacts` che modella la lista dei contatti di una chat ordinati sulla base dell'ultima interazione (dalla più alla meno recente).

```
public class RecentContacts {
    // Crea una lista di contatti vuota.
    public RecentContacts() {
        // Aggiunge un nuovo contatto in fondo alla lista.
        // Lancia l'eccezione ExistingContactException
        // se il contatto è già in lista.
        public void add(String contact) throws ExistingContactException;
    }

    // Rimuove il contatto dalla lista.
    // Lancia l'eccezione UnknownContactException se il contatto non è in lista.
    public void remove(String contact) throws UnknownContactException;

    // Metodo invocato quando avviene un'interazione
    // (invio o ricezione di un messaggio) con il contatto contact.
    // Lancia l'eccezione UnknownContactException se il contatto non è in lista.
    public void notifyInteraction(String contact) throws UnknownContactException;

    // Restituisce il contatto in posizione pos.
    public /*@ pure @*/ String get(int pos);

    // Restituisce il numero di contatti presenti in lista.
    public /*@ pure @*/ int size();
}
```

Domanda a)

Si specificino in JML i metodi `add`, `remove` e `notifyInteraction`.

Soluzione

```
//@requires contact != null
//@ensures \forall(int i; 0<=i<\old(size()); !contact.equals(\old(get(i)))) &&
//@           \forall(int i; 0<=i<\old(size()); get(i).equals(\old(get(i)))) &&
//@           get(size()-1).equals(contact) &&
//@           size() == \old(size())+1
//@signals(ExistingContactException)
//@           \exists(int i; 0<=i<\old(size()); contact.equals(\old(get(i)))) &&
//@           \forall(int i; 0<=i<\old(size()); get(i).equals(\old(get(i)))) &&
//@           size() == \old(size())
public void add(String contact) throws ExistingContactException;

//@requires contact != null
//@ensures \exists(int i; 0<=i<\old(size()));
//@           contact.equals(\old(get(i))) &&
//@           \forall(int j; 0<=j<i; get(j).equals(\old(get(j)))) &&
//@           \forall(int j; i<=j<size(); get(j).equals(\old(get(j+1)))) &&
//@           size() == \old(size())-1
//@signals(UnknownContactException)
//@           \forall(int i; 0<=i<\old(size()); !contact.equals(\old(get(i)))) &&
//@           \forall(int i; 0<=i<\old(size()); get(i).equals(\old(get(i)))) &&
//@           size() == \old(size())
public void remove(String contact) throws UnknownContactException;
```

```

//@requires contact != null
//@ensures \exists(int i; 0<=i<\old(size()));
//@      contact.equals(\old(get(i))) &&
//@      get(0).equals(contact) &&
//@      \forall(int j; 1<=j<=i; get(j).equals(\old(get(j-1)))) &&
//@      \forall(int j; i<j<size(); get(j).equals(\old(get(j))))) &&
//@      size() == \old(size())
//@signals(UnknownContactException)
//@      \forall(int i; 0<=i<\old(size()); !contact.equals(\old(get(i)))) &&
//@      \forall(int i; 0<=i<\old(size()); get(i).equals(\old(get(i)))) &&
//@      size() == \old(size())
public void notifyInteraction(String contact) throws UnknownContactException;

```

Domanda b)

Si specifichi in JML l'invariante pubblico del tipo `RecentContacts`. Esistono proprietà rilevanti per la classe che non si possono esprimere con un invariante?

Soluzione

```

//@public invariant
//@ size() >=0 &&
//@ \forall(int i; 0<=i<size(); get(i)!=null) &&
//@ \forall(int i; 0<=i<size());
//@   \forall(int j; 0<=j<size() && i!=j; !get(i).equals(get(j)))

```

Non è possibile esprimere proprietà evolutive, in particolare non si può affermare che l'ordine con cui si trovano i contatti riflette l'ordine di invocazione dei metodi `add`, `remove` e `notifyInteraction`. Tale proprietà viene comunque garantita dalle post-condizioni dei modificatori.

Domanda c)

Si consideri il tipo `Contacts`, che mantiene la lista dei contatti senza un ordine preciso (le postcondizioni dei modificatori non specificano un ordine preciso per gli elementi). Il tipo `Contacts` può essere sottotipo di `RecentContacts` secondo il principio di sostituibilità di Liskov? È possibile il contrario? Motivare le proprie risposte.

Soluzione

Il tipo `Contacts` non può essere sottotipo di `RecentContacts` perché non rispetterebbe il vincolo di ordinamento. Ad esempio, questo indebolirebbe le post-condizioni di `add` e `notifyInteraction`.

Il tipo `RecentContacts` può invece essere sottotipo di `Contacts`, in quanto l'ordine cronologico di interazione rappresenta un caso particolare dell'assenza di ordine (ovvero di ordinamento qualsiasi, non deterministico).

Esercizio 2

Si consideri la seguente classe `LongSequence` che realizza una sequenza di valori `long` positivi. Ogni elemento della sequenza è inizialmente “vuoto”, ovvero posto a `-1`. I metodi `get` e `set` consentono di leggere (rimuovendo) o scrivere un valore in una certa posizione della sequenza, sospendendo il chiamante se la posizione è vuota (o piena nel caso del `set`). Il metodo `getSize` restituisce la capacità della sequenza (ovvero il numero di elementi che può contenere).

```

public class LongSequence {
    private long[] data;
    public LongSequence(int size) {

```

```

        data = new long[size];
        for(int i=0; i<size; i++) data[i]=-1;
    }
    public long get(int pos) throws InterruptedException {
        long result;
        synchronized(data) {
            while(data[pos]<0) wait();
            result = data[pos];
            data[pos] = -1;
            notifyAll();
        }
        return result;
    }
    public void set(int pos, long val) throws InterruptedException {
        synchronized(data) {
            while(data[pos]>=0) wait();
            data[pos] = val;
            notifyAll();
        }
    }
    public int getSize() {
        return data.length;
    }
}

```

Domanda a)

La sincronizzazione è corretta per tutti i metodi? In caso affermativo motivare la risposta, in caso negativo indicare come andrebbe modificato il codice per avere una sincronizzazione corretta.

Soluzione

I metodi `get` e `set` correttamente acquisiscono il lock sull'oggetto `data` prima di modificarlo, ma invocano la `wait` e `notifyAll` su `this` invece che su `data` stesso. Occorre quindi cambiare le istruzioni `wait()` in `data.wait()` e analogamente cambiare `notifyAll()` in `data.notifyAll()`.

Si noti che il metodo `getSize` non richiede sincronizzazione perchè la lunghezza dell'array `data` è un valore immutabile.

Domanda b)

Si aggiunga il metodo `public void setAllAsync(long val)` alla classe `LongSequence`. Il metodo deve impostare tutti i valori della sequenza a `val` indipendentemente dal precedente stato dell'elemento (pieno o vuoto) e deve farlo in maniera asincrona rispetto al chiamante (cioè usando un thread separato). Si usino solo i meccanismi base della sincronizzazione di Java senza sfruttare classi di libreria.

Soluzione

```

public void setAllAsync(long val) {
    new Thread() {
        public void run() {
            synchronized(data) {
                for(int i=0; i<data.length; i++) data[i]=val;
                data.notifyAll();
            }
        }
    }
}

```

```

        }
    }.start();
}

```

Esercizio 3

Si consideri il metodo `findMyIntegers`, non completamente definito, della seguente classe `Exam`:

```

public class Exam {
    public static List<Integer> findMyIntegers(List<Integer> aList, Predicate<Integer> aPredicate) {
        BiFunction<List<Integer>, Predicate<Integer>, List<Integer>> onlyMine;
        onlyMine = (myListOfInt, myPredicate) -> {
            .....
            .....
        };
        return onlyMine.apply(aList, aPredicate);
    }

    public static void main(String[] s) {
        Predicate<Integer> predicate = x -> x % 2 == 0;
        List<Integer> startingList = Arrays.asList(1, 7, 4, 6, 8, 9);
        List<Integer> finalList = findMyIntegers(startingList, predicate);
        System.out.println(finalList);
    }
}

```

Il metodo applica il predicato `aPredicate`, definito con l'interfaccia funzionale `Predicate<Integer>`, al parametro `aList`, restituendone, come una nuova lista, tutti e soli i valori che soddisfano il predicato stesso.

A tale scopo, il codice utilizza l'interfaccia funzionale `BiFunction` per definire una funzione che riceve come argomenti una lista e un `Predicate` di `Integer`, restituendo una lista.

La definizione delle due interfacce funzionali viene ricordata di seguito:

```

@FunctionalInterface
public interface BiFunction<T, U, R> {
    // Apply this function to two arguments;
    R apply(T t, U u);
}

@FunctionalInterface
public interface Predicate<T> {
    // Evaluates this predicate on the given argument:
    boolean test(T t);
    ...
}

```

Completare il codice del metodo usando preferibilmente un approccio funzionale. Indicare inoltre il risultato dell'esecuzione del metodo `main` interno alla classe `Exam`.

Soluzione

Basta inserire al posto puntini la seguente riga:

```
return myListOfInt.stream().filter(myPredicate).collect(Collectors.toList());
```

Il `main` stampa 4, 6, 8.

Esercizio 4

Si consideri il seguente metodo Java:

```
public static int m(int arr[]) {  
    int sum = 0;  
    if (arr == null || arr.length == 0) {  
        sum = -1;  
    } else {  
        for (int i = 0; i < arr.length % 2; i++) {  
            if (arr[i] == i) {  
                sum += i;  
            }  
        }  
    }  
    return sum;  
}
```

Domanda a)

Si fornisca un insieme di casi di test per la copertura delle decisioni (branch/edge coverage).

Soluzione

Occorre un caso in cui si entra nel primo ramo `if` (array nullo o vuoto). Occorre valutare positivamente e negativamente la condizione del ciclo `for` (basterebbe un caso). Occorre valutare positivamente e negativamente la condizione del secondo `if` (servono due casi separati in quanto nel ciclo `for` si entra al massimo una volta).

```
arr = null,  
arr = [0], // Entra una volta nel for ed esegue if interno  
arr = [1] // Entra una volta nel for e non esegue if interno
```

Domanda b)

Si fornisca un insieme di casi di test per la copertura dei cammini.

Soluzione

In caso di array non nullo e non vuoto, il codice entra nel ciclo `for` nessuna volta (quando arr ha dimensione pari) o una volta (quando array ha dimensione dispari). Per coprire tutti i cammini occorre quindi avere un caso di test che non entri mai nel ciclo, uno che ci entri una volta ed esegua il ramo `if` e uno che ci entri una volta e non esegua il ramo `if`. In aggiunta serve un caso con array nullo oppure vuoto per coprire il cammino in cui si entra nel primo `if`.

```
arr = null,  
arr = [1, 1], // Non entra nel ciclo for  
arr = [0], // Entra nel ciclo for una volta e nell'if  
arr = [1] // Entra nel ciclo for una volta ma non nell'if
```

Ingegneria del Software — Soluzione del Tema 18/01/2021

Esercizio 1

Per incentivare l'uso di carte e app di pagamento, il governo ha definito un piano di rimborso per gli utenti che ne fanno uso. Un utente ha diritto a un rimborso normale (normal cashback) se effettua almeno 50 transazioni: il rimborso è del 10% delle spese effettuate fino a un massimo di 150 euro. I 100000 utenti che effettuano più transazioni hanno anche diritto a un rimborso super (super cashback) di 1500 euro. In caso di utenti con lo stesso numero di transazioni, si privileggiano gli utenti per cui la prima transazione effettuata è più vecchia. Si assume che due transazioni non possono mai essere perfettamente contemporanee.

Si consideri la classe `TransactionsDB` che definisce un database di transazioni ai fini di monitorare il piano di rimborsi. Il database utilizza la classe pura `Transaction` di cui sono riportati i metodi più rilevanti.

```
public class TransactionsDB {
    // @ensures (* Aggiunge la transazione , lanciando una NullPointerException se e' nulla *)
    public void addTransaction(Transaction t);

    // @ensures (* \result e' la lista di transazioni effettuate da uId in ordine cronologico *)
    public /*@ pure */ List<Transaction> userTransactions(UserId uId);

    // @ensures (* \result == ammontare del rimborso normale per uId *)
    public /*@ pure */ float normalCashBackFor(UserId uId);

    // @ensures (* \result == ammontare del rimborso super per uId *)
    public /*@ pure */ float superCashBackFor(UserId uId);
}

public /*@ pure */ class Transaction {
    // @ensures (* \result == utente che ha effettuato la transazione *)
    public UserId getUserId();

    // @ensures (* \result == ammontare della transazione *)
    public float getAmount();

    // @ensures (* \result <==> la transazione e' precedente a other *)
    public boolean before(Transaction other);
}
```

Domanda a)

Si specificino in JML i metodi `normalCashBackFor()` e `superCashBackFor()`.

Soluzione

```
//@requires uId != null
//@ensures userTransactions(uId).size() < 50 ==> \result == 0 &&
//@userTransactions(uId).size() >= 50 ==> \result ==
//@ (\sumof int i; i>=0 && i<userTransactions(uId).size());
//@ userTransactions(uId).get(i).getAmount() > 1500 ? 150 :
//@ (\sumof int i; i>=0 && i<userTransactions(uId).size());
//@ userTransactions(uId).get(i).getAmount() / 10;
public /*@ pure */ float normalCashBackFor(UserId uId);

//@requires uId != null
//@ensures \result ==
//@ (\numof UserId other; other!=null && !other.equals(uId));
//@ (userTransactions(other).size()>userTransactions(uId).size()) +
//@ (\numof UserId other; other!=null && !other.equals(uId));
//@ userTransactions(other).size()==userTransactions(uId).size() &&
//@ !userTransactions(other).isEmpty() &&
```

```

//@  userTransactions(other).get(0).before(userTransactions(uId).get(0)))
//@ < 100000 ? 1500 : 0;
public /*@ pure */ float superCashBack(UserId uId);

```

Domanda b)

Si consideri un'implementazione che utilizza una lista per contenere le transazioni degli utenti, come mostrato di seguito.

```

public class TransactionsDB {
    private final List<Transaction> list;
    ...
}

```

Per tale implementazione, si definiscano l'invariante di rappresentazione e la corrispondente funzione di astrazione.

Soluzione

Nell'invariante di rappresentazione escludiamo che la lista sia nulla, escludiamo possibili valori nulli nella lista ed escludiamo valori duplicati.

```

//(* Representation invariant RI *)
//@private invariant list!=null &&
//@(\forall int i; i>=0 && i<list.size(); list.get(i)!=null &&
//@ (\forall int j; j>=0 && j<i; !list.get(j).equals(list.get(i)) ) )

```

La funzione di astrazione definisce `userTransactions()` in funzione della lista usata nella rappresentazione. Gli altri metodi pubblici sono infatti definiti a partire da questo metodo. Si assume che il fatto che `userTransactions()` ritorni le transazioni in ordine cronologico sia definito come postcondizione del metodo.

```

//(* Abstraction function AF *)
//@private invariant
//@(\forall Transaction t; t!=null;
//@ userTransactions(t.getUserId()).contains(t) <=>
//@ (\exists int i; i>=0 && i<list.size(); list.get(i).equals(t)) )

```

Domanda c)

Si consideri ora una classe `TransactionsDB2` che aggiunge un metodo `totalCashBackFor(UserId uid)` che ritorna il cash back totale a cui ha diritto `uId`, come somma di cash back normale e super. È possibile definire `TransactionsDB2` come sottoclassa di `TransactionsDB` in accordo con il principio di sostituzione? È possibile il viceversa?

Soluzione

Trattandosi di un'estensione pura, che aggiunge solo un metodo e non cambia le proprietà, è possibile definire `TransactionsDB2` come sottoclassa di `TransactionsDB`.

Il viceversa non sarebbe possibile, in quanto non si possono rimuovere metodi nelle sottoclassi (regola della segnatura in Java).

Esercizio 2

Si consideri la seguente classe Java:

```
public class Vector {  
    private double x;  
    private double y;  
    public Vector(double x, double y) {  
        this.x = x; this.y = y;  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
    public Vector sum(Vector other) {  
        Vector res = new Vector(this.x+other.x, this.y+other.y);  
        return res;  
    }  
}
```

Domanda a)

E' possibile invocare i metodi della classe da thread separati senza che questo crei problemi? In caso affermativo motivare la risposta, in caso negativo indicare come andrebbe modificato il codice per avere una sincronizzazione corretta.

Soluzione

La classe rappresenta un oggetto immutabile. Come tale non presenta problemi di sincronizzazione, anche a fronte dell'invocazione dei suoi metodi da parte di thread concorrenti.

Domanda b)

Si aggiunga alla classe il metodo:

```
public void set(double x, double y)
```

che imposta il valore delle due componenti x e y del vettore. Si scriva il codice del metodo (e se necessario si cambi il codice degli altri metodi) in maniera che non si presentino problemi di concorrenza.

Soluzione

Il metodo `set` rende l'oggetto mutabile. Il metodo dovrà quindi essere `synchronized` e andranno sincronizzati anche i due getter. Più delicata la sincronizzazione del metodo `sum` che rischia di generare un deadlock. Si riporta il codice dei due metodi `set` e `sum`:

```
public synchronized void set(double x, double y) {  
    this.x = x; this.y = y;  
}  
  
public Vector sum(Vector other) {  
    double otherX, otherY;  
    synchronized(other) {  
        otherX = other.x;  
        otherY = other.y;  
    }  
    Vector res;  
    synchronized(this) {  
        res = new Vector(this.x+other.x, this.y+other.y);  
    }  
    return res;  
}
```

In alternativa il metodo `sum` poteva essere riscritto in modo più semplice come segue:

```
public Vector sum(Vector other) {  
    return new Vector(this.getX() + other.getX(), this.getY() + other.getY());  
}
```

eliminando corse critiche e rischio di deadlock a fronte di un comportamento che non garantisce atomicità: tra due invocazioni di `getX` e `getY` potrebbe essere invocato il metodo `set`.

Domanda c)

Si aggiunga alla classe il metodo:

```
public void suspendIfDifferent()
```

che sospende il thread chiamante in attesa che le componenti `x` e `y` del vettore diventino uguali. Si spieghi se e come vanno cambiati gli altri metodi.

Soluzione

Il metodo `suspendIfDifferent` può essere scritto come segue:

```
public synchronized void suspendIfDifferent() throws InterruptedException {  
    while(x!=y) wait();  
}
```

Inoltre va inserita l'istruzione `if (x==y) notifyAll();` all'interno del metodo `set`.

Esercizio 3

Si considerino le seguenti classi Java:

```
abstract class Vehicle {
    public void print(Vehicle other) {
        System.out.println("I am a " + who() + " and " + other + " is a " + other.who());
    }
    public abstract String who();
}

class Car extends Vehicle {
    public void print(Car other) {
        System.out.println("Someone says I am a " + who() + ", and " + other + " is a " + other.who());
    }
    public String who() { return "car"; }
}

class ElectricCar extends Car {
    public void print(Car other) {
        System.out.println("I am sure I am a " + who() + ", and " + other + " is a " + other.who());
    }
    public String who() { return "electric car"; }
}

class Bike extends Vehicle {
    public void print(Vehicle other) {
        System.out.println("I believe I am a " + who() + ", and " + other + " is a " + other.who());
    }
    public String who() { return "bike"; }
}
```

Si consideri poi il seguente frammento di codice Java che utilizza le classi dichiarate sopra:

```
1. Vehicle v1 = new Vehicle();
2. Vehicle v2 = new Car();
3. Vehicle v3 = new ElectricCar();
4. Vehicle v4 = new Bike();
5. Car c = new Car();
6. ElectricCar ec1 = new Car();
7. ElectricCar ec2 = new ElectricCar();
8. v1.print(ec1);
9. v2.print(v3);
10. v3.print(v4);
11. v3.print(c);
12. c.print(v3);
13. c.print(ec2);
14. ec2.print(c);
```

Domanda a)

Si indichino quali righe del codice sopra riportato generano un errore in fase di compilazione, motivando brevemente la propria risposta.

Soluzione

- Riga 1. Non è possibile instanziare una classe astratta (Vehicle)
- Riga 6. La classe Car non è sottoclasse di ElectricCar ma sopraclass. Non è possibile assegnare una Car and una ElectricCar.
- Riga 8. A seguito del primo errore v1 non è definito.

Domanda b)

Supponendo che tutte le righe che provocano errori siano state rimosse, scrivere l'output prodotto dal programma, motivando brevemente la risposta.

Soluzione

9. I am a car other is a electric car
10. I am a electric car other is a bike
11. I am a electric car other is a car
12. I am a car other is a electric car
13. Someone says I am a car, other is a electric car
14. I am sure I am a electric car, other is a car

Esercizio 4

Si consideri il seguente metodo statico:

```
1 public static int foolCode(int n, int x) {  
2     if (n > 0 && x < 100) {  
3         int j = 0;  
4         while (2 * j < n) {  
5             j++;  
6             n--;  
7         }  
8     }  
9     return n*x;  
10 }
```

Domanda a)

Si definisca un insieme minimo di casi di test che copra tutte le istruzioni.

Soluzione

Un caso basta, p.es. $n=1, x=1$.

Domanda b)

Si definisca un insieme minimo di casi di test che copra tutte le diramazioni (decisions – branch coverage).

Soluzione

Oltre al caso precedente anche p.es. $n=0, x=1$.

Domanda c)

Si definisca un insieme minimo di casi di test che copra tutte le diramazioni e tutte le condizioni (decisions and conditions coverage).

Soluzione

Oltre ai casi precedenti anche p.es. $n=1, x=100$.

Domanda d)

Si sintetizzi un caso di test per attraversare il cammino : 1 2 3 4 5 6 7 4 5 6 7 8 9 (che quindi attraversa il ciclo due volte).

Soluzione $n=4, x=1$

Ingegneria del Software — Soluzione del Tema 2/09/2021

Esercizio 1

Si consideri la seguente classe Java PhotoLibrary per la gestione di una collezione di foto. Ogni foto è rappresentata dalla classe immutabile Photo che contiene alcune informazioni tra cui l'insieme delle persone (classe immutabile Person) che appaiono nella foto.

```
public class PhotoLibrary {
    // Ritorna il numero di foto contenute nella collezione.
    public /*@ pure @*/ int size();

    // Ritorna l'i-esima foto contenuta nella collezione (in ordine di inserimento).
    public /*@ pure @*/ Photo get(int i);

    // Aggiunge una foto alla collezione.
    // Lancia una DuplicateException se la foto e' gia' presente nella collezione.
    public void add(Photo photo) throws DuplicateException;

    // Ritorna le foto che contengono tutte le persone in s.
    // s deve contenere almeno una persona.
    public /*@ pure @ */ Set<Photo> getPhotosWith(Set<Person> s);

    // Ritorna la lista di tutte le persone contenute in foto della collezione. La lista e'
    // ordinata dalla persona che appare in piu' foto alla persona che appare in meno foto
    // (in caso di uguale numero di apparizioni, l'ordine non e' specificato).
    public /*@ pure @ */ List<Person> getAllPeople();
}

public /*@ pure @*/ class Photo {
    // Ritorna l'insieme di persone contenute nella foto (puo' essere vuoto).
    public Set<Person> getPeople();
    ...
}
```

Domanda a)

Si specificino in JML i metodi add, getPhotosWith, getAllPeople.

Soluzione

```
//@requires photo != null
//@

//@ensures (\forall int i; i >= 0 && i < \old(size()); !\old(get(i)).equals(photo)) &&
//@ size() == \old(size()) + 1 && get(size() - 1).equals(photo) &&
//@ (\forall int i; i >= 0 && i < \old(size()); \old(get(i)).equals(get(i)))
//@

//@signals (DuplicateException e)
//@ (\exists int i; i >= 0 && i < \old(size()); \old(get(i)).equals(photo)) &&
//@ size() == \old(size()) &&
//@ (\forall int i; i >= 0 && i < \old(size()); \old(get(i)).equals(get(i)))
public void add(Photo photo) throws DuplicateException;

//@requires s != null && !s.isEmpty()
//@

//@ensures \result != null && (\forall Photo p; ; \result.contains(p) <==>
//@ (\exists int i; i >= 0 && i < size(); get(i).equals(p) && p.getPeople().containsAll(s)) )
public /*@ pure @ */ Set<Photo> getPhotosWith(Set<Person> s);

//@ensures \result != null &&
//@ (\forall Person p; ; \result.contains(p) <==>
//@ (\exists int i; i >= 0 && i < size(); get(i).getPeople().contains(p))) ) &&
//@ (\forall int i; i >= 0 && i < \result.size());
//@ (\forall int j; j >= 0 && j < i; !\result.get(i).equals(\result.get(j))) )
//@ (\forall int i; i >= 0 && i < \result.size() - 1;
//@ (\numof int j; j >= 0 && j < size(); get(j).getPeople().contains(\result.get(i))) >=
//@ (\numof int j; j >= 0 && j < size(); get(j).getPeople().contains(\result.get(i + 1)))
public /*@ pure @ */ List<Person> getAllPeople();
```

Domanda b)

Si consideri un'implementazione che utilizza una `List` per contenere le foto della collezione, come mostrato di seguito.

```
public class PhotoLibrary {  
    private final List<Photo> photos;  
    ...  
}
```

Per tale implementazione si definiscano l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

Nell'invariante di rappresentazione escludiamo che la `List` sia nulla, che contenga valori nulli e che contenga duplicati.

```
/*@(* Representation invariant RI *)  
//@private invariant photos != null && !photos.contains(null) &&  
//@ (\forall int i; i>=0 && i<photos.size());  
//@ (\forall int j; j>=0 && j<i; !photos.get(i).equals(photos.get(j))) )
```

La funzione di astrazione definisce `size()` e `get()` in funzione della `List` usata nella rappresentazione. Gli altri metodi pubblici sono infatti definiti a partire da essi.

```
/*@(* Abstraction function AF *)  
//@private invariant size() == photos.size() &&  
//@ (\forall int i; i>=0 && i<photos.size(); photos.get(i).equals(get(i)))
```

Domanda c)

Si consideri una classe `PhotoLibraryIter` che aggiunge un metodo `iter` che ritorna un iteratore `Iterator<Photo>` per iterare sulle foto della collezione. `PhotoLibraryIter` può essere definita come sottoclasse di `PhotoLibrary` nel caso in cui l'iteratore ritornato non definisca il metodo (opzionale) `remove?` E nel caso in cui definisca tale metodo?

Soluzione

Il metodo `iter` è puro, ma se il metodo `remove` è definito, l'iteratore può modificare la collezioni rimuovendo foto. Tale rimozione violerebbe una proprietà invariante della classe `PhotoLibrary`, che assume che le foto non possano essere rimosse. Di conseguenza `PhotoLibraryIter` può essere definita come sottoclasse di `PhotoLibrary` solo nel caso in cui il metodo `remove` non sia definito.

Esercizio 2

Si consideri la seguente classe Java:

```
public class WeatherSensor {  
    private double temp;  
    private double humidity;  
    public synchronized double getTemp() { return temp; }  
    public synchronized double getHumidity() { return humidity; }  
    public synchronized void setTemp(double temp) { this.temp = temp; }  
    public synchronized void setHumidity(double humidity) { this.humidity = humidity; }  
}
```

Domanda a)

Si consideri il seguente frammento di codice che crea due thread che accedono (uno in scrittura ed uno in lettura) alla medesima istanza condivisa della classe `WeatherSensor`:

```
WeatherSensor ws = new WeatherSensor();
```

```
new Thread() {  
    public void run() {  
        for(int i=0; i<10000; i++) {  
            ws.setTemp(i); ws.setHumidity(i);  
        }  
    }  
}.start();  
  
new Thread() {  
    public void run() {  
        double t = ws.getTemp();  
        double h = ws.getHumidity();  
        if(t == h) System.out.println("Same");  
        else System.out.println("Different");  
    }  
}.start();
```

E' corretto affermare che il secondo thread stamperà sempre `Same`? In caso non fosse così si modifichi il frammento di codice (senza toccare la classe `WeatherSensor`) affinchè sia certo che venga sempre stampato `Same`.

Soluzione Il frammento di codice va modificato come segue:

```
WeatherSensor ws = new WeatherSensor();  
  
new Thread() {  
    public void run() {  
        for(int i=0; i<10000; i++) {  
            synchronized(ws) {  
                ws.setTemp(i); ws.setHumidity(i);  
            }  
        }  
    }  
}.start();  
  
new Thread() {  
    public void run() {  
        synchronized(ws) {  
            double t = ws.getTemp();  
            double h = ws.getHumidity();  
        }  
        if(t == h) System.out.println("Same");  
        else System.out.println("Different");  
    }  
}.start();
```

Domanda b)

Si consideri il seguente metodo, parte della classe WeatherTemp, che ritorna `true` se la temperatura del sensore corrente e del sensore `other` sono uguali e `false` altrimenti:

```
public synchronized boolean sameTemp(WeatherSensor other) {  
    return this.temp == other.temp;  
}
```

Il metodo è correttamente sincronizzato? Spiegare perché e nel caso non lo fosse fornire una versione correttamente sincronizzata del metodo.

Soluzione

Il metodo non acquisisce il lock su `other` pur accedendo all'attributo privato `other.temp`. Va corretto come segue:

```
public boolean sameTempOk(WeatherSensor other) {  
    double t;  
    synchronized(other) { t = other.temp; }  
    synchronized(this) { return this.temp == t; }  
}
```

Domanda c)

Si aggiunga alla classe `WeatherSensor` un ulteriore metodo `getPositiveTemp` che restituisce la temperatura del sensore solo se positiva, sospendendo il chiamante fino a quando la temperatura non diventa positiva. Si indichi se e quali altri metodi della classe `WeatherSensor` vadano modificati e come.

Soluzione

```
public synchronized double getPositiveTemp() throws InterruptedException {  
    while(temp <= 0) wait();  
    return temp;  
}
```

Inoltre va aggiunta l'istruzione `notifyAll()` al metodo `setTemp`.

Esercizio 3

Si consideri il seguente programma Java.

```
class Veicolo {  
    protected void accensione(int tempoMax) {  
        System.out.println("accensione-Veicolo in "+tempoMax);  
    }  
    public void spegnimento() {  
        System.out.println("spengo-Veicolo");  
    }  
}  
  
class VeicoloAMotore extends Veicolo {  
    public void accensione(int tempoMax) {  
        System.out.println("accensione-VeicoloAMotore in "+tempoMax);  
    }  
    public void spegnimento(int tempo) {  
        System.out.println("spengo-VeicoloAMotore");  
    }  
}  
  
public class Motociclo extends VeicoloAMotore {  
    public void accensione(double tempoMax) {  
        System.out.println("accensione-Motociclo in "+tempoMax);  
    }  
    public void spegnimento() {  
        System.out.println("spengo-MotoCiclo");  
    }  
    public static void main(String[] args) { ... }  
}
```

Il main è costituito dalle seguenti istruzioni, numerate per riferimento:

```
1 Veicolo v = new Veicolo();  
2 Veicolo v1= new VeicoloAMotore();  
3 VeicoloAMotore vm1= new Veicolo();  
4 VeicoloAMotore vm2= new Motociclo();  
5 v.accensione(3);  
6 v1.accensione(3);  
7 vm1.accensione(3);  
8 vm2.accensione(3);  
9 v.spegnimento(1);  
10 v.spegnimento();  
11 v1.spegnimento();  
12 v1.spegnimento(2);  
13 vm1.spegnimento(2);  
14 vm2.spegnimento();  
15 vm2.spegnimento(2);
```

Domanda a)

Scrivere il numero di riga delle istruzioni (se ne esistono) che generano un errore in compilazione. Giustificare brevemente la risposta.

Soluzione

Righe 3, 7, 9, 12,13.

La riga 3, in quanto Veicolo non è una sottoclasse di VeicoloAMotore. Quindi devono essere eliminate anche le righe 7 e 13. Le righe 9 e 12, in quanto un Veicolo non possiede il metodo spegnimento(int).

Domanda b)

Supponendo di eliminare le eventuali righe che generano errori, si scriva, per ogni riga numerata, il valore stampato in uscita.

Soluzione

5: accensione-Veicolo in 3
6: accensione-VeicoloAMotore in 3
8: accensione-VeicoloAMotore in 3
10: spengo-Veicolo
11: spengo-Veicolo
14: spengo-MotoCiclo
15: spengo-VeicoloAMotore

NB: La riga 8 chiama il metodo VeicoloAMotore.accensione(int) in quanto il valore 3 è un int, non un double.

Esercizio 4

Si considerino i seguenti metodi Java e se ne completino le parti omesse, *utilizzando esclusivamente i concetti e i costrutti della programmazione funzionale*.

Domanda a)

Il metodo seguente prende come argomento una lista di nomi (di tipo `String`) (il parametro `people`) e restituisce il nome di maggiore lunghezza fra quelli interamente in maiuscolo. Per esempio, se la lista include i nomi "Paolo", "MARIO", "ISA", "Antonio", il metodo restituisce "MARIO". Se non esiste alcun nome interamente in maiuscolo, il metodo restituisce la stringa "n.a.".

Si ricorda che la classe `String` ha il metodo `length()` che restituisce la lunghezza della stringa.

```
public static String longestInUppercase(List<String> people) {  
    return ....;  
}
```

Suggerimento: per verificare se un nome è interamente in maiuscolo si può ad esempio usare il metodo `toUpperCase()` della classe `String`, che restituisce la conversione in maiuscolo della stringa `this` – un semplice confronto del risultato con l'originale può stabilire se l'originale era già interamente in maiuscolo.

Soluzione

```
public static String longestInUppercase(List<String> people) {  
    return people.stream()  
        .filter(name -> name.equals(name.toUpperCase()))  
        .reduce((name1, name2) -> name1.length() >= name2.length() ? name1 : name2)  
        .orElse("n.a.");  
}
```

Domanda b)

Il metodo seguente prende ancora come argomento una lista di nomi di tipo `String` (il parametro `people`), ma restituisce la lista dei nomi che non sono interamente in maiuscolo, convertiti in maiuscolo. Per esempio, se la lista include i nomi "Paolo", "MARIO", "ISA", "Antonio", il metodo restituisce una lista composta da "PAOLO", "ANTONIO".

```
public static List<String> inUpperCase(List<String> people) {  
    return ....;  
}
```

Soluzione

```
public static List<String> inUpperCase(List<String> people) {  
    return people.stream()  
        .filter(name -> !name.equals(name.toUpperCase()))  
        .map(name -> name.toUpperCase())  
        .collect(Collectors.toList());  
}
```

Ingegneria del Software — Soluzione del Tema 15/02/2021

Esercizio 1

Si consideri la seguente classe Java PackageManager per gestire l'installazione di pacchetti software e delle loro dipendenze. I singoli pacchetti sono rappresentati dalla classe pura Package di cui riportiamo i metodi rilevanti.

```
public class PackageManager {
    // Ritorna lo spazio disco disponibile in KByte.
    public /*@ pure */ int availableSize();

    // Ritorna true se e solo se dep e' una dipendenza diretta o indiretta di p e
    // dep non e' attualmente installata.
    public /*@ pure */ boolean depToInstall(Package dep, Package p);

    // Se p non e' gia' installato e c'e' sufficiente spazio disco, installa p e tutti
    // i pacchetti da cui p dipende direttamente o indirettamente e che non sono installati.
    // Se non c'e' sufficiente spazio disco lancia l'eccezione DiskFullException.
    public void install(Package p) throws DiskFullException;

    // Ritorna l'elenco dei pacchetti attualmente installati
    public /*@ pure */ Set<Package> getInstalled();
}

public /*@ pure */ class Package {
    // Ritorna l'identificativo univoco del pacchetto software
    public String getId();

    // Ritorna la dimensione del pacchetto software in KBytes
    public int getSize();

    // Ritorna l'elenco dei pacchetti da cui questo pacchetto dipende direttamente
    public List<Package> getDeps();

    // Due pacchetti sono uguali se hanno lo stesso id
    public boolean equals(Object o);
}
```

Domanda a)

Si specifichi in JML il metodo depToInstall().

Soluzione

```
//@requires dep != null && p != null
//@ensures \result <==> (
//@  ( p.equals(dep) && !getInstalled().contains(p) ) ||
//@  ( (\exists Package d; p.getDeps().contains(d); depToInstall(dep, d)) ) )
public /*@ pure */ boolean depToInstall(Package dep, Package p);
```

Domanda b)

Si specifichi in JML il metodo install().

Soluzione

```
//@requires p != null
//@
//@ensures
//@ (\sum Package dep; \old(depToInstall(dep, p)); dep.size()) <= \old(availableSize()) &&
//@ (\forall Package x; ; getInstalled().contains(x) <==>
```

```

//@  ( \old(getInstalled()).contains(x) || depToInstall(x, p) ) ) &&
//@ availableSize() == \old(availableSize()) -
//@           (\sum Package dep; depToInstall(dep, p); dep.size())
//@
// @signals DiskFullException &&
//@ (\sum Package dep; depToInstall(dep, p); dep.size()) > \old(availableSize()) &&
//@ getInstalled().containsAll(\old(getInstalled())) &&
//@ \old(getInstalled()).containsAll(getInstalled()) &&
//@ \old(availableSize()) == availableSize()
public void install(Package p) throws DiskFullException;

```

Domanda c)

Si consideri un'implementazione che utilizza una lista per contenere i pacchetti installati e un intero per mantenere lo spazio disponibile, come mostrato di seguito.

```

public class PackageManager {
    private final List<Package> list;
    private int availableSize;
    ...
}

```

Per tale implementazione, si definiscano l'invariante di rappresentazione e la corrispondente funzione di astrazione.

Soluzione

Nell'invariante di rappresentazione escludiamo che la lista sia nulla, che contenga valori nulli e che contenga duplicati. Richiediamo che lo spazio disponibile sia maggiore o uguale a zero

```

//@(* Representation invariant RI *)
//@private invariant list != null && availableSize >= 0 &&
//@ (\forallall int i; i>=0 && i<list.size(); list.get(i) != null
//@   (\forallall int j; j>=0 && j<i; !list.get(j).equals(list.get(i)) ) ) &&

```

La funzione di astrazione definisce `getInstalled()` in funzione della lista usata nella rappresentazione e `availableSize()` in funzione della variabile corrispondente. Gli altri metodi pubblici sono infatti definiti a partire da essi.

```

//@(* Abstraction function AF *)
//@private invariant availableSize() == availableSize &&
//@(\forallall Package p; p!=null;
//@  getInstalled().contains(p) <==> list.contains(p) )

```

Domanda d)

Si consideri ora una classe `PackageManager2` che aggiunge un metodo `uninstall(Package p)` per cancellare un pacchetto se questo è installato. È possibile definire `PackageManager2` come sottoclasse di `PackageManager` in accordo con il principio di sostituzione?

Soluzione

Non è possibile, in quanto l'aggiunta del metodo violerebbe un invariante della classe, ovvero il fatto che i pacchetti non possono essere rimossi.

Esercizio 2

Si consideri la seguente classe Java:

```
public class Matrix {  
    private double[][] data;  
    private int size;  
  
    public Matrix(int size) {  
        this.size = size;  
        data = new double[size][size];  
    }  
  
    public synchronized double get(int row, int column) {  
        return data[row][column];  
    }  
  
    public synchronized void set(int row, int column, double value) {  
        data[row][column] = value;  
    }  
  
    public synchronized void add(Matrix m) {  
        for (int r=0; r<size; r++) {  
            for (int c=0; c<size; c++) {  
                data[r][c] += m.data[r][c];  
            }  
        }  
    }  
  
    public synchronized void clear() {  
        for (int r=0; r<size; r++) {  
            for (int c=0; c<size; c++) {  
                data[r][c] = 0;  
            }  
        }  
    }  
}
```

Domanda a)

La classe è correttamente sincronizzata? In caso negativo si illustri come modificare la classe per risolvere il problema di sincronizzazione identificato.

Soluzione

Il metodo `add` accede a `m.data` senza sincronizzarsi su `m`. La seguente versione modificata del metodo aggira il problema senza rischiare di cadere in una situazione opposta, di potenziale deadlock:

```
public void add(Matrix m) {  
    for (int r=0; r<size; r++) {  
        for (int c=0; c<size; c++) {  
            set(r,c, get(r,c)+m.get(r,c));  
        }  
    }  
}
```

Domanda b)

Si aggiunga un metodo `public void asyncPrint()` che stampa a video il contenuto della matrice **in un thread separato rispetto al chiamante**. Ovvero, il chiamante dovrà tornare subito mentre un thread separato esegue la vera e propria stampa.

Soluzione

Aggiungiamo un metodo privato `print` che esegue la stampa (deve essere sincronizzato poichè accede allo stato della matrice), mentre il metodo `asyncPrint` (non è necessario che sia sincronizzato) si occuperà di creare ed avviare il thread separato che esegue `print`:

```
public void asyncPrint() {
    new Thread( () -> print() ).start();
    /* Oppure:
    new Thread() {
        public void run() {
            print();
        }
    }.start();
    */
}

private synchronized void print() {
    for(int r=0; r<size; r++) {
        for(int c=0; c<size; c++) {
            System.out.printf("%10.1f ", data[r][c]);
        }
        System.out.println("");
    }
}
```

Domanda c)

Si supponga che il metodo `add` non sia presente e si modifichi il metodo `set` in maniera che il chiamante venga sospeso in attesa che il valore in posizione `row, column` diventi zero prima di impostarlo al valore `value`.

Si indichi se altri metodi devono essere modificati e come (si ricorda che il metodo `add` è stato eliminato).

Soluzione

Il metodo `set` deve essere modificato come segue:

```
private synchronized void set(int row, int column, double value) throws InterruptedException {
    while(data[row][column]!=0) wait();
    data[row][column] = value;
}
```

Inoltre bisogna aggiungere l'istruzione `notifyAll()` al metodo `clear`.

Esercizio 3

Si consideri il seguente programma Java.

```
public abstract class Vector {
    public abstract double norm();
}

public class PlanarVector extends Vector {
    protected final double x;
    protected final double y;
    public PlanarVector(double x, double y) {this.x=x; this.y=y;}
    public double norm(){ return Math.sqrt(x*x+y*y); }
    public PlanarVector sum(PlanarVector v) {return new PlanarVector(x+v.x, y+v.y);}
    public String toString() {return "x=" + x + "y=" + y;}
}

public class SpaceVector extends PlanarVector {
    protected final double z;
    public SpaceVector(double x, double y,double z) {super(x,y); this.z=z;}
    public double norm(){ return Math.sqrt(x*x+y*y+z*z); }
    public SpaceVector sum(SpaceVector v) {return new SpaceVector(x+v.x, y+v.y, z+v.z);}
}
public String toString() {return super.toString() + "z=" + z;}
```

e la seguente classe di prova (situata nello stesso package).

```
1 public class TestVector {
2     public static void main (String [] args) {
3         Vector v0;
4         PlanarVector v1, v2;
5         SpaceVector v3, v4;
6         v1 = new PlanarVector (3,4);
7         v0 = v1;
8         v2 = new SpaceVector(2,3,6);
9         v3 = new SpaceVector(2,1,0);
10        v4 = v3;
11        System.out.println(v0.sum(v1));
12        System.out.println(v1.sum(v2));
13        System.out.println(v2.sum(v1));
14        System.out.println(v2.sum(v3));
15        System.out.println(v3.sum(v2));
16        System.out.println(v3.sum(v4));
17        System.out.println(v1.norm());
18        System.out.println(v2.norm());
19    }
20 }
```

Domanda a)

Si elenchi i numeri di riga corrispondenti alle eventuali istruzioni che generano errori di compilazione, giustificando la risposta.

Dopo avere eliminato le istruzioni corrispondenti, si scriva il risultato dell'esecuzione del programma (riportando esplicitamente i numeri di riga originali). Qualora un risultato non sia intero, lo si rappresenti simbolicamente, senza calcolarne il valore (es. "radice di 2").

Soluzione

L'unica istruzione da eliminare è quella alla riga 11, in quanto il tipo statico di v0 (ossia Vector) non ha il metodo sum.

Il risultato dell'esecuzione è il seguente:

```
12 X=5.0 y=7.0
13 x=5.0 y=7.0
14 x=4.0 y=4.0
15 x=4.0 y=4.0
16 x=4.0 y=2.0 z=0.0
17 5.0
18 7.0
```

Esercizio 4

Si consideri il seguente metodo statico Java.

```
1 static int test(int x) {  
2     int y = x%3;  
3     System.out.println(y);  
4     while (x>0 && y<3) {  
5         if (y==1)  
6             return y;  
7         y--;  
8     }  
9     return 0;  
10 }
```

Si determini un insieme minimo di test per ciascuno dei seguenti criteri di copertura. Qualora un criterio non sia soddisfacibile, calcolare la percentuale di copertura dell'insieme di test definito per quel criterio.

Domanda a)

Copertura delle istruzioni.

Soluzione

Servono almeno due casi di test, in quanto vi sono due return. Ad esempio, $x=2$ (o $x=5$) esegue il cammino $1, 2, 3, 4, 5, 7, 4, 5, 6$. Il caso $x=0$ copre anche la riga 9 (ignoriamo le righe 8 e 10 che sono solo dei terminatori sintattici).

Domanda b)

Copertura delle decisioni (branch).

Soluzione

I due casi precedenti vanno bene.

Domanda c)

Copertura delle condizioni e decisioni.

Soluzione

La copertura totale delle condizioni è impossibile, perché il resto della divisione per 3 è sempre minore di 3. Usando i due casi precedenti è quindi possibile coprire solo 2 condizioni su 3 (66%).

Domanda d)

In base eventualmente anche ai casi di test definiti, individuare la presenza di un possibile errore di programmazione.

Soluzione

Il programma entra in loop infinito quando $x \% 3$ è zero. Quindi ad esempio per $x=3$.



Politecnico di Milano

Anno accademico 2010-2011

Ingegneria del Software

Appello del 9 Febbraio 2012

Cognome:

Nome:

Matricola:

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h45m.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 1

Si scrivano in JML le pre- e post- condizioni dei seguenti metodi:

- **cornici** prende in ingresso una matrice bidimensionale quadrata di interi e controlla che i numeri di ogni cornice siano tutti uguali tra loro. Ad esempio, la matrice a sinistra di Figura 1 non rispetta la condizione, mentre quella di destra è corretta. Si noti inoltre che la matrice con un solo elemento rispetta sempre la condizione e che lo stesso vale per tutte le matrici quadrate i cui elementi sono tutti uguali tra loro.

			1	1	1	1	1
1	1	1	1	3	3	3	1
2	2	2	1	3	2	3	1
3	3	3	1	3	3	3	1
			1	1	1	1	1

Figure 1: Matrici d'esempio.

```
public static boolean cornici(int[][] a)
```

- **somme** prende in ingresso un array di interi, di lunghezza n , e controlla che ogni elemento, di posizione compresa tra 0 e $n - 3$, sia la somma dei due elementi successivi.

```
public static boolean somme(int[] a)
```

Esercizio 2

Un Dizionario è una struttura dati adatta a memorizzare e ricercare efficientemente degli elementi di un tipo assegnato. Ogni elemento contenuto è caratterizzato da una chiave.

Si consideri la seguente specifica della classe Dizionario, generica rispetto ai parametri $\langle V, K \rangle$ che rappresentano rispettivamente il tipo dei valori restituiti dal dizionario e il tipo delle chiavi. Il Dizionario utilizza una classe Elemento, anch'essa generica, di cui ogni oggetto è individuato da un valore e da una chiave.

```
import java.util.*;
/*@ pure @*/ class Elemento<Val,Key> {
//@ensures k == key() && v == value();
public Elemento (Key k, Val v) { };
//restituisce il valore di this;
public Val value() {return v;}
//restituisce la chiave di this;
public Key key() {return k;}
}

public class Dizionario<V,K> {
private Elemento<V,K> el;

//costruisce un DizionarioVuoto
public Dizionario() {}

//restituisce numero di elementi del dizionario
public int size() {}

//@ensures (* se esiste un elemento di chiave 'chiave' allora \result.key().equals(chiave))
//@ealtrimenti \result == null
public Elemento<V,K> find(K chiave) {}

//restituisce una lista di tutti e soli gli elementi di this, in ordine arbitrario;
public ArrayList<Elemento<V,K>> elementi() {}

//inserisce in this e restituisce un nuovo elemento di chiave k e valore v
public Elemento<V,K> insert (K k, V v) {}

//rimuove e da this
public void remove (Elemento<V,K> e) {}
}

a) Scrivere la specifica JML di insert e remove.
```

Sketch Soluzione

```
//@\result.key() == k && elementi().contains(\result) &&
//@\forall int i; 0\le i && i < \old(elementi().size());
//@      \old(elementi().get(i)).key() != k ==>
//@      elementi().contains(\old(elementi().get(i))) &&
//@ elementi().size() == \old(elementi().size()) + (\old(find(k)) == null ? 1 : 0)
public Elemento<V,K> insert (K k, V v) {}

//@\ !elementi().contains(\result) &&
//@\forall int i; 0\le i && i < \old(elementi().size());
//@      \old(elementi().get(i)).key() != k ==>
//@      elementi().contains(\old(elementi().get(i))) &&
//@ elementi().size() == \old(elementi().size()) + (\old(find(k)) == null ? 0 : -1)
public void remove (Elemento<V,K> e){}
```

b) La specifica del Dizionario viene successivamente estesa a quella di un Dizionario Ordinato, per tenere conto del caso in cui le chiavi siano ordinate totalmente. L'utilità delle chiavi ordinate è ovviamente di permettere implementazioni più efficienti dei metodi insert, remove, find.

Descrivete una possibile soluzione al problema della specifica delle chiavi ordinate, motivando le vostre scelte. La vostra soluzione deve soddisfare i principi fondamentali della progettazione a oggetti. Mostrate anche quali sono i principi utilizzati, e come mai sono soddisfatti.

Sketch Soluzione Si pu specificare che le chiavi sono ordinate richiedendo ad esempio che il costruttore di DizionarioOrdinato debba ricevere come parametro un oggetto che implementa l'interfaccia Comparator<K>.

Quindi DizionarioOrdinato puo' essere definita come la seguente estensione di Dizionario:

```
public class DizionarioOrdinato<V,K> extends Dizionario<V,K> {

    //costruisce un DizionarioOrdinato vuoto usando un comparatore per le chiavi:
    public DizionarioOrdinato(Comparator<K> comparatore);

    //restituisce una lista di tutti e soli gli elementi di this,
    //secondo l'ordine delle chiavi;
    public ArrayList<Elemento<V,K>> elementi();
}
```

La Classe DizionarioOrdinato verifica il principio di sostituzione: il costruttore infatti non e' ereditato e l'unico metodo ridefinito, elementi(), rafforza la postcondizione originale. Si noti che la classe DizionarioOrdinato, quasi certamente, reimplementera' opportunamente i metodi find, insert e remove per sfruttare l'ordinamento, anche se non ne modifica (necessariamente) la specifica.

Una soluzione alternativa e' usare l'interfaccia Comparable:

Ad esempio:

```
public class DizionarioOrdinato2<V, K extends Comparable<? super K>> extends Dizionario<V,K> {

    //costruisce un DizionarioOrdinato2 vuoto
    public DizionarioOrdinato2() {}

    //restituisce una lista di tutti e soli gli elementi di this, nell'ordine delle chiavi
    public ArrayList<Elemento<V,K>> elementi() {return new ArrayList<Elemento<V,K>>();}
}
```

Questa classe impone un vincolo aggiuntivo su K, "suggerendo" che vi sia una qualche violazione della regola dei metodi (il vincolo su K e' una specie di precondizione). Tuttavia questo vincolo riguarda solo il costruttore (che riceve come parametro K_{\cup}), quindi non comporta violazione della regola dei metodi. Anche questa soluzione verifica quindi il principio di sostituzione.

Per confrontare le due soluzioni, si consideri il seguente metodo che riceve come parametro un Dizionario in cui i valori sono String e le chiavi sono istanze di una opportuna classe Orario:

```
public static void foo(Dizionario<String,Orario> d) { . . . }
```

La soluzione del DizionarioOrdinato, che non impone vincoli sul parametro K, permette di istanziare e passare a foo(..) un parametro di tipo *DizionarioOrdinato < String, Orario >*, purche' per Orario sia definita una classe a parte che implementa *Comparator < Orario >* (una cui istanza viene passata al costruttore di DizionarioOrdinato). Non e' quindi necessario modificare Orario.

La soluzione del DizionarioOrdinato2 impone invece il vincolo che il parametro K implementi l'interfaccia *Comparable < Orario >* (o anche un *Comparable < T >* in cui T e' sovratipo di Orario) Quindi se Orario non implementa Comparable, diventa necessario modificare la classe Orario stessa. Non solo, questa soluzione potrebbe essere impossibile qualora l'ordine che si vuole usare per le chiavi non sia lo stesso ordinamento implementato da Comparable.

In linea di massima, la prima soluzione e' da preferire in quanto piu' generale: non costringe a modificare classi esistenti usate come chiavi, e permette maggiore flessibilita' nel definire l'ordinamento.

Esercizio 3

Si progetti, utilizzando un diagramma delle classi UML, la struttura di un quotidiano. Ogni giornale è composto da un certo numero di pagine; ogni pagina è organizzata in trenta slot: sei segmenti per ognuna delle cinque colonne. Il giornale contiene articoli e inserzioni pubblicitarie. Ogni articolo ha un titolo, un autore e un corpo. Ogni inserzione pubblicitaria contiene un'immagine e una dimensione. Gli articoli possono essere accompagnati da fotografie.

Articoli, fotografie e inserzioni devono essere contenuti in uno o più slot. Ogni pagina può contenere un numero ben preciso di articoli, fotografie e inserzioni; alcuni elementi potrebbero essere disposti su più pagine. La prima pagina ha un'organizzazione leggermente diversa: la prima fila di slot contiene il nome del giornale, la data e altre informazioni accessorie.

Il giornale impaginato correttamente deve occupare tutte le pagine a disposizione. Ovvero non devono restare pagine occupate solo parzialmente o contenuti non stampati per mancanza di spazio. Dopo aver identificato attributi e metodi necessari, si scriva anche la specifica del metodo **impaginabile** della classe **Quotidiano** per controllare se i contenuti a disposizione occupano esattamente il numero di slot della foliazione stabilita.

Esercizio 3

Si consideri il metodo seguente e si generi un insieme “minimo” di casi di test che soddisfi il criterio di copertura delle diramazioni (*branch coverage*) e che copra anche le condizioni limite.

```
00 int cerca(int[] a, int v) {  
01     int r = -1, i = 0;  
02     boolean t = false;  
03  
04     if (a == null)  
05         return r;  
06  
07     while(i < a.length && t == false) {  
08         if (a[i] == v) {  
09             t = true;  
10             r = i;  
11         }  
12         else  
13             i++;  
14     }  
15     return r;  
16 }
```

Appello 25 settembre 2015



Politecnico di Milano
Anno accademico 2014-2015

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Baresi <input type="checkbox"/> Ghezzi <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica di un tipo di dato astratto `gara` che rappresenta una gara di atletica leggera (salti e lanci). Alla gara partecipano un numero di atleti fissato nel momento in cui la gara viene creata. Ad ogni atleta corrisponde un numero di pettorale: $1, 2, \dots, n$, dove n è il numero dei partecipanti alla gara. L'operazione `gareggia(x, ris)` indica che l'atleta con il numero di pettorale x ha gareggiato ottenendo il risultato ris (per esempio, se si tratta di una gara di salto in alto, ris rappresenta l'altezza di salto in cm). L'operazione `primo()` restituisce il numero di pettorale dell'atleta in testa al momento in cui l'operazione viene eseguita.

```
public class Gara{  
  
    // se p <=0 genera l'eccezione NonEsisteGara, altrimenti crea  
    //una gara con p partecipanti  
    public Gara(int p) throws NonEsisteGaraException  
  
    //metodo mutator  
    //se x<0 o x e' maggiore del num. dei partecipanti genera l'eccezione  
    //AtletaInesistente altrimenti modifica il valore del risultato  
    //ottenuto dall'atleta x senza modificare i risultati degli  
    //altri atleti  
    public void gareggia(int x, float result) throws AtletaInesistenteException  
  
    //metodi observer:  
  
    //restituisce il vincitore corrente della gara.  
    public /*@ pure @*/ int primo()  
  
    //restituisce i risultati ottenuti da ogni atleta partecipante alla gara.  
    //Se un atleta non ha ancora ancora gareggiato il suo risultato  
    //e' convenzionalmente fissato a -1.  
    public /*@ pure @*/ ArrayList<Float> risultati;b  
  
}
```

1. Si specificino i metodi `gareggia` e `primo` in JML.

SOLUZIONE:

Per completezza specificiamo anche il costruttore.

```
    //@ensures p>0 && risultati().size()==p &&  
    //@      (\forall int i; 0<= i && i<p; risultati().get(i)==-1);  
    public Gara(int p) throws NonEsisteGaraException  
  
    //@ensures 0<x && x<= risultati().size() &&  
    //@ risultati().size()== \old(risultati().size()) &&  
    //@ (\forall int i; 0<= i && i<risultati().size();  
    //@      i==x-1 ? risultati().get(i) == result  
    //@            :risultati().get(i) == \old(risultati().get(i));  
    //@signals (AtletaInesistenteException e) (x<=0 || x>risultati().size()) &&  
    //@      risultati().equals(\old(risultati().clone()));  
    public void gareggia(int x, float result) throws AtletaInesistenteException  
  
    //@ensures risultati().get(\result -1) ==  
    //@      (\max int i; 0<=i && i<risultati().size(); risultati().get(i));
```

```
public /*@ pure @*/ int primo()
```

Si e' ipotizzato che primo() un pettorale qualunque quando sia chiamato prima di avere inserito almeno un risultato.

2. Si definisca un'implementazione ragionevolmente efficiente per il calcolo del vincitore. Si scrivano pertanto una rep, il suo invarianti di rappresentazione e la sua funzione di astrazione.

SOLUZIONE:

Per semplificare il calcolo del vincitore e' utile memorizzare il pettorale dell'atleta col migliore risultato. I risultati possono essere memorizzati semplicemente in un array (la cui dimensione non cambia dopo la costruzione). Occorre poi fare attenzione che i pettorali partono da 1 mentre gli indici dell'array da 0.

```
private int primo;
private float[] ris;

//RI:
private invariant ris !=null &&
    ris[primo-1] == (\max int i; 0<=i && i<ris.length; ris[i]);

//AF:
private invariant primo()== primo && ris.equals(risultati().toArray())
```

3. Si fornisca un'implementazione del costruttore e del metodo gareggia() descrivendo brevemente, per ciascuno di essi, (1) perchè il rep invariant viene effettivamente mantenuto e (2) perchè la loro specifica viene rispettata. SOLUZIONE:

```
public Gara(int p) throws NonEsisteGaraException {
    if (p<=0) throw new NonEsisteGaraException();
    ris = new Float[p-1];
    primo=1;
    Arrays.fill(ris,-1);
}
```

Il RI è banalmente vero (ogni elemento dell'array ris è inizializzato a -1 e il primo è il pettorale numero 1, che è un pettorale valido). La specifica è pure banalmente vera: in base alla funzione di astrazione ris corrisponde esattamente a risultati(), e ris ha dimensione p ed è inizializzato interamente a -1.

```
public void gareggia(int x, float result) throws AtletaInesistenteException {
    if (x<=0 || x>= ris.length) throw new AtletaInesistenteException();
    ris[x-1] = result;
    if (result>ris[primo-1])
        primo = x;
}
```

Se RI vale all'entrata del metodo allora ris[primo-1] contiene il risultato migliore (ossia massimo). Se result è maggiore del massimo corrente allora il codice garantisce di memorizzare in primo il pettorale x. La specifica vale in quanto il codice memorizzare in ris[x-1] il valore di result e, per la funzione di astrazione, ris[x-1] corrisponde a risultati().get(x-1); tutti gli altri valori restano invariati.

4. Si vuole ora trattare un caso leggermente più generale di gara: il vincitore può essere scelto anche come l'atleta che ha il punteggio minimo invece che quello con il punteggio massimo (ad esempio quando il punteggio rappresenta il tempo impiegato per una gara di corsa). Si definisca quindi una classe GaraEstesa, che estende la classe Gara. In essa il costruttore riceve un parametro addizionale tipo, il cui valore può essere solo il carattere '<' o '>', per stabilire l'ordinamento, che non può essere modificato dopo la costruzione.

```

public class GaraEstesa extends Gara {

    public GaraEstesa(int p, char tipo)

        //restituisce il vincitore corrente della gara secondo l'ordinamento.
        @override
        public /*@ pure */ int primo()

    }
}

```

a) Si completi la specifica dei metodi di `GaraEstesa`, aggiungendo eventualmente tutti e soli i metodi che si ritengono necessari.

SOLUZIONE:

E' necessario aggiungere un metodo puro che restituisca il tipo della gara:

```
public /*@ pure */ char tipo();
```

Specifiche (sotto l'ipotesi che i risultati validi siano positivi).

```

//@requires tipo == '<' || tipo == '>';
//@ensures p>0 && risultati().size() == p &&
//@      tipo() == tipo &&
//@      (\forall int i; 0 <= i && i < p; risultati().get(i) == -1);
public GaraEstesa(int p, char tipo) throws NonEsisteGaraException;

//@ensures risultati().get(-1) ==
//@ (tipo() == '>' ? (\max int i; 0 <= i && i < risultati().size(); risultati().get(i)) :
//@ : (\min int i; 0 <= i && i < risultati().size() && risultati().get(i) > 0; risultati().get(i));
public /*@ pure */ int primo()

```

b) La classe `GaraEstesa` verifica il principio di sostituzione? Giustificare la risposta.

SOLUZIONE:

NO, in quanto il metodo `primo()` viola la regola dei metodi: quando il tipo e' '<', restituisce il concorrente arrivato ultimo, anziche' il primo. La postcondizione e' pertanto violata.

Esercizio 2

Un impianto industriale coordinata tre bracci meccanici per il confezionamento di caramelle:

- Ogni robot preleva le caramelle da confezionare da un unico contenitore;
- Il contenitore è riempito ad intervalli regolari con 10.000 caramelle;
- Ogni robot può solo prelevare 5 caramelle per volta dal contenitore comune;
- Ogni robot può gestire confezioni da 5, 10 e 15 caramelle;
- L'accesso al contenitore è consentito solamente a due robot per volta;

Si scriva il codice Java delle classi Robot, Controllore e Impianto tenendo presente che: (a) il sistema deve essere progettato per la massima flessibilità, ovvero in futuro il numero di robot e di contenitori della caramelle potrebbero cambiare. Inoltre, in un certo istante, tutti i robot devono gestire confezioni dello stesso tipo. Ad esempio, se un robot è configurato per confezioni da 10 caramelle, anche gli altri devono confezionare allo stesso modo.

Esercizio 3

Si consideri il seguente metodo, in cui ogni istruzione è numerata per comodità:

```
1 static void esegui(int a, int b) {  
2             if (b>= 0 && a>0) {  
3                 while (b !=0) {  
4                     if (b >= 3)  
5                         a++;  
6                     else a--;  
7                     b--;  
8                 };  
9             };  
10            return a;  
11 }
```

1. Si scriva la path condition e si sintetizzi un dato di test per percorrere il cammino seguente: 1, 2, 3, 4, 5, 7, 3, 4, 6, 7, 3, 4, 6, 7, 8. SOLUZIONE:

b=3 && a>0 (es. b=3 && a = 1)

2. Qual'è il numero minimo di dati di test necessari per coprire tutte le condizioni del programma?
SOLUZIONE:

Il caso trovato nel punto precedente copre già le condizioni del while e del secondo if (durante le varie iterazioni del ciclo). Basta quindi coprire il caso false del primo if (con due casi di test). i casi di test minimi sono quindi solo 3:

- 1) b<0 (es- b=-1)
- 2) b>=0 && a<=0 (es. b=1 && a=0)
- 3) b=3 && a=1

3. Si sintetizzino i dati di test che soddisfano il requisito di cui al punto (2).

Esercizio 4

Si considerino le seguenti classi Java:

```
public class Albero {  
    public String toString(){  
        return "Albero";  
    }  
    public void innesto(Albero a) {  
        System.out.println("Albero: " + this + " " + a);  
    }  
}  
  
public class Arancio extends Albero {  
    public String toString(){  
        return "Arancio";  
    }  
  
    public void innesto(Arancio a){  
        System.out.println("Arancio: " + this + " " + a);  
    }  
}  
  
public class Melo extends Albero {  
    public String toString(){  
        return "Melo";  
    }  
  
    public void innesto(Melo m) {  
        System.out.println("Melo: " + this + " " + m);  
    }  
}
```

e si spieghi cosa stamperebbe il seguente metodo main, motivando brevemente le risposte:

```
public class Test {  
    public static void main(String args[]) {  
        Albero a1, a2;  
        Melo m;  
        Arancio a;  
  
        a1 = new Albero();  
        a2 = new Arancio();  
        m = new Melo();  
        a = new Arancio();  
  
        m.innesto(a1);  
        a2.innesto(a1);  
        m.innesto(m);  
        a2 = m;  
        a1 = a;  
        a2.innesto(m);  
        a1.innesto(a);  
    }  
}
```

SOLUZIONE:

Albero: Melo Albero

Albero: Arancio Albero

Melo: Melo Melo

Albero: Melo Melo

Albero: Arancio Arancio



Dipartimento di Elettronica e Informazione

20133 Milano (Italia)
Piazza Leonardo da Vinci,
32
Tel. (39) 02-2399.3400
Fax (39) 02-2399.3411

Giovanni Denaro

prof.

Carlo Ghezzi

prof.

Angelo Morzenti

prof.

Pierluigi San Pietro

I Prova di Ingegneria del Software

29 Aprile 2002

Cognome

Nome

Matricola

Sezione (segnarne una) Denaro Ghezzi Morzenti SanPietro

Istruzioni

- La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
- Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno presi in considerazione.
- È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
- Non è possibile lasciare l'aula conservando il tema della prova in corso.
- Tempo a disposizione: 3h.
- Punteggio totale a disposizione: 40 punti nominali (10 per ogni esercizio), correspondenti a 13/30 disponibili per la prima prova. La sufficienza corrisponde a circa 18 punti.

Esercizio 1:

Parte (a):

Si considerino le seguenti classi Punto2D e Punto3D, dichiarate in file separati Punto2D.java, Punto3D.java del package Geometria. Completare opportunamente le parti con i puntini, aggiungendo, ove necessario, modificatori di visibilità, dichiarazioni e/o implementazioni di metodi, ecc. Si trascuri l'implementazione di altri metodi oltre a distanza, equals e i costruttori (come ad esempio toString() e repOk()).

```

package Geometria;
public ..... class Punto2D {
    ..... //OVERVIEW: Tipo immutabile che rappresenta un punto in uno spazio
    ..... //euclideo a due dimensioni.
    ..... //Un punto tipico è individuato dalle sue coordinate cartesiane (x,y).
    ..... //AF(C) = (c.X, c.Y)
    ..... //R: true
    ..... //rep: basato su coordinate cartesiane
    ..... protected ..... float x, y; //protected perché x e y usate da Punto3D
    ..... this.x = x; this.y=y; .....//aggiungeremo mascheramento
}

public float distanza (Punto2D p) {
    ..... return (float) Math.sqrt((p.x-x) * (p.x-x) +(p.y-y) * (p.y-y));
}

public boolean equals(Punto2D p) {
    ..... return(x == p.x && y == p.y);
}

public float distanza (Punto2D p) {
    ..... return (float) Math.sqrt((p.x-x) * (p.x-x) +(p.y-y) * (p.y-y));
}

public boolean equals(Punto2D p) {
    ..... return(x == p.x && y == p.y);
}

package Geometria;
public ..... class Punto3D extends Punto2D {
    ..... //OVERVIEW: Tipo immutabile che rappresenta un punto in uno spazio
    ..... //euclideo a tre dimensioni.
    ..... //Un punto tipico è individuato dalle sue coordinate cartesiane (x,y,z).
    ..... protected ..... float z;
    ..... public Punto3D (float x, float y, float z) {
        ..... super(x,y); this.z=z;
        ..... // usare super(x,y) per inizializzare le componenti ereditate; altrimenti
        ..... // chiamato costruttore di default di Punto2D, che non esiste!
    }
}

/N.B. il metodo distanza che segue non è una ridefinizione di quello della
//classe Punto2D, in quanto i parametri hanno tipo diverso. Pertanto il
//metodo della classe Punto2D continua ad essere usabile, in quanto
//ereditato. Il metodo che segue dunque semplicemente realizza overloading.

public float distanza (Punto3D p) {
    ..... return (float) Math.sqrt((p.x-x) * (p.x-x) +(p.y-y) * (p.y-y) +(p.z-z) * (p.z-z));
}

public boolean equals (Punto2D p) {
    ..... if (p instanceof Punto3D) return equals((Punto3D) p);
    ..... return super.equals(p);
}

public boolean equals (Punto3D p) {

```

```

// il metodo è chiamato da equals(Punto2D): se assente, chiamata di
// equals con un Punto3D causa ricorsione infinita (che provoca Stack Overflow)
if (z==p.z) return super.equals(p);
//NB si puo' anche scrivere x == p.x &&y==p.y ecc. al posto di super.equals
return false;
}
.....
}

```

Alcune osservazioni:

L'esercizio precedente non è un esempio di buona progettazione, ma è servito solo allo scopo di verificare la vostra comprensione dei meccanismi della programmazione object oriented. L'ereditarietà è stata infatti usata in maniera impropria. Un punto a 3 dimensioni NON È un punto a 2 dimensioni; non ne conserva le proprietà.

Un'organizzazione migliore per questo stesso problema è quella dell'esercizio 2.

Si osservi che in generale sarebbe stato meglio che in Punto2D fosse ridefinita anche la equals(Object);

```

public boolean equals (Object o) {
    if (o instanceof Punto2D) return equals((Punto2D) o);
    return false;
}

```

In questo modo si evitano problemi di casting quando si chiama equals (v. esercizio 4, parte a). Ovviamente questo non era richiesto per l'esame.

```

Esercizio 1, Parte (b)
In un file separato di nome PuntoUser.java, contenuto in un altro package, è riportata la seguente classe (NB: si rammenta che import permette di importare una classe e utilizzarne i metodi pubblici senza doverlo specificare):
import Geometria.Punto2D, Geometria.Punto3D;

public class PuntoUser {
    public static void main(String args[]) {
        Punto2D p1,p2;
        Punto3D p3;
        p1 = new Punto2D(3,7);
        p2 = new Punto3D(3,7, 4); OK, assegnamento polimorfico
        System.out.println(p1.distanza(p2)); 0.0: Chiamato metodo di Punto2D
        System.out.println(p2.distanza(p1)); 0.0 Punto2D
        p3 = new Punto3D(6,7, 5);
        System.out.println(p2.distanza(p3)); 3.0 Punto2D
        System.out.println(p3.distanza(p1)); 3.0 Punto2D
        System.out.println(p1.distanza(p3)); 3.0 Punto2D
    }
}

Alcune osservazioni:
L'esercizio precedente non è un esempio di buona progettazione, ma è servito solo allo scopo di verificare la vostra comprensione dei meccanismi della programmazione object oriented. L'ereditarietà è stata infatti usata in maniera impropria. Un punto a 3 dimensioni NON È un punto a 2 dimensioni; non ne conserva le proprietà.
Un'organizzazione migliore per questo stesso problema è quella dell'esercizio 2.
Si osservi che in generale sarebbe stato meglio che in Punto2D fosse ridefinita anche la equals(Object);
public boolean equals (Object o) {
    if (o instanceof Punto2D) return equals((Punto2D) o);
    return false;
}
In questo modo si evitano problemi di casting quando si chiama equals (v. esercizio 4, parte a). Ovviamente questo non era richiesto per l'esame.

Punto3D p4 = new Punto3D(6,1, 5);
System.out.println(p3.distanza(p4)); 6.0 Punto3D
System.out.println(p1.x); scorretto: x è protected
p1 = p2; OK, assegnamento polimorfico
p3 = new Punto3D(); scorretto: manca costruttore di default
p3 = p1; scorretto: tipo statico di p1 (Punto2D) non è sottotipo del tipo statico di p3 : correggere in p3 = (Punto3D) p1; perché tipo dinamico di p1 è compatibile con Punto3D, che è il tipo statico di p3.
p3 = p2; scorretto: tipo statico di p2 non è sottotipo del tipo di p3.
Correggere in p3 = (Punto3D) p2; perché tipo dinamico di p2 è compatibile con Punto3D, che è il tipo statico di p3.
}
}

```

Tutte le linee sono sintatticamente corrette, ma ci possono essere errori legati alla visibilità e al controllo statico dei tipi. Segnare le linee non corrette, spiegandone brevemente il motivo, e proporre, se possibile, una correzione. Si ricorda che il metodo statico System.out.println() stampa in output il valore dell'espressione passata come argomento. Per ogni istruzione di stampa, riportare a fianco il risultato stampato durante l'esecuzione (sotto l'ipotesi di eliminazione dal codice di tutte le linee scorrette). Per ogni chiamata del metodo distanza indicare se viene chiamato il metodo definito in Punto2D o in Punto3D.

Esercizio 2,

Parte (a)

Definire un'interfaccia Punto, che rappresenta un punto in un generico spazio euclideo. Non vi è alcuna indicazione del numero di coordinate, ma deve essere previsto un metodo per la distanza.

public.....interface Punto

```
{  
    public.....float distanza (Punto p.....);  
    .....  
}
```

Riservare Punto2D e Punto3D in modo che siano implementazioni di Punto, e in modo che Punto3D non sia erede di Punto2D. Per brevità si trascurino i metodi equals, la Overview, AF e RI.

.....class Punto2D.....

```
.....  
.....  
.....  
.....  
.....  
.....  
}

```
.....
.....
.....
.....
.....
.....
}
```


```

Soluzione:

```
public interface Punto  
{  
    public float distanza (Punto p);  
}
```

```
public class Punto2D implements Punto {  
    private float x, y; //NB: anche protected accettabile  
    public Punto2D (float x, float y) {  
        this.x = x; this.y=y;  
    }  
    public float distanza (Punto p) {  
        if (! (p instanceof Punto2D)) throw new ClassCastException();  
        //NB: la linea precedente si poteva anche omettere perche' ClassCastException  
        // (eccezione unchecked) puo essere lanciata dal cast a Punto2D  
        Punto2D p1 = (Punto2D) p;  
        return (float) Math.sqrt((p1.x-x) * (p1.x-x) + (p1.y-y) * (p1.y-y));  
    }  
}
```

Esercizio 2. Parte (b)

Si ipotizzi che l'interfaccia Punto e le classi Punto2D e Punto3D siano incluse in un package Punti. In un file separato di nome PuntoUser2.java, contenuto in un altro package, è riportata la seguente classe:

```
import Punti.*;
public class PuntoUser2 {
    public static void main(String args[]) {
        Punto p1, p2, p3;
        p1 = new Punto2D(3,7);
        p2 = new Punto3D(3,7,9);
        p3 = new Punto3D(4,7,9);
        System.out.println(p2.distanza(p3));
        System.out.println(p1.distanza(p2));
        System.out.println(p2.distanza(p1));
    }
}
```

Quale è il risultato dell'esecuzione del codice?

Stampa 1, poi lancia eccezione ClassCastException in quanto p2 ha tipo apparente Punto compatibile con p1.distanza(Punto), ma tipo effettivo Punto3D è invece incompatibile e provoca eccezione in metodo distanza.

Se anche si eliminasse la seconda print, anche la terza print lancerebbe eccezione.

Esercizio 3
Sono date le seguenti specifiche della classe Segmento e della classe Spezzata, che sono dichiarate nel package Geometria. Le classi usano la classe Punto2D dell'esercizio 1 e una classe DuplicatePointException per le eccezioni, da non definire.

```
public class Segmento {
    //OVERVIEW: un segmento è una coppia ordinata di punti distinti. Tipo immutable.
    // Un segmento tipico è una coppia (inizio, fine) con:
    //|inizio|=fine, inizio e fine di tipo Punto2D.
    //costruttori:
    public Segmento(Punto2D inizio, Punto2D fine) throws DuplicatePointException
    //EFFECTS: if inizio.equals(fine) throw DuplicatePointException
    // else costruisce this come segmento di due punti: (inizio, fine)
    //observers:
    public float lunghezza() {
        //EFFECTS: restituisce la lunghezza di this
    }
    public Punto2D puntoIniziale()
    //EFFECTS: restituisce l'estremo iniziale di this
    public Punto2D puntoFinale()
    //EFFECTS: restituisce l'estremo finale di this
}

public class Spezzata {
    //OVERVIEW: una spezzata è un insieme ordinato di almeno 2 punti di tipo Punto2D, tutti distinti.
    // Una spezzata tipica è [p1, p2, ..., pn], corrispondente alla linea spezzata che unisce //p1, p2, ..., pn nell'ordine. Tipo mutable.
    //costruttori
    public Spezzata(Punto2D inizio, Punto2D fine) throws DuplicatePointException
    //EFFECTS: se Inizio != Fine restituisce una spezzata di due punti:
    //|(inizio, fine), altrimenti throw DuplicatePointException
    //observers:
    public Punto2D puntoIniziale(){}
    //EFFECTS: restituisce il punto di partenza p1 di this
    public Punto2D puntoFinale(){}
    //EFFECTS: restituisce il punto di arrivo pn di this
    public float lunghezza(){}
    //EFFECTS: restituisce la lunghezza complessiva di this
    public boolean puntoIncluso(Punto2D p){}
    //EFFECTS: ritorna true se p è equals a un punto di this, altrimenti false.
    //modifiers:
    public void addInizio(Punto2D p){} throws DuplicatePointException;
    //EFFECTS: this._post = [p, p1, ..., pn]; aggiunge p a this all'inizio (se p !=pi).
    public void addFine (Punto2D p){} throws DuplicatePointException;
    //EFFECTS: this._post = [p1, ..., pn, p]; aggiunge p a this in fondo (se p !=pi).
}
```

(l'esercizio continua sulla pagina successiva)

(continua l'Esercizio 3).....

Completere l'implementazione parziale della classe Spezzata descritta qui, in cui si usa come struttura dati un Vector di Segmento: il primo elemento del Vector è il segmento che unisce p1 a p2; il secondo è il segmento che unisce p2 a p3, ecc. Completezza la classe con:

- i modificatori di visibilità e la dichiarazione delle variabili.
 - il codice del costruttore e dei metodi (salvo che per il metodo puntoIncluso(), che non è riportato).
 - AF e RI, senza implementarli.
- Infine, verificare che RI è effettivamente un invariant per Spezzata.
- Si noti che non è necessario invece implementare la classe Segmento.

NB: la classe Vector del package java.util ha i seguenti metodi, di cui alcuni possono essere utili nell'implementazione di Spezzata:

```
public Object get(int i) : restituisce elemento in posizione i
public Object elementAt(int i) : come Object.get(int i)
public Object lastElement() : restituisce ultimo elemento del Vector
public Object firstElement() : restituisce primo elemento del Vector
public Iterator iterator() : restituisce generatore agli elementi del Vector
public int size() : restituisce il numero di elementi del Vector
public void add(Object o) : inserisce O come nuovo elemento alla fine del Vector
public void add(int i, Object o) : inserisce o come nuovo elemento in posizione i del Vector, shiftando opportunamente gli elementi a destra
```

```
import java.util.*;
public class Spezzata {
    //OVERVIEW (v. specifica)
    //AF(c) =
    ....
```

```
[segmenti.firstElement().puntoIniziale(), segmenti.get(0).puntoFinale(),
segmenti.get(1).puntoFinale(), ... , segmenti.lastElement().puntoFinale]
```

/attenzione: il Vector nel rep contiene dei segmenti, non dei punti, ma la specifica astratta è composta solo da punti!!!

```
//RI(c) =
segmenti != null && segmenti.size() > 0 &&
for all i, 0 <= i < segmenti.size(), segmenti.get(i) è un Segmento &&
for all i, 0 <= i < segmenti.size() - 1,
    segmenti.get(i).puntoFinale() == segmenti.get(i+1).puntoIniziale() &&
for all i,j, 0 <= i < segmenti.size() - 1,
    segmenti.get(i).puntoFinale() != segmenti.get(j).puntoFinale() & &
```

```
lung =  $\sum_{i=0}^{\text{segmenti.size()}-1} \text{segmenti.get}(i).\text{lunghezza}$ 
```

NB: la condizione che tutti i punti finali sono distinti può anche essere omessa, in quanto implicata dalla specifica della classe.....

```
.....
//rep:
private Vector segmenti;
```

```
private float lung;
```

```
//costruttore:
public Spezzata(Punto2D inizio, Punto2D fine) throws DuplicatePointException{
    Segmento s = new Segmento(inizio, fine);
    //costruttore di Segmento lancia già eccezione...
    segmenti = new Vector(1);
    //il Vector segmenti deve essere inizializzato!!!
    segmenti.add(s);
    lung = s.lunghezza(); // anche lung deve essere inizializzata!
```

```
}
```

//observers:

```
public Punto2D puntoIniziale(){
    return ((Segmento) segmenti.firstElement()).puntoIniziale();
}
//attenzione: il Vector contiene dei segmenti, non dei punti!!!
public Punto2D puntoFinale(){
    return ((Segmento) segmenti.lastElement()).puntoFinale();
}
public float lunghezza () {return lung;} //l'esercizio continua sulla pagina successiva)
```

NB: il senso inteso di "lunghezza" di una spezzata è quello di somma delle lunghezze dei segmenti che la compongono (la lunghezza di un segmento è la distanza fra i suoi due punti estremi). Si considera comunque accettabile anche la soluzione in cui la lunghezza di una spezzata è pari al numero di segmenti di cui è composta, anche se questa interpretazione è meno naturale.

(continua l'Esercizio 3).....

```
//Modifiers:
public void addInizio(Punto2D p) throws DuplicatePointException {
    if (puntoIncluso(p)) throw new DuplicatePointException("Segmenti addInizio");
    //Inserisce il nuovo segmento all'inizio del Vettore:
    Segmento s = new Segmento(p, puntoIniziale());
    segmenti.add(0,s);
    .....
    lung = lung + s.lunghezza(); //aggiornare lung!!!
}
public void addFine (Punto2D p) throws DuplicatePointException {
    if (puntoIncluso(p)) throw new DuplicatePointException("Segmenti addFine");
    //Inserisce il nuovo segmento alla fine del Vettore:
    Segmento s = new Segmento(puntoFinale(),p);
    segmenti.add(s);
    lung = lung + s.lunghezza();
}
.....
.....
.....
}
```

public Iterator punti() { return new Puntilter (this);}

```
private static class Puntilter implements Iterator {
    int posizione;
    Vector elementi;
```

```
Puntilter(Spezzata s) { posizione = -1; elementi = s.segmenti();}
```

```
public Object next() throws NoSuchElementException {
    if (posizione >= elementi.size()) throw new NoSuchElementException();
    //il throw qui sopra è previsto dalla specifica di next()
    if (posizione++ == -1) return null;
    (Segmento)elementi.firstElement().puntoIniziale();
    return ((Segmento)elementi.get(posizione)).puntoFinale();
}

public boolean hasNext() { return (posizione < elementi.size());}
public void remove() {throw new UnsupportedOperationException();}
//NB: la remove poteva essere omessa.
```

}

//class Spezzata

Si osservi che la soluzione qui illustrata è solo un esempio fra le tante possibili, più o meno semplici. L'importante è che:

- 1) next() restituisca i punti, e non i segmenti, componenti una spezzata. Non ci devono essere ripetizioni (pes. restituisca tutti i punti iniziali e l'ultimo punto finale)
- 2) next() lanci l'eccezione NoSuchElementException quando non ci sono punti.

Esercizio 4

- a) Supponre che la classe Spezzata dell'esercizio 3 fornisca un metodo iteratore punti() che restituisce un generatore per la classe Spezzata. Il generatore restituisce i punti di una Spezzata nello stesso ordine in cui i punti vi sono presenti. Utilizzare l'iteratore così definito per implementare il metodo puntoIncluso() della classe Spezzata, supponendo di usare per il confronto fra punti il metodo public boolean equals(Punto2D p) definito in Punto2D.
- b) Implementare il metodo iteratore punti() e la classe del generatore, senza il metodo remove().

```
class Spezzata {...{
    public boolean puntoIncluso(Punto2D p) {
        //NB: non è necessario, ma non è errato, fare riflessione o try...catch di
        //l'effetto S: ritorna true se p è equals a un punto di this, altrimenti false.
        for (Iterator i = punti(); i.hasNext();)
            if (p.equals((Punto2D) i.next())) return true;
        //NB: serve cast perché Punto2D non ha ridefinito equals( Object )
        //è corretto anche scrivere: ((Punto2D) i.next()).equals(p)
        return false;
    }

    public Iterator punti() { return new Puntilter (this);}

    private static class Puntilter implements Iterator {
        int posizione;
        Vector elementi;
        Puntilter(Spezzata s) { posizione = -1; elementi = s.segmenti();}

        public Object next() throws NoSuchElementException {
            if (posizione >= elementi.size()) throw new NoSuchElementException();
            //il throw qui sopra è previsto dalla specifica di next()
            if (posizione++ == -1) return null;
            (Segmento)elementi.firstElement().puntoIniziale();
            return ((Segmento)elementi.get(posizione)).puntoFinale();
        }

        public boolean hasNext() { return (posizione < elementi.size());}
        public void remove() {throw new UnsupportedOperationException();}
        //NB: la remove poteva essere omessa.
```

}

//class Spezzata

Si osservi che la soluzione qui illustrata è solo un esempio fra le tante possibili,

- 1) next() restituisca i punti, e non i segmenti, componenti una spezzata. Non ci devono essere ripetizioni (pes. restituisca tutti i punti iniziali e l'ultimo punto finale)
- 2) next() lanci l'eccezione NoSuchElementException quando non ci sono punti.

Ingegneria del Software – a.a. 2005/06

Appello del 8 settembre 2006

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h30min.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
1

Esercizio 1 (punti 10)

L'ADT immutabile Carta fornisce un'astrazione per una carta da briscola: asso ("1"), due ("2"), tre ("3"), quattro ("4"), cinque ("5"), sei ("6"), sette ("7"), fante ("F"), cavallo ("C") e re ("R") di spade ("S"), coppe ("C"), bastoni ("B") e denari ("D") per un totale di quaranta possibili carte distinte. L'ADT *MazzoCarte* fornisce un'astrazione per un mazzo di carte da briscola, ossia di quaranta carte.

Un giocatore può pescare una carta dal mazzo usando il metodo *pesca()*. Questo metodo restituisce una carta (a caso) e la elimina dal mazzo. Quando un giocatore pesca l'ultima carta del mazzo, il metodo solleva anche ---dopo aver restituito la carta e dopo averla eliminata dal mazzo--- l'eccezione *ExceptionMazzoVuoto*. L'ADT fornisce anche i seguenti i metodi puri *nelMazzo* e *carteDelMazzo*:

```
//@ requires true ;
//@ ensures (*restituisce true se c è parte del mazzo*) ;
public /*@pure@*/ boolean nelMazzo(Carta c)
```

```
//@ requires true ;
//@ ensures (*restituisce il numero di carte rimaste nel mazzo*) ;
public /*@pure@*/ int carteDelMazzo()
```

a- Si scriva la specifica del metodo *pesca* tenendo presente che:

- ✓ Si assume che il mazzo debba contenere almeno una carta
- ✓ Il numero di carte nel mazzo viene decrementato di uno
- ✓ La carta pescata non è più presente nel mazzo
- ✓ La carta pescata è l'unica carta eliminata dal mazzo

//@ requires

//@ ensures

```
public Carta pesca()
```

```
public Carta pesca()
```

Soluzione:

```
requires carteDelMazzo() >=1
ensures   \old(nelMazzo(\result)) &&
           !nelMazzo(\result) &&
           carteDelMazzo() == \old(carteDelMazzo()) -1 &&
           (\forall Carta c; c != \result; nelMazzo(c) <==> \old(nelMazzo(c)));
```

b- Si trasformi la specifica del metodo *pesca* eliminando l'assunzione che il mazzo non sia vuoto e includendo il lancio di un'eccezione *ExceptionMazzoVuoto*. Si mostri solo la parte di specifica modificata rispetto a quella definita in (a).

//@ requires

```
//@ ensures
```

```
public Carta pesca() throws MazzoVuotoException
```

Soluzione:

```
requires true;
```

Aggiungere alla ensures: `\old{carteNelMazzo()} >= 1`

Aggiungere la clausola:

```
signals(MazzoVuotoException e) \old{carteNelMazzo()} == 0 ;
```

c- Sapendo che l'ADT *Carta* fornisce i seguenti metodi puri per conoscere il valore e il seme di una carta:

```
//@ requires true ;
//@ ensures (*restituisce il valore di this*) ;
public /*@pure@*/ char valore()
```

```
//@ requires true ;
//@ ensures (*restituisce il seme di this*)
public /*@pure@*/ char seme()
```

Si scriva l'invariante pubblico per la classe *MazzoCarte* per imporre che un mazzo non può avere carte ripetute.

```
//@public invariant
```

Soluzione: non ci possono essere due reference c1 e c2, c1 != c2, a carte con stesso valore e seme:

```
public invariant
```

```
(\forall Carta c1; nelMazzo(c1);
  (\forall Carta c2; nelMazzo(c2); c1.valore() != c2.valore() || c1.seme() != c2.seme() || c1==c2));
```

Esercizio 2 (punti 7)

Supponendo che sia definita la classe :

```
public class InsiemeInteri  
    private int[] insieme;
```

si scriva in Java un opportuno iteratore che restituisca in sequenza i numeri primi contenuti nell'array, definendo sia il metodo *primi()* di *InsiemeNumeri* che restituisce l'iteratore, sia la classe *PrimiGen* che definisce l'iteratore (ossia il generatore). L'iteratore deve scandire gli elementi dell'array e restituire solo i numeri primi. Ad esempio, se l'array contenesse i numeri 5, 7, 23, 45, 13, 48, 17, l'iteratore dovrebbe restituire i numeri 5, 7, 23, 13, e 17.

```
public Iterator<Integer> primi() {  
    ....  
    ....  
    ....  
}
```

Classe del generatore :

```
private static class PrimiGen .....
```

```
}
```

Soluzione :

```
public Iterator<Integer> primi() {
    return new PrimiGen(this);
}

private static class PrimiGen implements Iterator<Integer> {
    private InsiemeInteri i ;
    private int n ;
    PrimiGen(InsiemeInteri ins){
        i=ins ;
        n=0 ;
        while(n<i.insieme.length && !isPrime(i.insieme[n])) n++ ;
            /* isPrime metodo statico implementato in modo ovvio */
    }

    public boolean hasNext() { return n<i.insieme.length ;}

    public Integer next() throws NoSuchElementException {
        Integer res ;
        if (this.hasNext()) res = i.insieme[n] ;
        else throw new NoSuchElementException("PrimiGen") ;
        while(n<i.insieme.length && !isPrime(i.insieme[n])) n++ ;
        return res ;
    }
}
```

Esercizio 3 (punti 8)

Si consideri una classe Java **MiaClasse**:

```
public class MiaClasse {  
    protected static int num = 0;  
    protected int val;  
    public MiaClasse(int a){  
        val = a;  
        num += a;  
    }  
}
```

a- Supponendo di creare due oggetti o1 e o2 in sequenza:

```
o1 = new MiaClasse(5);  
o2 = new MiaClasse(3);
```

quali valori assumono i campi val e num dei due oggetti?

Sol.

```
o1 = new MiaClasse(5);      val = 5 e num = 5  
o2 = new MiaClasse(3);      val = 3 e num = 8
```

b- Aggiungendo due sottoclassi **MiaSottoclasse1** e **MiaSottoclasse2**:

```
public class MiaSottoclasse1 extends MiaClasse {...}  
public class MiaSottoclasse2 extends MiaClasse {...}
```

```
MiaClasse a;  
MiaSottoclasse1 b;  
MiaSottoclasse2 c;
```

Quali delle seguenti righe di codice darebbe un errore in compilazione? (mettere una X per ogni riga che si ritiene scorretta aggiungendo una breve spiegazione)

```
a = new MiaClasse();  
b = new MiaClasse();  
c = new MiaSottoclasse1();  
b = a;  
a = b;  
b = new MiaSottoclasse2();
```

Sol.

```
a = new MiaClasse();  
b = new MiaClasse();      X assegnamento di un oggetto a una variabile di una sottoclasse  
                           della sua classe  
c = new MiaSottoclasse1(); X assegnamento di un oggetto a una variabile di una classe non  
                           antenato della sua classe  
b = a; X assegnamento di una variabile a una variabile di una sottoclasse della sua classe  
a = b;  
b = new MiaSottoclasse2(); X assegnamento di un oggetto a una variabile di una classe non  
                           antenato della sua classe
```

Appello 2 Settembre 2016



Politecnico di Milano
Anno accademico 2015-2016

Ingegneria del Software

Cognome:

LAUREANDO

Nome:

Matricola:

Sezione (segnarne una): Ghezzi

Mottola

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Esercizio 1

Una TabellaSemiordinata è un array associativo che mappa indici interi positivi a valori di un tipo Dato. Per il tipo Dato esiste una relazione di ordinamento definita da un opportuno ComparatoreDato che implementa l'interfaccia standard Comparator. Ad esempio, se d1 e d2 sono istanze di Dato, e comparatore è un'istanza di ComparatoreDato, allora comparatore.compare(d1, d2) ritorna un intero negativo se d1 precede d2, un intero positivo se d2 precede d1, o zero se d1 e d2 sono uguali.

Una istanza di TabellaSemiordinata deve garantire che quando viene aggiunto un nuovo elemento dNew in un indice i già occupato da un elemento dOld, l'elemento pre-esistente viene ri-assegnato al primo indice libero inferiore a quello originale se dOld precede dNew, oppure al primo indice libero superiore a quello originale se dNew precede dOld. Se invece il nuovo elemento dNew è uguale a dOld, la tabella non subisce variazioni.

L'interfaccia di TabellaSemiordinata è riportata di seguito.

```
public interface TabellaSemiordinata {  
  
    boolean indiceOccupato(int d);  
  
    void aggiungi(Dato d);  
  
    void rimuovi(Dato d);  
  
    Dato ritorna(int d);  
}
```

Vi viene chiesto di:

1. Fornire la specifica JML di tutti i metodi definiti in TabellaSemiordinata.
2. Supponendo che la classe MiaTabella implementi TabellaSemiordinata utilizzando un oggetto map di tipo Hasmap, specificare la funzione di astrazione come private invariant.

SOLUZIONE

1. Specifica JML dei metodi: totalleftmargin=-2cm

```
public interface TabellaSemiordinata {  
  
    //@requires d>=0  
    //@ensures \result <==> ritorna(d)!=null  
    boolean /*@ pure */ indiceOccupato(int d);  
  
    /*@ requires d!=null && i>=0  
     * ensures indiceOccupato(i) && ritorna(i).equals(d) &&  
     * (!\old(indiceOccupato(i)) ==>  
     *   (\forall int idx; idx >= 0 && idx != i; indiceOccupato(idx) <==> \old(indiceOccupato(idx)) &&  
     *      indiceOccupato(idx) ==> ritorna(idx).equals(\old(ritorna(idx)))) ) &&  
     *   (\!\old(indiceOccupato(i)) ==>  
     *     (comparatore.compare(d,\old(ritorna(i))) < 0 ==> (indiceOccupato(L) &&  
     *       ritorna(L).equals(\old(ritorna(i))) &&  
     *       (\forall int idx; idx >= 0 && idx != L; indiceOccupato(idx) <==> \old(indiceOccupato(idx))  
     *          indiceOccupato(idx) ==> ritorna(idx).equals(\old(ritorna(idx)))) ) &&  
     *     (comparatore.compare(d,\old(ritorna(i))) > 0 ==> (indiceOccupato(R) &&  
     *       ritorna(R).equals(\old(ritorna(i))) &&  
     *       (\forall int idx; idx >= 0 && idx != R; indiceOccupato(idx) <==> \old(indiceOccupato(idx))  
     *          indiceOccupato(idx) ==> ritorna(idx).equals(\old(ritorna(idx)))) ) &&  
     *     (comparatore.compare(d,\old(ritorna(i))) == 0 ==>  
     *       (\forall int idx; idx >= 0 ; indiceOccupato(idx) <==> \old(indiceOccupato(idx)) &&  
     *          indiceOccupato(idx) ==> ritorna(idx).equals(\old(ritorna(idx)))) ))  
     * where L is \max(int idx; idx < i ; !\old(indiceOccupato(idx)))  
     * and R is \min(int idx; idx > i ; !\old(indiceOccupato(idx)))  
    */  
    void aggiungi(Dato d);  
}
```

```

//@ requires d!=null
//@ ensures (\forall int i; i>=0 \&& \old(ritorna(i)).equals(d); !indiceOccupato(i)) &&
//@ (\forall int i; i>=0 \&& !\old(ritorna(i)).equals(d); indiceOccupato(i) <==> \old(indiceOccupato(i)) &&
//@ indiceOccupato(i) ==> ritorna(i).equals(\old(ritorna(i)))) &&
void rimuovi(Dato d);

//@ requires d >= 0
//@ ensures \result==null <==> !indiceOccupato(d)
Dato /*@ pure @*/ ritorna(int d);
}

```

2. Funzione di astrazione (come private invariant) della classe `MiaTabella`:

```

public class MiaTabella implements TabellaSemiordinata {

    //@ private invariant map != null && !map.values().contains(null) &&
    //@ (\forall int i; map.keySet().contains(i); i>=0) &&
    //@ (\forall int i;; map.keySet().contains(i) <==> indiceOccupato(i) &&
    //@ (map.keySet().contains(i) ==> map.get(i).equals(ritorna(i))) &&
private Map<Integer,Dato> map = new HashMap<Integer,Dato>();

    //...
}

```

Esercizio 2

Il sistema informativo di un'università deve gestire le informazioni relative ai propri dipendenti, dipartimenti, istituti, corsi erogati, e progetti di ricerca.

Le informazioni relative ai dipendenti comprendono la normale anagrafica (nome, cognome, data di nascita, ...). I dipendenti si dividono in ricercatori, attivi in un dato campo di ricerca, e personale amministrativo. Un dipendente può essere chiamato a dirigere un determinato dipartimento, che a sua volta comprende uno o più istituti.

I ricercatori fanno parte di uno o più istituti, e possono lavorare per un dato numero di ore al mese in determinati progetti di ricerca. Alcuni ricercatori sono anche impiegati come docenti di corsi erogati dall'università.

I corsi sono caratterizzati dal nome, il numero di ore totali, ed eventuali precedenze (obbligatorie o facoltative) rispetto ad altri corsi.

Vi viene richiesto di:

1. Creare un diagramma UML a vostra scelta per descrivere il sistema informativo.
2. Modificare il diagramma al punto precedente qualore il sistema informativo dovesse gestire anche le informazioni relative agli studenti e ai corsi già superati da uno studente.

Esercizio 3

Si consideri il seguente frammento di codice Java:

```

1. public static int mistero(int i, int x) {
2.     int k = 1;
3.     while (k<x && x<=3)
4.         if (i > k || x == 1)
5.             break;
6.         i = k/(x-1);
7.         k++;
8.     }
9.     return i;
}

```

1. Si definisca il diagramma del flusso di controllo.
2. Si definisca l'insieme *minimo* di casi di test, supponendo che sia i che x siano non negativi, per coprire:
(a) le istruzioni, (b) le decisioni (branch) e (c) le condizioni.

Qualora non si riesca ad arrivare ad una copertura completa, calcolare la percentuale di copertura.

SOLUZIONE

a) E' evidente che per entrare nel ciclo while, il valore di x deve essere 2 oppure 3. Per potere eseguire il break (istr. 5), occorre rendere vera la condizione dell'if: lo si puo' fare alla prima iterazione del ciclo, scegliendo un valore per la variabile i maggiore di k , ossia $x \geq 1$. Basta quindi scegliere ad esempio il caso (2,2) (ma anche (2,3) andrebbe bene).

Per potere eseguire le istruzioni 6 e 7, e' necessario che l'if sia falso, ossia che valga la condizione $i \leq k \ \&\ \& x != 1$: per entrare alla prima iterazione del ciclo (in cui $k=1$), basta quindi scegliere ad esempio $i=2$, ossia il caso (0,2) (ma anche (0,3), (1,2) e (1,3) andrebbero bene).

Questa scelta e' minimale, perche' non vi e' modo di rendere vero l'if dopo avere eseguito le istruzioni 6 e 7 (in quanto i diventa per forza minore di k), mentre dopo avere eseguito il break le istruzioni 6 e 7 saranno saltate. Quindi servono almeno due casi di test.

b) I due casi precedenti coprono anche il branch else "nascosto" dell'if. Sono quindi coperte anche tutte le diramazioni.

c) Con i due casi precedenti sono coperte anche le seguenti possibilita' per le costituenti delle condizioni:

```

k < x: TRUE, FALSE
x <= 3 : TRUE
i > k : FALSE, TRUE
x == 1 : FALSE
  
```

Per completare la copertura delle condizioni nell'AND nel while, serve coprire anche il caso $x <= 3 : FALSE$ (naturalmente in questo caso $k < x : TRUE$), per il quale basta ad es. selezionare (0,4).

Per completare la copertura delle condizioni nell'OR dell'If, servirebbe coprire il caso il caso $x == 1 : TRUE$. Questo e' pero' evidentemente impossibile (se x , che non e' modificata, fosse uguale a 1 non si entrerebbe nel while, in quanto $k <= x$ non sarebbe verificata). Quindi delle 8 possibilita' per le 4 costituenti delle condizioni, se ne coprono 7, ossia 87.5%.

Esercizio 4

Il seguente codice Java implementa una semplice versione della tradizionale interazione tra un thread produttore di dati ed un thread consumatore degli stessi dati, tramite una coda condivisa. La gestione delle eccezioni e' omessa per brevita'.

```

public class ProducerConsumer{
    public static void main(String args[]) {
        List<Integer> sharedQueue = new ArrayList<Integer>();
        Thread prodThread = new Thread(new Producer(sharedQueue), "Producer");
        Thread consThread = new Thread(new Consumer(sharedQueue), "Consumer");
        prodThread.start();
        consThread.start();
    }

    class Producer implements Runnable {
        private final List<Integer> sharedQueue;
        public Producer(List<Integer> sharedQueue) {
            this.sharedQueue = sharedQueue;
        }
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println("Produced: " + i);
            }
        }
    }
}
  
```

```

        produce(i);
    }
}
private synchronized void produce(int i) {
    sharedQueue.add(i);
    notifyAll();
}

class Consumer implements Runnable {
private final List<Integer> sharedQueue;
public Consumer(List<Integer> sharedQueue) {
    this.sharedQueue = sharedQueue;
}
public void run() {
    while (true) {
        System.out.println("Consumed: " + consume());
    }
}
private synchronized int consume() {
    while (sharedQueue.isEmpty())
        wait();
    notifyAll();
    return sharedQueue.remove(0);
}
}

```

Vi viene chiesto di:

- Argomentare con precisione il motivo per cui questa implementazione risulta corretta rispetto alla sincronizzazione di produttore e consumatore. Viceversa, se l'implementazione presenta qualche problema, descrivere con precisione il problema e a quali conseguenze può portare. Descrivere inoltre (mediante codice Java) le modifiche al codice necessarie per rimediare al problema. **SOLUZIONE**

I due thread non sono fra loro sincronizzati! Una soluzione e' la seguente: Nel Consumer:

```

.....
private int consume() throws InterruptedException {
    while (sharedQueue.isEmpty()) {
        synchronized (sharedQueue) {
            sharedQueue.wait();
        }
    }

    synchronized (sharedQueue) {
        sharedQueue.notifyAll();
        return sharedQueue.remove(0);
    }
}

```

e nel Producer:

```

.....
private void produce(int i) throws InterruptedException {

    synchronized (sharedQueue) {

```

```

        sharedQueue.add(i);
        sharedQueue.notifyAll();
    }
}

```

2. Independentemente dalla risposta alla domanda precedente, modificare l'implementazione in maniera che la coda condivisa non possa crescere arbitrariamente, ma possa al massimo contenere MAX elementi, dove MAX è una costante intera. **SOLUZIONE**

Si puo' modificare il producer in questo modo, per metterlo in attesa quando il numero di elementi prodotti supera il massimo: totaleftmargin=-2cm

```

private void produce(int i) throws InterruptedException {
    //wait if queue is full
    while (sharedQueue.size() == size) {
        synchronized (sharedQueue) {
            System.out.println("Queue is full " + Thread.currentThread().getName()
                + " is waiting , size: " + sharedQueue.size());
            sharedQueue.wait();
        }
    }

    //producing element and notify consumers
    synchronized (sharedQueue) {
        sharedQueue.add(i);
        sharedQueue.notifyAll();
    }
}

```

e corrispondentemente si modifica il consumer come segue: totaleftmargin=-2cm

```

...
private int consume() throws InterruptedException {
    //wait if queue is empty
    while (sharedQueue.isEmpty()) {
        synchronized (sharedQueue) {
            System.out.println("Queue is empty " + Thread.currentThread().getName()
                + " is waiting , size: " + sharedQueue.size());

            sharedQueue.wait();
        }
    }

    //Otherwise consume element and notify waiting producer
    synchronized (sharedQueue) {
        sharedQueue.notifyAll();
        return sharedQueue.remove(0);
    }
}

```

Nel rispondere alle domande, non ci si preoccupi della gestione delle eventuali eccezioni.

Esercizio 5

Trasformare il seguente codice Java in stile funzionale.

```

public static void numeroPari(final List<Integer> numeri) {

    Integer found = null;
    for (Integer i: numeri) {
        if (i % 2 == 0) {
            found = i;
            break;
        }
    }
}

```

```
        }
    }
    if (found != null) {
        System.out.print("Il primo numero pari " + found);
    } else {
        System.out.println("Nessun numero pari!");
    }
}
```

SOLUZIONE

Una soluzione semplice è:

```
package Functional;

import java.util.List;
import java.util.Optional;

public class Functions {
    public static void numeroPari(final List<Integer> numeri) {
        final Optional<Integer> trovato = numeri.stream().filter(i -> (i % 2 == 0)).findFirst();

        if (trovato.isPresent())
            System.out.println("Il primo numero pari" + trovato);
        else
            System.out.println("Nessun numero pari!");
    }
}
```

Ingegneria del Software – a.a. 2009/10

Appello del 23 Luglio 2010

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 100 punti nominali. La sufficienza si raggiunge indicativamente con 60 punti.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
.....

Esercizio 1 (punti 25)

E' data l'astrazione procedurale **calcolo**. Il metodo riceve come parametro un array x di almeno 2 interi e restituisce un array di interi, i cui elementi sono così costituiti:
il primo elemento è la somma di tutti gli elementi di x, il secondo è la somma dei primi $x.length - 1$ elementi di x, il terzo dei primi $x.length - 2$ ecc.

a- Si scrivano la pre- e post-condizione del metodo evidenziando tutti gli elementi significativi

SOL:

```
//@ requires (x != null) && (x.length >= 2)
```

```
/*@ ensures \result != null && \result.length == x.length &&
```

```
@     (\forall int i; i >= 0 && i < x.length; (\result[i]  
@           == (\sum int j; j >= 0 && j < (x.length - i); x[j])))  
@*/
```

```
static int[] calcolo(int[] x)
```

b- Si identifichino le eccezioni necessarie per rendere totale il metodo e si scrivano le opportune modifiche alla specifica di cui al punto (a).

SOL:

```
//@ requires true
```

```
/*@ ensures ((x != null) && (x.length >= 2)) =>
```

```
@     \result != null && \result.length == x.length &&  
@           (\forall int i; i >= 0 && i < x.length; \result[i]  
@               == (\sum int j; j >= 0 && j < (x.length - i); x[j])))  
@*/
```

```
//@ signals (NullPointerException e) (x == null)
```

```
//@ signals (ArrayTooSmallException e) ((x != null) && (x.length < 2))
```

Esercizio 2 (punti 50)

Si consideri la seguente specifica di una classe HeapInt.

```
public class HeapInt{
/* OVERVIEW: Collezione mutabile di oggetti, di tipo intero. */

//@ensures (*inizializza this alla HeapInt vuota*).
public HeapInt()
//osservatori puri:

//@ensures (* \result è la cardinalità di x in this, ossia il suo numero di comparsa *)
public /*@ pure @*/ int occurrences(int x)

//@ensures (* \result è la cardinalità di this, contando ogni elemento con la propria cardinalità *)
public /*@ pure @*/ int size()

//@requires size()>0;
//@ensures (\forall int y; occurrences(y)>0; \result>=y)
public /*@ pure @*/ int max()

//mutators:
//@ensures (* aggiunge una comparsa di x a this *);
public void insert(int x)

//@requires size()>0;
//@ ensures (* elimina e restituisce una comparsa dell'elemento massimo da this *)
public int extract()
}
```

Domanda A) Specificare in JML le postcondizioni dei metodi insert e extract

SOL:

```
//@ensures occurrences(x) == \old(occurrences(x)+1) && size() == \old(size()+1) &&
//@\(\forall int y; x!=y; occurrences(y) == \old(occurrences(y)));
public void insert(int x)

//@ensures \ result == \old(max()) && occurrences(\result) == \old(occurrences(\result)-1) &&
//@\ size() == \old(size()-1) && (\forall int y; \result!=y; occurrences(y) == \old(occurrences(y)));
public int extract()
```

Si consideri ora una classe HeapInt2, identica a HeapInt ma che modifica il metodo extract in modo che questo elimini tutte le comparse dell'elemento massimale.

Domanda B): Specificare in JML questa versione del metodo extract(),

Sol:

```
//@ensures \ result == \old(max()) && occurrences(\result) == 0 &&
//@ size() == \old(size()-occurrences(\result)) &&
//@(\forall int y; \result!=y; occurrences(y) == \old(occurrences(y)));
public int extract()
```

Domanda C) Dimostrare analiticamente, in base alle specifiche date, che HeapInt2 non può essere erede di HeapInt senza violare il principio di sostituzione.

SOL:

La dimostrazione è per assurdo. Detta post1 la postcondizione di HeapInt.extract() e detta post2 la postcondizione di HeapInt2.extract(), condizione necessaria per la verifica del principio è che post2 ==> post1. Ma se vale post1, allora vale $\text{occurrences}(\text{\result})=0$, mentre se vale post2 allora vale $\text{occurrences}(\text{\result})=\text{\old}(\text{occurrences}(\text{\result})-1)$.

Per uno heap in cui l'elemento max() ha cardinalità 2, da post1 deriva che $\text{occurrences}(\text{\result})=0$, mentre da post2 deriva che $\text{occurrences}(\text{\result})=1$. Allora se post2 implicasse post1 si dedurrebbe una contraddizione.

Si consideri nuovamente la classe *HeapInt* della Domanda A. È data un'implementazione realizzata nel modo seguente. Un attributo *heap* di tipo *ArrayList* memorizza gli elementi di *this* come *Integer*, senza ripetizioni, in ordine decrescente. Un attributo *comparse* di tipo *ArrayList* memorizza in *comparse.get(i)* il numero di comparse dell'elemento *heap.get(i)*. Il numero di comparse non può mai essere nullo o negativo.

Il rep quindi è:

```
private ArrayList<Integer> heap;
private ArrayList<Integer> comparse;
```

Data la seguente implementazione del costruttore:

```
public HeapInt() {
    heap = new ArrayList<Integer>;
    comparse = new ArrayList<Integer>;
}
```

Domanda D) Scrivere l'invariante di rappresentazione di *HeapInt*.

SOL:

```
//@private invariant heap !=null && comparse != null && heap.size() == comparse.size() &&
//@(\forall int i; 0< i && i< heap.size(); heap.get(i-1) > heap.get(i)) &&
//@(\forall int i; 0< i && i< comparse.size(); comparse.get(i) >0);
```

NB: la condizione che gli elementi di *heap* siano tutti distinti deriva dal fatto che sono in ordine strettamente decrescente.

Domanda E) Completare la seguente implementazione del metodo *extract()*

```
public int extract() {
    int massimo = this.heap.get(0); // memorizza in massimo il valore da restituire
    .....comparse.set(0,comparse.get(0)-1);
    .....if (comparse.get(0)==0 {
        .....comparse.remove(0);
        .....heap.remove(0);
    .....}
    return massimo;
}
```

Esercizio 3 (punti 25)

Implementare il seguente metodo iteratore *elementi()* della classe *HeapInt* dell'Es. 2. Si suggerisce di descrivere la struttura delle eventuali classi aggiuntive rispetto a *HeapInt*.

//@ensures (* restituisce un iteratore agli elementi di this, in ordine decrescente dal più grande al più piccolo. Gli elementi che compaiono ripetutamente sono restituiti un numero di volte pari alla loro cardinalità.*);

public Iterator<Integer> elementi()

SOL:

```
public class HeapInt {  
    private ArrayList<Integer> heap;  
    private ArrayList<Integer> comparse;  
    public Iterator<Integer> elementi() {return new HeapIterator();}  
    private class HeapIterator implements Iterator<Integer> {  
        private Integer current;  
        private int count;  
        private Integer next;  
        public HeapIterator() {  
            if (heap.size() == 0) {  
                this.count = comparse.get(0);  
                this.current = heap.get(0);  
            }  
        }  
        public boolean hasNext() {  
            if (current != null)  
                return true;  
        }  
        public Integer next() throws NoSuchElementException {  
            if (!hasNext()) {  
                throw new NoSuchElementException();  
            }  
            next = current;  
            count --;  
            if (!count) {  
                int nuovoIndice = heap.indexOf(current) + 1;  
                if (nuovoIndice == heap.size())  
                    current = null;  
                else {  
                    current = heap.get(nuovoIndice);  
                    count = comparse.get(nuovoIndice);  
                }  
            }  
            return next;  
        }  
        public void remove() throws IllegalStateException {  
            int index;  
            if (next == null) throw new IllegalStateException();  
            index = heap.indexOf(next);  
            if (comparse.get(index) == 1) {  
                comparse.remove(index);  
                heap.remove(index);  
            }  
            next = null;  
        }  
    }  
}
```

Politecnico di Milano

Ingegneria del Software – a.a. 2008/09

Appello del 02 Luglio 2009

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Total
.....

Esercizio 1 (punti 8)

Si consideri il metodo statico **controllaMatrice**. Questo ha due parametri m e N, con m, una matrice quadrata di interi NxN (con N>2). Il metodo restituisce true nei casi seguenti:

- Ogni casella m[i][j] sopra la diagonale principale, e con indice j maggiore di 2, contiene la somma delle due celle precedenti della stessa riga. Ad esempio, m[3][4] = m[3][2] + m[3][3]
- Ogni casella m[i][j] sotto la diagonale principale, e con i maggiore di 2, contiene la somma delle due celle precedenti della stessa colonna. Ad esempio, m[3][1] = m[1][1] + m[2][1]
- I numeri sulla diagonale principale, a partire dal terzo, rispettano una regola analoga: ogni elemento è la somma dei due precedenti (sulla diagonale). Ad esempio m[4][4] = m[2][2] + m[3][3]

Si noti che i primi due elementi delle prime due righe sono valori arbitrari.

a- Si scrivano la pre- e post-condizione del metodo evidenziandone tutti gli elementi significativi

```
//@requires N > 2 &&
    m != null && m.length == N &&
    (\forall int i; 0 <= i && i < N; m[i] != null && m[i].length == N );
//@ensures (\result == true) <==>
    (\forall int i; 2 <= i && i < N; (\forall int j; 2 <= j && j < N;
        ( (i == j) => (m[i][j] == m[i-1][j-1] + m[i-2][j-2]) ) &&
        ( (i > j)  => (m[i][j] == m[i-1][j] + m[i-2][j]) ) &&
        ( (i < j)  => (m[i][j] == m[i][j-1] + m[i][j-2]) )
    ));
```

```
public static boolean controllaMatrice(int[][] m, int N)
```

b- Si identifichino le eccezioni necessarie per rendere totale il metodo, **mostrandone come cambia la specifica del metodo.**

```
//@requires true;
//@ensures N > 2 &&
m != null && m.length == N &&
(\forall int i; 0 <= i && i < N; m[i] != null && m[i].length == N) &&
(\result == true) <==>
(\forall int i; 2 <= i && i < N; (\forall int j; 2 <= j && j < N;
  (i == j) => (m[i][j] == m[i-1][j-1] + m[i-2][j-2])) &&
  (i > j) => (m[i][j] == m[i-1][j] + m[i-2][j])) &&
  (i < j) => (m[i][j] == m[i][j-1] + m[i][j-2]));
);
//@signals (NullPointerException e) m == null ||
  (\exists int i; 0 <= i && i < N; m[i] == null);
//@signals (MatrixTooSmallException e) N <= 2;
//@signals (WrongMatrixException e) m != null &&
  (m.length != N || (\exists int i; 0 <= i && i < N;
    m[i] != null && m[i].length != N))
```

Esercizio 2 (punti 12)

L'azienda *happyCompany* deve preparare dei rendiconti per i propri dipendenti a progetto. Questo significa che ogni dipendente deve tenere traccia delle ore dedicate ogni giorno a ciascun progetto. Dopo vari tentativi rudimentali, *happyCompany* ha deciso di dotarsi di un'applicazione specifica.

Ogni **Dipendente** (si supponga che esista una classe con questo nome) è associato a un insieme di oggetti **RapportoGiornaliero** e questi vengono opportunamente aggiornati ogni mese. Ogni progetto corrisponde a un codice, che è un intero positivo, definito univocamente per quel progetto.

- a- La classe **RapportoGiornaliero** ha una data, osservabile con un metodo **public Data data()**, e un metodo **public void aggiungiOra(int prg, int oi)** per segnare che il dipendente relativo a quel rapporto ha lavorato al progetto di codice **prg** nell'ora che inizia da **oi**. Ad esempio, **aggiungiOra(345, 9)** serve per dire che dalle 9 alle 9.59 la persona ha lavorato al progetto 345. Una persona può lavorare nella stessa ora ad un solo progetto per volta e non può lavorare per più di 8 ore al giorno. Per ragioni di flessibilità, ogni dipendente può spalmare il proprio lavoro come crede sulle 24 ore, ma senza mai eccedere il limite suddetto. Si definiscano in JML la pre- e post-condizione del metodo **aggiungiOra** sapendo che la classe **RapportoGiornaliero** offre anche un metodo **public int[] rendiconto()** che restituisce un array di 24 interi (ogni casella, una per ogni ora della giornata, contiene il codice del progetto su cui si e' lavorato, oppure 0 se l'ora in questione non è stata lavorata).
- Si supponga che la classe Data abbia metodi **giorno()**, **mese()**, **anno()** con significato ovvio, che restituiscono ciascuno un int.

```
//@ requires prg != 0 && 0 <= oi && oi <= 23 &&
rendiconto()[oi] == 0 &&
(\numof int i; 0 <= i && i <= 23; rendiconto()[i] != 0) <= 7;
//@ensures rendiconto()[oi] == prg &&
(\forallall int i; 0 <= i && i <= 23;
(oi != i) => (rendiconto()[i] == \old(rendoconto()[i]) );
public void aggiungiOra(int prg, int oi)
```

b- La classe Dipendente offre i metodi:

public ArrayList<Integer> progetti(int mese, int anno) e
public ArrayList<RapportoGiornaliero> rapporti(int mese, int anno)
che restituiscono rispettivamente l'insieme dei progetti su cui il dipendente ha lavorato nel mese **mese**, e l'insieme di tutti i rapporti giornalieri del mese per il dipendente, come ArrayList di **RapportoGiornaliero**. Questi metodi servono per il consolidamento dei rapporti mensili, attraverso un ulteriore metodo della classe:

public void consolida(int mese, int anno)

che viene chiamato alla fine del **mese** per verificare che il dipendente non abbia lavorato più di 160 ore nel mese e abbia lavorato almeno 4 ore a ogni progetto dichiarato. Nel caso le due regole fossero violate, il metodo solleva le eccezioni **TroppoLavoroException** e **PocheOreException**, rispettivamente.

b1-*Si definiscano in JML la pre- e post-condizione, comprensive della gestione delle eccezioni, per il metodo **consolida()**.*

```
//@requires 0 <= mese && mese < 12;
//@ensures (* il dipendente non lavora per più di 160 ore / mese *) &&
    \sum(int i; 0 <= i && i < rapporti(mese,anno).size());
        \numof(int j; 0 <= j && j < 23;
            rapporti(mese,anno).get(i).rendiconto()[j] != 0
        ) ) <= 160 &&
        (* il dipendente lavora almeno 4 ore ad ogni progetto *) &&
        \forallall(int p; 0 <= p && p < progetti(mese,anno).size());
            \sum(int i; 0 <= i && i < rapporti(mese,anno).size();
                \numof(int j; 0 <= j && j < 23;
                    rapporti(mese,anno).get(i) .rendiconto()[j] ==
                    progetti(mese,anno).get(p)
                ) ) >= 4
            );
//@signals (TroppoLavoroException e)
    \sum(int i; 0 <= i && i < rapporti(mese,anno).size());
        \numof(int j; 0 <= j && j < 23;
            rapporti(mese,anno).get(i) .rendiconto()[j] != 0
        ) ) > 160;
//@signals (PocheOreException e)
    \exists(int p; 0 <= p && p < progetti(mese,anno).size());
        \sum(int i; 0 <= i && i < rapporti(mese,anno).size();
            \numof(int j; 0 <= j && j < 23;
                rapporti(mese,anno).get(i) .rendiconto()[j] ==
                progetti(mese,anno).get(p)
            ) ) < 4
        );
public void consolida(int mese, int anno)
    throws TroppoLavoroException, PocheOreException;
```

b2- Si definiscano in JML la pre- e post-condizione per il metodo **rapporti (mese, anno)**, assicurando la consistenza dei dati, e cioè che vengano restituiti solo i rapporti giornalieri del mese e dell'anno specificati, senza giorni duplicati.

Si ipotizza che chiedendo i rapporti per un giorno non lavorativo, venga restituita una lista vuota (non nulla)

```
//@ requires 0 <= mese && mese < 12;
//@ ensures \result != null &&
    \forall(int g; 0 <= g && g < \result.size());
        \result.get(g) != null &&
        \result.get(g).anno() == anno &&
        \result.get(g).data().mese() == mese &&
        \forall(int z; g < z && z < \result.size());
            \result.get(g).data().giorno() !=
            \result.get(z).data().giorno()
    );
public @/ pure @/ ArrayList<RapportoGiornaliero> rapporti(int mese, int
anno);
```

b3- Invece di specificare la proprietà di cui al punto b2 mediante pre e post-condizione, si sarebbe potuto esprimere la stessa mediante un invariante pubblico? Si o no? Perchè?

Sintatticamente è possibile definire la postcondizione in un invariante, in quanto il metodo rapporti è un metodo puro, e le proprietà che vengono espresse validano la consistenza dell'intero oggetto. L'invariante pubblico esprimerebbe le stesse proprietà, definendo che, per ogni mese e per ogni anno, valgono le post-condizioni specificate.

Tuttavia, in JML occorre comunque che sia l'invariante ad essere garantito in base alla specifica dei metodi pubblici, e non viceversa. Quindi l'invariante non può semplicemente rimpiazzare la postcondizione.

c- *Happy Company* ha molto lavoro e deve ricorrere a consulenti esterni. Per questo, la classe **Consulente** viene aggiunta al sistema come sottoclasse di **Dipendente**. Il consulente però non ha limiti di orario: può lavorare quante ore vuole su qualunque progetto, senza ovviamente superare le 24 ore al giorno. Considerando l'ipotesi precedente, il metodo **consolida()**, ridefinito per la sottoclasse, rispetta il principio di sostituzione? Motivare brevemente la risposta.

NO

Il metodo **consolida()**, pur rispettando la regola delle signature, violerebbe la regola dei metodi. La specifica del metodo in *Dipendente* assicura che se questo termina correttamente, allora il dipendente ha lavorato meno di 160 ore al mese; in caso contrario viene sollevata l'eccezione *TroppoLavoroException*.

Il testo richiede di definire un *Consulente*, in cui vengono rimossi i vincoli di orario. Tuttavia, viene richiesto che *Consulente* sia sottoclasse di *Dipendente*. Questo comporta che, nel ridefinire il metodo *consolida()*, la classe *Consulente* deve mantenere le promesse fatte nel medesimo metodo dalla classe *Dipendente*. Riassumendo, il metodo deve sia rimuovere i vincoli di orario (le nuove richieste) sia sollevare un'eccezione nel caso in cui il dipendente lavori più di 160 ore al mese. Si può notare che le due cose sono in palese contrasto, che è causato dalla violazione della regola dei metodi.

Esercizio 3 (punti 5)

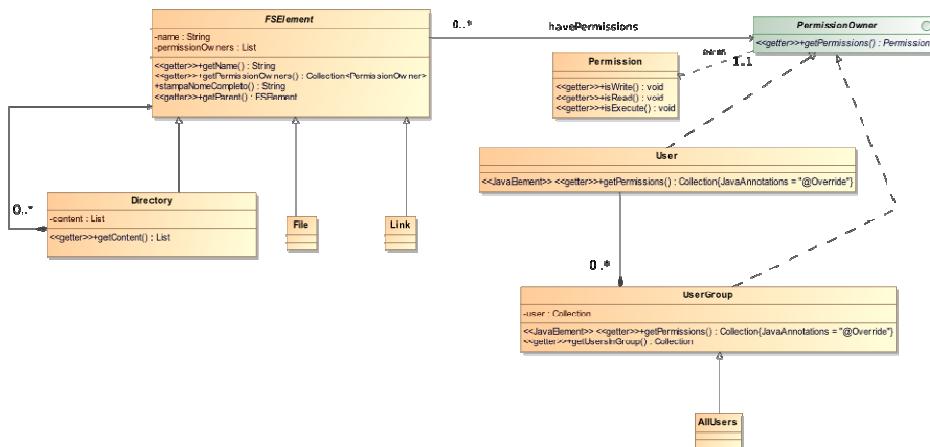
Si consideri la struttura tipica di un file system. Le directory sono organizzate gerarchicamente: ogni directory può contenere altre directory, file, oppure link. Un link è un riferimento a un file fisicamente memorizzato in un'altra directory; in questo modo il file riferito diventa virtualmente parte anche della directory che contiene il link. Una directory ha un nome. Ogni file è caratterizzato da un nome, una dimensione e un tipo. Un link ha un nome e un tipo.

Ogni elemento (directory, file o link) è associato con un insieme di diritti d'accesso: lettura, scrittura e esecuzione. Questi sono concessi al proprietario di una risorsa (un singolo utente), a gruppi di utenti o a tutti gli utenti.

- a- Si modelli il problema descritto con un diagramma delle classi UML, evidenziando i metodi e gli attributi principali delle diverse classi.
- b- In funzione delle classi identificate nel diagramma precedente, si scriva il corpo del metodo **stampaNomeCompleto** per scrivere a video il nome completo di un file partendo dalla radice del file system.
- c- Si scriva l'invariante privato della classe **Directory** per dire che un suo oggetto non può mai contenere più di 200 file e deve esistere almeno un utente in grado di leggere il contenuto della directory.

NOTA: Se si ritiene che la descrizione informale contenga ambiguità, si descriva a parole come il diagramma UML proposto risolve tale ambiguità.

3.a



3.b

```
public String stampaNomeCompleto(){
    FSElement parent = getParent();
    if (parent != null){
        return parent.stampaNomeCompleto() + "/" + name;
    } else{
        return name;
    }
}
```

3.c

```
//@ private invariant
\numof( int i; i >= 0 && i <= content.size());
    content.get(i) instanceof File) <= 200&&
\nexist(int j; j >= 0 && j < permissionOwners.size());
    permissionOwners.get(j) instanceof User &&
    permissionOwners.get(j).getPermissions().isRead());
```

Ingegneria del Software – a.a. 2008/09

Appello del 8 Luglio 2008

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
.....

Esercizio 1 (punti 12)

La classe **Magazzino** consente di gestire un semplice magazzino per lo stoccaggio delle merci con 1000 slot. Non interessa sapere quali merci vengono immagazzinate, ma solo la quantità totale di ciascun tipo di merce. Si ipotizzi per semplicità che la quantità sia misurabile in unità; essa non deve mai eccedere la capacità massima. La classe mette a disposizione i seguenti metodi observer:

quantita(int cod), restituisce il numero di unità della merce di codice cod presenti in magazzino
ArrayList<Integer> merci(), restituisce tutti e soli i codici di tutte le merci stoccate (senza restituirne la quantità)

domanda a

Fornire la specifica JML dei metodi seguenti, che rilanciano eccezioni nel caso di superamento dei limiti di capacità.

//@ requires qta >= 0;

//@ ensures

// precondizione spostata come postcondizione

//@ (\sum int i; i<=0 && i<\old(merci().size()); \old(quantita(merci().get(i)))) <=1000-qta &&

// Tutti i vecchi codici devono continuare ad esistere e se sono diversi da cod anche la quantità
// deve restare uguale

//@ (\forall int i; 0 <= i && i < \old(merci().size());

//@ (\exists int j; 0 <= j && j < merci().size(); \old(merci().get(i)) == merci().get(j) &&

//@ merci().get(j) != cod ==> \old(quantita(merci().get(j))) == quantita(merci().get(j))) &&

// La quantità di merce di codice cod deve essere incrementata di qta

//@ quantita(cod) == \old(quantita(cod)) + qta &&

// Se il cod non era presente, la lunghezza di merci deve essere incrementata di uno; altrimenti
// resta uguale. Questo dovrebbe bastare se facciamo l'ipotesi ovvia che i codici non siano
// ripetuti nella lista

//@ \old(merci().contains(cod)? merci().size() == \old(merci().size):

//@ merci().size() == \old(merci().size + 1;

//@ signals (CapacitaRaggiuntaException e)

//@ (\sum int i; i<=0 && i<\old(merci().size()); \old(quantita(merci().get(i)))) > 1000-qta;

public void aggiungi(int cod, int qta) throws CapacitaRaggiuntaException

//@ requires qta >= 0;

//@ ensures

// precondizione spostata come postcondizione

//@ \old(quantita(cod))>= qta &&

// Tutti i nuovi codici devono continuare ad esistere e se sono diversi da cod anche la quantità
// deve restare uguale

//@ (\forall int i; 0 <= i && i < merci().size();

//@ merci().get(j) != cod ==> (\exists int j; 0 <= j && j < \old(merci().size());

//@ merci().get(i) == \old(merci().get(j)) &&

```

//@ \old(quantita(merci().get(j))) == quantita(merci().get(j))) &&
// La quantita di merce di codice cod deve essere decrementata di qta
//@ quantita(cod) == \old(quantita(cod)) - qta &&
// Se la quantita di cod diventa zero, la lunghezza di merci deve essere decrementata di uno;
// altrimenti resta uguale. Questo dovrebbe bastare se facciamo l'ipotesi ovvia che i codici
// non siano ripetuti nella lista
//@ quantita(cod)==0? merci().size() == \old(merci().size()-1:
//@           merci().size() == \old(merci().size);

```

public void preleva(int cod, int qta) throws mercelnsufficientException

domanda b

La classe **MagazzinoVirtuale** fornisce le stesse operazioni di **Magazzino**, ma senza limite di stoccaggio. Dopo aver specificato in JML il metodo **aggiungi(int cod, int qta)**, opportunamente definito, spiegare brevemente e chiaramente se è possibile definire **MagazzinoVirtuale** come estensione di **Magazzino**, rispettando il principio di sostituzione.

//@ requires qta >= 0;

//@ ensures

// incremento di quantita della sola merce con codice cod

```

//@ (\forall int i; i<=0 && i<merci().size());
//@   merci().get(i) == cod? quantita(merci().get(i)) == \old(quantita(cod))+qta :
//@           quantita(merci().get(i)) == \old(quantita(merci().get(i))) &&

```

// eventualmente aggiungo solo cod alla lista di codici

```

//@ !(\exists int j; j<=0 && j <\old(merci().size()));
//@   \old(merci().get(j)==merci().get(i)) ==> merci().get(i)==cod) &&

```

// e lo faccio una volta sola

```

//@ merci().size()-\old(merci().size()) <= 1;
public void aggiungi(int cod, int qta) throws CapacitaRaggiuntaException

```

Non è possibile rispettare il principio di sostituzione.

La regola delle segnature sarebbe in realtà verificata, in quanto tutti i metodi di MagazzinoVirtuale hanno la stessa segnatura che in Magazzino, salvo l'assenza dell'eccezione CapacitaRaggiuntaException.

Tuttavia, la postcondizione completa del metodo aggiungi() include anche la parte signals, ossia il lancio dell'eccezione al raggiungimento di 1000 unità. Formalmente, la postcondizione di Magazzino.aggiungi è del tipo:

(CapacitaOK&& aggiornaQuantita && nonLanciareCapacitaRaggiuntaException) ||
 (!!CapacitaOK && quantitaInvariata && lancia CapacitaRaggiuntaException)

La postcondizione di MagazzinoVirtuale.aggiungi è invece del tipo:

aggiornaQuantita && nonLanciareCapacitaRaggiuntaException

Ma questa condizione non implica la postcondizione precedente (in quanto ci sono casi in cui CapacitaOK è falso). Pertanto, si viola la regola dei metodi.

Ad esempio, la nuova classe “sorprenderebbe” il codice che si aspettasse di inserire 1000 oggetti per poi ricevere un’eccezione. Ad esempio, il codice seguente inserisce 1000 oggetti in magazzino.

```
public void riempi(Magazzino m,int cod) {  
try {  
    for (;;) m.aggiungi(cod,1);  
catch { CapacitaRaggiuntaException e}  
}
```

Se chiamata su un oggetto di tipo MagazzinoVirtuale, il codice risulterebbe in un loop infinito.

domanda c

c1- Si consideri un’altra versione della classe **Magazzino**, in cui la specifica del metodo **aggiungi** non prevede il lancio dell’eccezione **CapacitaRaggiuntaException**: il superamento del limite è considerato un prerequisito sul chiamante. Scrivere la specifica del metodo **aggiungi** in questa nuova versione:

```
//@ requires qta >= 0 &&  
//@ (\sum int i; i<=0 && i<\old(merci().size()); \old(quantita(merci().get(i)))) <=1000-qta  
  
//@ ensures  
// incremento di quantita della sola merce con codice cod  
  
//@ (\forallall int i; i<=0 && i<merci().size());  
//@ merci().get(i) == cod? quantita(merci().get(i)) == \old(quantita(cod))+qta :  
//@ quantita(merci().get(i)) == \old(quantita(merci().get(i))) &&  
  
// eventualmente aggiungo solo cod alla lista di codici  
  
//@ !(\exists exists int j; j<=0 && j <\old(merci().size()));  
//@ \old(merci().get(j)==merci().get(i)) ==> merci().get(i)==cod) &&  
  
// e lo faccio una volta sola  
  
//@ merci().size()-\old(merci().size()) <= 1;  
  
public void aggiungi(int cod, int qta)
```

c2- Si consideri ora la stessa classe MagazzinoVirtuale specificata al punto (b). Se questa fosse definita come estensione di Magazzino, il principio di sostituzione sarebbe verificato? Motivare la risposta.

Sì, in quanto il metodo aggiungi di MagazzinoVirtuale indebolisce la precondizione, mantendo invariata la postcondizione.

Esercizio 2 (punti 7)

Si consideri il seguente metodo statico

```
public static boolean triangolare (int[][] m, char t)
```

che presa in ingresso una matrice quadrata, restituisce true se t vale ‘s’ e la matrice è una matrice triangolare superiore (ovvero, tutti gli elementi al di sotto della diagonale principale sono uguali a zero), oppure se t vale ‘i’ e la matrice è una matrice triangolare inferiore (tutti gli elementi al di sopra della diagonale principale sono uguali a zero).

Si definisca la specifica del metodo statico senza considerare possibili eccezioni

```
//@ requires
// matrice non nulla o vuota:

//@ m != null && m.length > 0

// matrice quadrata senza righe nulle:

//@ (\forall int i; i <= 0 && i < m.length; m[i] != null && m[i].length == m.length) &&

// carattere corretto:

//@ t == 's' || t == 'i';

//@ ensures

//@ t == 's' &&
//      (\forall int i; 0 < i && i < m.size();
//       (\forall int j; 0 <= j && j < i; m[i][j] == 0))
// || t == 'i' &&
//      (\forall int i; 0 <= i && i < m.size() - 1;
//       (\forall int j; i < j && j < m.size(); m[i][j] == 0));
// la matrice non e' modificata al termine:
//@ assignable nothing;
```

Si aggiungano le relative eccezioni, e si modifichino le pre- e post-condizioni ove necessario, per rendere il metodo totale.

Si dichiari che il metodo può lanciare le eccezioni `NullPointerException`, `EmptyException`, `NonQuadrataException`, `CarattereException` si tolga la `requires` e si aggiungano:

`m!=null && (\forall int i; 0 <= i && i < m.size(); m[i].length == m.length) && (t == 's' || t == 'i')`
in AND alla `ensures`, e inoltre le `signals`:

```
signals (NullPointerException e) m == null || (\exists int i; 0 < i && i < m.length; m[i] == null);
signals (EmptyException e) m.length == 0;
signals (NonQuadrataException e) m != null && !(\forall int i; 0 < i && i < m.length; m[i].length == m.length);
signals (CarattereException e) !(t == 's' || t == 'i')
```

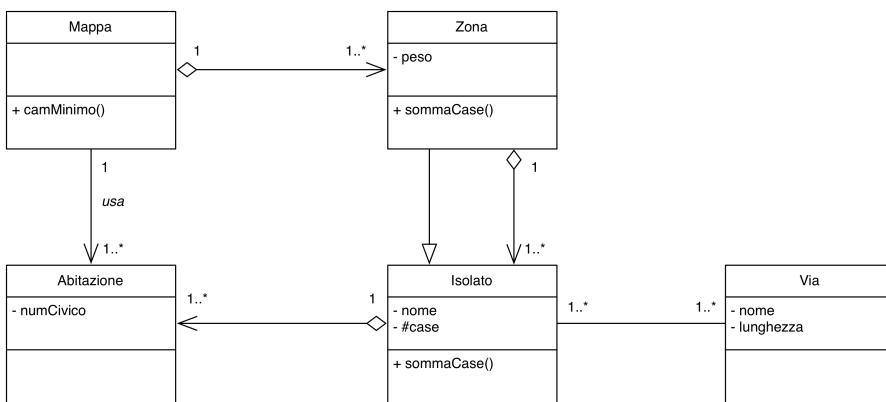
Esercizio 3 (punti 6)

La mappa di una città può essere vista a diversi livelli gerarchici. In prima approssimazione possiamo pensare che sia un insieme di zone connesse da vie principali. Ogni zona ha un nome, un peso e al proprio interno può contenere altre zone e/o isolati. Ogni via ha un nome e una lunghezza, connette due o più zone, ma può anche collegare (separare) due o più isolati. Ogni isolato è identificato da un nome e dal numero di case (numeri civici) in esso contenute. Una via può solo collegare isolati della stessa zona, oppure zone diverse, ma non isolati di zone diverse.

Ogni zona deve poter conoscere la somma delle case in essa contenute, direttamente o transitivamente. La mappa deve essere in grado di calcolare il cammino minimo (rispetto alla somma delle lunghezze delle vie traversate) tra due abitazioni di due zone qualsiasi.

domanda a

Descrivere con un diagramma delle classi UML la specifica sopra riportata, aggiungendo anche tutti e soli gli attributi e metodi rilevanti per la specifica.



domanda b

Scrivere la specifica JML del modo che calcola il numero di case in una zona, in funzione del modello UML definito al punto precedente.

Supponendo di usare un array zone per implementare l'aggregazione tra Zona e Isolato, la postcondizione sarebbe:

```
//@ ensures \result == (\sum int i; i>=0 && i< zone.length; zone[i].sommaCase());
```

Appello del 12 Settembre 2018



Politecnico di Milano
Anno accademico 2017-2018

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una): <input type="checkbox"/> Cugola	<input type="checkbox"/> Mottola
	<input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente specifica di una classe Sequence.

```
public class Sequence<T> {
    //una sequenza di elementi di tipo T.

    //inizializza alla sequenza vuota
    public Sequence() {
        //inserisce info all'inizio della sequenza ipotizzando che key non esista
        public void enter(String key, T info) {
            //cancella l'elemento con chiave key
            public void delete(String key) {
                //modifica l'info dell'elemento con chiave key ipotizzando che tale elemento esista
                public void change(String key, T info);
                //restituisce l'elemento con chiave key. Restituisce null se key non e' presente
                public /*@ pure */ T lookup(String key)
                //restituisce tutte le chiavi nell'ordine della sequenza
                public /*@ pure */ ArrayList<String> allKeys();
                //restituisce il numero di elementi nella sequenza
                public /*@ pure */ int size();
            }
        }
    }
}
```

Domanda a)

Si scriva un invarianto pubblico per la classe. Si specifichino inoltre in JML i metodi `enter` e `change`. Si argomenti che l'invariante è effettivamente verificato.

Soluzione

```
public invariant  (\forall String key; allKeys().contains(key));
                  lookup(key)!= null &&
                  (\forall key'; allKeys().contains(key'); key'!=key;))

//@requires key!= null && T!= null && !allKeys().contains(key);
//@ensures  lookup(key)==info && allKeys().get(0)==key &&
//           allKeys().size()==\old(allKeys().size()+1) &&
//           (\forall key'; \old(allKeys().contains(key')));
//                  allKeys().indexOf(key')==1+\old(allKeys().indexOf(key')) &&
//                  lookup(key')==\old(lookup(key'));
public void enter(String key, T info);

//@requires key!= null && allKeys().contains(key);
//@ensures  lookup(key)==info &&
//           allKeys().size()==\old(allKeys().size()) &&
//           (\forall key'; \old(allKeys().contains(key')));
//                  allKeys().indexOf(key')==\old(allKeys().indexOf(key')) &&
//                  key!=key' ==> lookup(key')==\old(lookup(key'));
public void change(String key, T info);
```

Il costruttore costruisce una sequenza vuota, che verifica banalmente l'invariante. Gli unici metodi che potrebbero inserire un duplicato sono `enter`, `change`, ma la loro requires lo impedisce. Le loro postcondizioni includono l'asserzione `lookup(key)==info`, lasciando poi immutato il resto della sequenza, quindi anche la condizione `lookup(key) != null` change e' verificata.

A rigore, servirebbe conoscere anche la specifica di `delete`, che non e' qui precisata.

Domanda b)

Si consideri la seguente implementazione parziale della classe Sequence<T>.

```
class Nodo<T> {
    protected String key;
    protected T info;
    protected Nodo<T> next;
}

public class Sequence<T> {
    protected Nodo<T> first;
    protected size;

    public Sequence() {first=null; size=0}

    public void enter(String key, T info) {
        Nodo<T> nuovo = new Nodo();
        nuovo.key = key;
        nuovo.info = info;
        nuovo.next = first;
        first=nuovo;
        size++;
    }

    public size() {return size;}
...
}
```

Scrivere l'invariante di rappresentazione della classe, ove possibile usando JML.

Soluzione

```
private invariant size==0 <==> first == null &&
    size>0 ==> (* il numero di elementi linkati a partire da first e' uguale a size *) &&
        (* non vi sono due elementi con lo stesso next *)
```

Domanda c)

Si vuole definire una classe SequenzaOrdinata, che ha la stessa interfaccia di Sequence, ma in cui gli elementi sono disposti secondo l'ordine lessicografico delle chiavi anzichè secondo l'ordine di arrivo.

E' possibile definire la classe come estensione di Sequence rispettando il principio di sostituzione di Liskov? e viceversa? Giustificare la risposta.

Soluzione

Non è possibile, in quanto la specifica di `enter` per la nuova classe (“inserire in ordine”) è incompatibile con quella di `Sequence` (“inserire nella prima posizione”).

Esercizio 2

Si consideri il seguente snippet di codice.

```
import java.util.Vector;
public class Driver {
    private StringContainer b = null;

    public static void main(String[] args) {
        Driver d = new Driver();
        d.run();
    }
}
```

```

    }
    public void run() {
        b = new StringContainer();
        b.add("One");
        b.add("Two");
        b.remove("One");
    }
}

class StringContainer {
    private Vector v = null;

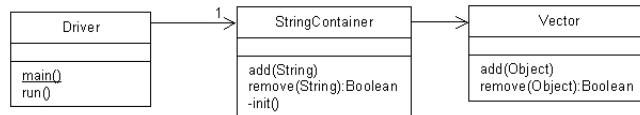
    public void add(String s) {
        init();
        v.add(s);
    }
    public boolean remove(String s) {
        init();
        return v.remove(s);
    }
    private void init() {
        if (v == null)
            v = new Vector();
    }
}

```

Domanda a)

Si disegni un class diagram che esprime le relazioni strutturali tra tutte le classi coinvolte nell'esecuzione del programma.

Soluzione



Domanda b)

Si disegni un sequence diagram consistente con il class diagram precedente che illustri la specifica esecuzione descritta nello snippet di codice.

Soluzione

Esercizio 3

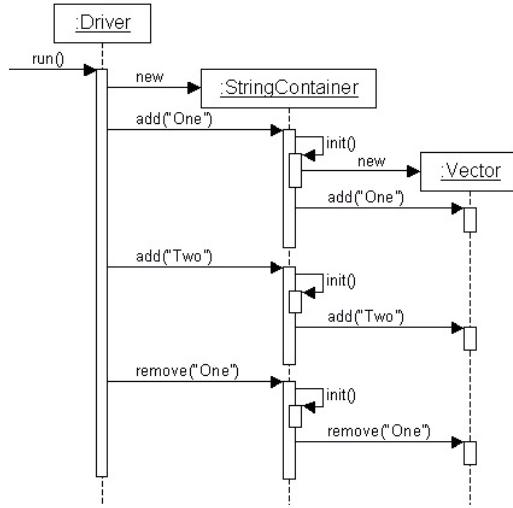
Si considerino le seguenti dichiarazioni di classi Java.

```

abstract class Oggetto {
    public abstract void parla();
}

class OggettoVolante extends Oggetto {
    public void parla() { System.out.println("Ciao, sono un oggetto volante"); }
    public void parla(Oggetto altro) {
        System.out.println("Ciao, chi sei?");
        altro.parla();
    }
}

```



```

class Uccellino extends OggettoVolante {
    public void parla() { System.out.println("Ciao, sono un uccellino"); }
}
class UFO extends OggettoVolante {
    public void parla(OggettoVolante altro) {
        System.out.println("Ciao, sono un oggetto volante non identificato, tu?");
        altro.parla();
    }
}

```

Si consideri poi il seguente frammento di codice Java che utilizza le classi sopra.

1. Oggetto o1, o2;
2. OggettoVolante o3;
3. UFO o4;
4. o1 = new Oggetto();
5. o2 = new OggettoVolante();
6. o3 = new UFO();
7. o4 = o3;
8. o2.parla();
9. o2.parla(o3);
10. o3.parla(o2);
11. o4 = (UFO) o2;
12. o4.parla();
13. o4 = (UFO) o3;
14. o4.parla();
15. o4.parla(o4);

Domanda a)

Si indichino dapprima quali righe del codice sopra riportato generano un errore in fase di compilazione e quali un errore in fase di esecuzione, motivando brevemente la propria risposta.

Soluzione

- La riga 4 non compila: Oggetto è una classe astratta e non può essere istanziata.
- La riga 7 non compila: o3 è staticamente un OggettoVolante, sopraccalss di UFO.
- La riga 9 non compila: o2 è staticamente di classe Oggetto e tale classe possiede il solo metodo parla() senza parametri.

- La riga 11 compila ma solleva un'eccezione a run-time: `o2` potenzialmente potrebbe contenere un riferimento ad un UFO e per questo il compilatore accetta il cast, ma a tempo di esecuzione si verifica che l'oggetto effettivamente riferito da `o2` sia di classe UFO e non essendo così viene sollevata un'eccezione.

Domanda b)

Supponendo che tutte le righe che provocano errori in fase di compilazione e/o di esecuzione siano state rimosse, illustrare l'output del programma, motivando brevemente la propria risposta.

Soluzione

Supponiamo di eliminare le righe 4,7,9,11 e anche la riga 12 che dipende dalla 11. Consideriamo le righe rimanenti ottenendo:

```

8. Ciao, sono un oggetto volante
10. Ciao, chi sei?
     Ciao, sono un oggetto volante
14. Ciao, sono un oggetto volante
15. Ciao, sono un oggetto volante non identificato, tu?
     Ciao, sono un oggetto volante

```

Esercizio 4

Si consideri il seguente codice sorgente, in un linguaggio pseudo C.

```

while (a<2) {
    if (b<1) {
        c++;
    }
    if (a>b) {
        b=a;
    }
    else if (c==3) {
        c++;
    }
    else {
        a++;
    }
}

```

Domanda a)

Si identifichi un insieme minimo di casi di test che garantisca copertura completa delle istruzioni. Tra tutti i possibili insiemi minimi di casi di test che offrono tale garanzia se ne privilegi uno che minimizza anche il numero di iterazioni del ciclo while.

Soluzione

Esaminando il codice sorgente è possibile notare che il secondo `if` ha tre possibili diramazioni, che sono quindi in mutua esclusione tra loro. Se vogliamo puntare a un unico caso di test saranno quindi necessarie *almeno* tre diverse iterazioni del ciclo, ciascuna delle quali deve essere studiata in maniera da entrare in una diversa diramazione del secondo `if`. Ciò si può ottenere ad esempio con il seguente test case: `{a=0; b=0; c=2}`.

Domanda b)

Per l'insieme di casi di test individuato al punto precedente calcolare la percentuale di copertura secondo (1) il criterio delle diramazioni e (2) il criterio delle condizioni.

Soluzione

Secondo il criterio delle diramazioni, il test case sopra ha copertura del 100%. Infatti, è progettato per coprire le tre diramazioni del secondo `if`, ma nell'ultima iterazione sarà `b=1`, per cui anche il ramo `else` del primo `if` viene coperto. Non essendovi condizioni composte, il test case ha copertura totale anche secondo il criterio delle condizioni.

Appello 18 Luglio 2016



Politecnico di Milano
Anno accademico 2015-2016

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Ghezzi <input type="checkbox"/> Mottola <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta il divieto di partecipazione all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Esercizio 1

La classe mutabile Sequence rappresenta una lista sequenziale di elementi, di un tipo Dato, descritto parzialmente piu' avanti. La sequenza non e' mai vuota, ossia contiene sempre almeno un elemento. Una specifica informale di Sequence e' data di seguito, con il metodo affinita' che e' spiegato piu' avanti.

```
public class Sequence {  
  
    //costruisce una sequenza di un solo dato.  
    public Sequence(Dato d)  
  
    //restituisce il numero di elementi nella sequenza  
    public int size()  
  
    //suppone che d non \`e nullo  
    //inserisce un nuovo dato in ultima posizione  
    public void insert(Dato d)  
  
    // restituisce il dato in posizione i, se 0<=i<size()  
    //altrimenti lancia IndexOutOfBoundsException  
    public Dato get(int i) throws IndexOutOfBoundsException  
  
    //calcola l'affinita' totale della sequenza, ignorando i dati duplicati  
    public double affinitaTotale()  
  
    //cancella dalla sequenza l'occorrenza piu' a sx di d,  
    //ma lancia eccezione EmptyException se d e' l'unico elemento.  
    public void delete(Dato d) throws EmptyException  
  
    ...  
}
```

Ogni elemento di tipo Dato ha un'affinita' rispetto ad ogni altro elemento di tipo Dato. L'affinita' e' un valore reale compreso fra 0 e 1. L'affinita' massima (uno) e' raggiunta solo con gli elementi per cui vale equals, ossia `d1.affinita(d2) = 1` se e solo se `d1.equals(d2)`. La classe immutabile Dato ha, oltre a un costruttore non meglio specificato, un metodo double `affinita(Dato d)`, che calcola l'affinita di this con il dato d. L'affinita' totale di una sequenza e' data dal prodotto delle affinita' delle coppie di dati contigue. Es. se la sequenza e' d1 d2 d3, allora `affinitaTotale` vale `affinita(d1,d2)*affinita(d2,d3)`.

Parte (a)

1. Si scriva la specifica JML dei metodi `affinitaTotale()` e `delete`.

SOLUZIONE:

```
//@requires size()>1  
//@ensures \result ==  
//@ (\product int i; 0<=i<size()-1;  
//@ (\exists int j; 0<=i && i<j;  
//@ get(i).equals(get(j)) && get(i+1).equals(get(j+1)))  
//@ ? 1  
//@ : get(i).affinita(get(i+1))  
public /*@ pure @*/ double affinitaTotale()
```

Def. per semplicita' un metodo `minPos` che restituisce la posizione piu' a sx di un dato d, -1 se non esiste:

```

//@requires d!=null;
//@ensures \result ==
//@      (\exists int i; 0<=i && i<size(); get(i).equals(d))
//@      ? (\min int i; 0<=i<size()&& get(i).equals(d); i)
//@      : -1
public /*@ pure */ boolean minPos(Dato d)

//@ensures \old(size()) > 1 &&
//@    minPos(d)>-1
//@    ? size()==\old(size()-1) &&
//@        (\forall int j; 0<=j && j<size();
//@            get(j).equals(\old(get(j<minPos(d) ? j : j+1))))
//@    : size()==\old(size() &&
//@        (\forall int j; 0<=j<size(); \old(get(j)).equals(get(j))))
//@signals(EmptyException e) \old(size())==1 && get(0).equals(\old(get(0)))
public void delete(Dato d)

```

2. Si definisca in JML l'invariante pubblico della classe Sequence, in base alle specifiche e alle descrizioni fornite

SOLUZIONE:

```

//@ public invariant size()>0 &&
//@      (\forall int i; 0<=i<size(); get(i)!=null) &&
//@      affinitaTotale()>=0 &&
//@      affinitaTotale()<=1

```

3. Si stabilisca, argomentando in modo rigoroso, se l'invariante pubblico è verificato dalla specifica. SOLUZIONE:

La condizione nel public invariant che affinitaTotale() e' fra 0 e 1 deriva dalla specifica di affinitaTotale(), per cui il valore restituito e' sempre un prodotto di affinita', che per ipotesi sono comprese fra 0 e 1. La condizione *size()* > 0 e' verificata in quanto il costruttore costruisce una sequenza non vuota e l'unico metodo che puo' ridurre la dimensione della sequenza (ossia delete) lancia un'eccezione se *size()* == 1, senza cancellare l'elemento: quindi se anche la nuova *size()* diminuisce di 1, sara' sempre positiva.

Parte (b)

Si consideri un'implementazione di Sequence in cui si utilizza una LinkedList per memorizzare gli elementi in sequenza. La seguente classe Elemento, nello stesso package di Sequenza, contiene un dato e un riferimento all'elemento successivo, in modo da definire un elemento della LinkedList. Se un Elemento e' l'ultimo, allora il campo next==null.

```

class Elemento{
    Dato info;
    Elemento next;
    Elemento(Elemento n, Dato x) {
        next= n;
        info =x;
    }
}

public class Sequence{
    //rep
    private Elemento inizio;
    private Elemento fine;
}

```

```

private double affinitaTot;

public Sequenza(Dato d) {
    inizio= new Elemento(null,d);
    fine = inizio;
    affinitaTot=1;
}
public void insert(Dato d) {
    affinitaTot= affinitaTot* fine.info.affinita(d);
    fine.next = new Elemento(null, d);
    fine = fine.next;
}
...
}
}

```

1. (FACOLTATIVO) Scrivere l'invariante rappresentazione della classe Sequence, in base al frammento di implementazione dato.

SOLUZIONE:

Def. un metodo puro ausiliario getElem:

```

//@requires 0<= i && i>0 ==> e.next!=null
//@ensures (i==0|| e==null) ? \result == e :
//@ \result ==getElem(Elem.next, i-1);
public static /*@ pure @*/ Elemento getElem(Elemento e, int i)

```

L'inv. asserisce che la lista e' sequenziale:

```

private invariant inizio!=null && fine != null && fine.next == null &&
                affinitaTot>=0 && affinitaTot<=1 &&
                (\forall int i; 0<=i && getElem(inizio,i)!=null;
                 (\forall int j; i<j && getElem(inizio,j)!=null;
                  getElem(inizio,i)!= getElem(inizio,j) )) &&
                (\exists int j; 0<=j; fine=getElem(inizio,j));

```

2. Definire la funzione di astrazione della classe Sequence. Si puo' usare un private invariant o scrivere un'implementazione del metodo toString().

SOLUZIONE:

per scrivere un private invariant, si puo' riutilizzare il metodo puro ausiliario getElem:

```

private invariant
    (\forall int i; 0<=i && getElem(inizio,i)!=null;
     getElem(inizio,i)== get(i));

```

in alternativa, si puo' definire la toString:

```

@Override
public String toString() {
    Elemento e = inizio;
    String s = "Sequenza: e.info";
    while (e.next!= null) {
        e=e.next;
        s = s+ ", " + e.info;
    }
}

```

```
    }  
    return s;  
}
```

Parte (c)

Si consideri una variante di Sequence, detta ThinSequence, che ha anche un metodo che consente di cancellare un dato da una posizione assegnata, cosi' definito:

```
//@ensures (*elimina il dato in posizione i *)
//@signals (IndexOutOfBoundsException e) i<0 || i>=size();
public void deleteAt(int i) throws IndexOutOfBoundsException
```

1. Completare la specifica JML del metodo deleteAt, formalizzandone la **postcondizione**. **SOLUZIONE:**

```
//@requires size()>0;
//@ensures (\forall int j; 0<=j && j<size();
//@           get(j).equals(\old(get(j<i ? j : j+1))) &&
//@           size() == \old(size()-1);
```

2. E' possibile definire questa classe ThinSequence come derivata da Sequence, o viceversa, rispettando il principio di sostituzione? Motivare la risposta.

Non e' possibile che ThinSequence erediti da Sequence, in quanto viene violata la regola delle proprietà (l'invariante di Sequence prevede che una Sequence non sia mai vuota). Anche il viceversa non e' possibile, in quanto una classe Sequence definita come erede di ThinSequence non verificherebbe il proprio invarianto.

Esercizio 2

Si consideri la seguente classe che calcola le radici di una equazione di secondo grado. Il metodo che calcola le radici contiene un errore nel ramo commentato. E' invece una scelta di progetto restituire sempre due radici, anche nel caso in cui la radice e' una sola (in tal caso i due valori restituiti sono uguali) o non esistono radici reali (in tal caso il valore restituito e' -1).

```
//calcolo radici di ax^2+bx+c come (-b+-sqrt(b^2-4ac))/(2a)
class Roots {
    double root1, root2;
    int num_roots;
    public Roots(double a, b, c) {
        double q, r;
        q=b*b-4*a*c;
        if (q>0 && a!=0) {
            num_roots = 2;
            r = (double) Math.sqrt(q);
            root1= (r-b)/(2*a);
            root2=(-r-b)/(2*a);
        } else if (q==0) { //in questo ramo c'e' un errore: l'assunzione che
                           //esista esattamente 1 radice
            num_roots=1;
            root1=(-b)/(2*a);
            root2=root1;
        } else {
            num_roots=0;
            root1=-1;
            root2=-1;
        }
    }
    public int num_roots() {return num_roots;}
    public double first_root() {return root1; }
    public double second_root() {return root2; }
}
```

1. Fornire un insieme minimo di casi di test che coprono tutte le istruzioni (statements);
2. L'insieme minimo di cui sopra copre anche tutti i branch?
3. L'insieme minimo di cui sopra soddisfa anche il criterio di condition coverage? In caso negativo, fornire un insieme minimo che soddisfi il criterio.
4. L'insieme di casi di test fornito in risposta ai punti 2 e 3 consente di rilevare l'errore presente nel codice? In caso negativo, quale test addizionale deve essere fornito per rilevare l'errore?
5. Si supponga di generare invece i casi di test in maniera casuale. Ipotizzando che il computer abbia parole di 64 bit per rappresentare i valori numerici double, qual'e' la probabilita' che si generino dati che portano ad attraversare il ramo che contiene l'errore e a generare un malfunzionamento?

SOLUZIONE:

Per il punto 5: deve essere $q = 0$ per attraversare il ramo errato e $a = 0$ per causare l'errore. Quindi solo $a = 0 \wedge b = 0$ portano alla generazione del malfunzionamento. Scegliendo i valori a e b in modo casuale, la prob sarebbe $\frac{1}{2^{64} \times 2^{64}} = 2^{-128}$.

Esercizio 3

Si consideri un'applicazione di commercio elettronico per la gestione degli ordini di acquisto. Un ordine, una volta creato dal cliente, puo' essere da pagare, pagato, cancellato, spedito. Gli ordini non pagati o cancellati non possono essere spediti, mentre quelli spediti non possono essere cancellati o spediti di nuovo.

a) Si definisca una struttura (ad es. in UML) per rappresentare una soluzione estendibile alla gestione degli ordini, utilizzando design pattern opportuni. Accennare anche all'implementazione in Java di classi significative.

b) Si delinei come estendere la soluzione precedente a nuovi requisiti. In particolare, la gestione deve considerare che gli ordini spediti possono andare in restituzione (ad esempio, perche' il materiale ordinato e' guasto o il cliente esercita il diritto di recesso).

SOLUZIONE:

parte (a): Si tratta di applicare il pattern State.

```
public interface StatoOrdine {
    boolean pagato(Ordine ordine);
    StatoOrdine spedisci(Ordine ordine);
    boolean cancellabile(Ordine ordine);
    StatoOrdine cancella(Ordine ordine);
    SituazioneOrdine status();
}

public class Ordine {
    public final static DA_PAGARE=0, SPEDITO=1, CANCELLATO=2, PAGATO=3;
    private StatoOrdine _statoOrdine;

    public Ordine(StatoOrdine statoOrdine) {
        _statoOrdine = statoOrdine;
    }

    int idOrdine;
    String cliente;
    Date dataOrdine;
    public int status() {
        return _statoOrdine.status();
    }
    public boolean cancellabile() {
        return _statoOrdine.cancellabile(this);
    }
    public void cancella() {
        if (cancellabile())
            _statoOrdine.cancella(this);
    }
    public boolean pagato() {
        return _statoOrdine.pagato(this);
    }
    public void spedisci() {
        if (pagato())
            _statoOrdine.spedisci(this);
    }

    void modifica(StatoOrdine statoOrdine) {
        _statoOrdine = statoOrdine;
    }
}
```

```
}
```

Poi ad esempio uno stato puo' essere implementato in questo modo:

```
public class StatoCancelato implements StatoOrdine
{
    public boolean pagato(Ordine ordine) {
        return false;
    }

    public StatoOrdine spedisci(Ordine ordine) {
        throw new UnaNuovaException("non Spedibile! Ordine cancellato!");
    }

    public boolean cancellabile(Ordine ordine) {
        return false;
    }

    public StatoOrdine cancella(Ordine ordine) {
        throw new UnaNuovaException("Ordine gia' cancellato!");
    }

    public int status() {
        return Ordine.CANCELLATO;
    }
}
```

Parte (b). La via piu' semplice e'

- definire una nuova interfaccia StatoOrdineRestituibile, con il nuovo metodo restituisce(...), che eredita da StatoOrdine;
- implementare la nuova interfaccia con una classe StatoRestituito,
- definire una nuova classe OrdineRestituibile che eredita da Ordine e gestisce la restituzione ridefinendo opportunamente i metodi secondo le modalita' di gestione della restituzione.

Esercizio 4

Sono dati i seguenti frammenti di codice, in cui il significato delle variabili e' autoesplicativo:

```
long primesCount = 0;
for(long number : numbers) {
    if(isPrime(number))
        primesCount += 1;
}

int max = 0;
for(int price : prices) {
    max = Math.max(max,price);
}
```

Riscrivere i frammenti usando la notazione funzionale di Java 8.

SOLUZIONE:

```
final long primesCount =
    numbers .stream()
        .filter(number -> isPrime(number)) .count();

final int max = prices.stream().reduce(0, Math::max);
```

Esercizio 5

Si consideri il seguente programma Java

```
1  class Hotel {  
2      public int bookings;  
3      public void book() {  
4          bookings++;  
5      }  
6  }  
7  public class SuperHotel extends Hotel {  
8      public void book() {  
9          bookings--;  
10     }  
11     public void book(int size) {  
12         book();  
13         super.book();  
14         bookings += size;  
15     }  
16     public static void main(String args[]) {  
17         Hotel hotel = new Hotel();  
18         hotel.book(2);  
19         System.out.print(hotel.bookings);  
20     }  
21 }
```

Domanda 1

Questo programma genera un qualche errore? Se sì, si tratta di uno (o più) errori segnalati a compile-time o a run-time? Se sono presenti errori, come correggerli? Scegliere la/le alternativa/e e motivarla/e opportunamente.

- A. Aggiungere un parametro intero al metodo dichiarato alla linea 3.
- B. Eliminare il parametro 2 alla linea 18.
- C. Cambiare il tipo dinamico dell'oggetto hotel alla riga 17 in SuperHotel
- D. Cambiare il tipo statico dell'oggetto hotel alla riga 17 in SuperHotel
- E. Entrambe la C e D.
- F. Nessuna correzione necessaria.

SOLUZIONE:

Il programma non compila a causa della chiamata alla linea 18 con un parametro int, non previsto nella segnatura di Hotel.book().

Esaminiamo alternative:

- A) da sola causa errore di compilazione alla riga 13, dove ci si aspetta che super.book() non abbia parametri.

B) da sola e' sufficiente perche' il programma compili e si esegua correttamente.

C) da sola causa errore di compilazione

D) da sola causa errore di compilazione

E) C e D assieme sono sufficienti perche' il programma compili e si esegua correttamente.

F) e' ovviamente errata.

La risposta e' quindi che sono corrette le alternative B ed E.

Ingegneria del Software — Soluzione del Tema 26/08/2022

Esercizio 1

Si consideri la seguente classe `SeriesStore` per memorizzare episodi di serie TV. Gli episodi sono rappresentati dalla classe immutabile `Episode` che espone tra gli altri un metodo per ottenere la serie di appartenenza di un episodio.

```
public class SeriesStore {
    // Ritorna il primo episodio della serie passata per parametro.
    // Ritorna null se la serie non esiste.
    public /*@ pure @*/ Episode getFirstEpisode(String series);

    // Ritorna l'episodio successivo a quello passato per parametro.
    // Ritorna null se tale episodio non esiste, ovvero se quello passato per parametro
    // e' l'ultimo della serie.
    public /*@ pure @*/ Episode getNextEpisode(Episode ep);

    // Ritorna il numero di episodi della serie successivi a quello passato per parametro.
    // Ritorna 0 se la serie non esiste.
    public /*@ pure @*/ int numNextEpisodes(Episode ep);

    // Aggiunge l'episodio in fondo alla sua serie di appartenenza. Se la serie ancora
    // non esiste, l'episodio viene aggiunto come il primo di quella serie.
    // Lancia una IllegalEpisodeException se l'episodio passato come parametro e' null.
    public void addEpisode(Episode ep) throws IllegalEpisodeException;
}

public /*@ pure @*/ Episode {
    // Ritorna la serie di appartenenza dell'episodio.
    public /*@ pure @*/ String getSeries();
    ...
}
```

Domanda a)

Specificare in JML il metodo `numNextEpisodes`.

Soluzione

```
/*@ requires ep != null
*/
/*@ ensures \result == (getNextEpisode(ep) == null) ? 0 : 1+numNextEpisodes(getNextEpisode(ep))
public /*@ pure @*/ int numNextEpisodes(Episode ep);
```

Domanda b)

Specificare in JML il metodo `addEpisode`.

Soluzione

Definiamo `addEpisode` solo a partire da `getFirstEpisode` e `getNextEpisode` in quanto `numNextEpisodes` deriva da questi.

```
/*@ ensures ep != null &&
*/
/*@ (\forall String series; series != null && !series.equals(ep.getSeries()));
/*@   getFirstEpisode(series) == \old(getFirstEpisode(series)) ) &&
/*@ (\forall Episode e; e != null && !e.getSeries().equals(ep.getSeries()));
/*@   getNextEpisode(e) == \old(getNextEpisode(e)) ) &&
/*
/*@ getFirstEpisode(ep.getSeries()) == \old(getFirstEpisode(ep.getSeries())) == null ?
/*@   ep : \old(getFirstEpisode(ep.getSeries())) &&
/*@ getNextEpisode(ep) == null &&
/*@ (\forall Episode e; e.getSeries().equals(ep.getSeries()) && !e.equals(ep));
/*@   getNextEpisode(e) == (\old(getNextEpisode(e))) == null ?
/*@   ep : \old(getNextEpisode(ep.getSeries())) )
/*

```

```

//@ signals (IllegalEpisodeException exc) ep == null &&
//@ (\forall Episode e; e != null; getNextEpisode(e) == \old(getNextEpisode(e))) &&
//@ (\forall String series; series != null;
//@   getFirstEpisode(series) == \old(getFirstEpisode(series)) )
public void addEpisode(Episode ep) throws WrongEpisodeException;

```

Domanda c)

Si consideri la seguente implementazione della classe `SeriesStore` che usa una mappa per associare a ogni serie la lista dei suoi episodi. Si ricorda che una `Map<K, V>` è una struttura dati che associa a chiavi di tipo K (`String` in questo caso) valori di tipo V (`List<Episode>` in questo caso). Una Map fornisce, fra gli altri, i metodi osservatori `get(K key)`, che restituisce il valore univoco associato alla chiave `key`, e `containsKey(K key)`, che restituisce `true` se esiste un valore associato alla chiave `key`. Altri metodi sono ad esempio `size()` e `isEmpty()`, di significato ovvio.

```

public class SeriesStore {
    private Map<String, List<Episode>> episodes;
    ...
}

```

Per tale implementazione, si forniscano l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

L'invariante di rappresentazione richiede che la mappa non sia nulla, non contenga chiavi nulle e non contenga liste di episodi nulle o vuote. Gli episodi in ogni lista non devono essere duplicati e la loro serie deve corrispondere a quella della chiave.

```

//@ private invariant
//@ episodes != null && !episodes.containsKey(null) &&
//@ (\forall String s; episodes.containsKey(s);
//@   episodes.get(s) != null && !episodes.get(s).isEmpty() &&
//@   (\forall int i; i>=0 && i<episodes.get(s).size();
//@     episodes.get(s).get(i).getSeries().equals(s) &&
//@     !(\exists int j; j>=0 && j<i;
//@       episodes.get(s).get(i).equals(episodes.get(s).get(j)) ) ) )

```

La funzione di astrazione definisce `getFirstEpisode` e `getNextEpisode` in base al contenuto di `episodes`. Tutti gli altri metodi sono definiti a partire da questi.

```

//@ private invariant
//@ (\forall String s; s!=null; getFirstEpisode(s) ==
//@   episodes.containsKey(s) ? episodes.get(s).get(0) : null) &&
//@ (\forall Episode e; e!=null; getNextEpisode(e) ==
//@   (episodes.containsKey(e.getSeries()) &&
//@     (\exists int i; i>=0 && i<episodes.get(e.getSeries()).size()-1;
//@       episodes.get(e.getSeries()).get(i).equals(e) ) ) ?
//@     episodes.get(e.getSeries()).get(i+1) : null

```

Domanda d)

Si consideri una classe `SeriesStore2` che aggiunge un metodo `addEpisodes` che prende in ingresso una lista di episodi e li aggiunge allo store. Può tale classe essere definita come sottoclasse di `SeriesStore` in accordo con il principio di sostituzione?

Soluzione

La classe può essere definita come sottoclasse di `SeriesStore` perché aggiunge un metodo senza alterare i metodi esistenti e senza violare alcun invariante pubblico di `SeriesStore`.

Esercizio 2

Si consideri la seguente classe Java:

```
public class SyncCouple {
    private int x, y;
    private String xlock = new String("xlock"), ylock = new String("ylock");
    public void setX(int x) {
        synchronized(xlock) { this.x=x; }
    }
    public void setY(int y) {
        synchronized(ylock) { this.y=y; }
    }
    public void setXYIfNotEqual(int i) {
        synchronized(xlock) {
            synchronized(ylock) {
                if(x!=y) { x=i; y=i; }
            }
        }
    }
    public boolean equalXY() {
        synchronized(ylock) {
            synchronized(xlock) {
                return x==y;
            }
        }
    }
}
```

Domanda a)

La classe è correttamente sincronizzata? In caso negativo spiegare, anche con un esempio, quale problema possa verificarsi e indicare come correggere il problema minimizzando i cambiamenti e senza stravolgere l'approccio alla sincronizzazione adottato.

Soluzione

La chiamata dei metodi `setXYIfNotEqual` e `equalXY` sullo stesso oggetto da parte di due thread diversi potrebbe portare ad un deadlock. Infatti il thread T1, che esegue il primo metodo, potrebbe acquisire il lock sull'attributo `xlock` per poi cedere il controllo al secondo thread T2 che, eseguendo il metodo `equalXY`, acquisirebbe il lock su `ylock` per poi sospendersi in attesa di poter acquisire il lock su `xlock`, ma tale lock sarebbe in possesso di T1 che si troverebbe a sua volta bloccato in attesa di acquisire il lock su `ylock` (in possesso di T2).

La soluzione più semplice al problema, che non modifica l'approccio alla sincronizzazione, è scambiare l'ordine di acquisizione dei lock in uno dei due metodi `setXYIfNotEqual` o `equalXY`, in modo da acquisire i lock nel medesimo ordine. Ad esempio modificando il metodo `equalXY` come segue:

```
public boolean equalXY() {
    synchronized(xlock) {
        synchronized(ylock) {
            return x==y;
        }
    }
}
```

Domanda b)

Si aggiunga alla classe `SyncCouple` un metodo `waitForZeroX()` che sospende il chiamante fino a quando l'attributo `x` non assume il valore zero.

Soluzione Il metodo va implementato come segue:

```
public void waitForZeroX() throws InterruptedException {
    synchronized(xlock) {
        while(x!=0) xlock.wait();
    }
}
```

Inoltre bisognerà aggiungere l'istruzione `xlock.notifyAll()` ai metodi `setX` e `setXYIfNotEqual` (all'interno del blocco sincronizzato su `xlock`).

Esercizio 3

```
1 public static void mystery(int x) {
2     if (x <= 1) {
3         return;
4     }
5     int m = x - 1;
6     while (x >= 0) {
7         x = x % m;
8         if (m > 1 || x >= 0) {
9             x = x - 1;
10        }
11    }
12    return;
13 }
```

Si determini un insieme minimo di casi di test per ciascuno dei casi seguente di copertura.

1. Copertura delle istruzioni (statement coverage)
2. Copertura delle decisioni (edge coverage)
3. Copertura delle decisioni e delle condizioni (edge and condition coverage)

Soluzione

1. Copertura delle istruzioni (statement coverage). Essendoci due return, servono almeno due casi. Ad es. $x = 1$ (per uscire dal primo return) e $x = 2$ (per percorrere il while una volta) sono in realtà anche sufficienti.
2. Copertura delle decisioni (edge coverage). La copertura completa delle decisioni è impossibile, in quanto la condizione dell'if all'interno del ciclo è sempre verificata: x non può essere negativa, perché il corpo del ciclo è eseguito con $x \geq 0$ e l'istruzione $x = x \% m$ non può rendere x negativo ($m > 0$). I casi precedenti $x = 1, x = 2$ sono sufficienti per massimizzare la copertura delle decisioni.
3. Copertura delle decisioni e delle condizioni (edge and condition coverage). Con i casi precedenti $x = 1, x = 2$ si copre la condizione $m > 1$ falsa; per rendere $m > 1$ vera, basta prendere $x = 3$.

Esercizio 4

Si considerino le seguenti classi Java:

```
abstract class Date {
    public abstract String dateDistance(Date d);
}

class DayDate extends Date {
    int day;
    public String dateDistance(Date p) {
        return "Distance of the two dates is" + dateDistance((DayDate) p);
    }
    public String dateDistance(DayDate p) {
        return ":" + (day-p.day) + " days";
    }
    public DayDate(int d) { day=d; }
}

class DayMonthDate extends DayDate {int month;
    DayMonthDate(int d, int m) { super(d); month = m; }
    public String dateDistance(Date p) {
        return "Distance in days and months is" + dateDistance((DayMonthDate) p);
    }
    public String dateDistance(DayMonthDate p) {
        return super.dateDistance(p) + ":" + (month - p.month) + " months";
    }
}

class DayMonthYearDate extends DayMonthDate {int year;
    DayMonthYearDate(int d, int m, int y) { super(d, m); year = y; }
    public String dateDistance(Date p) {
        return "Distance in d-m-y is" + dateDistance((DayMonthYearDate) p);
    }
    public String dateDistance(DayMonthYearDate p) {
        return super.dateDistance(p) + ":" + (year - p.year) + " years";
    }
}
```

e il seguente frammento di codice le cui righe sono state numerate per riferimento:

```
1 DayDate d1 = new DayDate(25);
2 DayDate d2 = new DayMonthDate(25, 2);
3 DayMonthDate d3 = new DayMonthDate(20, 4);
4 DayMonthYearDate d4 = new DayMonthDate(25, 2);
5 DayMonthYearDate d5 = new DayMonthYearDate(24, 3, 2022);
6 Date d6 = new DayDate(24);
7 System.out.println(d1.dateDistance(d2));
8 System.out.println(d1.dateDistance(d3));
9 System.out.println(d1.dateDistance(d6));
10 System.out.println(d2.dateDistance(d3));
11 System.out.println(d3.dateDistance(d2));
12 System.out.println(d3.dateDistance(d5));
13 System.out.println(d4.dateDistance(d5));
14 System.out.println(d5.dateDistance(d3));
15 System.out.println(d5.dateDistance(d6));
```

Domanda a)

Scrivere il numero di riga delle istruzioni (se ne esistono) che generano un errore in compilazione o un'eccezione a run time, chiarendo se il problema è a compile time o a run time, e spiegandone brevemente le ragioni.

Soluzione

La variabile d4 è inizializzata incorrettamente, in quanto DayMonthDate non è sottoclasse di DayMonthYearDate. Sono quindi da eliminare anche le righe in cui d4 viene riferita (compile time).

La chiamata d5.dateDistance(d6) genera a run time un ClassCastException, in quanto il cast del metodo DayMonthYearDate.dateDistance(Date) da Date a DayMonthYearDate non è possibile: il tipo dinamico di d6 è infatti DayDate.

Sono quindi da eliminare le righe 4, 13 e 15.

Domanda b)

Dopo aver rimosso le linee identificate al punto precedente, scrivere cosa stampa il programma (indicando il numero di riga che produce ogni stampa).

Soluzione

- 7) : 0 days
- 8) : 5 days
- 9) Distance of the two dates is: 1 days
- 10) : 5 days
- 11) : -5 days
- 12) : -4 days: 1 months
- 14) : 4 days: -1 months

Appello 14 settembre 2015



Politecnico di Milano
Anno accademico 2014-2015

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Baresi <input type="checkbox"/> Ghezzi <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri il seguente metodo che conta le occorrenze della stringa param in testo:

```
static ArrayList<Integer> trovaOccorrenze(String testo, String param)
```

Si scrivano le pre- e post-condizioni JML del metodo ipotizzando che:

1. Il metodo restituisce sempre un `ArrayList` di un elemento. Se il metodo trova almeno un'occorrenza, il primo elemento della lista contiene la posizione in `testo` del primo carattere di `param`. Se il metodo non trova occorrenze, il primo elemento della lista vale `-1`;
2. Il metodo restituisce un `ArrayList` con un numero di elementi variabile. Se la lista non è vuota, i suoi elementi contengono le posizioni in ordine crescente (primo carattere) di tutte e sole le occorrenze di `param` in `testo`. Se il metodo non trova occorrenze, la lista contiene solo il valore `-1`;
3. l'intestazione del metodo contenga un terzo parametro `int prec`, ovvero la precisione da usarsi per la ricerca delle occorrenze. Un'occorrenza di `param` in `testo` si considera valida se è una sottostringa di `testo` che ha la stessa lunghezza di `param` e il numero dei caratteri diversi, nelle stesse posizioni, è al piu' uguale a `prec`.
Ad esempio se `testo` valesse `cassa` e `prec` 3, valori accettati di `param` sono per esempio `colla`, `carta`, `masso`.

SOLUZIONE:

Usiamo per comodita' i metodi `indexOf` della classe `String`, cosi' definiti:

```
public int indexOf(String str): restituisce l'indice della prima comparsa della string str  
nella stringa this, oppure -1 se la stringa str non compare.
```

```
public int indexOf(String str, int fromIndex): restituisce l'indice della prima comparsa della string str  
nella stringa this partendo dall'indice fromIndex, oppure -1 se la stringa str non compare.
```

1.

```
//@requires testo!=null && param!=null  
//@ensures \result.get(0).equals(testo.indexOf(param, 0))  
static ArrayList<Integer> trovaOccorrenze(String testo, String param)
```

2.

```
//@requires testo!=null && param!=null  
//@ensures  
(\forall int y;  
 (\result.contains(y) <=>  
 ( (testo.indexOf(param, y)==y && 0<=y<testo.length())  
 ||  
 (y==-1 && testo.indexOf(param, 0)==-1)  
 ))  
 ) &&  
// result e' ordinato in modo crescente  
(\forall int i; 0<=i && i<\result.size()-1;  
 \result.get(i)<\result.get(i+1));  
static ArrayList<Integer> trovaOccorrenze(String testo, String param)
```

3.

```
//@requires testo!=null && param!=null  
//@ensures  
(\forall int i; ;  
 \result.contains(i) <=>
```

```
// la stringa corrisponde
(
    (\num_of int j; 0<=j && j<param.length();
     testo.charAt(i+j)<param.charAt(j)) <=prec
    &&
    0<=i && i<testo.length()-param.length()
    ||
    (
        i===-1 &&
        !(\exists int k; 0<=k && k<testo.length()-param.length();
         (\num_of int j; 0<=j && j<param.length();
          testo.charAt(k+j)<param.charAt(k)) <=prec
        )
    )
)
) &&
// result e' ordinato in modo crescente
(\forall int i; 0<=i && i<\result.size()-1;
 \result.get(i)<\result.get(i+1));
static ArrayList<Integer> trovaOccorrenze(String testo, String param)
```

Esercizio 2

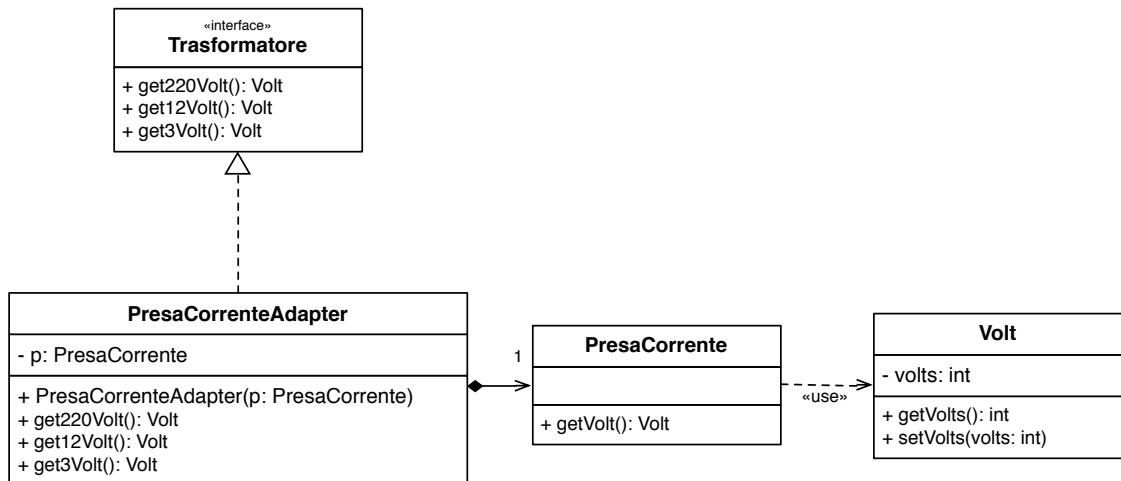
Date le classi Java Volt e PresaCorrente, che restituisce corrente a 220 volt:

```
public class Volt {  
    private int volts;  
  
    public Volt(int v) {  
        this.volts=v;  
    }  
  
    public int getVolts() {  
        return volts;  
    }  
  
    public void setVolts(int volts) {  
        this.volts = volts;  
    }  
}  
  
public class PresaCorrente {  
    public Volt getVolt(){  
        return new Volt(220);  
    }  
}
```

Si utilizzi il design pattern **Adapter** per “realizzare” un trasformatore capace di erogare corrente a 3, 12, e 220 volt:

```
public interface Trasformatore {  
    public Volt get220Volt();  
    public Volt get12Volt();  
    public Volt get3Volt();  
}
```

Si definisca la classe Java da aggiungere, specificando anche le relazioni con le classi esistenti.



SOLUZIONE:

```
public class PresaCorrenteAdapter implements Trasformatore {
```

```
private final PresaCorrente presaCorrente;

public PresaCorrenteAdapter(PresaCorrente presaCorrente) {
    super();
    this.presaCorrente = presaCorrente;
}

private Volt convertTo(int value) {
    Volt result = presaCorrente.getVolt();
    result.setVolts(value);
    return result;
}

@Override
public Volt get220Volt() {
    return convertTo(220);
}

@Override
public Volt get12Volt() {
    return convertTo(12);
}

@Override
public Volt get3Volt() {
    return convertTo(3);
}
}
```

Esercizio 3

Un gruppo di 10 amici ha a disposizione una botte con 100 litri di vino e 2 rubinetti per berli. Di conseguenza possono bere solo due persone alla volta, sempre che la botte non sia vuota. Si supponga, per semplicità, che ogni persona beve sempre 1/2 litro di vino. Si scriva un programma (concorrente) Java che simuli la situazione appena descritta. Il thread Bevitore, dopo aver atteso un tempo casuale, deciderà di bere e si metterà in attesa se il numero di rubinetti liberi fosse 0. Dopo aver ottenuto la disponibilità di un rubinetto, controllerà la quantità di vino rimasta e, se possibile, completerà la bevuta in sorsate di 10ml. Se la botte fosse vuota, il Bevitore terminerà la sua esecuzione.

La soluzione deve prevedere:

- una classe Botte, con un metodo bevi() (usabile in maniera concorrente) che consente di decrementare il contenuto della botte di 1/2 litro;
- una classe Gestore, che fornisce due metodi (usabili in maniera concorrente) occupaRubinetto() e rilasciaRubinetto() che consentono di acquisire uno dei due rubinetti, se libero, e rilasciarlo dopo la bevuta,

si suppone che ogni amico che vuole bere dalla botte esegua sempre la sequenza di operazioni:

- gestore.occupaRubinetto();
- botte.bevi();
- gestore.rilasciaRubinetto();

Si devono pertanto implementare le classi Gestore e Botte.

SOLUZIONE:

```
public class Botte {  
    private int mlRimasti;  
  
    public Botte(){  
        this.mlRimasti=100000;  
    }  
    public int bevi() throws InterruptedException{  
        int effettivamenteBevuto=0;  
        int sorseggio=10;  
        while(effettivamenteBevuto<500 && !isEmpty()) {  
            effettivamenteBevuto=effettivamenteBevuto+this.sorseggia(sorseggio);  
        }  
        return effettivamenteBevuto;  
    }  
    private synchronized boolean isEmpty(){  
        return this.mlRimasti==0;  
    }  
    private synchronized int sorseggi(int quantita){  
        if(mlRimasti>quantita){  
            this.mlRimasti=this.mlRimasti-quantita;  
            return quantita;  
        }  
        else{  
            int bevuta=this.mlRimasti;  
            this.mlRimasti=0;  
            return bevuta;  
        }  
    }  
}
```

```
        }
    }

public class GestoreRubinetti {
    private int rubinettiLiberi;

    public GestoreRubinetti(){
        this.rubinettiLiberi=2;
    }
    public synchronized void occupaRubinetto() throws InterruptedException{
        while(rubinettiLiberi==0){
            wait();
        }
        rubinettiLiberi--;
    }
    public synchronized void rilasciaRubinetto(){
        rubinettiLiberi++;
        notify();
    }
}
```

Esercizio 4

Si consideri il metodo Java seguente:

```
public static int foo(int n) {
    int t=0;

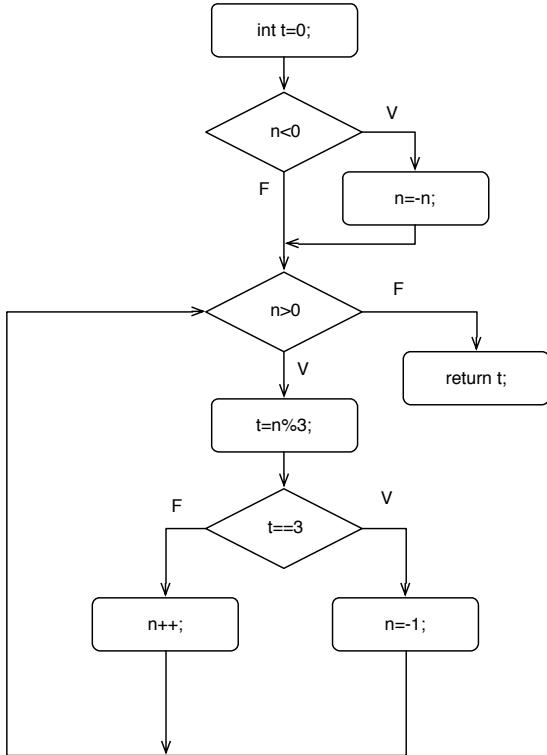
    if (n < 0) n = -n;

    while (n >0) {
        t = n%3;
        if (t == 3) n = -1;
        else n++;
    }
    return t;
}
```

e si definisca

1. Il diagramma del flusso di controllo.
2. Un insieme minimo di test che copra tutte le istruzioni. Nel caso non fosse possibile, definire la copertura (in percentuale) dell'insieme di test definiti.
3. La condizione che n deve rispettare per eseguire il programma senza entrare nel ciclo *while*. È possibile? Cosa restituirebbe il programma?
4. Si modifichi il test dell'istruzione *if* da $t==3$ in $t==2$ e si calcoli la precondizione che n dovrebbe rispettare affinche' si entri nel ciclo due volte e la prima volta si esegua il ramo *then* mentre la seconda si esegua il ramo *else*. Calcolare anche un caso di test, se esiste, che soddisfa la precondizione.

SOLUZIONE: 1.



2.

$n = -1$

Copre tutte le istruzioni tranne il ramo then del secondo if. Infatti, all'ingresso del ciclo while, $n=1$. Ad ogni iterazione il valore di n viene sempre incrementato finché il valore di n supera il massimo valore contenibile in un intero, e di conseguenza per overflow, il valore di n diviene negativo. A quel punto si esce dal ciclo while.

Copertura 8/9.

3.

$n=0$.

ritorna $t=0$;

4.

$$\begin{cases} n \neq 0 \\ |n| \% 3 = 2 \\ -1 > 0 \\ -1 \% 3 \neq 2 \\ 0 \leq 0 \end{cases}$$

Non è possibile coprire il suddetto cammino in quanto $-1 \geq 0$ non può mai essere soddisfatta.

Ingegneria del Software – a.a. 2004/05

Appello del 9 settembre 2005

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 28/30.

Esercizio 1:

Esercizio 2:

Totale

Esercizio 1 (punti 20)

Si vuole realizzare un insieme di elementi "pesati", in cui ogni oggetto dell'insieme è costituito da un nome e da un peso, che può andare da 0 a 63. Le operazioni che si possono effettuare su un insieme sono le seguenti: aggiungere un nuovo elemento (con il relativo peso); togliere uno o più elementi tra quelli di peso maggiore presenti nell'insieme; cambiare il peso di uno degli elementi presenti nell'insieme.

Per realizzare gli elementi da inserire nell'insieme si sceglie di usare il seguente tipo **Elemento**.

```
public class Elemento {
    //@public invariant nome != null && 0 <= peso && peso < 64;
    public final String nome;
    public final int peso;

    //@ensures n != null && 0 <= peso && peso < 64 &&
    //@      nome == n && peso == p;
    //@signals (IllegalArgumentException e)
    //@      n == null || peso < 0 ||  peso >= 64;
    public Elemento(String n, int p){ /* ... */ }

    public boolean equals(Elemento e){
        return e.nome.equals(nome) && e.peso == peso;
    }
    public boolean equals(Object e){
        if(e instanceof Elemento) return equal((Elemento)e);
        return false;
    }
}
```

Sia data inoltre la seguente specifica di un tipo **StringSet**.

```
public class StringSet {
    // OVERVIEW: insiemi di stringhe illimitati e modificabili;
    // per es.: {alfa, pippo, pallino}

    //@ensures (*\result è true sse x è fra gli elementi di this*);
    public /*@ pure @*/ boolean isIn(String x){ /* ... */ }

    //@ensures (*\result è cardinalità di this *);
    public /*@ pure @*/ int size(){ /* ... */ }

    //@ensures (\forall String y; !this.isIn(y));
    public StringSet(){ /* inizializza this come insieme vuoto */ }

    //@ensures this.isIn(x) &&
    //@      (\forall String y; x!=y;
    //@           \old(this.isIn(y)) <==> this.isIn(y));
    public void insert(String x){ /* inserisce x in this */ }

    //@ensures !this.isIn(x) &&
    //@      (\forall String y; x!=y; \old(this.isIn(y)) <==> this.isIn(y));
    public void remove(String x){ /* rimuove x da this */ }

}
```

1. Si completi la seguente dichiarazione della classe **WeightedSet** con le opportune specifiche JML.

```

public class WeightedSet {
    // OVERVIEW: insiemi di elementi pesati <nome, peso> tali per cui ad ogni "nome"
    //             nell'insieme e' associato al massimo un peso (non sono presenti
    //             cioe' nell'insieme due coppie con nome uguale e peso diverso).
    //@public invariant
    //@(\forall Elemento x; isIn(x));
    //@ (\forall Elemento y; isIn(y) && !x.equals(y);
    //@     x.nome != y.nome)

    //@ensures (*\result è true sse x è fra gli elementi di this*);
    public /*@ pure */ boolean isIn (Elemento x){ /* ... */ }

    //inizializza this come insieme vuoto
    //@ensures
    //@ (\forall Elemento y;; !this.isIn(y))
    public WeightedSet(){ /* ... */ }

    //Aggiunge x a this. x deve essere diverso da null e tale che non ci sia in
    //this un elemento con lo stesso nome di x, anche se con peso diverso.
    //@requires
    //@ x!=null && !(\exists Elemento y; isIn(y); y.nome==x.nome && y.peso!=x.peso)

    //@ensures
    //@isIn(x) && (\forall Elemento y;; isIn(y) <==> y.equals(x) || \old(isIn(y)))
    public void add(Elemento x){ /* ... */ }

    //Ritorna un qualunque sottoinsieme di nomi presi tra quelli di peso maggiore
    //presenti in this. Gli elementi i cui nomi vengono ritornati sono eliminati
    //da this.
    //@ensures
    //@(\forall String s; \result.isIn(s);
    //@   (\exists Elemento x;;
    //@     \old(isIn(x)) &&
    //@     (\forall Elemento y; \old(isIn(y)); (y.peso <= x.peso)) &&
    //@     !(\exists Elemento x; isIn(x); x.nome==s))
    public StringSet get(){ /* ... */ }

    //Cambia il peso associato all'elemento di nome n in p; siccome gli attributi
    //della classe Elemento sono final, si noti che per "modificare" un peso
    //occorre in realta' creare un nuovo oggetto Elemento con il nuovo peso.
    //Se nell'insieme non e' presente alcun elemento con nome n, oppure se il peso
    //p non e' un peso valido, solleva un'eccezione checked WeightedSetExc.
    //@requires n!=null
    //@ensures 0<=p && p<64 &&
    //@   (\exists Elemento x; \old(isIn(x)); x.nome==n) &&
    //@   (\exists Elemento x;;
    //@     isIn(x) && x.nome==n && x.peso==p &&
    //@     (\forall Elemento y; !y.equals(x) && isIn(y); y.nome!=x.nome)) &&
    //@     (\forall Elemento y; y.nome!=n; \old(isIn(y)) <==> isIn(y))
    //@signals (WeightedSetExc e): p<0 || p>=64 ||
    //@           !(\exists Elemento x; \old(isIn(x)); x.nome==n)
    public void cambiaPeso(String n, int p) throws WeightedSetExc { /* ... */ }
}

```

2. Come cambiano (se cambiano) la dichiarazione e la specifica JML del metodo **add** se si adotta uno stile di programmazione difensiva, cioè si sceglie di sollevare un'eccezione **WeightedSetExc** nel caso in cui l'oggetto *x* di tipo **Elemento** abbia un nome che è già presente nell'insieme pesato, ma associato ad un peso diverso da quello di *x*? Se ne riscrivano dichiarazione e specifica in modo da rispecchiare la modifica.

```
//@ensures: x!=null &&
//@      !(\exists Elemento y; \old(isIn(y)); y.nome==x.nome && y.peso!=x.peso) &&
//@      isIn(x) &&
//@      (\forall Elemento y; :isIn(y)<=> y.equals(x)) || \old(isIn(y)))
//@signals (WeightedSetExc e):
//@      x==null ||
//@      (\exists Elemento y; \old(isIn(y)); y.nome==x.nome && y.peso!=x.peso)
```

3. Si decide di implementare l'insieme mediante un array di 64 insiemi di tipo **StringSet**. Ogni insieme corrisponde ad un peso tra 0 e 63, e contiene tutti (e soli) i nomi degli elementi di quel peso presenti nell'insieme. Si completi la dichiarazione sottostante con l'indicazione del rep invariant della classe **WeightedSet**.

```
public class WeightedSet {
    //rep:
    private StringSet elPesati[64];
    //@private invariant
    //@(\forall int i; 0<=i && i<64; elPesati[i]!=null) &&
    //@(\forall int i; 0<=i && i<64;
    //@      (\forall int j; i<j && j<64;
    //@      !(\exists String x; ; elPesati[i].isIn(x) && elPesati[j].isIn(x))))
```

}

4. Si implementi il metodo **add** della classe **WeightedSet** (*una* versione a scelta tra quella del punto 1 e quella del punto 2).

Implementiamo la versione del punto 2, quella che lancia l'eccezione.

```
void add(Elemento x) throws WeightedSetExc {  
    if(x==null) throw new WeightedSetExc();  
    for(int i=0; i<64; i++){  
        if((i!=x.peso && elPesati[i].isIn(x.nome))  
            throw new WeightedSetExc();  
    }  
  
    elPesati[i].add(x.nome);  
}
```

5. Si dimostri che l'implementazione di **add** del punto 4 mantiene l'invariante di rappresentazione scritto al punto 3, e che rispetta la specifica del metodo.

Il rep invariant è garantito dal ciclo for. Infatti se il rep invariant è vero all'ingresso del metodo ciò significa che non ci sono in *elPesati* array diversi contenenti lo stesso nome. Ora, *x.nome* deve andare nel set di indice *x.peso*, per cui, perché il rep invariant sia garantito, *x.nome* non comparire in nessuno degli array con indice diverso da *n.peso*. Ciò è garantito dal ciclo for quindi il rep invariant è mantenuto.

D'altro canto, la specifica è garantita in quanto:

- ? la clausola della signals corrisponde esattamente al primo if e alla condizione per cui, nel primo ciclo for, viene sollevata un'eccezione;
- ? le prime due condizioni della ensures sono vere all'uscita del ciclo for;
- ? la terza condizione è garantita dalla add;
- ? la quarta condizione è garantita dal fatto che non viene rimosso nulla dall'insieme.

6. Sia data una classe **WeightedSetAll** che estende **WeightedSet** e ridefinisce il metodo **get** in modo che esso ritorni *tutti* gli elementi di *this* che hanno il peso più grande tra quelli presenti nell'insieme (se per esempio il peso più grande presente nell'insieme è 50, il metodo ritorna tutti gli elementi dell'insieme di peso 50).
- a. Si dia la specifica JML del nuovo metodo **get**.
 - b. La classe **WeightedSetAll** rispetta il principio di sostituibilità? Si motivi brevemente, ma con precisione, la risposta.

Basta aggiungere alla clausola **ensures** del padre

```
//@ensures: ... /* la clausola ensures del padre */ &&
//@  (\forall Elemento x; \old(isIn(x)) &&
//@                  (\forall Elemento y; \old(isIn(y)); y.peso<=x.peso));
//@                  \result.isIn(x.nome));
```

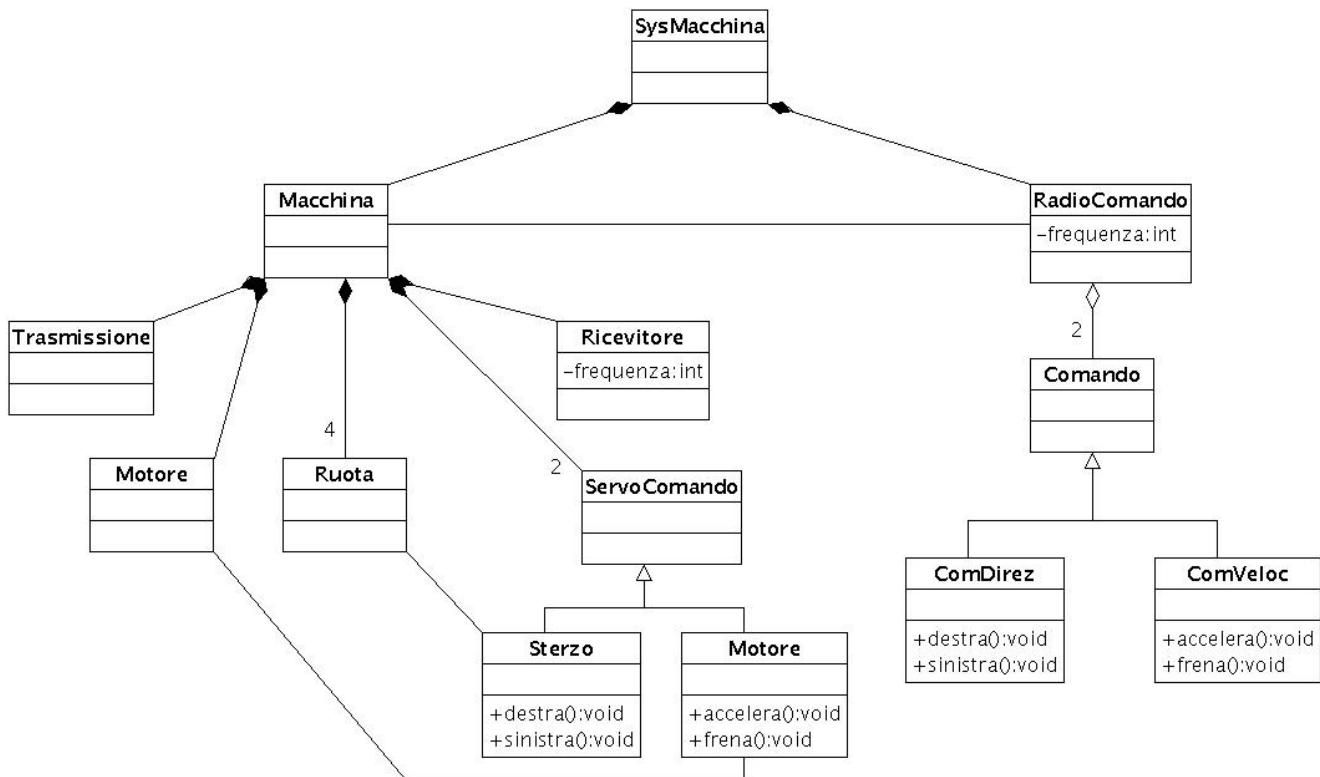
rispetta il principio di sostituibilità perché sono verificate sia la regola della segnatura, che quella dei metodi, che quella delle proprietà. In particolar modo, la regola della segnatura e quella delle proprietà sono banalmente verificate. Quella dei metodi è verificata in quanto le postcondizioni di **WeightedSetAll** sono più forti di quelle di **WeightedSet** (infatti **WeightedSetAll** precisa quali elementi vengono ritornati, mentre il padre questo lo lasciava indeterminato). Ciò è evidenziato dal fatto che la specifica del nuovo metodo è data dalla vecchia specifica, con una *aggiunta* di condizioni.

Esercizio 2 (punti 8)

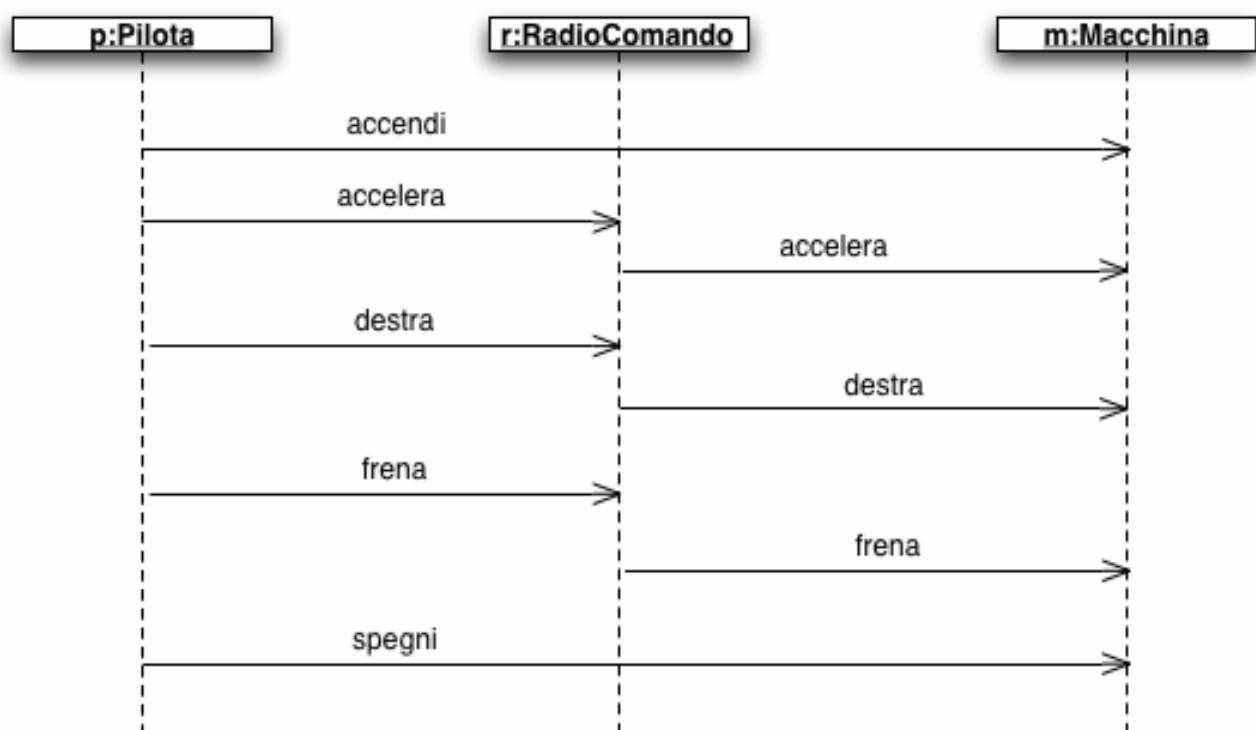
Una macchina radiocomandata è formata da due parti: la macchina stessa ed il radiocomando. Ci sono 2 tipi di comandi, quelli di direzione e quelli di velocità. Mediante i comandi di direzione si può scegliere se girare a destra o a sinistra; i comandi di velocità invece permettono di scegliere se accelerare (in questo caso, di quanto accelerare) o se frenare. Ogni radiocomando può essere configurato per funzionare su frequenze diverse.

La macchina, invece, si compone di un motore, di un sistema di trasmissione, di 4 ruote, di un ricevitore/controllore, e di 2 servocomandi. I servocomandi possono essere di 2 tipi, di direzione o di velocità. Una macchina ha un servocomando per la direzione (piazzato sulle ruote anteriori), e uno per la velocità (installato sul motore). Il ricevitore/controllore della macchina può anch'esso essere configurato per funzionare su diverse frequenze (la frequenza del ricevitore deve essere la stessa del radiocomando perché i due possano comunicare).

- Si rappresenti mediante un diagramma delle classi UML il sistema di cui sopra.



- b. Si scriva un Sequence Diagram UML che descrive la seguente situazione: il guidatore accende la macchina, quindi inizialmente accelera in direzione rettilinea, quindi curva a destra; durante la curva frena, quindi rimette le ruote in posizione rettilinea. Dopo avere ri-messo le ruote in posizione rettilinea frena, quindi spegne il motore.



Appello 23 luglio 2015



Politecnico di Milano
Anno accademico 2014-2015

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una):	<input type="checkbox"/> Baresi <input type="checkbox"/> Ghezzi <input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la seguente interfaccia Java Albero<T> che specifica i metodi pubblici di un ADT Albero. I nodi dell'albero sono oggetti di una classe immutabile, detta Nodo<T>, precisata nel seguito. Ogni nodo memorizza un valore di tipo T.

```
public interface Albero<T> {
    //OVERVIEW: Un albero con radice, dalla quale esiste un unico cammino ad ogni altro nodo.
    //Ogni nodo puo' avere 0, 1 o piu' figli, disposti in un ordine qualunque.
    //L'albero non puo' essere vuoto. Ogni nodo e' etichettato con un valore di tipo T.
    //ensures (* \result e' il numero di nodi dell'albero *);
    public /*@ pure */ int size();

    //ensures (* \result e' la radice dell'albero *)
    public /*@ pure */ Nodo<T> radice();

    //ensures (* \result e' un iteratore a tutti e soli i figli del nodo n
    //in ordine qualunque.*);
    public Iterator<Nodo<T> figli(Nodo<T> n);

    //ensures (* restituisce il padre del nodo n, lanciando eccezione
    //se n e' la radice o se n non esiste nell'albero*);
    public /*@ pure */ Nodo<T> padre(Nodo<T> n) throws InvalidNodeException;

    //ensures (* \result e' l'insieme di tutti e soli i nodi dell'albero *);
    public /*@ pure */ Set<Nodo<T>> nodi();

    //ensures (* inserisce come figlio del nodo n un nuovo nodo contenente x;
    //il nuovo nodo. Lancia eccezione se n non esiste nell'albero *);
    public Nodo<T> inserisci(Nodo<T> n, T x) throws InvalidNodeException;
}
```

Se utile, si consideri anche la seguente semplice implementazione di Nodo<T>:

```
public /*@ pure */ class Nodo<T> {
    private T valore;
    public Nodo(T x){valore = x;}
    public T getValore{return valore;}
}
```

Domanda a

Considerando come date le specifiche dei metodi puri `padre()`, `nodi()` e `radice()` descrivere in JML la specifica dei metodi `size()` e `inserisci()`. Definire anche, in base alla overview, un opportuno *invariante astratto*.

```
//ensures \result==nodi().size();
public /*@ pure */ int size();
```

Il metodo `inserisci` deve garantire che il nodo creato (rappresentato da `result`) è effettivamente un nodo nuovo, di valore x , ed è figlio del nodo n (che deve essere parte dell'albero al momento della chiamata). Inoltre, il resto dell'albero non cambia, ossia ogni nodo esistente è ancora presente nell'albero ed è ancora figlio dello stesso padre. Il lancio dell'eccezione comporta che il nodo n non sia parte dell'albero al momento della chiamata; occorre anche segnalare che l'albero no cambia (con una condizione molto simile a quella della postcondizione normale).

```
//ensures n !=null && \old(nodi().contains(n)) &&
//\old(nodi().contains(\result)) &&
//!\old(nodi().contains(\result)) &&
//\result !=null && \result.getValore().equals(x) &&
```

```

//@      padre(\result) == n &&
//@      nodi().size()==1+\old(nodi().size()) &&
//@      nodi().containsAll(\old(nodi()) &&
//@      (\forall Nodo<T> n1; \old(nodi().contains(n1));
//@          n1!= radice() ==> n1.padre() == \old(n1.padre()));
//@signals (InvalidNodeException e) (!\old(nodi().contains(n)) || n == null) &&
//@      nodi().equals(\old(nodi())) &&
//@      (\forall Nodo<T> n1; \old(nodi().contains(n1));
//@          n1!= radice() ==> n1.padre() == \old(n1.padre()));
public Nodo<T> inserisci(Nodo<T> n, T x);

```

Per esprimere le proprietà della Overview è sufficiente scrivere nell'invariante che l'albero abbia almeno un nodo e che ogni nodo diverso dalla radice abbia un padre.

```

//@public invariant size()>0 &&
//@ (\forall Nodo<T> n; nodi().contains(n);
//@      n == radice() || nodi().contains(padre(n)));

```

Domanda b

Si consideri ora la specifica della seguente interfaccia:

```

public interface AlberoBinario<T> extends Albero<T> {
    \\OVERVIEW: Un Albero in cui ogni nodo ha 0, 1 o 2 figli.
    //@ensures (*\result e' il figlio sx di n, se esiste*);
    public /*@ pure @*/ Nodo<T> sx(Nodo<T> n) throws InvalidNodeException;

    //@ensures (*\result e' il figlio dx di n, se esiste*);
    public /*@ pure @*/ Nodo<T> dx(Nodo<T> n) throws InvalidNodeException;
}

```

Un `AlberoBinario<T>` è sostituibile a un `Albero<T>` in base al principio di sostituzione di Liskov? La specifica dell'interfaccia è in grado di garantire il proprio invariante, così come descritto nella overview? Motivare la risposta.

SOLUZIONE: Si, in quanto regole della segnatura e dei metodi sono ovviamente verificate e la regola delle proprietà vale: un `AlberoBinario` è solo un caso particolare di un `Albero`. Tuttavia, la specifica della `inserisci()` stabilisce che è possibile inserire un numero arbitrario di figli per ogni nodo e quindi la proprietà invariante di essere un albero binario non può essere garantita.

Domanda c

Sia data ora la seguente classe che implementa `AlberoBinario<T>`.

```

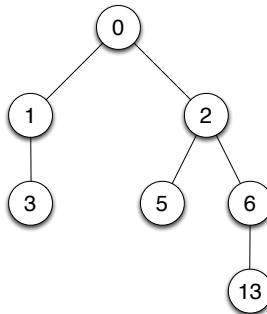
class AlbBinVettore<T> implements AlberoBinario<T> {
    private ArrayList<Nodo<T>> vettore;
    ...implementazione di tutti i metodi.
}

```

I nodi dell'albero binario sono memorizzati in un `ArrayList<Nodo><T>` `vettore` che ha le seguenti caratteristiche. È dato un metodo `p()` che per ogni nodo dell'albero ne calcola l'indice, numerando i nodi per livelli, da sinistra a destra, partendo da 0 per la radice: 1 e 2 per i suoi figli, poi 3 e 4 per i figli del nodo 1, ecc. Si noti che alcuni numeri possono essere saltati. Nelle posizioni che non corrispondono a un nodo, `vettore` vale `null`. Formalmente, `p$` è definita come segue:

- Se n è la radice dell'albero binario, $p(n) = 0$;
- Se n è il figlio sinistro di un nodo m , allora $p(n) = 2p(m) + 1$;
- Se n è il figlio destro di un nodo m , allora $p(n) = 2p(m) + 2$.

Quindi ad esempio:



Si scrivano invarianti di rappresentazione e funzione di astrazione della classe `AlbBinVettore<T>`.
SOLUZIONE:

RI afferma che il vettore non ha nodi duplicati (altrimenti non sarebbe un albero) e che ogni nodo, salvo la radice, ha un padre. Il padre di un nodo di indice i ha indice $(i - 1)/2$.

```

private invariant vettore!= null &&
    (\forall int i; 0\le i && i<vettore.size();
     vettore.get(i)!=null ==>
        i=0 || vettore.get((i-1)/2)!=null &&
        (\forall int j; i< j && j<vettore.size();
         vettore.get(i)\neq vettore.get(j)));

```

AF: La funzione di astrazione produce la radice del nodo e stabilisce che `nodi()` restituisce tutti e soli i nodi di `vettore` che sono diversi da `null`.

```

private invariant radice()==vettore.get(0) &&
    vettore.containsAll(nodi()) &&
    (\forall Nodo<T> n; vettore.contains(n);
     n!=null ==> nodi().contains(n)) &&
    (\forall Nodo<T> n; nodi().contains(n);
     vettore.contains(n));

```

La versione con `toString()` puo' in questo caso essere piu' precisa e restituire un stringa in cui i nodi sono disposti nell'ordine per livelli:

```

@Override
public String toString() {
    String s = vettore.get(0).toString() &&
        for (int i=1; i<vettore.size();i++) {

```

```

        int k = 1
            if (vettori.get(i) != null) s= s+ " " + vettore.get(i).toString()
        }
return s;

```

Si potrebbe anche inserire un carattere separatore fra ogni livello (quando i ha la forma 2^k-1 per un intero $k \geq 0$).

Esercizio 2

Si consideri la seguente classe Java

```

public class Counter {
    private int count = 0;
    void inc() { count++; }
    void dec() { count--; }
    int getCount() { return count; }
}

```

Si consideri anche la seguente classe che genera threads che usano Counter

```

public class TroubleMaker implements Runnable {
    private Counter count;

    public void run() {
        for(int i=0; i<10000; i++){
            if ( i%2 == 0 )
                count.inc();
            else
                count.dec();
            System.out.println("Count is: " + count.getCount());
        }
    }

    public TroubleMaker(Counter c) {
        count = c;
    }

    public static void main(String args[]) {
        Counter c = new Counter();
        for(int i=1; i<11; i++)
            (new Thread(new TroubleMaker(c))).start();
    }
}

```

Riportare le modifiche che occorre apportare per far sì che

1. L'accesso al contatore avvenga in mutua esclusione tra i diversi thread
2. Eventuali thread che invocano il decremento del contatore vengano sospesi fino a che il contatore non diventa positivo, onde evitare che esso possa diventare negativo.
3. Si consideri l'operazione `println` effettuata da un thread. Così come è scritto il programma non è assicurato che il valore stampato sia il valore prodotto dall'incremento (o decremento) che il task ha effettuato con la precedente istruzione. Si fornisca una sintetica spiegazione del perché. Si mostri anche quale modifica deve essere fatta al metodo `run` per assicurare che ciò invece avvenga.

SOLUZIONE:

1. Basta rendere synchronized tutti i metodi di Counter.

2. Si modifichi la classe Counter come segue:

```
public class Counter {  
  
    private int count = 0;  
    synchronized void inc() {  
        count++;  
        this.notifyAll();  
    }  
    synchronized void dec() {  
        while(count==0) {  
            try {  
                this.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        count--;  
    }  
    synchronized int getCount() {  
        return count;  
    }  
}
```

3. A causa del nondeterminismo, *count* potrebbe essere modificato da un altro thread subito dopo l'incremento o il decremento della variabile. Per ovviare al problema basta ad esempio che il metodo prenda un lock sull'oggetto *count*:

```
public void run() {  
    for(int i=0; i<10000; i++){  
        synchronized (count) {  
            if ( i%2 == 0 )  
                count.inc();  
            else  
                count.dec();  
            System.out.println("Count is: " + count.getCount());  
        }  
    }  
}
```

Non serve invece a nulla rendere il metodo synchronized (come sempre per i metodi run). Si potrebbe invece definire un oggetto lock su cui run si sincronizza, ma solo se non vi sono altri thread che accedono a count.

Esercizio 3

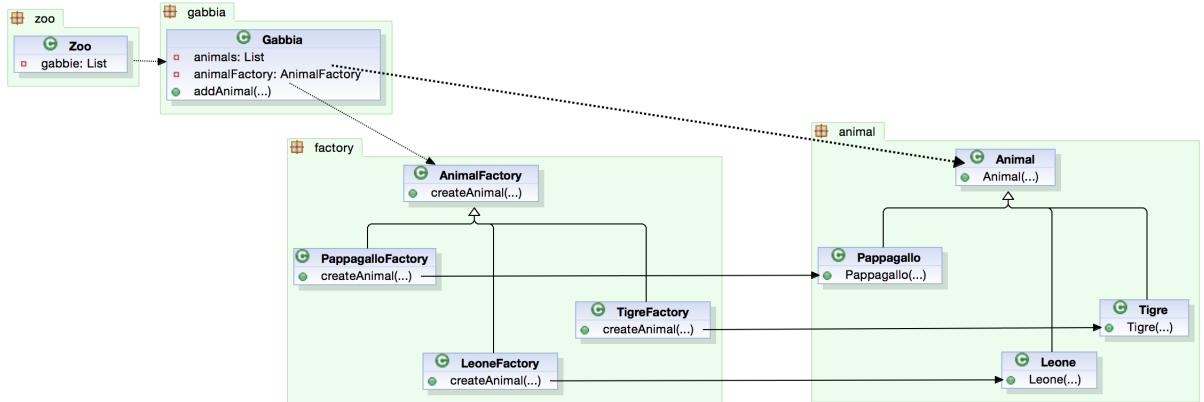
Uno Zoo è composto da più gabbie. Una Gabbia può contenere animali. Esistono tre tipi di animali: Tigre, Leone e Pappagallo.

1. Fornire il diagramma delle classi UML che specifica il problema.
2. Si vuole garantire che una Gabbia possa contenere animali di uno stesso tipo, ma che questo tipo non sia noto a priori. Per questo si usa il pattern *factory method*. Quando viene invocato il costruttore di

Gabbia, viene passato come parametro la factory che crea animali del tipo voluto. Questa factory viene chiamata quando viene invocato il metodo addAnimal su una Gabbia.

Mostrare le aggiunte al diagramma UML che rappresentano quanto detto sopra.

3. Fornire l'implementazione completa della classe Gabbia: rappresentazione, costruttore e metodo addAnimal.



```

public class Gabbia {

    private List<Animal> animals = new ArrayList<Animal>();
    private AnimalFactory animalFactory;

    public Gabbia(AnimalFactory af) {
        animalFactory=af;
    }
    public void addAnimal(){
        animals.add(animalFactory.createAnimal());
    }
}
  
```

Esercizio 4

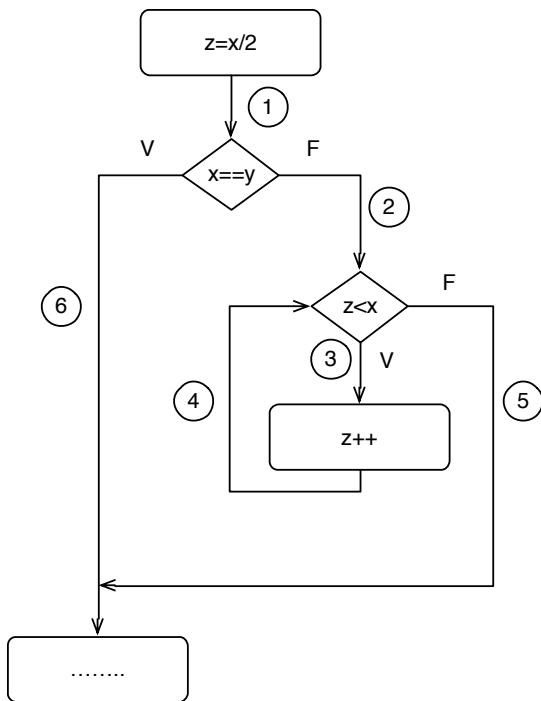
Si consideri questo frammento di programma che opera su interi:

```
static void method(int x, int y) {  
    int z = x/2;  
    if (x == y)  
        ...  
    else  
        while (z < x)  
            z++;  
}
```

1. Si definisca un insieme di casi di test che copre tutti i *branch*
2. Si calcoli il predicato che deve essere soddisfatto da un caso di test che
 - Entra nel ramo else e percorre il ciclo 0 volte;
 - Entra nel ramo else e percorre il ciclo 1 volta;
 - Entra nel ramo else e percorre il ciclo 2 volte;
 - Entra nel ramo then.
3. Per ciascun predicato, si sintetizzi il caso di test che provoca la copertura del relativo cammino.

SOLUZIONE

Punto 1



Dopo aver rappresentato il control flow è necessario coprire i branch/edge marcati con ①, ②, ③, ④, ⑤, ⑥.
 $x = 2, y = 2$ copre i branches ① and ⑥.

$x = 2, y = 1$ copre i branches ①, ②, ③, ④ and ⑤.

Punto 2a

$$x \neq y \wedge x \leq 0$$

Punto 2b
$$x \neq y \wedge 1 \leq x \leq 2$$
Punto 2c
$$x \neq y \wedge 3 \leq x \leq 4$$
Punto 2c
$$x = y$$
Punto 2d
$$x = 0, y = 1$$
$$x = 2, y = 3$$
$$x = 4, y = 3$$
$$x = y = 1.$$

Ingegneria del Software – a.a. 2005/06 sede di Cremona

Appello del 10 luglio 2006

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
 2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
 3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
 4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
 5. Tempo a disposizione: 1h30min.
 6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Total 100

Esercizio 1 (punti 10)

Il metodo statico *trasposta* ha come argomenti due interi $n, m \geq 1$, e due matrici A e T (di interi) costituite, rispettivamente, da n righe e m colonne, e da m righe e n colonne. Il metodo, senza alterare il contenuto di A, memorizza in T la matrice trasposta di A. Si rammenta che una matrice T è la trasposta di una matrice A se, e solo se, T è ottenuta da A scambiando le righe con le colonne. Si ricorda anche che in Java le matrici sono dichiarate come array di array. Ad esempio, A[0] è l'array che contiene la prima riga della matrice, A[1] la seconda, ecc. A[0][1] è quindi l'elemento A_{12} della matrice A.

a- Si scriva la specifica formale in JML del metodo.

//@ requires

.....
.....
.....

Sol: $n \geq 1 \& m \geq 1 \& A.length == n \& T.length == m \&$

$(\forall \text{int } i; 0 \leq i \& i < n; A[i].length == m) \& (\forall \text{int } i; 0 \leq i \& i < m; T[i].length == n);$

//@ensures

.....
.....
.....
.....

Sol: $(\forall \text{int } i; 0 \leq i \& i < n; \forall \text{int } j; 0 \leq j \& j < m; A[i][j] == T[j][i] \& A[i][j] == \text{old}(A[i][j]));$
Opzionalmente, si poteva aggiungere anche la clausola opzionale:

\@\Assignable T[*][*];

public static void trasposta (int n, int m, int [][] A, int [][] T)

b- Si trasformi la specifica JML del punto precedente sostituendo una (o più) a scelta delle precondizioni nel lancio di opportune eccezioni. Si mostri solo la parte di specifica modificata rispetto a quella definita al punto a.

Sol: si toglie ad esempio la condizione $n \geq 1 \& m \geq 1$ dalla requires, congiungendola alla clausola ensures.

Si aggiunge inoltre la clausola signals per lanciare ad esempio una EmptyException:
\@signals(EmptyException e) !(n>=1 & m>=1)

c- Si scelga opportunamente un insieme minimale di casi di test funzionali, che si possa considerare sufficiente per verificare il metodo statico, in base al criterio dei casi limite.

Sol: si possono considerare i seguenti casi, scegliendo una qualunque matrice A che soddisfi le condizioni date:

$n == 1, m == 1$

$n > 1, m == 1$

$n == 1, m > 1$

$n == m > 1$

$n > 1, m > 1, n != m$

Esercizio 2 (punti 15)

Si consideri la classe **Agenda** che rappresenta un insieme di appuntamenti (detti anche eventi). Ogni evento è un oggetto della classe **Evento**, specificata informalmente al punto b. La classe **Agenda** ha un metodo di inserimento, che aggiunge un evento se questo non esiste. Se l'evento è già presente in **Agenda**, viene sollevata l'eccezione **DuplicateException**. Si ha un *conflitto* quando si inserisce un evento sovrapposto anche solo per un minuto con un altro evento in **Agenda**. In tal caso, l'evento è comunque inserito in **Agenda**, ma viene segnalata l'eccezione **ConflictException**.

La classe **Agenda** ha la seguente specifica:

```
public class Agenda {  
    //OVERVIEW: Un'Agenda è un insieme mutabile, anche vuoto e senza duplicazioni, di Eventi.  
    //@ ensures (* costruisce un'Agenda vuota*);  
    public Agenda()  
  
    //@ requires e!=null;  
    //@ ensures inAgenda(e) && (\forall Evento e1; \old(inAgenda(e1)); inAgenda(e1));  
    //@ signals (ConflictException e) inAgenda(e) &&  
    //@ (\exists Evento e1; \old(inAgenda(e1)); e.confitto(e1));  
    //@ signals (DuplicateException e) \old(inAgenda(e)) ;  
    public void inserisci(Evento e) throws ConflictException, DuplicateException  
  
    //@ requires e!=null;  
    //@ ensures !inAgenda(e) && \old(inAgenda(e));  
    //@ signals (EventDoesNotExist exc) \old(!inAgenda(e));  
    public void rimuovi(Evento e) throws EventDoesNotExistException  
  
    //@ requires e!=null;  
    //@ ensures (\result==\exists Evento e1; this.eventiInData(e.dataEvento()); e1.equals(e));  
    public /*@ pure */ boolean inAgenda(Evento e)  
  
    //@ requires data!=null;  
    //@ ensures (* \result è la lista di tutti e soli gli eventi che accadono in data d *);  
    public /*@ pure */ ArrayList<Evento> eventiInData(Date d)  
}
```

a- Completare in JML, nelle parti con i puntini, la seguente specifica della classe **Evento**:

```
public /*@ pure */ class Evento {  
    //OVERVIEW: Un Evento è un oggetto immutabile, con alcune informazioni, una data e  
    //un orario di inizio e di fine. L'orario è nel formato hhmm: ad es. 1630 corrisponde alle  
    //ore 16.30. Due eventi sono in conflitto quando sono sovrapposti anche solo per un minuto.  
  
    //@ requires nome!=null && data !null && .....
```

```

.....  

//@ ensures ...  

.....  

public Evento(String nome, Date data, int orainizio, int oraFine)  

//@ requires .....  

//@ ensures .....  

public boolean conflitto(Evento e)  

//@ ensures (* \result è il nome dell'evento *);  

public String nomeEvento()  

//@ ensures (* \result è la data dell'evento *);  

public Date dataEvento()  

//@ ensures (* \result è l'orario di inizio dell'evento *);  

public int orarioInizio()  

//@ ensures (* \result è l'orario di fine dell'evento *);  

public int orarioFine()  

}

```

Sol: la prima requires richiede anche `orainizio<=oraFine && orainizio>= 0 && orainizio<=2359 && oraFine>= 0 && oraFine<=2359.`

Il metodo conflitto ha la specifica:

```

//@ requires e!=null;  

//@ ensures \result == (e.orarioInizio()>=this.orarioInizio && e.orarioInizio<=this.orarioFine) ||  

//@           (this.orarioInizio()>=e.orarioInizio && this.orarioInizio<=e.orarioFine) )

```

b- Si consideri la seguente estensione di Agenda e si scriva, *motivando la risposta nel modo più preciso possibile*, se verifica il principio di sostituzione:

La classe AgendaSenzaConflitti estende Agenda, modificando il metodo *inserisci(Evento e)* in modo che nel caso di conflitto di e con altri eventi dell'Agenda, il metodo non inserisca l'evento e, pur lanciando l'eccezione ConflictException.

Risposta:

Sol: No, perche' si violerebbe la regola della postcondizione più forte. Infatti la postcondizione di *inserisci* prevede l'inserimento dell'elemento anche in caso di conflitto: per non inserire l'elemento dovremmo pertanto indebolire la postcondizione.

c- Un possibile rep della classe Agenda è costituito da un array di eventi riempito nella parte iniziale e da un intero che indica il numero di eventi effettivamente memorizzati nell'array:

```

private Evento [] evList;
private int numEv;

```

Per ragioni di efficienza implementativa, nell'array gli eventi con la stessa data sono disposti in ordine crescente rispetto all'orario di inizio (così come definita dal metodo **orarioInizio()**), anche se non necessariamente in posizioni consecutive.

Scrivere in JML l'invariante di rappresentazione corrispondente a questa soluzione:

Sol: `private invariant evList !=null &&`

```

  (forall int i; 0<=i && i< numEv; (forall int j; i<j && j<= numEv;
    evList[i].dataEvento() == evList[j].dataEvento() ==>

```

```
evList[i].orarioInizio() <=evList[j].orarioInizio());
```



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Appello di Ingegneria del Software

9 Febbraio 2005

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h30m.
6. Punteggio totale a disposizione: 26/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 1 (punti 10)

Il metodo statico suGiuOgiuSu riceve come parametro un array *a* contenente tre numeri interi e restituisce al chiamante un valore di tipo *boolean*; se *a*[0]>*a*[1] allora se *a*[1]<*a*[2] suGiuOgiuSu restituisce true, altrimenti restituisce false; se *a*[0]<*a*[1] allora se *a*[1]>*a*[2] restituisce true, altrimenti false.

- a) Considerando il paragrafo precedente come una specifica del metodo suGiuOgiuSu *evidenziare un suo difetto* particolarmente rilevante, motivando adeguatamente la risposta.
- b) Sempre considerando il primo paragrafo come una specifica del metodo, *fornire 4 casi di test funzionali* per tale metodo, scegliendoli, e motivando tale scelta, sulla base del criterio delle combinazioni proposizionali.
- c) Si consideri la seguente implementazione del metodo, non necessariamente corretta rispetto alla specifica.

```
static boolean suGiuOgiuSu (int [] a) {  
    if ( a[0]>a[1] && a[1]<a[2] )  
        return true;  
    else    if ( a[0]<a[1] )  
        if ( a[1]>a[2] )  
            return true;  
        else return false;  
    else return false;  
}
```

Indicare, motivando la risposta, *quanti dati di test occorrono* per effettuare un test strutturale che copra tutte le diramazioni. Fornire un esempio di tali dati di test.

d) Per ottenere la copertura rispetto al criterio delle condizioni è *necessario aggiungere altri dati di test oltre a quelli indicati* al punto (c)? Se sì, indicare quali, se no, spiegare perché.

e) Con riferimento al punto (c), *definire un'eccezione TuttiUgualiException* di tipo checked, che possieda solo due costruttori, uno senza argomenti e uno con un argomento di tipo stringa. *Modificare il codice del metodo suGiuOgiuSu* in modo che esso lanci un'eccezione *TuttiUgualiException* nel caso in cui i tre elementi dell'array siano tutti uguali.

Esercizio 2 (punti 8)

Si vuole definire la classe *ExtremelyLongInt* che permetta di rappresentare e manipolare numeri interi anche estremamente elevati. Allo scopo di facilitare l'effettuazione di calcoli algebrici (p.es., il calcolo del massimo comune divisore e del minimo comune multiplo tra due oggetti *ExtremelyLongInt*) si sceglie di rappresentare un qualsiasi valore *ExtremelyLongInt* con un REP che consista nella scomposizione del numero in fattori primi.

```
class ExtremelyLongInt {  
    ...  
    // il REP  
    private int [] base; // contiene tutti i fattori primi dell'oggetto ExtremelyLongInt  
    private int [] exp; // contiene gli esponenti nello stesso ordine dei fattori  
    private int segno; // contiene +1 se il numero è positivo, -1 altrimenti  
    ...  
}
```

Per esempio, il valore astratto 360 di tipo *ExtremelyLongInt* ha una rappresentazione in cui segno =1. base = {2, 3, 5} ed exp = {3, 2, 1}, dal momento che $360 = 2^3 \cdot 3^2 \cdot 5^1$.

a) Scrivere (senza implementarlo) l'*invariante di rappresentazione* per tale astrazione, usando formule aritmetiche commentate con frasi in linguaggio naturale.

b) Scrivere (senza implementarla) la *funzione di astrazione*.

Esercizio 3 (punti 8)

1. Si descriva mediante un class diagram in UML i dati utilizzati dal seguente sistema di controllo degli accessi a un edificio.

Il sistema si compone di un controllore centrale e di una serie di cancelli agli accessi dell'edificio. Il controllore centrale mantiene anche un database con i dati degli utenti che possono accedere all'edificio. Ci sono 3 tipi di cancelli: a bassa, media, ed alta sicurezza. I cancelli a bassa sicurezza verificano l'identità degli utenti solo mediante un lettore di badge. I cancelli a media sicurezza, invece, verificano l'identità mediante un lettore di impronte digitali. Infine, i cancelli ad alta sicurezza verificano l'identità sia mediante un lettore di impronte digitali, che mediante un lettore di retina.

Ogni cancello ha un controllore locale, il quale riceve i dati dai vari lettori e comunica con il controllore centrale per verificare l'identità degli utenti. Ogni utente e' caratterizzato da un nome, da un badge, da delle impronte digitali, e dai dati della sua retina.

2. Si descriva mediante un sequence diagram il seguente caso di funzionamento del sistema.

Un utente arriva ad un cancello ad alta sicurezza, e fa leggere prima le impronte digitali, poi la retina all'apposito lettore. Ogni lettore spedisce i dati al controllore locale, il quale li rimanda al controllore centrale, e ne riceve indietro un oggetto con i dati dell'utente corrispondente. Se l'utente ricevuto dal controllore centrale e' lo stesso entrambe le volte, il controllore locale invia un segnale di apertura al cancello.

Ingegneria del Software — Soluzione del Tema 18/01/2022

Esercizio 1

Si consideri la seguente classe **immutable** Java `OlympicsTable` per la gestione dei risultati di un'edizione dei giochi olimpici. La classe permette di inserire i risultati (`Result`) per ogni disciplina (`Game`) e di consultare informazioni statistiche. Si assuma che ogni gara assegna esattamente una medaglia d'oro, una d'argento e una di bronzo, ovvero non sono possibili casi di parità o medaglie non assegnate.

```
public /*@ pure @*/ class OlympicsTable {
    // Ritorna l'insieme dei risultati.
    public Set<Result> getResults();

    // Aggiunge un risultato (creando e ritornando un nuovo oggetto di tipo OlympicsTable).
    // Lancia una DuplicateException se e' gia' presente un risultato per
    // la stessa disciplina e per lo stesso sesso.
    public OlympicsTable add(Result res) throws DuplicateException;

    // Ritorna l'elenco dei paesi i cui atleti hanno vinto una medaglia di tipo Medal.
    // L'elenco e' ordinato per numero di medaglie vinte, in ordine decrescente.
    public List<Country> getRank(Medal med);
}

public /*@ pure @*/ class Result {
    // Ritorna la disciplina di questa gara
    public Game getGame();

    // Ritorna il sesso degli atleti partecipanti
    public Sex getSex();

    // Ritorna l'atleta che ha ottenuto la medaglia Medal
    public Athlete getAthlete(Medal med);
    ...
}

public /*@ pure @*/ class Athlete {
    // Ritorna la nazionalita' dell'atleta
    public Country getNationality();
    ...
}

public enum Medal { GOLD, SILVER, BRONZE }
public enum Sex { F, M }
```

Domanda a)

Si specifichi in JML il metodo `add`.

Soluzione

```
//@requires res != null
//@
//@ensures
//@ !(\exists Result r; getResults().contains(r);
//@     r.getGame().equals(res.getGame()) && r.getSex().equals(res.getSex()) ) &&
//@ \result != null && \result.getResults().contains(res) &&
//@ \result.getResults().size() == getResults().size() + 1 &&
//@ \result.getResults().containsAll(getResults())
//@

//@signals (DuplicateException e)
//@ !(\exists Result r; getResults().contains(r);
//@     r.getGame().equals(res.getGame()) && r.getSex().equals(res.getSex()) )
public OlympicsTable add(Result res) throws DuplicateException;
```

Domanda b)

Si specifichi in JML il metodo `getRank`.

Soluzione

```
//@requires med != null
//@
//@ensures \result != null &&
//@ (\forall Country c; ; \result.contains(c) <=>
//@   (\exists Result r; getResults().contains(r);
//@     r.getAthlete(medal).getNationality().equals(c) ) ) &&
//@ (\forall int i; i>=0 && i<\result.size();
//@   (\forall int j; j>=0 && j<i;
//@     !\result.get(i).equals(\result.get(j)) &&
//@     (\exists Result r; getResults().contains(r);
//@       r.getAthlete(med).getNationality().equals(\result.get(i))) >=
//@       (\exists Result r; getResults().contains(r);
//@         r.getAthlete(med).getNationality().equals(\result.get(j))) ) )
public List<Country> getRank(Medal med);
```

Domanda c)

Si consideri un'implementazione di `OlympicsGame` in cui i risultati sono salvati in due array, uno per le gare femminili e uno per le gare maschili, come mostrato di seguito.

```
public /*@ pure @*/ class OlympicsTable {
    private Result femaleResults[];
    private Result maleResults[];
    ...
}
```

Per tale implementazione, fornire l'invariante di rappresentazione (representation invariant) e la funzione di astrazione.

Soluzione

L'invariante di rappresentazione indica che i due array non sono null, che non contengono risultati nulli o duplicati e che contengono solo risultati per gare femminili (`femaleResults`) e maschili (`maleResults`).

```
private invariant
femaleResults != null && maleResults != null &&
(\forall int i; i>=0 && i<femaleResults.length;
 femaleResults.get(i) != null && femaleResults.get(i).equals(Sex.F) &&
 (\forall int j; j>=0 && j<femaleResults.length;
 femaleResults.get(i).getGame().equals(femaleResults.get(j)) ) ) &&
(\forall int i; i>=0 && i<maleResults.length;
 maleResults.get(i) != null && maleResults.get(i).equals(Sex.M) &&
 (\forall int j; j>=0 && j<femaleResults.length;
 maleResults.get(i).getGame().equals(maleResults.get(j)) ) )
```

La funzione di astrazione si limita a definire `getResults` come unione di tutti e soli gli elementi presenti nei due array. Tutti gli altri metodi pubblici sono infatti definiti a partire da `getResults`.

```
private invariant
(\forall Result r; ; getResults().contains(r) <=>
 (\exists int i; i>=0 && i<femaleResults.length; femaleResults.get(i).equals(r)) ||
 (\exists int i; i>=0 && i<maleResults.length; maleResults.get(i).equals(r)) )
```

Domanda d)

Si consideri una classe `OlympicsGame2` che aggiunge un metodo puro `remove` così definito.

```
// Rimuove un risultato (creando e ritornando un nuovo oggetto di tipo OlympicsTable).
// Lancia una NoResultException se il risultato non e' presente.
public OlympicsTable remove(Result res) throws DuplicateException;
```

`OlympicsGame2` può essere definita come sottoclasse di `OlympicsGame` in accordo con il principio di sostituzione?

Soluzione

Si, in quanto `OlympicsGame2` non modifica alcun metodo di `OlympicsGame`, ma aggiunge un unico metodo che, essendo puro, non modifica l'istanza su cui è chiamato e non può quindi violare alcuna proprietà pubblica di `OlympicsGame`.

Esercizio 2

Si consideri la seguente classe Java che modella un testo fatto di un numero finito di paragrafi (stringhe immutabili).

```
public class Text {
    private String[] paragraphs;
    public Text(int maxPar) {
        paragraphs = new String[maxPar];
    }
    public synchronized void setPar(int parNum, String par) {
        paragraphs[parNum] = par;
    }
    public void clearPar(int parNum) {
        synchronized(paragraphs) {
            paragraphs[parNum] = null;
        }
    }
}
```

Domanda a)

La classe è correttamente sincronizzata? In caso affermativo motivare la propria risposta. In caso negativo proporre una modifica del solo metodo `setPar` che risolva il problema identificato.

Soluzione

La classe non è correttamente sincronizzata. Il metodo `setPar` è sincronizzato su `this` mentre il metodo `clearPar` è sincronizzato sull'attributo `paragraphs`. Dovremo sincronizzare anche il metodo `setPar` su `paragraphs`, come segue:

```
public void setPar(int parNum, String par) {
    synchronized(paragraphs) {
        paragraphs[parNum] = par;
    }
}
```

Domanda b)

Si modifichi il codice del metodo `setPar` affinché sospenda il chiamante se il paragrafo da inserire è già presente (ovvero è diverso da `null`). Si spieghi se e come vadano cambiati gli altri metodi affinchè la sincronizzazione rimanga corretta.

Soluzione

Il metodo `setPar` va modificato come segue:

```
public void setPar(int parNum, String par) throws InterruptedException {
    synchronized(paragraphs) {
        while(paragraphs[parNum] != null) paragraphs.wait();
        paragraphs[parNum] = par;
    }
}
```

bisogna inoltre aggiungere l'istruzione: `paragraphs.notifyAll()` all'interno del blocco sincronizzato nel metodo `clearPar`.

Domanda c)

Si aggiunga alla classe `Text` un metodo: `public void print()` che stampi, paragrafo per paragrafo, tutto il testo a terminale, **eseguendo in parallelo rispetto al chiamante**.

Soluzione

```
public void print() {
    String[] temp = new String[paragraphs.length];
    synchronized(paragraphs) {
        for(int i=0; i<paragraphs.length; i++) temp[i] = paragraphs[i];
    }
```

```
new Thread() {
    public void run() {
        for(String p : temp) if(p!=null) System.out.println(p);
    }
}.start();
}
```

oppure (usando una lambda come corpo del thread):

```
public void print() {
    String[] temp = new String[paragraphs.length];
    synchronized(paragraphs) {
        for(int i=0; i<paragraphs.length; i++) temp[i] = paragraphs[i];
    }
    new Thread( () -> {for(String p : temp) if(p!=null) System.out.println(p); }).start();
}
```

Esercizio 3

Si consideri il programma seguente:

```
1 static void function(int x, int y) {
2     if (x>0) {
3         while (x!=0 && y>0) {
4             y = y - x;
5             x = x - 2;
6             if (x<0) x = -x;
7         }
8     }
9 }
```

Si determini un insieme minimo di casi di test per ciascuno dei casi seguenti di copertura.

1. Copertura delle istruzioni (statement coverage)
2. Copertura delle decisioni (edge coverage)
3. Copertura delle decisioni e delle condizioni (edge and condition coverage)

Soluzione

1. Basta un caso di test, p.es. $x = 1, y = 1$, che è quindi anche l'insieme minimo.
2. Il caso precedente non copre le due diramazioni "nascoste" degli if (ovvero i casi in cui gli if vengono valutati negativamente). Servono almeno due casi, in quanto il caso $x = 0$ (y qualunque) copre la diramazione nascosta del primo if, senza eseguire più nulla. Per coprire entrambe le diramazioni del secondo if, occorre che il ciclo sia eseguito almeno due volte. Ad esempio usando $x = 3, y = 4$. L'insieme è quindi minimo.
3. Per coprire tutte le condizioni, occorre coprire l'uscita dal while sia con $x == 0$ che con $y <= 0$. Il caso precedente $x = 3, y = 4$ esce con y negativo; per uscire con $x == 0$ occorre aggiungere un terzo caso: ad esempio $x = 2, y = 3$, in modo che x diventi zero al termine della prima iterazione. Anche questo insieme è quindi minimo.

Esercizio 4

Si consideri il seguente programma Java.

```
public class Oven {
    public void bake(Pizza p) { p.bakedIn(type()); }
    public String type() { return "oven"; }
}

class ExpressOven extends Oven {
    public void bake(CheesePizza p) { p.bakedIn("very " + type()); }
    public void bake(VegetablePizza p) { p.bakedIn("really " + type()); }
    public String type() { return "fast oven"; }
}

abstract class Pizza {
    public void bakedIn(String oven) { System.out.println("Baked in " + oven); }
}

class CheesePizza extends Pizza {
    public void bakedIn(String oven) { System.out.println("Cheese baked in " + oven); }
}

class VegetablePizza extends Pizza {
    public void bakedIn(String oven) { System.out.println("Veg baked in " + oven); }
}
```

Il main è costituito dalle seguenti istruzioni, numerate per riferimento:

```
1 Pizza pizza = new Pizza();
2 CheesePizza pizzal = new VegetablePizza();
3 Pizza vegPizza = new VegetablePizza();
4 Pizza chsPizza = new CheesePizza();
5 CheesePizza chsPizzal = new CheesePizza();
6 Oven ov = new Oven();
7 Oven expr = new ExpressOven();
8 ExpressOven expr1 = new ExpressOven();
9 ov.bake(pizza);
10 ov.bake(pizzal);
11 ov.bake(vegPizza);
12 ov.bake(chsPizza);
13 ov.bake(chsPizzal);
14 expr1.bake(vegPizza);
15 expr1.bake(chsPizza);
16 expr1.bake(chsPizzal);
17 expr.bake(vegPizza);
18 expr.bake(chsPizza);
19 expr.bake(chsPizzal);
20 expr.bake(pizzal);
```

Domanda a)

Scrivere il numero di riga delle istruzioni (se ne esistono) che generano un errore in compilazione. Giustificare brevemente la risposta.

Soluzione

Righe 1 (si istanzia una classe astratta) e 2 (CheesePizza non è sottotipo di VegetablePizza). Di conseguenza, anche 9, 10 e 20.

Domanda b)

Supponendo di eliminare le eventuali righe che generano errori, si scriva, per ogni riga numerata, il valore stampato in uscita.

Soluzione

- 11: Veg baked in oven
- 12: Cheese baked in oven
- 13: Cheese baked in oven
- 14: Veg baked in fast oven
- 15: Cheese baked in fast oven
- 16: Cheese baked in very fast oven
- 17: Veg baked in fast oven
- 18: Cheese baked in fast oven
- 19: Cheese baked in fast oven

Appello 16 Settembre 2016



Politecnico di Milano
Anno accademico 2015-2016

Ingegneria del Software

Cognome:

LAUREANDO

Nome:

Matricola:

Sezione (segnarne una): Ghezzi

Mottola

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Esercizio 1

La classe Borsa realizza una collezione i cui elementi sono esclusivamente numeri interi nell'intervallo 0...MAX, con MAX una costante non negativa e stabilita al momento della costruzione. Gli elementi possono essere ripetuti con cardinalita' arbitraria.

La seguente specifica descrive in modo informale il comportamento della classe. *Non vi sono altre eccezioni oltre a quelle segnalate nel codice.*

```
public class Borsa {  
    //overview: Una collezione non ordinata di elementi  
    //il cui valore \`e limitato fra 0 e MAX.  
  
    //la costante MAX:  
    public final int MAX;  
    //costruisce una Borsa vuota con elementi fino a max>0  
    public Borsa(int max)  
  
    //inserisce un elemento  
    public void insert(int x) throws OutofRangeException  
  
    //rimuove una comparsa di un elemento e  
    //restituisce true se una comparsa dell'elemento  
    //\`e ancora presente dopo l'eliminazione,  
    //false altrimenti  
    public boolean remove(int x)  
  
    //restituisce il numero di comparsa dell'elemento i  
    public int card(int i)  
  
    //restituisce il numero di elementi distinti nella borsa  
    public int distinct()  
}
```

1. Si indichi quali metodi della classe Borsa sono puri. Scrivere inoltre la specifica JML per il costruttore e i metodi insert e remove della classe Borsa, eventualmente utilizzando un opportuno oggetto astratto tipico se ritenuto conveniente.
2. Scrivere in JML l'invariante pubblico della classe e argomentare che è verificato dalla specifica da voi definita.

Soluzione

La specifica di tutti i metodi descrive l'effetto della chiamata mostrando come cambiano i valori di card(i). Esplicitiamo, pur se non richiesto, anche la specifica di distinct(), che è semplicemente derivata da card(i).

```
public class Borsa {  
    //overview: Una collezione non ordinata di elementi  
    //il cui valore \`e limitato fra 0 e MAX.  
  
    //la costante MAX:  
    public final int MAX;  
    //@requires max>=0;  
    //@ensures MAX==max &&  
    //@( \forall int i; 0<=i && i<= MAX; card(i)==0 );  
    public Borsa(int max)
```

```

//@ensures 0<= x && x<= MAX &&
//@ (\forall int i; 0<=i && i<=MAX; card(x)==\old(card(x)) + (i==x ? 1 : 0);
//@signals (OutOfRangeException e) !(0<= x && x<= MAX) &&
//@ (\forall int i; 0<=i&&i<= MAX; card(i)==\old(card(i)));
public void insert(int x) throws OutofRangeException

//@ensures \result == (x>0 && x<=MAX && card(x)>0) &&
//@ (\forall int i; 0<=i&&i<= MAX;
//@ card(i)== \old(card(x)) - (i==x&&\old(card(x)>0 ? 1 : 0)));
public boolean remove(int x)

//restituisce il numero di comparse dell'elemento i
public /*@ pure @*/ int card(int i)

//@ensures \result == (\num_of int i; 0<=i && i<=MAX; card(i)>0);
public /*@ pure @*/ int distinct()
}

public invariant MAX>=0 && distinct()>=0 &&
(\forall int i; i<0 || i>MAX; card(i)==0) &&
(\forall int i; 0<=i && i<=MAX; card(i)>=0);

```

Esercizio 2

Nell'implementazione della classe Borsa dell'esercizio 1, si decide di rappresentare la Borsa come un array elems di dimensioni MAX+1, L'intero contenuto in posizione i di elems rappresenta il numero di comparse dell'elemento i nella Borsa. Un intero numDistinti memorizza quanti elementi distinti sono presenti nell'array (ossia proprio il valore restituito dal metodo distinct).

```
//rep:  
private int[] elems;  
private int numDistinti;
```

Si definiscano in JML l'invariante di rappresentazione e la funzione di astrazione. Inoltre, si implementi il metodo remove nella maniera ritenuta piu' opportuna.

Soluzione

Funzione di astrazione:

```
(\forall int i; 0<=i && i<=MAX; card(i)==elems[i];) &&  
numDistinti == distinct();
```

L'invariante di rappr. afferma che la lunghezza di elems è uguale a MAX+1, che elems contiene solo valori nonnegativi e stabilisce infine che numDistinti contiene effettivamente il numero di valori distinti in elems:

```
private invariant MAX+1==elems.length &&  
(\forall int i; 0<=i && i<=MAX; elems[i]>=0;) &&  
numDistinti == (\# of int j; 0<=j & j<=MAX; elems[j]>0);
```

Una semplice implementazione, basata sul RI, è:

```
public boolean remove(int x) {  
    if (x<0 || x>MAX || elems[x]==0)  
        return false;  
    elems[x]--;  
    if (elems[x]==0) {  
        numDistinti--;  
        return false;  
    }  
    return true  
}
```

Esercizio 3

Si consideri la seguente definizione di una classe Person:

```
public class Person {  
    private final String name;  
    private final int age;  
    public Person(final String theName, final int theAge) {  
        name = theName;  
        age = theAge;  
    }  
    public int ageDifference(final Person other) {  
        return age - other.age;  
    }  
    public String toString() {  
        return String.format("%s - %d", name, age);  
    }  
    ...  
}
```

Il seguente frammento di codice definisce un ArrayList people di Person e, a mero titolo di esempio, lo inizializza con alcuni valori.

```
final List<Person> people = Arrays.asList(  
    new Person("John", 20),  
    new Person("Sara", 21),  
    new Person("Jane", 21),  
    new Person("Greg", 35));
```

Si rammenta qui la definizione in Java 8 del notissimo interfaccia Comparator:

```
@FunctionalInterface  
public interface Comparator<T> {  
    // \result<0 se o1 minore di o2, =0 se uguali, altrimenti >0  
    int compare(T o1, T o2)  
    ...  
}
```

Scrivere un frammento di programma Java 8 in stile funzionale che stampa le persone contenute in people in ordine di eta' decrescente (nell'esempio, Greg, Sara, Jane, John).

Si suggerisce di utilizzare l'operazione sorted() definita per tutti gli stream, che prevede un argomento funzionale che sia un Comparator. Ad esempio, se la variabile stringCollection è una List di oggetti String (la cui classe fornisce un metodo compareTo), il seguente frammento definisce uno stream ordinato in ordine crescente.

```
stringCollection  
    .stream()  
    .sorted((a, b) -> b.compareTo(a))
```

Soluzione:

```
people.stream()  
    .sorted((person1, person2) -> person2.ageDifference(person1))  
    .forEach(System.out::println);
```

Si noti che la lambda (person1, person2) -> person2.ageDifference(person1) definisce una funzione anonima che rispetta l'interfaccia Comparator.

E' possibile scrivere una versione ancora piu' compatta:

```
people.stream()
    .sorted(Person::ageDifference)
    .forEach(System.out::println);
```

Esercizio 4

Si deve progettare un sistema per la gestione in tempo reale dei tempi di arrivo di autobus urbani presso le fermate della linea che percorrono. Una linea include almeno una fermata oltre al capolinea. Le fermate e il capolinea sono caratterizzate dalla loro posizione geografica, ad esempio in coordinate GPS. Un autobus è caratterizzato dalla linea che percorre, dalla posizione aggiornata in tempo reale, e dal fatto di essere completo oppure no. Ogni volta che l'autobus aggiorna la propria posizione, il sistema calcola il tempo stimato di arrivo a tutte le fermate della linea che sta percorrendo ed informa gli utenti del servizio tramite il loro smartphone, oltre che visualizzare i tempi di attesa presso pannelli informativi alle fermate stesse.

Si disegni un diagramma UML a scelta che illustra le relazioni tra le varie entità coinvolte. Nel caso in cui si ritenga di voler applicare un determinato pattern, si indichi quale e quali entità dell'applicazione rivestono quale ruolo nel pattern.

Soluzione:

Si tratta di un'applicazione del pattern Observer.

Esercizio 5

Si consideri il seguente frammento di codice Java:

```
public class ContoCorrente {  
    private double saldo;  
    private double numero;  
  
    public synchronized void deposita(double quantita){  
        saldo += quantita;  
    }  
  
    public synchronized void ritira(double quantita){  
        saldo -= quantita;  
    }  
  
    public synchronized void giroConto(ContoCorrente destinazione, double quantita){  
        this.ritira(quantita);  
        destinazione.deposita(quantita);  
    }  
}
```

Il frammento funziona correttamente in caso di accesso concorrente? In caso positivo, motivare la risposta. In caso negativo, mostrare un esempio del problema individuato e indicare come correggerlo.

Soluzione:

Il programma puo' andare in deadlock se, dati due conti correnti c1 e c2, un thread esegue ad es.
c1.giroConto(c2,100); e un altro thread esegue c2.giroConto(c1,50);.

Infatti, il primo thread deve bloccare c1, l'altro c2. Pero', l'esecuzione di c1.giroConto(c2,100) ha bisogno di prendere il lock anche su c2 per continuare; analogamente, l'esecuzione di c2.giroConto(c1,50); ha bisogno del lock su c1. I due thread restano quindi entrambi in attesa perenne che l'altro liberi il lock.

Si puo' risolvere il problema molto semplicemente eliminando la dichiarazione synchronized dalla dichiarazione di giroConto().

Si noti che il programma consente al saldo di diventare negativo (il metodo ritira non ha alcun vincolo): questo fatto puo' essere accettabile oppure no, a seconda della specifica del sistema, ma non è in questo caso un problema di concorrenza.

Ingegneria del Software – a.a. 2006/07

Appello del 23 luglio 2007

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 8)

Il seguente metodo statico

```
public static int rigaSommaMax(int [][] m) {... }
```

accetta come ingresso un array a due dimensioni m di cui si assume che rappresenti una matrice rettangolare (cioè tutte le righe hanno lo stesso numero di elementi; si noti che una matrice quadrata, in cui il numero delle righe è uguale a quello delle colonne, è considerato un caso particolare di matrice rettangolare, quindi accettabile come parametro) avente almeno due righe e due colonne, e produce come risultato l'indice della riga che i cui elementi hanno somma massima (se più righe hanno lo stesso valore massimo il metodo restituisce l'indice di una qualsiasi di queste).

a) Si scriva una specifica, in JML del metodo indicando la pre- e la post-condizione.

Soluzione

```
//@requires m!= null &&
// la matrice ha almeno due righe
//@ m.length>1 &&
// tutte le righe hanno almeno due elementi
//@ (\forall int i; 0<=i<m.length; m[i]!=null && m[i].length>1) &&
//tutte le righe hanno la stessa lunghezza
//@ (\forall int i; 0<=i<m.length; (\forall int j; i<j<m.length; m[i].length == m[j].length) )&&

//@ensures
// non esiste una riga per la quale la somma degli elementi sia maggiore che per la riga di indice \result
//@ !(\exists int i; 0<=i<m.length;
//@ (\sum int j; 0<=j<m[i].length; m[i][j]) > (\sum int j; 0<=j<\result.length; m[\result][j]))
```

b) si trasformi la specifica precedente facendo in modo che il metodo accetti in ingresso una qualsiasi “matrice” con almeno due righe e due colonne, ma se la matrice non è rettangolare (cioè le righe non sono tutte della stessa lunghezza) esso lanci un’eccezione *MatriceNonRettangolareException* (tutti gli altri requisiti sono invariati).

Soluzione

```
//@requires m!= null &&
// la matrice ha almeno due righe
//@ m.length>1 &&
// tutte le righe hanno almeno due elementi
//@ (\forall int i; 0<=i<m.length; m[i].length>1) &&

//@ensures
//tutte le righe hanno la stessa lunghezza
(\forall int i; 0<=i<m.length; (\forall int j; i<j<m.length; m[i].length == m[j].length) ) &&
// non esiste una riga per la quale la somma degli elementi sia maggiore che per la riga di indice \result
!(\exists int i; 0<=i<m.length;
(\sum int j; 0<=j<m[i].length; m[i][j]) > (\sum int j; 0<=j<\result.length; m[\result][j]))&&
//@signals (MatriceNonRettangolareException e)
//@ (! (\forall int i; 0<=i<m.length; (\forall int j; i<j<m.length; m[i].length == m[j].length) ))
```

c) Si indichi quale clausola deve essere *aggiunta* alla post-condizione del punto a) per specificare che, nel caso in cui la matrice abbia più righe con lo stesso valore massimo della somma degli elementi, viene restituito come risultato l'indice della prima tra queste (quella che ha cioè indice minimo).

Soluzione: occorre aggiungere la clausola

```
&& !(\exists int k; k<\result;
      (\sum int j; 0<=j<\m[k].length; \m[k][j]) ==
      (\sum int j; 0<=j<\result.length; \m[\result][j]))
```

Esercizio 2 (punti 13)

Si consideri un sistema di gestione del traffico telefonico. Il sistema, fra le altre cose, tiene traccia degli utenti e delle loro chiamate. Una chiamata ha una data, un'ora, una durata e un numero. Le chiamate possono essere vocali, e in tal caso hanno anche uno scatto alla risposta e un costo al secondo, o di tipo dati, che invece hanno un volume di dati scambiati (in Kbyte) e un costo in volume (ossia al Kbyte).

Formalmente, le tre classi sono:

```
abstract public class Chiamata {
    abstract public Data data();
    abstract public int ora(); //HHMMSS
    abstract public int durataInSecondi();
    abstract public int costoTotale();
    abstract public String numeroChiamato();
}

public class ChiamataVocale extends Chiamata {
    public int costoScatto();
    public int costoAlSecondo();
}

public class ChiamataDati extends Chiamata {
    public int costoKB();
    public int volumeKByte();
}
```

La specifica della classe Data è:

```
public classe Data {
    public int giorno();
    public int mese();
    public int anno();
    //@ ensures (*\result è true sse this precede strettamente d *)
    public boolean precede(Data d);
}
```

Per ogni utente, il sistema tiene traccia del nome, del credito residuo e delle chiamate effettuate. E' inoltre possibile effettuare una ricarica di un numero intero di euro.

L'interfaccia della classe è così definita:

```
public class Utente {
    //@ensures (* \result è il credito in centesimi di euro ancora disponibile *)
    public int creditoResiduo();
    //@ensures ....
```

```

public void ricarica(int euro);
//@ensures (*\result è un generatore, in ordine cronologico, delle chiamate effettuate alla data
d*)
public Iterator<Chiamata> chiamateInData(Data d);
}

```

a) Scrivere in JML la postcondizione del metodo ricarica.

SOLUZIONE: `this.creditoResiduo() == 100*euro + \old(this.creditoResiduo());`

Sia dato il seguente rep per la classe Utente:

```

private ArrayList<Chiamata> log;
//contiene tutte le chiamate effettuate in ordine cronologico
int spesaTotaleAnnoCorrente; //la spesa complessiva in centesimi di euro dell'utente
nell'anno ) corrente fino alla data attuale
Data dataCorrente; //la data attuale
int totaleRicariche; //il credito complessivo, in euro, acquistato dall'utente
int creditoResiduo; //il credito ancora disponibile

```

b) Scrivere l'invariante di rappresentazione che stabilisce la correttezza e la mutua consistenza dei dati che costituiscono il rep.

SOLUZIONE:

`log in ordine crescente && ultima chiamata del log in data <=dataCorrente && totaleRicariche è uguale al totale dei costi delle chiamate sommato al credito residuo && spesaTotaleAnno è pari al costo complessivo delle chiamate per il solo anno corrente.`

```

log !=null && dataCorrente!=null && (\forall int j; 0<= j && j<log.size(); log.get(j) !=null) &&
(\forall int j; 0<= j && j<log.size()-1;
    (log.get(j).data().precede(log.get(j+1).data()) ||
     log.get(j).data().equals(log.get(j+1).data()) &&
     log.get(j).ora()<log.get(j+1).ora()) &&
    (log.get(log.size()-1).data().precede(dataCorrente) ||
     log.get(log.size()-1).data().equals(dataCorrente) )
totaleRicariche == 100*(\sum int j; 0<= j && j<log.size(); log.get(j).costoTotale())+creditoResiduo
&&
spesaTotaleAnno ==
(\sum int j; 0<= j && j<log.size()&& log.get(j).data().anno()==dataCorrente.anno();
 log.get(j).costoTotale())

```

c) Con riferimento al rep introdotto al punto precedente, implementare il metodo iteratore chiamateInData(), includendo anche la definizione di tutte le classi appropriate.

SOLUZIONE:

```
public Iterator<Chiamata> chiamateInData(Data d) {return new UtenteGen(this,d);}
private static class UtenteGen implements Iterator<Chiamata> {
    private Utente p; //l'Utente sul cui log si vuole iterare
    private int n; //posizione del prox el. da considerare
    private Data dataAttuale;
    //@requires u!=null;
    UtenteGen(Utente u,Data d) {
        dataAttuale=d;
        p=u;
        for(int i; 0<=i && i<p.log.size() && p.log.get(i).data().precede(dataAttuale);i++);
        n=i;
    }
    public boolean hasNext () {
        return n<p.log.size() && dataAttuale.equals(p.log.get(n).data());
    }
    public int next () throws NoSuchElementException {
        if (!hasNext()) throw new NoSuchElementException();
        return p.log.get(n++);
    }
}
```

Esercizio 3 (punti 4)

Si consideri la seguente variante della classe Utente introdotta nell'esercizio 2, con parte dell'interfaccia definita come segue.

```
public class Utente {  
    ...  
    //@ensures (*\result è la somma del costo industriale di tutte le chiamate effettuate  
    // alla data d*)  
    public int costolIndustrialeChiamateInData(Data d);  
  
    //@ensures \result > costolIndustrialeChiamateInData(d) / 2 &&  
    // \result <= costolIndustrialeChiamateInData(d) * 2  
    public int importoFatturabileInData(Data d);  
}
```

Vengono inoltre definite le due classi UtenteAgevolato e UtentePrivilegiato, possibili classi eredi di Utente, che usufruiscono di tariffe particolarmente favorevoli.

```
public class UtenteAgevolato {  
    ...  
    //@ensures \result == costolIndustrialeChiamateInData(d) * 0.7  
    public int importoFatturabileInData(Data d);  
}  
  
public class UtentePrivilegiato {  
    ...  
    public final int TETTO = ...;  
    //@ensures \result ==  
    //@ costolIndustrialeChiamateInData(d) * 1.5 <= TETTO ?  
    //@ costolIndustrialeChiamateInData(d) * 1.5 : TETTO  
    public int importoFatturabileInData(Data d);  
}
```

Le classi UtenteAgevolato e UtentePrivilegiato sono definibili come classi eredi della classe Utente senza violare il principio di sostituzione? Motivare adeguatamente la risposta, nel caso positivo argomentando che sono soddisfatte le regole di sostituibilità, oppure nel caso negativo mostrando che non lo sono mediante un controesempio.

Soluzione

La classe UtenteAgevolato soddisfa il principio di sostituzione, perchè la postcondizione

$\text{\result} == \text{costolIndustrialeChiamateInData}(d) * 0.7$

Implica logicamente la postcondizione

$\text{\result} > \text{costolIndustrialeChiamateInData}(d) / 2 &&$
 $\text{\result} < \text{costolIndustrialeChiamateInData}(d) * 2$

Invece la classe UtentePrivilegiato non lo soddisfa, perchè l'imposizione del TETTO all'importo fatturabile può far sì che venga violata la condizione di fatturare almeno metà del costo industriale; in particolare, per valori di costolIndustrialeChiamateInData(d) sufficientemente elevati, si possono verificare entrambe le seguenti condizioni

$\text{costolIndustrialeChiamateInData}(d) * 1.5 > \text{TETTO}$

che fa sì che $\text{\result} == \text{TETTO}$, e

TETTO < costoIndustrialeChiamateInData(d)/2

violando quindi la postcondizione del metodo importoFatturabileInData che prescrive che
 $\text{result} \geq \text{costoIndustrialeChiamateInData}(d)/2$

Dipartimento di Elettronica e Informazione



Politecnico di Milano
20133 Milano (Italia)
Piazza Leonardo da Vinci, 32
Tel. (39) 02-2399.3400
Fax (39) 02-2399.3411
prof. **Giovanni Denaro**
prof. **Carlo Ghezzi**
prof. **Angelo Morzenti**
prof. **Pierluigi San Pietro**

Il Prova di Ingegneria del Software

1 Luglio 2002

Cognome

Nome

Matricola

Sezione (segnare una) Denaro Ghezzi Morzenti SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno presi in considerazione.
3. È possibile consultare libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.30m.
6. Punteggio totale a disposizione: 40 punti nominali (10 per ogni esercizio), corrispondenti a 13/30.

Esercizio 1:

Sono date le seguenti definizioni di una classe Stack e di una classe BoundedStack

```
public class Stack {  
    //OVERVIEW: una pila di elementi di tipo Object  
    public Stack() {  
        //EFFECTS: inizializza this alla pila vuota  
        public void push(Object v)  
            //EFFECTS: inserisce v in cima a this  
            public Object topPop() throws EmptyStackException  
                //REQUIRES: this.size()>0  
                //EFFECTS: restituisce e elimina cima di this  
                public int size()  
                    // EFFECTS: restituisce il numero di elementi di this  
    }  
  
    public class BoundedStack extends Stack {  
        //OVERVIEW: una pila di elementi di tipo Object di dimensioni massime pari a cento  
        public BoundedStack()  
            //EFFECTS: inizializza this alla pila vuota  
            public void push(Object v)  
                //REQUIRES: 0<=this.size()<=100  
                //EFFECTS: inserisce v in cima alla pila  
    }  
  
    Si supponga che il rep della classe Stack sia dichiarato privato.  
    La soluzione proposta verifica il principio di sostituzione di Liskov?  
    La risposta deve essere motivata.
```

Soluzione: La soluzione non verifica il principio di sostituzione perché la precondizione di BoundedStack.push (this.size()<100) è più forte di quella di Stack.push (true), violando la regola dei metodi.

Esercizio 2

Si consideri la classe Mystery riportata di seguito.

Si modifichi la classe in modo che gli oggetti che essa genera possano essere usati da più thread concorrenti, garantendo che:

1. i metodi *bango* e *bingo* vengano eseguiti da ciascun thread in mutua esclusione;
2. un thread che esegue *bingo* e trova che la condizione $n > m$ non è vera viene sospeso fino al momento in cui la condizione diventa vera a causa dell'esecuzione di *bango* da parte di un altro thread;
3. analogamente, un thread che esegue *bango* e trova che la condizione $m > l$ non è vera viene sospeso fino al momento in cui la condizione diventa vera a causa dell'esecuzione di *bingo* da parte di un altro thread.

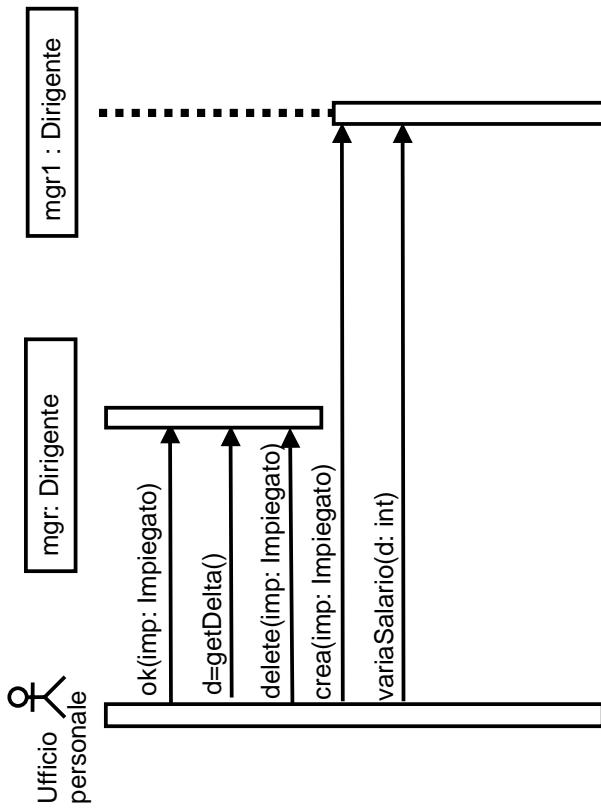
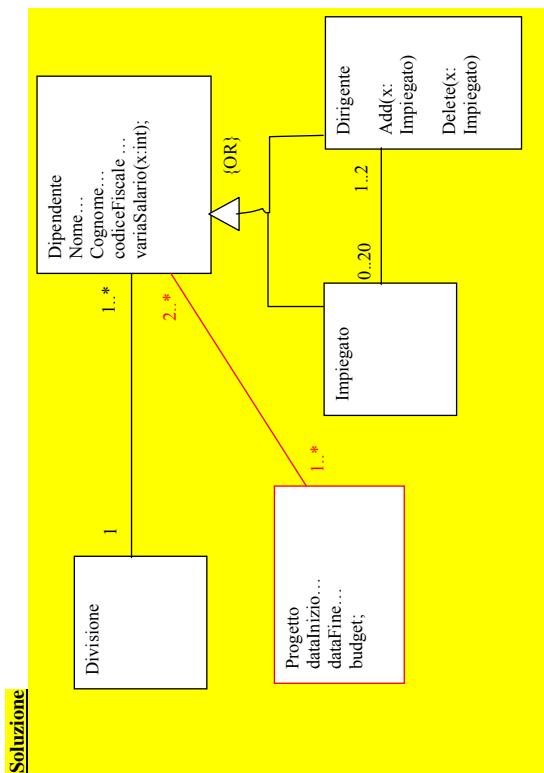
```
class Mystery {  
    private int n=10; private int m=1;  
    ...  
    public bingo {  
        if (n>m) {  
            n--; m++;  
            esegui operazione 1;  
        }  
        public bango {  
            if (m>l) {  
                n++; m--;  
                esegui operazione 2;  
            }  
        }  
    }  
}
```

Esercizio 3 (parte a)
Si consideri la fase di analisi e specifica dei requisiti di un nuovo software che deve essere realizzato e il seguente frammento di descrizione del dominio applicativo (la struttura di una certa azienda):

“Le divisioni aziendali hanno almeno 1 dipendente, ma il numero massimo non è predeterminato. Ogni dipendente ha un codice fiscale, un nome, un cognome e un salario; deve essere possibile modificare il salario mediante un’operazione variaSalario(x), dove x è un parametro intero che rappresenta la variazione di salario. Un dipendente è assegnato a una e una sola divisione. I dipendenti possono essere impiegati o dirigenti. Un dirigente coordina al massimo 20 impiegati e fornisce due operazioni, add e delete, con i quali si può aggiungere o eliminare un impiegato dall’insieme degli impiegati coordinati dal dirigente. Un impiegato è coordinato da uno o anche da due dirigenti.”

1. Rappresentare questa descrizione mediante un class diagram di UML.
2. Che cosa cambierebbe nel diagramma se si cambiisse il vincolo “un dirigente coordina al massimo 20 impiegati” in “un dirigente coordina da 5 a 20 impiegati”?
3. Che cosa cambierebbe nel diagramma se si aggiungesse questo frammento di specifica: “ogni dipendente lavora in almeno un progetto e in ciascun progetto lavorano almeno 2 dipendenti. Ogni progetto ha una data di inizio, una data di fine e un budget.”

Soluzione
public class Mystery {
.....
 private int n=10;
 private int m=1;
 public synchronized void bango {
.....
 try {
..... if (n>m)
..... try {
..... wait();
..... catch(InterruptedException e) {
..... n++;
..... m++;
..... esegui operazione 1;
..... notifyAll();
..... }
..... }
..... }
.....
 public synchronized void bingo {
..... while (! (m>1))
..... try {
..... wait();
..... catch(InterruptedException e) {
..... n++;
..... m--;
..... esegui operazione 2;
..... notifyAll();
..... }
..... }

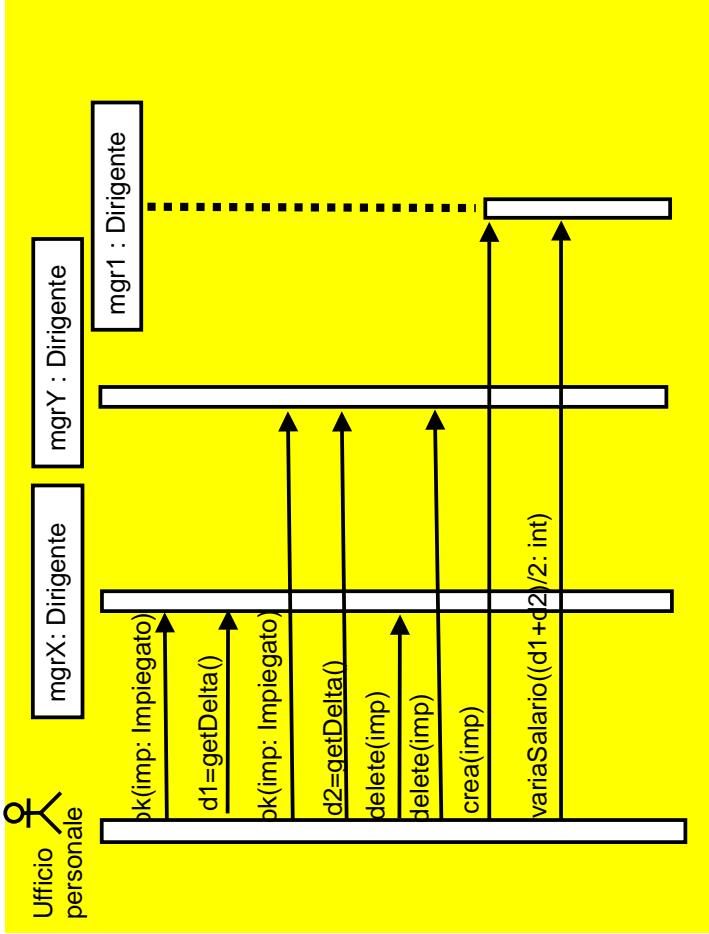


Esercizio 3 (parte b)

Si consideri il sequenze diagram riportato in figura, che descrive la promozione di un impiegato a dirigente da parte dell'ufficio personale. Il diagramma descrive il fatto che l'ufficio personale deve avere l'approvazione del dirigente da cui l'impiegato dipende e da questo deve farsi dare l'incremento di stipendio da applicare all'impiegato nella promozione a dirigente. (Si noti che il diagramma descrive solo lo scenario in cui il dirigente dà l'approvazione).

4. Si definisca un nuovo sequenze diagram che descrive il caso in cui il dipendente è coordinato da due dirigenti. Se entrambi danno l'approvazione, l'incremento di stipendio è il valor medio delle due proposte di incremento.

1. vedi diagramma delle classi qui sopra
2. sostituire 0..20 con 5..20
3. aggiungere la parte in rosso
4. vedi sequenze diagram qui sotto



Esercizio 4 (parte a)

Si consideri la specifica del seguente metodo:

```

static int triangolo (int a, int b, int c) {
    // REQUIRES: a>0 and b>0 and c>0
    // EFFECTS: restituisce 1 se i tre lati definiscono un triangolo equilatero,
    // 2 se definiscono un triangolo isoscele,
    // 3 se definiscono un triangolo scaleno,
    // 0 se a, b, e c non possono essere lati di un triangolo
}

```

Si definisca un insieme di dati di test in base a una strategia di tipo “black box” (cioè funzionale), motivando sinteticamente ciascuna scelta.

Esercizio 4 (parte b) Si consideri la seguente implementazione (che non tiene conto dell’ultima parte della clausola EFFECTS) del metodo precedente:

```

static int triangolo (int a, int b, int c) {
    if ((a==b) && (b==c)) return 1;
    if ((a==b) || (b==c) || (a==c)) return 2;
    else return 3;
}

```

Si consideri la seguente formulazione di una strategia di tipo “white box”(cioè strutturale): deve esistere, per ciascun cammino percorribile del flusso di controllo, uno e un solo dato di test che ne genera la copertura.

1. Quanti casi di test sono necessari? _____
2. Si descrivano i test scelti:

Esercizio 4 (parte c) Si consideri la seguente implementazione alternativa della funzione precedente:

```

static int triangolo (int a, int b, int c) {
    if ((a==b) && (b==c)) return 1;
    else if (a==b) return 2;
    else if (b==c) return 2;
    else if (a==c) return 2;
    else return 3;
}

```

Considerando sempre la precedente formulazione del criterio di copertura dei cammini:

1. Quanti casi di test sono necessari per questo programma? _____
2. Si descrivano i test scelti:

Soluzione a:

I dati $\langle 5, 5 \rangle$, $\langle 5, 7, 7 \rangle$, $\langle 5, 6, 7 \rangle$ e $\langle 5, 6, 12 \rangle$ corrispondono alle quattro parti della clausola EFFECTS e per essi il metodo deve restituire, rispettivamente, i valori 1, 2, 3 e 0.

Soluzione b:

Servono 3 dati di test. I dati possono essere $\langle 5, 5, 5 \rangle$, $\langle 5, 5, 7 \rangle$, $\langle 5, 6, 7 \rangle$.

Soluzione c:

Servono 5 dati di test. I dati possono essere $\langle 5, 5, 5 \rangle$, $\langle 5, 5, 7 \rangle$, $\langle 5, 7, 7 \rangle$, $\langle 5, 7, 5 \rangle$, $\langle 5, 6, 7 \rangle$.



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da Vinci,

32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Recupero di Ingegneria del Software – 17 Settembre 2002

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno presi in considerazione.
3. È possibile consultare liberamente libri, manuali o appunti, e scrivere a matita. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h30.
6. Punteggio totale a disposizione: 26 punti. La prova è sufficiente ottenendo almeno 16 punti.

Esercizio 1 (12 punti)

Le seguenti classi Punto, Segmento, Triangolo, TriangoloIsoscele rappresentano le note entità geometriche in uno spazio euclideo bidimensionale con coordinate cartesiane.

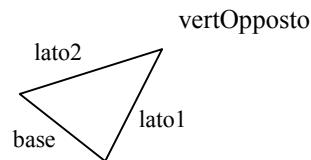
```
public class Punto {  
    // Punto è immutabile  
    private float x, y;  
    public Punto(float x, float y) {this.x = x; this.y = y; }  
    public boolean equals (Punto p) {  
        return (p.x ==x && p.y ==y);  
    }  
    public float distanza (Punto p) {  
        return (float)Math.sqrt((p.x-x) * (p.x-x) +(p.y-y) * (p.y-y));  
    }  
}
```

Della classe Segmento si riporta qui solo la specifica (da non implementare):

```
public class Segmento {  
    //tipo immutabile, è una coppia ordinata di punti distinti .  
    public Segmento(Punto inizio, Punto fine) {  
    //REQUIRES: inizio e fine sono distinti  
    // EFFECTS: this come segmento di due punti: (inizio, fine)  
    }  
    public Punto puntoIniz() {  
        //EFFECTS: restituisce l'estremo iniziale di this  
    }  
    public Punto puntoFin() {  
        //EFFECTS: restituisce l'estremo finale di this  
    }  
    public float lung() { //EFFECTS: restituisce la lunghezza di this  
    }  
} // fine di Segmento
```

a) Completare negli spazi previsti l'implementazione della classe Triangolo, inclusi AFe RI
(supporre come data la definizione della classe TriangoloScorrettoException):

```
public class Triangolo {  
    // tipo mutabile, corrispondente a un triangolo NON DEGENERE rappresentato  
    // internamente da una terna di segmenti  
    //Uno dei tre segmenti è individuato come la base del triangolo  
    private Segmento base, lato1, lato2;  
    private static boolean verificaTriangolo(Segmento lato0, Segmento lato1,Segmento lato2) {  
        //EFFECTS: restituisce true se lato0, lato1, lato2 individuano un triangolo non degenere  
        //NB: da non implementare....  
  
        public Triangolo (Punto p0, Punto p1, Punto p2) throws TriangoloScorrettoException {  
            // EFFECTS: se p0, p1, p2 individuano un triangolo non degenere, costruisce un triangolo  
            // con base(p0,p1), lato1(p1,p2), lato2(p2,p0) altrimenti lancia TriangoloScorrettoException  
            this.base = new Segmento(p0, p1); .....  
            this.lato1 = new Segmento(p1, p2); .....  
            this.lato2 = new Segmento(p2, p0); .....  
            if (!verificaTriangolo(base,lato1,lato2)) throw new  
            TriangoloScorrettoException(); .....  
        }  
  
        //completare AF  
        AF(this) = il triangolo individuato dai tre vertici base.puntoIniz(), base.puntoFinale(),  
        lato1.puntoFinale().....
```



b) Dimostrare che RI è un invariante per Triangolo

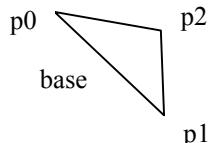
b) Dimostrare che RI è un invarianto per Triangolo
Traccia soluzione: il costruttore costruisce base, lato1, lato2 diversi da null e che verificano le altre condizioni dello RI (se non fossero verificate verrebbe lanciata eccezione).
Anche cambiaVertOpposto verifica tutte condizioni dello RI.

c) Si completi inoltre la seguente classe TriangoloIsoscele. Si dica se il principio di sostituzione di Liskov può essere verificato. Si commenti opportunamente la risposta.

```
public class TriangoloIsoscele extends Triangolo {
    // tipo mutabile, corrispondente a un triangolo isoscele NON DEGENERE
    //completare, solo se ritenuto necessario, il rep
    il rep non va completato.....
    // completare AF:
    AF(this) =AF(super).....
    // completare fornendo una descrizione dell'invariante di rappresentazione
    RI(this) = RI(super) && this.lato1.lung() == this.lato2.lung()

    // implementare RI con il metodo RepOK()
    public boolean RepOK() {
        return (super.RepOk() && this.lato1().lung() == this.lato2().lung());
        //si noti che devono essere chiamati i metodi lato1() e lato2(),
        //perché le due variabili omonime sono private nella superclasse
        .....
        .....
        .....
        .....
        .....

    //completare il costruttore:
    public TriangoloIsoscele (Punto p0, Punto p1, Punto p2) .....throws
    TriangoloScorrettoException{
        // EFFECTS: se p0, p1, p2 individuano un triangolo isoscele non degenere
        // (in modo che le lunghezze dei segmenti (p0,p2) e (p1,p2) siano uguali),
        // costruisce un triangolo con base (p0,p1), altrimenti lancia TriangoloScorrettoException
```



```
super(p0,p1,p2);
if (this.lato1().lung != this.lato2().lung) throw new
TriangoloScorrettoException();.....
```

```
.....
```

```
.....
```

} //fine costruttore
// Inserire qui la specifica e l'implementazione di altri metodi di Triangolo che si ritiene *necessario* (ri)definire in TriangoloIsoscele

Il princ. di Liskov non vale perche' la cambiaVerticeOpposto deve essere ridefinita in modo da garantire che il triangolo sia isoscele: ha quindi di fatto una precondizione più forte (inclusa nella clausola EFFECTS).

```
public void cambiaVerticeOpposto(Punto p) throws TriangoloScorrettoException {
    //EFFECTS: se this.base.puntoIniz(),this.base.puntoFin() e p individuano
    // un triangolo non degenere e ISOSCELE,
    //allora this.lato1.puntoFin()_post = this.lato2.puntoIniz()_post = p;
    //altrimenti lancia TriangoloScorrettoException
    super.cambiaVerticeOpposto(p);
    if (this.lato1().lung()!=this.lato2().lung()) throws new TriangoloScorrettoException();
}
```

.....
.....
.....
.....
.....

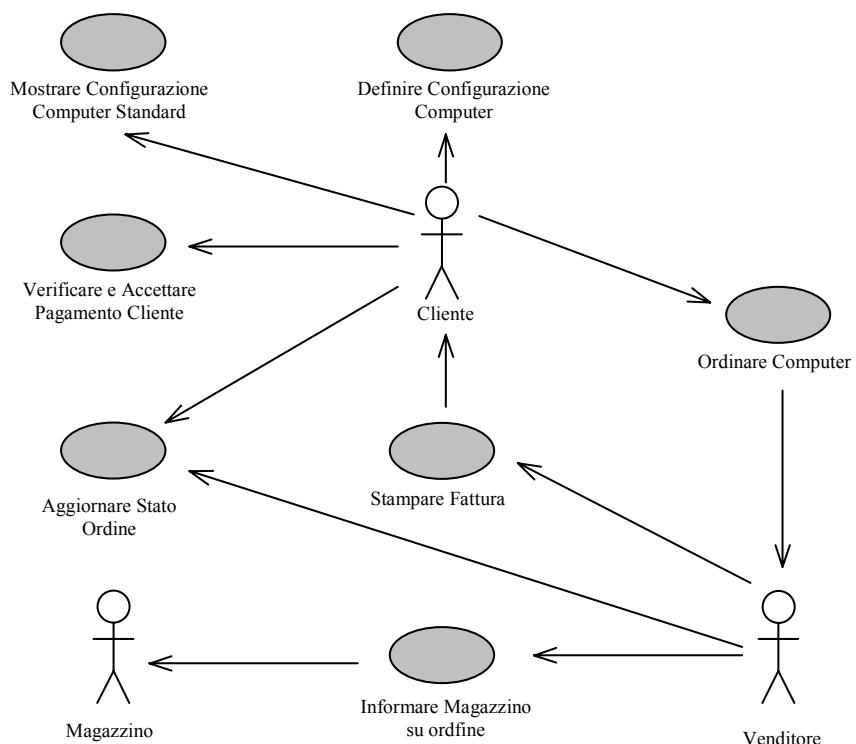
//fine Triangolososcele

Esercizio 2 (8 punti)

Un produttore di computer offre la possibilità di acquistare computer via internet. Un cliente può selezionare un computer sulla pagina web del produttore. I computer sono classificati in server, desktop e portatili. Il cliente può selezionare una configurazione standard o può definire interattivamente la configurazione desiderata. I componenti configurabili (a esempio le memorie) sono presentati in una lista di opzioni disponibili. Il cliente può ottenere una valutazione del costo di ogni configurazione di computer. Per ordinare, il cliente deve fornire le informazioni per la consegna e il pagamento. Le modalità di pagamento accettate sono: carta di credito e assegno. Una volta che l'ordine è stato inviato, il sistema manda un e-mail di conferma al cliente con i dettagli dell'ordine. Il cliente, mentre attende l'arrivo del computer, può in ogni momento controllare *on line* lo stato dell'ordine. Inoltre il sistema informativo di vendita verifica la solvibilità del cliente, richiede al magazzino la configurazione ordinata, stampa la fattura, richiede al magazzino di spedire il computer al cliente.

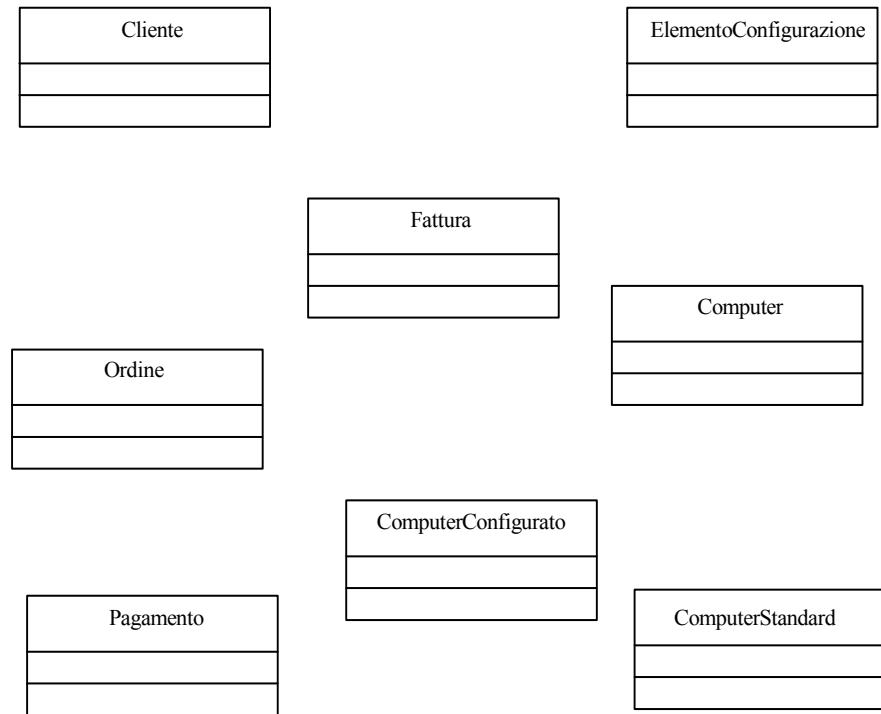
- a) Immaginando che nella specifica dei requisiti in UML siano presenti i tre attori: **cliente, venditore, magazzino**, disegnare i diagrammi di casi d'uso per il sistema sopra descritto.

Sol.

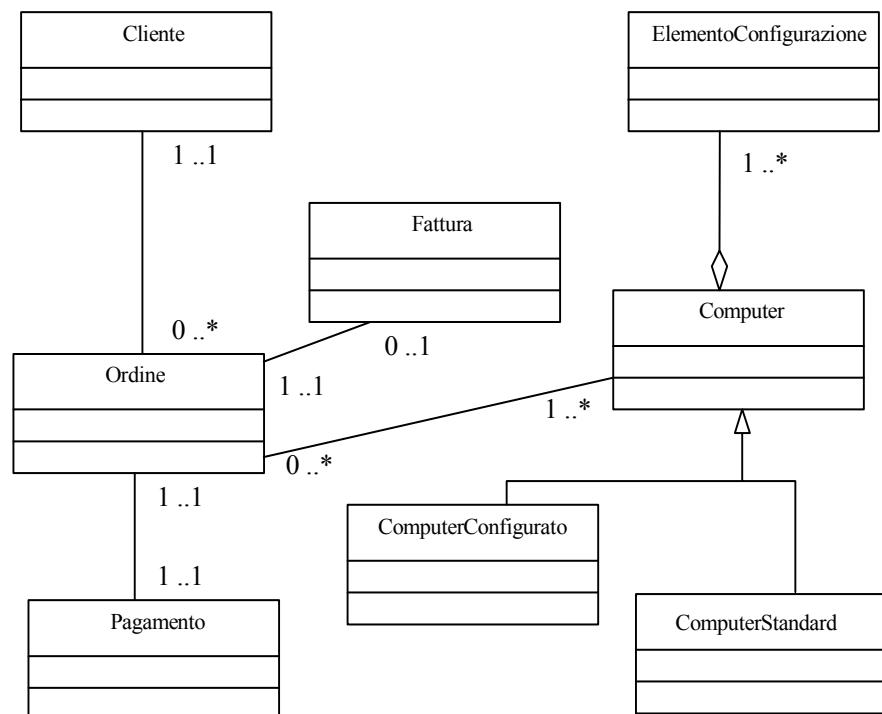


**b) Descrivere a parole un possibile scenario significativo e illustrarlo con un sequence diagram.
Sol ovvia.**

c) Assumendo di aver individuato le classi mostrate in figura, costruire a partire da essa un diagramma delle classi che includa informazioni relative alle associazioni tra classi (indicando le relative molteplicità), aggregazioni e generalizzazioni.



Sol



Esercizio 3 (6 punti)

1. Si individuino dati di test per il costruttore public TriangoloIsoscele di cui all'Esercizio 1 secondo una strategia di tipo “black box”, specificando il motivo per cui ciascun dato viene scelto.
2. Si individuino, per lo stesso costruttore, dati di test di tipo “white box” che coprano tutti i cammini della implementazione data nell'Esercizio 1.

Soluzione

1. bisogna considerare tutti i modi con i quali si puo' generare una TriangoloScorrettoException: un lato maggiore della somma degli altri due, due angoli coincidenti, tre angoli allineati, i due lati diversi dalla base di lunghezza non uguale; inoltre va considerato il caso di triangolo non degenere e isoscele.
2. a un esame superficiale, esistono nell'implementazione due cammini, corrispondenti al caso in cui il triangolo e' isoscele oppure no. Basterebbero quindi per coprire tutti i cammini due casi di test. In realta', bisogna tener presente che la chiamata di super puo' a sua volta generare una eccezione, che provoca l'immediata uscita dal costruttore della sottoclassa con la stessa eccezione. Pertanto in tal caso esistono tre cammini, e quindi 3 casi di test per la copertura secondo un metodo “white box”.

Ingegneria del Software — Soluzione del Tema 13/07/2021

Esercizio 1

Si consideri la seguente classe Java MusicLibrary per la gestione di una collezione di brani musicali. Ogni brano è rappresentato dalla classe immutabile Song che contiene alcune informazioni tra cui l'artista o gruppo musicale (Artist) che esegue il brano e la data (Date) in cui il brano è stato registrato, come riportato di seguito.

```
public class MusicLibrary {
    // Ritorna l'insieme di artisti che eseguono almeno un brano nella collezione.
    public /*@ pure @*/ Set<Artist> getArtists();

    // Ritorna la lista di brani eseguiti dall'artista artist.
    // Lancia una UnknownArtistException se la collezione non contiene brani di artist.
    public /*@ pure @*/ List<Song> getByArtist(Artist artist) throws UnknownArtistException;

    // Ritorna l'insieme dei brani eseguiti in data date.
    // Lancia una UnknownDateException se nessun brano della collezione e'
    // stato composto in data date.
    public /*@ pure @*/ Set<Song> getByDate(Date date) throws UnknownDateException;

    // Aggiunge il brano alla collezione.
    public void addSong(Song s);
}

public /*@ pure @*/ class Song {
    // Ritorna l'artista o gruppo che esegue il brano
    public Artist getArtist();

    // Ritorna la data di registrazione del brano
    public Date getDate();
}
```

Domanda a)

Si specifichi in JML il metodo getByDate().

Soluzione

Definiamo getByDate() in funzione di getByArtist() e getArtists().

```
//@requires date != null
//@

//@ensures \result != null && !\result.isEmpty() &&
//@ (\forall Artist a; getArtists().contains(a);
//@   (\forall Song s; getByArtist(a).contains(s);
//@     \result.contains(s) <==> date.equals(s.getDate())))
// (\forall Song s; \result.contains(s);
//@   s.getDate().equals(date) && getArtists().contains(s.getArtist()) &&
//@   getByArtist(s.getArtist()).contains(s))
//@

//@signals (UnknownDateException e)
//@ (\forall Artist a; getArtists().contains(a);
//@   (\forall Song s; getByArtist(a).contains(s); !date.equals(s.getDate())))
public /*@ pure @*/ Set<Song> getByDate(Date date) throws UnknownDateException;
```

Domanda b)

Si specifichi in JML il metodo addSong().

Soluzione

```
//@requires s != null
//@

//@ensures
//@ getArtists().contains(s.getArtist()) &&
//@ getByArtist(s.getArtist()).contains(s) &&
```

```

//@
//@ (\forall Artist a; \old{getArtists()}.contains(a) && !a.equals(s.getArtist());
//@   getArtists().contains(a) && getByArtist(a).size() == \old{getByArtist(a)}.size() &&
//@   getByArtist(a).containsAll(\old{getByArtist(a)})) &&
//@
//@ !\old{getArtists()}.contains(s.getArtist()) ==>
//@   getArtists().size() == \old{getArtists()}.size() + 1 &&
//@   getByArtist(s.getArtist()).size() == 1 &&
//@ \old{getArtists()}.contains(s.getArtist()) ==>
//@   getArtists().size() == \old{getArtists()}.size() &&
//@   getByArtist(s.getArtist()).size() == \old{getByArtist(s.getArtist())}.contains(s) ?
//@                               \old{getByArtist(s.getArtist())}.size() :
//@                               \old{getByArtist(s.getArtist())}.size() + 1 &&
//@   getByArtist(s.getArtist()).containsAll(\old{getByArtist(s.getArtist())})
public void addSong(Song s) ;

```

Domanda c)

Si consideri un'implementazione che utilizza un Set per contenere le canzoni, come mostrato di seguito.

```

public class MusicLibrary {
    private final Set<Song> songs;
    ...
}

```

Per tale implementazione si definiscano l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

Nell'invariante di rappresentazione escludiamo che il Set sia nullo o contenga valori nulli.

```

//@(* Representation invariant RI *)
//@private invariant songs != null &&
//@ (\forall Song s; songs.contains(s); s != null)

```

La funzione di astrazione definisce `getArtists()`, `getByArtist()` in funzione del Set usato nella rappresentazione. Gli altri metodi pubblici sono infatti definiti a partire da essi.

```

//@(* Abstraction function AF *)
//@private invariant
//@ (\forall Artist a; ; getArtists().contains(a) <=>
//@   (\exists Song s; songs.contains(s); s.getArtist().equals(a)) ) &&
//@ (\forall Song s; s != null && getArtists().contains(s.getArtist()));
//@   getByArtist(s.getArtist()).contains(s) <=> songs.contains(s) )

```

Domanda d)

Si consideri la classe `SortedMusicLibrary` che modifica la specifica del metodo `getByArtist()` per garantire che la lista ritornata contenga le canzoni in ordine di data di registrazione (dalla meno alla più recente). È possibile definire `SortedMusicLibrary` come sottoclasse di `MusicLibrary` in accordo con il principio di sostituzione?

Soluzione

È possibile in quanto la classe si limita a rafforzare la post-condizione del metodo `getByArtist()` della sopraclasse.

Esercizio 2

Si consideri la seguente classe Java:

```
public class RainfallDB {  
    private List<Double> data;  
    public RainfallDB() {  
        data = new ArrayList<Double>();  
    }  
    public double average() {  
        double tot = 0.0;  
        for(double rf : data) tot += rf;  
        return tot / data.size();  
    }  
    public RainfallDB addReading(double rainfall) {  
        RainfallDB temp = new RainfallDB();  
        temp.data.addAll(this.data);  
        temp.data.add(rainfall);  
        return temp;  
    }  
}
```

Domanda a)

È possibile invocare i metodi della classe RainfallDB da thread paralleli senza rischi di conflitti nell'accesso ai dati? Si motivi la propria risposta e in caso di risposta negativa si spieghi come cambiare il codice della classe per ovviare ai problemi di sincronizzazione identificati.

Soluzione

La classe rappresenta oggetti immutabili e come tale non presenta necessità di ulteriore sincronizzazione.

Domanda b)

Si aggiunga alla classe precedente un metodo:

```
public void clear() { ... }
```

che "svuota" la collezione di dati, garantendo che la classe resti correttamente sincronizzata (si indichi se necessario come cambiare gli altri metodi).

Soluzione

Il metodo deve essere implementato come segue (come metodo sincronizzato):

```
public synchronized void clear() {  
    data.clear();  
}
```

e tutti gli altri metodi andranno anch'essi sincronizzati visto che la classe non rappresenta più oggetti immutabili.

Domanda c)

Si aggiunga un ulteriore metodo:

```
public void clone(RainfallDB db) {
```

che sostituisce agli elementi dell'oggetto corrente (this) quelli contenuti nell'oggetto db passato come parametro.

Soluzione

L'implementazione del metodo `clone` va accuratamente sincronizzata per evitare deadlock. Una possibile implementazione è la seguente:

```
public void clone(RainfallDB db) {  
    List<Double> temp;  
    synchronized(db) {  
        temp = new ArrayList<Double>(db.data);  
    }
```

```
synchronized( this ) {  
    data . clear ();  
    data . addAll( temp );  
}  
}
```

Esercizio 3

Si consideri un'applicazione per un negozio (reale o virtuale), da realizzare in Java. Una parte dell'applicazione gestisce il calcolo del prezzo finale di vendita del "carrello" dell'acquirente. Il pagamento viene poi gestito da un'altra parte dell'applicazione. A tale scopo, il progetto del sistema prevede una classe `Sale` con due metodi di calcolo (prima dello sconto e dopo lo sconto), oltre ad ulteriori dati e operazioni che qui ignoriamo.

```
public class Sale {  
    ...  
    public Money totaleNonScontato() { ... }  
    public Money totaleScontato() { ... }  
    ...  
}
```

Il calcolo del totale deve essere effettuato in base al costo dei singoli prodotti moltiplicato per loro quantità, considerando però anche la politica di sconto. Il negozio infatti prevede una serie di iniziative promozionali, non sempre uguali, per cui ad esempio ci potrebbe essere uno sconto, di percentuale variabile, il martedì oppure uno sconto del 5% la sera dopo le 23, oppure un'offerta speciale del 15% il mercoledì per gli anziani, ecc. ecc. Tali politiche non sono interamente previste o prevedibili nella specifica del sistema. Occorre quindi progettare un sistema software che sia in grado di gestire facilmente politiche diverse e che renda possibile introdurre facilmente nuove politiche di sconto, o variazioni delle politiche esistenti.

Si utilizzi un design pattern opportuno per la progettazione del sistema. Si trateggino in Java le classi necessarie e le loro relazioni, utilizzando opportuni frammenti di codice o diagrammi UML per definire, illustrare ed esemplificare il sistema così progettato e il suo funzionamento.

Soluzione

Una politica di sconto è naturalmente un algoritmo. Occorre quindi progettare un sistema che permetta di scegliere a runtime fra diversi algoritmi e che consenta l'aggiunta di nuovi algoritmi. Si tratta quindi senza dubbio di applicare il pattern Strategy. A tale scopo introduciamo un'interfaccia:

```
public interface IPoliticaSconto {  
    public Money getTotal(Sale)  
}
```

Ogni politica di sconto è una classe che implementa l'interfaccia. Esempio:

```
public class ScontoDelMartedì implements IPoliticaSconto {  
    private percentuale;  
  
    public ScontoDelMartedì(int sconto) {  
        float percentuale = sconto / 100.0;  
    }  
  
    public Money getTotal(Sale s) {  
        import java.time.LocalDate;  
        LocalDate localDate = LocalDate.now();  
        if (localDate.getDayOfWeek()==Tuesday) {  
            return new Money(s.totaleNonScontato().value() * (1-percentuale));  
        } else {  
            return s.totaleNonScontato();  
        }  
    }  
}  
  
public class Sale {  
    private IpoliticaSconto politica;  
    ...  
    public void inizializzaPoliticaSconto(IPoliticaSconto politica) {  
        this.politica = politica;  
    }  
  
    public Money totaleScontato() {  
        return politica.getTotal(this);  
    }  
    ...  
}
```

Esercizio 4

Si considerino i seguenti metodi Java e se ne completino le parti omesse, *utilizzando esclusivamente i concetti e i costrutti della programmazione funzionale*.

Domanda a)

Il metodo seguente prende come argomento una lista di persone (Person) e restituisce una lista delle persone con più di 60 anni.

Si ipotizzi che la classe Person abbia il metodo getAge () che restituisce l'età della persona.

```
public static List<Person> older(List<Person> people) {  
    List<Person> elder = ... ;  
    return elder;  
}
```

Soluzione

```
public static List<Person> older(List<Person> people) {  
    List<Person> elder =  
    people.stream()  
        .filter(person -> person.getAge() > 60)  
        .collect(Collectors.toList());  
    return elder;  
}
```

Domanda b)

Il metodo seguente prende come argomento una lista di nomi e stampa il nome più lungo, se la lista non è vuota, altrimenti non stampa nulla.

```
public static void longest(List<String> names) {  
    final Optional<String> theLongest = ... ;  
    theLongest.ifPresent(.....);  
}
```

Soluzione

```
public static void longest(List<String> names) {  
    final Optional<String> theLongest = friends.stream()  
        .reduce((name1, name2) ->  
            name1.length() >= name2.length() ? name1 : name2);  
    theLongest.ifPresent(name ->  
        System.out.println(String.format("The longest name: %s", name)));  
}
```

Ingegneria del Software – a.a. 2009/10

Appello del 01 Luglio 2010

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h15.
6. Gli studenti che hanno una prova valida devono svolgere il solo Esercizio 3, con consegna in 1h15m. In caso di consegna oltre il termine, la prova precedente perde di validità e si valuterà pertanto l'intero tema.
7. Punteggio totale a disposizione: 100 punti nominali. La sufficienza si raggiunge indicativamente con 60 punti per il tema completo, 30 punti per chi deve svolgere solo l'esercizio 3.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
.....

Esercizio 1 (punti 35)

Si consideri il metodo statico **verificaAnagrammi**. Il metodo riceve come parametro una matrice rettangolare MxN (con N≥2, M≥1) di caratteri. Una matrice, come di consueto in Java, è organizzata come array di array: m[i][j] rappresenta l'elemento sulla riga i, colonna j.

Si noti che ogni riga e colonna della matrice può essere intesa come una stringa (partendo dal carattere in posizione 0, poi in posizione 1, ecc.). Il metodo restituisce true se tutte le seguenti condizioni sono verificate:

- Ogni colonna da 1 a N-1 (intesa come rappresentazione di una stringa a partire dal carattere in posizione 0) è un'anagramma della colonna 0.
- Le righe sono in ordine lessicografico crescente a partire dalla riga 0.

a- Si scrivano la pre- e post-condizione del metodo evidenziando tutti gli elementi significativi

Sol:

```
/*@requires m!= null && m.length>=2 &&m[0] != null && m[0].length>=1 &&
@          (\forall int i; 0<=i&&i< m.length; m[i]!=null && m[i].length == m[0].length);
@ ensures \result == (\forall int i; 0< i && i< m.length;
@                      anagramma(m[0],m[i])) && minoreUguale(m[i-1], m[i]);
@*/
```

dove anagramma e minoreUguale sono funzioni ausiliarie così definite:

```
/*@requires a!=null && b!=null && a.length == b.length;
@ensures \result ==
@      (\forall int i; 0<=i && i<a.length;
@           (\exists int j; 0<=j && j<=a.length-1; a[i] == b[j]) ==
@           (\exists int h; 0<=h && h<a.length; a[i]==a[h]));
@*/
public static boolean anagramma(char[] a, char[] b)
```

```
/*@requires a!=null && b!=null && a.length == b.length;
@ensures \result ==
@      (\exists j; 0<=j && j<=a.length;
@           (\forall int i; 0<=i && i<j; a[i].compareTo(b[i]) =0) &&
@           (j<a.length ==> a[j].compareTo(b[j])<0);
public static boolean minoreUguale(char[] a, char[] b)
```

static boolean verificaAnagrammi(char[][] m)

b- Si identifichino le eccezioni necessarie per rendere totale il metodo e si scrivano le opportune modifiche alla specifica di cui al punto (a).

SOL:

Detta PRE la precondizione di cui al punto a, e POST la postcondizione, si ottiene

```
/*@ ensures \old(PRE) && POST;
@signals (MatricellegaleException e) !\old(PRE);
```

Esercizio 2 (punti 15)

Mostrare l'output prodotto dal seguente programma:

```
class Alfa {  
    int i;  
    int j = 3;  
}  
  
abstract class Beta {  
    protected int i;  
    Beta(int i) { this.i = i; }  
    abstract int metodo(Alfa p);  
}  
  
class Gamma extends Beta {  
    private int j = 1;  
    Gamma(int i) { super(i); }  
    int metodo(Alfa p) {  
        p.i++;  
        int v = i + j;  
        return v;  
    }  
}  
  
public class Delta extends Gamma {  
    private int k = 3;  
    Delta(int i) { super(i); }  
    public int metodo(Alfa p) {  
        int w = super.metodo(p);  
        p.j--;  
        return w + k;  
    }  
    public static void main(String[] args) {  
        Delta q = new Delta(6);  
        Alfa p = new Alfa();  
        System.out.println(q.metodo(p));  
        System.out.println(p.i + ", " + p.j);  
    }  
}
```

RISPOSTA:

SOL:

10

1, 2

Esercizio 3 (punti 50)

Si consideri dapprima il seguente interfaccia Comparable, dalle librerie Java standard:

```
public interface Comparable<T> {  
    // @ensures (* if this minore di t then \result <0, else if this uguale a t then  
    // @ \result == 0, else \result ==1 *)  
    int compareTo(T t)  
}
```

Il metodo *compareTo* definisce un ordinamento totale sugli elementi del tipo generico T. Si consideri la seguente specifica di una classe generica PriorityQueue, sempre tratta dalle librerie standard Java.

```
public class PriorityQueue<E> {  
    /* OVERVIEW: Collezione mutabile di oggetti, di tipo generico E in cui gli elementi sono disposti in ordine  
    dal più piccolo (il primo elemento, ossia la testa) al più grande. Per l'ordinamento si usa il metodo  
    compareTo di E. Il tipo E deve implementare l'interfaccia Comparable. */  
  
    // @ensures (*inizializza this alla coda vuota*).  
    public PriorityQueue ()  
    //osservatori puri:  
  
    // @ensures (* \result è la cardinalità di this *)  
    public /*@ pure */ int size()  
  
    // @ensures (* \result è true se x compare in this *)  
    public /*@ pure */ boolean contains(Object x)  
  
    // @ensures (* \result è l'elemento in posizione i in this *)  
    public /*@ pure */ E get(int i)  
  
    // @ensures (size()>0 ? (forall E y; contains(y); \result.compareTo(y)<=0)  
    // @ : \result == null);  
    public /*@ pure */ E peek()  
  
    //mutators:  
    // @ensures x!= null && (* x è aggiunto a this *);  
    // @signals (NullPointerException e) x==null;  
    public void add(E x) throws NullPointerException  
  
    // @ ensures (\old(size())>0 ? \result == \old(peek()) &&(* \result è eliminato da this *)  
    // @ : \result == null && size()==0;  
    public E poll()  
}
```

Si osservi che 1) *add(x)* inserisce l'elemento x nella coda, purché x!=null; 2) *peek()* restituisce il primo elemento della coda, ossia il più piccolo secondo l'ordinamento definito da E per il metodo *compareTo* (se la coda è vuota restituisce null); 2) *poll()*, oltre a restituire il primo elemento nello stesso modo di *peek*, lo rimuove dalla coda.

Domanda A) Scrivere un invariante pubblico per la classe, che catturi le caratteristiche essenziali di PriorityQueue così come descritte nella Overview.

SOL:

```
public invariant (\forall int i; 0<=i && i<this.size()-1; this.get(i).compareTo(this.get(i+1))<=0;
```

È data un'implementazione di PriorityQueue realizzata nel modo seguente. Un attributo *coda* di tipo Vector memorizza tutti gli elementi della PriorityQueue, dal più grande (in posizione 0) al più piccolo (in ultima posizione). Un attributo *testa* memorizza la testa della coda, ossia l'elemento minimo.

Il rep quindi è:

```
private Vector<E> coda;  
private E testa;
```

Si ipotizza che man mano che gli elementi vengono inseriti o tolti dalla coda, il vector *coda* mantenga l'ordine dal più grande al più piccolo.

Sono date le seguenti implementazioni del costruttore:

```
public PriorityQueue() {  
    coda = new Vector<E>;  
}
```

e di peek() e di poll():

```
public E peek() {  
    if (coda.isEmpty()) throw new EmptyException();  
    return testa;  
}  
  
public E poll() {  
    if (coda.isEmpty()) throw new EmptyException();  
    if (coda.size()>=2)  
        testa = coda.get(lastElement()-1);  
    return coda.removeElementAt(coda.lastElement());  
}
```

Domanda B) : Scrivere l'invariante di rappresentazione di PriorityQueue.

SOL

```
private invariant coda!=null && (coda.size()>0 ==> coda.lastElement()==testa) &&  
(\forall int i; 0<i&& i<coda.size();coda.get(i-1).compareTo(coda.get(i))>=0;
```

Domanda C): Scrivere la funzione di astrazione di PriorityQueue, utilizzando un private invariant.

SOL

```
private invariant (\forall int i; 0 < i && i < coda.size(); coda.get(i) == this.get(size() - i - 1);
```

Domanda D): Implementare la funzione di astrazione con `toString()`.

```
public String toString() {
    String s = "";
    (for int i=0; i<coda.size(); s = s+coda.get(coda.size()-i-1);
    return s;
}
```

Si consideri ora una classe DistinctElements-PriorityQueue, che ha gli stessi metodi di PriorityQueue, ma che ridefinisce add in modo non ammettere l'inserimento di elementi duplicati: add non aggiunge l'elemento x qualora x.compareTo(y) restituisca 0 per almeno un elemento y in this.

Domanda E): Discutere se è possibile definire DistinctElements-PriorityQueue come estensione di PriorityQueue, rispettando il principio di sostituzione di Liskov.

SOL: No, perché per definire la nuova classe occorre ridefinire la specifica di add. Ci sono due soli modi: o si rafforza la precondizione, richiedendo che add(x) non possa essere chiamato con un elemento x già esistente in this; oppure, si indebolisce la postcondizione, non inserendo x in this qualora sia un duplciato. In entrambi i casi, si viola la regola dei metodi.

Appello 03 marzo 2015



Politecnico di Milano
Anno accademico 2013-2014

Ingegneria del Software

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h45'.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri la classe `Poligonale` classe mutabile che rappresenta una sequenza (senza duplicazioni) di punti nello spazio tridimensionale. Ogni punto è un elemento di una classe `Punto`, accennata in seguito. La classe `Poligonale` è così specificata:

```
public class Poligonale {  
  
    public Poligonale(); //costruisce una poligonale vuota.  
  
    //@ensures /result e' la cardinalita' dell'insieme  
    public int size() {...}  
  
    //@ensures /result e' il punto i-esimo della sequenza, con 0<=i<size().  
    //@signals IndexOutOfBoundsException e) (i<0 || i>=size());  
    public Punto get(int i) throws IndexOutOfBoundsException {...}  
  
    //@ensures (*inserisce un nuovo punto al termine della sequenza)  
    //@signals DuplicateException e) (*eccezione se p esiste in this*)  
    public void inserisci(Punto p) throws IndexOutOfBoundsException {...}  
  
    //@ensures /result e' un iteratore a tutti i punti, nell'ordine della sequenza,  
    //@ senza ripetizioni.  
    public Iterator<Punto> punti() {...}  
  
    //@ensures /result e' la lunghezza della linea spezzata che unisce,  
    //@in ordine, i punti della Poligonale.  
    public double lunghezza() {...}  
  
    ....  
}
```

La classe immutabile `Punto` ha un costruttore `Punto(double x, double y, double z)` e gli osservatori: `double getX()`, `double getY()`, `double getZ()`, di significato ovvio. La classe fornisce anche un metodo `distanza(Punto p)`, che calcola la distanza euclidea fra `p` e `this`. Infine, la classe `Punto` definisce `equals` in modo che due punti siano `equals` se, e solo se, hanno identiche coordinate.

- Si scriva la specifica JML dei metodi `inserisci(Punto)`, `lunghezza()`.

- b) Si definisca un opportuno invariante pubblico della classe Poligonale, in base alla descrizione che è stata data della classe stessa, e si argomenti se esso è verificato oppure no dalla specifica data.

- c) Si consideri la seguente implementazione parziale di Poligonale:

```
public class Poligonale {  
    private class Nodo{  
        Punto p;  
        Nodo n;  
        Nodo(Punto p, Nodo n) {  
            this.p = p;  
            this.n=n;  
        }  
    }  
    private Nodo testa;  
    private int len; //rep  
    public Poligonale() {  
        testa = null;  
        len=1;  
    }  
    public void inserisci(Punto p) throws DuplicateException {  
        ----  
        ----  
        ----  
        testa= new Nodo(p,testa);  
        len++;  
    }  
    public int size() {  
        ----  
        ----  
    }  
}
```

- c1) Completare il codice di inserisci dove sono presenti i trattini ----.

- c2) Scrivere l'invariante rappresentazione della classe.
- c3) Definire la funzione di astrazione della classe `Poligonale`, tramite un opportuno `private invariant` o un'implementazione del metodo `toString()`.
- c4) Si implementi il metodo iteratore `punti()`, definendo le classi opportune.

Esercizio 2

Si consideri una variante di `Poligonale`, detta `PoligonaleBis`, che ha anche un metodo che consente di inserire un punto come successivo a un punto dato.

```
//@ensures (*inserisce un nuovo punto daInserire dopo p*)
//@signals DuplicateException e) (*eccezione se p esiste in this*);
public void inserisciDopo(Punto p,Punto daInserire) throws IndexOutOfBoundsException
```

1. Specificare il metodo dato in JML.

2. E' possibile definire questa classe `PoligonaleBis` come derivata da `Poligonale`, rispettando il principio di sostituzione? Motivare la risposta.

3. Si consideri un'altra variante di `Poligonale`, detta `PoligonaleTer`, che ha la stessa definizione di `Poligonale`, ma il cui metodo `inserisci` e' cosi' descritto:

```
//@ensures (*inserisce un nuovo punto in una posizione qualunque*);
//@signals DuplicateException e) (*eccezione se p esiste in this*);
public void inserisci(Punto p) throws IndexOutOfBoundsException {...}
```

E' possibile definire questa classe `PoligonaleTer` come derivata da `Poligonale`, rispettando il principio di sostituzione? E viceversa, `Poligonale` come derivata da `PoligonaleTer`? Motivare la risposta.

Esercizio 3

Si considerino le seguenti classi Java:

```
public class Frutto {  
    public String toString(){  
        return "Frutto";  
    }  
    public void macedonia(Frutto f){  
        System.out.println("Frutto " + f);  
    }  
}  
  
public class Mela extends Frutto {  
    public String toString(){  
        return "Mela";  
    }  
    public void macedonia(Frutto f){  
        System.out.println("Frutto (Mela) " + f);  
    }  
    public void macedonia(Mela f){  
        System.out.println("Mela " + f);  
    }  
}  
  
public class Arancia extends Frutto {  
    public String toString(){  
        return "Arancia";  
    }  
    public void macedonia(Frutto f){  
        System.out.println("Frutto (Arancia) " + f);  
    }  
    public void macedonia(Arancia f){  
        System.out.println("Arancia " + f);  
    }  
}
```

e si spieghi cosa stamperebbe il seguente metodo `main`, motivando brevemente le risposte:

```
public class Test {  
    public static void main(String args[]){  
        Frutto f1, f2;  
        Mela m;  
        Arancia a;  
        f1 = new Frutto();  
        f2 = new Arancia();  
        m = new Mela();  
        a = new Arancia();  
        f1.macedonia(f1);  
        f1.macedonia(f2);  
        f2 = m;  
        f1 = m;  
        f1.macedonia(f2);  
        a.macedonia(m);  
        f1.macedonia(a);  
    }  
}
```

Esercizio 4

Si consideri il seguente frammento di programma:

```
0. numProdotti=0;
1. while (numProdotti <= PROD_MAX) {
2.     ----
3.     if (costoProdotto <= restaDaSpendere) {
4.         ----
5.         restaDaSpendere = restaDaSpendere - costoProdotto;
6.         if (restaDaSpendere > 0) {
7.             numProdotti++;
8.             System.out.println("Soldi finiti!");
9.             break;
10.        }
11.    }
12.};
```

1. Si definiscano casi di test che coprano tutte le istruzioni (si considerino le parti tralasciate ---- come se fossero una singola macro-istruzione che non modifica alcuna delle variabili che appaiono nel frammento);
2. Se il criterio di copertura fosse quello di copertura dei branch si otterebbe lo stesso risultato? Perché sì? Perché no?
3. Si calcoli la condizione logica che impone che il ciclo venga eseguito due volte, ma in modo che alla seconda iterazione si esca con l'istruzione break.



Politecnico di Milano

Anno accademico 2013-2014

Ingegneria del Software – Appello del 9 settembre 2014

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si consideri il seguente metodo statico:

```
private static ArrayList<String> tagliaLista(ArrayList<String> l, int n)
```

Il metodo prende in ingresso una lista `l` di Stringhe e un intero `n` e restituisce un'altra lista. La nuova lista deve contenere tutte le stringhe contenute in `l` di lunghezza non inferiore a `n` ordinandole dalla più lunga alla più corta. Se nessuna stringa in `l` dovesse avere lunghezza almeno pari a `n`, il metodo deve comunque restituire un `ArrayList` vuoto.

Si definisca:

1. La pre-condizione sui parametri. Si noti che il metodo non lancia eccezioni.

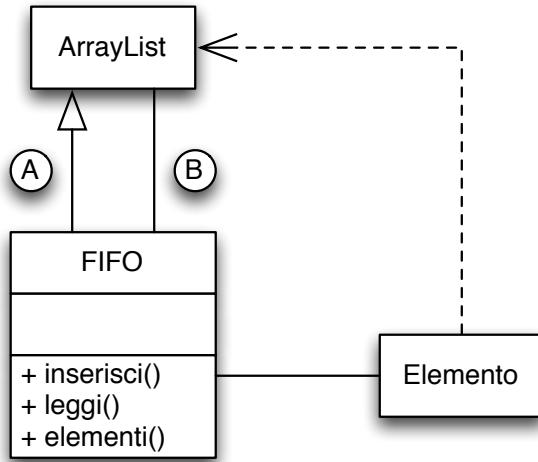
2. La post-condizione relativa alle stringhe contenute nel risultato

3. La post-condizione relativa all'ordinamento delle stringhe nel risultato

4. Ogni altra condizione da aggiungere alla post-condizione e non definita sopra

Esercizio 2

Si consideri il seguente diagramma (schematico e parziale) delle classi UML, che mostra una classe FIFO che realizza una struttura dati con politica di accesso FIFO (ossia, una coda):



Il diagramma illustra due possibili alternative, denotate con (A): ereditarietà e (B): associazione per la relazione fra la classe FIFO e la classe ArrayList di Java.

1. Quale delle due alternative (A o B) usereste per implementare la classe FIFO attraverso un ArrayList Java? Giustificare la risposta.
2. Scrivere un possibile rep della classe FIFO e spiegare come implementare il metodo `void inserisci(Elemento e)`.

3. Implementare il metodo iteratore `public Iterator<Elemento> elementi()`, assieme a tutte le classi necessarie. Il metodo consente di ottenere tutti gli elementi di una FIFO, secondo l'ordine di accesso FIFO.

Esercizio 3

Si consideri un'applicazione software che gestisce gli ordini e i conti di una pizzeria. La pizzeria prevede tre tipi base di pizza, denominati Margherita, Pomodoro, Focaccia. Ogni pizza è associata a una descrizione (di tipo stringa) e ha un proprio prezzo per la versione base. Il cliente può aggiungere a piacere alla pizza base diversi tipi di supplemento, che supponiamo essere Prosciutto, Funghi, Carciofi, Bufala. Ciascun supplemento ha un prezzo e una descrizione. Il prezzo finale di una pizza è calcolato sommando al prezzo base il prezzo di ciascun condimento. Ad esempio, se il prezzo di una Margherita fosse quattro euro, il costo del Prosciutto due euro e quello dei Funghi un euro, il costo di una Pizza Prosciutto e Funghi sarebbe di sette euro.

Si richiede di descrivere una struttura di classi adatta a risolvere il problema, tramite ad esempio un diagramma UML e, se necessario, qualche frammento di codice Java. In particolare, la soluzione prospettata deve consentire di:

- aggiungere combinazioni a piacere dei supplementi;
- calcolare correttamente il prezzo finale di una pizza;
- essere facilmente estendibile nel caso in cui si introducano nuovi supplementi o nuovi tipi di pizza base.

Esercizio 4

Si consideri il metodo Java seguente:

```
public static int foo(int n) {  
    int t = n%3;  
  
    while (n != 0)  
        if (t == 3) return 3;  
        else t = 2;  
    return t;  
}
```

e si definisca

1. Il diagramma del flusso di controllo
2. Un insieme minimo di test che copra tutte le istruzioni. Nel caso non fosse possibile, definire la copertura (in percentuale) dell'insieme di test definiti.
3. Quali (probabili) errori sono evidenziati dall'insieme di test definito al punto precedente.

Ingegneria del Software – a.a. 2009/10

Appello del 23 Luglio 2010

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 100 punti nominali. La sufficienza si raggiunge indicativamente con 60 punti.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale
.....

Esercizio 1 (punti 25)

E' data l'astrazione procedurale **calcolo**. Il metodo riceve come parametro un array x di almeno 2 interi e restituisce un array di interi, i cui elementi sono così costituiti:
il primo elemento è la somma di tutti gli elementi di x, il secondo è la somma dei primi $x.length - 1$ elementi di x, il terzo dei primi $x.length - 2$ ecc.

a- Si scrivano la pre- e post-condizione del metodo evidenziando tutti gli elementi significativi

SOL:

```
//@ requires (x != null) && (x.length >= 2)
```

```
/*@ ensures \result != null && \result.length == x.length &&
```

```
@     (\forall int i; i >= 0 && i < x.length; (\result[i]  
@           == (\sum int j; j >= 0 && j < (x.length - i); x[j])))  
@*/
```

```
static int[] calcolo(int[] x)
```

b- Si identifichino le eccezioni necessarie per rendere totale il metodo e si scrivano le opportune modifiche alla specifica di cui al punto (a).

SOL:

```
//@ requires true
```

```
/*@ ensures ((x != null) && (x.length >= 2)) =>
```

```
@     \result != null && \result.length == x.length &&  
@           (\forall int i; i >= 0 && i < x.length; \result[i]  
@               == (\sum int j; j >= 0 && j < (x.length - i); x[j])))  
@*/
```

```
//@ signals (NullPointerException e) (x == null)
```

```
//@ signals (ArrayTooSmallException e) ((x != null) && (x.length < 2))
```

Esercizio 2 (punti 50)

Si consideri la seguente specifica di una classe HeapInt.

```
public class HeapInt{
/* OVERVIEW: Collezione mutabile di oggetti, di tipo intero. */

//@ensures (*inizializza this alla HeapInt vuota*).
public HeapInt()
//osservatori puri:

//@ensures (* \result è la cardinalità di x in this, ossia il suo numero di comparsa *)
public /*@ pure */ int occurrences(int x)

//@ensures (* \result è la cardinalità di this, contando ogni elemento con la propria cardinalità *)
public /*@ pure */ int size()

//@requires size()>0;
//@ensures (\forall int y; occurrences(y)>0; \result>=y)
public /*@ pure */ int max()

//mutators:
//@ensures (* aggiunge una comparsa di x a this *);
public void insert(int x)

//@requires size()>0;
//@ ensures (* elimina e restituisce una comparsa dell'elemento massimo da this *)
public int extract()
}
```

Domanda A) Specificare in JML le postcondizioni dei metodi insert e extract

SOL:

```
//@ensures occurrences(x) == \old(occurrences(x)+1) && size() == \old(size()+1) &&
//@\(\forall int y; x!=y; occurrences(y) == \old(occurrences(y)));
public void insert(int x)

//@ensures \ result == \old(max()) && occurrences(\result) == \old(occurrences(\result)-1) &&
//@\ size() == \old(size()-1) && (\forall int y; \result!=y; occurrences(y) == \old(occurrences(y)));
public int extract()
```

Si consideri ora una classe HeapInt2, identica a HeapInt ma che modifica il metodo extract in modo che questo elimini tutte le comparse dell'elemento massimale.

Domanda B): Specificare in JML questa versione del metodo extract(),

Sol:

```
//@ensures \ result == \old(max()) && occurrences(\result) == 0 &&
//@ size() == \old(size()-occurrences(\result)) &&
//@(\forall int y; \result!=y; occurrences(y) == \old(occurrences(y)));
public int extract()
```

Domanda C) Dimostrare analiticamente, in base alle specifiche date, che HeapInt2 non può essere erede di HeapInt senza violare il principio di sostituzione.

SOL:

La dimostrazione è per assurdo. Detta post1 la postcondizione di HeapInt.extract() e detta post2 la postcondizione di HeapInt2.extract(), condizione necessaria per la verifica del principio è che post2 ==> post1. Ma se vale post1, allora vale $\text{occurrences}(\text{\result})=0$, mentre se vale post2 allora vale $\text{occurrences}(\text{\result})=\text{\old}(\text{occurrences}(\text{\result})-1)$.

Per uno heap in cui l'elemento max() ha cardinalità 2, da post1 deriva che $\text{occurrences}(\text{\result})=0$, mentre da post2 deriva che $\text{occurrences}(\text{\result})=1$. Allora se post2 implicasse post1 si dedurrebbe una contraddizione.

Si consideri nuovamente la classe *HeapInt* della Domanda A. È data un'implementazione realizzata nel modo seguente. Un attributo *heap* di tipo *ArrayList* memorizza gli elementi di *this* come *Integer*, senza ripetizioni, in ordine decrescente. Un attributo *comparse* di tipo *ArrayList* memorizza in *comparse.get(i)* il numero di comparse dell'elemento *heap.get(i)*. Il numero di comparse non può mai essere nullo o negativo.

Il rep quindi è:

```
private ArrayList<Integer> heap;
private ArrayList<Integer> comparse;
```

Data la seguente implementazione del costruttore:

```
public HeapInt() {
    heap = new ArrayList<Integer>;
    comparse = new ArrayList<Integer>;
}
```

Domanda D) Scrivere l'invariante di rappresentazione di *HeapInt*.

SOL:

```
//@private invariant heap !=null && comparse != null && heap.size() == comparse.size() &&
//@(\forall int i; 0< i && i< heap.size(); heap.get(i-1) > heap.get(i)) &&
//@(\forall int i; 0< i && i< comparse.size(); comparse.get(i) >0);
```

NB: la condizione che gli elementi di *heap* siano tutti distinti deriva dal fatto che sono in ordine strettamente decrescente.

Domanda E) Completare la seguente implementazione del metodo *extract()*

```
public int extract() {
    int massimo = this.heap.get(0); // memorizza in massimo il valore da restituire
    .....comparse.set(0,comparse.get(0)-1);
    .....if (comparse.get(0)==0 {
        .....comparse.remove(0);
        .....heap.remove(0);
    .....}
    return massimo;
}
```

Esercizio 3 (punti 25)

Implementare il seguente metodo iteratore *elementi()* della classe *HeapInt* dell'Es. 2. Si suggerisce di descrivere la struttura delle eventuali classi aggiuntive rispetto a *HeapInt*.

//@ensures (* restituisce un iteratore agli elementi di this, in ordine decrescente dal più grande al più piccolo. Gli elementi che compaiono ripetutamente sono restituiti un numero di volte pari alla loro cardinalità.*);

public Iterator<Integer> elementi()

SOL:

```
public class HeapInt {  
    private ArrayList<Integer> heap;  
    private ArrayList<Integer> comparse;  
    public Iterator<Integer> elementi() {return new HeapIterator ();}  
    private class HeapIterator implements Iterator<Integer> {  
        private Integer current;  
        private int count;  
        private Integer next;  
        public HeapIterator () {  
            if (heap.size() == 0) {  
                this.count = comparse.get(0);  
                this.current = heap.get(0);  
            }  
        }  
        public boolean hasNext() {  
            if (current != null)  
                return true;  
        }  
        public Integer next() throws NoSuchElementException {  
            if (!hasNext()) {  
                throw new NoSuchElementException ();  
            }  
            next = current;  
            count --;  
            if (!count) {  
                int nuovoIndice = heap.indexOf(current) + 1;  
                if (nuovoIndice == heap.size())  
                    current = null;  
                else {  
                    current = heap.get(nuovoIndice);  
                    count = comparse.get(nuovoIndice);  
                }  
            }  
            return next;  
        }  
        public void remove() throws IllegalStateException {  
            int index;  
            if (next == null) throw new IllegalStateException();  
            index = heap.indexOf(next);  
            if (comparse.get(index) == 1) {  
                comparse.remove(index);  
                heap.remove(index);  
            }  
            next = null;  
        }  
    }  
}
```

Ingegneria del Software – a.a. 2008/09

Appello del 11 settembre 2008

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi , Ghezzi , San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Totale

Esercizio 1 (punti 7)

Si consideri la seguente specifica dell'ADT Audio. Questo è un tipo immutabile che rappresenta un clip audio (p.es. brano musicale) codificato in digitale. Un tipico Audio è una quadrupla [codec, durata, traccia e bitRateMedio], dove “codec” è una stringa che descrive il codec utilizzato per la codifica del brano, “durata” è la durata in secondi del brano, “traccia” è la rappresentazione digitale (un array di byte) della musica del brano e “bitRateMedio” è il numero medio di Kbit al secondo della codifica, pari al numero di byte diviso la durata.

```
public /*@ pure */@class Audio {  
  
    //costruttore:  
    //@ensures(*crea un nuovo Audio sulla base dei valori passati come parametri*);  
    public Audio(String codec, int durata, byte[] traccia);  
  
    //metodi observer:  
    //@ensures(*restituisce il codec, autore e titolo*);  
    public String codec();  
  
    //@ensures(*restituisce la durata in secondi del brano*);  
    public int durata();  
  
    //@ensures(*restituisce la rappresentazione digitale dell'Audio *);  
    public byte[] traccia();  
  
    //restituisce il bitrate medio del brano:  
    //@ensures \result == traccia().length / durata();  
    public int bitRateMedio();  
}
```

a- Si consideri anche l'ADT Video. Questo ha gli stessi metodi di Audio, ma tiene conto anche della traccia Video e quindi aggiunge i metodi seguenti:

```
public class Video {  
  
    //@ensures(*restituisce la rappresentazione digitale dell'Audio *);  
    public byte[] tracciaVideo();  
  
    //@ensures(*crea un nuovo clip video sulla base dei valori passati come parametri*);  
    public Video(String info, int durata, byte[] tracciaAudio, byte[] tracciaVideo);  
  
    //e ridefinisce il seguente:  
    //@ ensures bitrate()*durata == traccia().length + tracciaVideo().length;  
    public int bitRateMedio();
```

È possibile definire l'ADT MP3 come estensione di Audio, soddisfacendo il principio di sostituzione di Liskov? Giustificare la risposta.

L'ADT non puo' soddisfare il principio di sostituzione, in quanto occorre ridefinire il metodo bitRateMedio con una postcondizione che *non* implica la postcondizione originale.

b- Si consideri ora l'ADT MP3, che ha gli stessi metodi di Audio con stessa specifica, salvo per la seguente definizione del metodo codec:

```
public class MP3 {  
    //@\result == "MPEG3";  
    public String codec() {}
```

MP3 ha inoltre il seguente costruttore:

```
//@ensures(*crea un nuovo MP3di sulla base dei valori passati come parametri*);  
public MP3(int durata, byte[] traccia);
```

È possibile definire l'ADT MP3 come estensione di Audio, soddisfacendo il principio di sostituzione di Liskov? Giustificare la risposta.

Si, in quanto la postcondizione di codec() è più forte di quella originale. Non sono inoltre violate proprietà di Audio, perché certo MPEG3 era un codec valido anche per Audio.

Esercizio 2 (punti 8)

a- Si consideri la specifica della seguente classe Supporto, che rappresenta una memoria per clip audio (es. CD, Scheda SD, lettore MP3 ecc.). Supporto è un tipo mutable. Un supporto contiene elementi di tipo Audio. Un Supporto tipico è identificato dalla coppia [CAPIENZA, brani], dove; “CAPIENZA” è una costante (per ogni supporto) che rappresenta la capacità in byte del supporto, “brani” è una sequenza <b1 b2 ... bn> di oggetti della classe Audio. Descrive la post-condizione dei metodi spazioDisponibile e inserisci.

```
public class Supporto{  
  
    public final int CAPIENZA;  
  
    //@ensures (* /result contiene tutti e soli i brani di this, nell'ordine di memorizzazione *);  
    public ArrayList<Audio> brani() {}  
  
    //@requires max_capienza>0;  
    //@ensures (* crea una nuova Memoria di capienza specificata, inizialmente vuota*)  
    public Supporto(int max_capienza){  
  
        //restituisce true se nella Memoria this c'è ancora abbastanza spazio per il brano b  
        //@ensures  
  
        CAPIENZA>= b.traccia().length +  
        (\sumof int i; 0<=i && i<brani().size(); brani().get(i).traccia().length))  
        public boolean spazioDisponibile(Audio b) {  
  
            //inserisce il brano b in this  
            //@requires spazioDisponibile(b);  
            //@ensures  
  
            brani().size() == \old(brani().size())+1 && b== brani().lastElement() &&  
            (\forallall int i; 0<=i && i<brani().size()-1; brani().get(i) == \old(brani().get(i)))  
            public void inserisci (Audio b) {}
```

}

- b- È dato il seguente rep per la classe Supporto.

```
private ArrayList<BranoMusicale> contenuto;  
private int durata; //durata complessiva dei brani memorizzati in contenuto.
```

e la seguente implementazione del costruttore:

```
public Supporto(int max_capienza){  
    CAPIENZA=max_capienza;  
    this.contenuto=new ArrayList<Audio>();  
    this.durata=0;  
}
```

Descrivere l'invariante privato e implementare i metodi seguenti:

```
//@ private invariant
```

contenuto!=null && contenuto.size() == brani().size() &&
durata == (\sumof int i; 0<=i && i<contenuto.size(); contenuto.get(i).getDurata()) &&
(\forallall i; 0<=i && i<brani().size(); brani().get(i)==contenuto.get(i);
(NB: si suppone una relazione uno a uno fra la posizione nel metodo *brani()* e nella variabile d'istanza
contenuto)

```
public ArrayList<Audio> brani() {
```

}

```
return contenuto.clone()
```

```
public boolean spazioDisponibile(Audio b) {
```

```
    int somma =0;  
    (for int i; i>0 && i< contenuto.size(); i++)  
        somma+=contenuto.getMusica().length;  
    return (b.traccia().length>=capienza -somma);.  
}
```

```
public void inserisci (Audio b) { contenuto.add(b);
```

```
    }  
}
```

c- Con riferimento al punto (b) dell'esercizio 1, si consideri la classe MP3Memory, che definisce una memoria contenente solo brani di tipo MP3.

```
public class MP3Memory extends Supporto{  
    // @also  
    // @public invariant  
  
    (forall int i; 0<=i && i<brani().size(); brani().get(i) instanceof MP3);  
  
    // @requires max_capienza>0;  
    // @ensures (* crea una nuova MP3Memory, di capienza specificata, inizialmente vuota*)  
    public MP3Memory(int max_capienza) {}  
}
```

Si completi il public invariant e si indichi, motivando accuratamente ma sinteticamente la risposta, se questo ADT soddisfa o meno il principio di sostituzione di Liskov rispetto a Supporto.

L'ADT (un contenitore) non soddisfa il principio di sostituzione (anche se la versione data soddisfa la regola dei metodi), in quanto il suo invariante deve impedire di fatto l'inserimento di brani che non siano di tipo MP3. Per soddisfare l'overview (e quindi l'invariante) occorrerebbe allora ridefinire la specifica del metodo inserisci, per impedire l'inserimento di Audio che non siano MP3. Questo tuttavia violerebbe la regola della postcondizione.

Esercizio 3 (punti 6)

Supponendo che esistano le classi Mia, Tua e Sua, dove Mia è super classe sia di Tua che di Sua, quali delle seguenti affermazioni sono corrette?

1. Tua e Sua ereditano i costruttori di Mia

SCORRETTA

2. Se Mia definisce un attributo x privato, qualunque metodo di Tua e Sua potrebbe leggerne e scriverne il valore direttamente

SCORRETTA

3. Scrivere: Mia m;

m = new Mia();

darebbe un errore in compilazione

SCORRETTA

4. Scrivere: Mia m;

m = new Tua();

darebbe un errore in compilazione

SCORRETTA

5. Scrivere: Tua m;

m = new Mia();

darebbe un errore in compilazione

CORRETTA

6. Scrivere: Tua m;

m = new Sua();

darebbe un errore in compilazione

CORRETTA

7. Se Mia, Tua e Sua fossero definite in tre package diversi, ci sarebbe almeno un errore in compilazione

SCORRETTA

8. Se Mia definisce un attributo x ‘friendly’, qualunque metodo di Tua e Sua (definite in package diversi rispetto a Mia) potrebbe leggerne e scriverne il valore direttamente

SCORRETTA

Esercizio 4 (punti 4)

Un distributore automatico di merendine è composto da un display, una tastiera, una gettoniera e un distributore vero e proprio. Questi elementi hardware sono controllati da software opportuno per consentire all'utente di scegliere un prodotto, pagare con il proprio dispositivo attivo (chiavetta), o in contanti, e recuperare il prodotto acquistato.

Chiaramente, ogni prodotto ha un prezzo, leggermente inferiore se il cliente paga con la propria chiavetta, e il distributore non eroga nulla se la cifra pagata non è sufficiente (il credito sul dispositivo non è sufficiente, oppure le monete inserite sono troppo poche). Se il totale delle monete inserite è superiore al prezzo richiesto, la macchina, attraverso la gettoniera, deve dare il resto. Il cliente può anche usare la gettoniera per caricare la propria chiavetta; questo avviene inserendo soldi (monete e banconote) senza selezionare un prodotto.

a- Si modelli il sistema attraverso un opportuno diagramma delle classi UML, specificando quali classi "rappresentano" elementi puramente software e quali definiscono l'interfaccia dei componenti hardware elencati in precedenza.

b- Si scriva la specifica JML del metodo definito per la selezione di un prodotto. Considerando che il codice scelto deve esistere (altrimenti il sistema deve prevedere un opportuno messaggio d'errore), il prodotto non deve essere esaurito e l'ammontare pagato sufficiente. Se il cliente paga con la chiavetta, il costo deve essere detratto dal credito del cliente, se paga in contanti, si deve prevedere l'eventuale resto.



Politecnico di Milano

Anno accademico 2012-2013

Ingegneria del Software – Appello del 19 luglio 2012

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione.
È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si scrivano in JML le pre- e post- condizioni del seguente metodo **Substrings**. Questo prende in ingresso due array di caratteri x e y e restituisce, in un array di interi, l'elenco, in ordine qualunque, di tutti e soli gli indici in cui y compare come sottostringa di x . Ad esempio, se $x = [abbbbbabbbabbbabb]$ e $y = [abb]$, il metodo restituisce l'array $[0\ 6\ 10\ 14]$.

Soluzione:

```
//@assignable \nothing;
//@requires x!=null && y!=null;
//@ensures \result != null &&
//  \result contiene solo indici da cui inizia
//  una comparsa di y in x:
//@  (\forall int i; 0 \leq i && i < \result.length;
//@    (\forall int j; 0\leq j && j < y.length;
//@      \result[i]+j < x.length && x[\result[i]+j]== y[j])) &&
//  \result contiene tutti gli indici in cui compare y:
//@  (\forall int i; 0 \leq i && i < x.length;
//@    (\forall int j; 0\leq j && j < y.length;
//@      i+j < x.length && x[i+j]== y[j])
//@      ==>
//@      (\exists k; 0 \leq k && k<\result.length; \result[k]==i));
public static int[] substrings(char[] x,char[] y)
```

Esercizio 2

Si consideri un distributore di caramelle. Il costo di ogni caramella è di una singola moneta da 50 cents. Il controllo del distributore è implementato in Java, con codice che include fra l'altro la seguente classe, qui specificata solo parzialmente:

```
public class Distributore {
    public final int MONETA = 50;
    public static final int ESAURITO=0;
    public static final int PRONTO=1;
    public static final int VENDUTO=2;
    public static final int FORNITO=3;

    //@requires caramelleIniziali>0;
    //@ensures caramelle() == caramelleIniziali;
    //@ensures monete() == 0;
    //@ensures stato()==PRONTO;
    public Distributore (int caramelleIniziali);

    public void inserisciMoneta();

    public int restituisciMoneta();

    public void giraManopola();

    public void preleva();

    //@ensures (* rilascia una caramella *);
    public void rilascia();

    //@requires nuoveCaramelle>0;
    //@ensures caramelle() == nuoveCaramelle && stato==PRONTO;
```

```

public void riempi(int nuoveCaramelle);

//@ensures (*\result e' il numero di caramelle rimaste*);
public /*@ pure @*/ caramelle();

//@ensures (*\result e' il numero di monete nel distributore*);
public /*@ pure @*/ int numMonete();

//@ensures (*\result e' lo stato (ESAURITO, PRONTO, ecc.)
//@           in cui si trova il distributore; *)
public /*@ pure @*/ int stato();

}

```

I metodi inserisciMoneta(), restituisciMoneta(), giraManopola() e preleva() corrispondono alle quattro possibili azioni dell’utente sul distributore, mentre gli altri metodi rappresentano funzionalità disponibili al codice Java del distributore. Il metodo rilascia() (che non corrisponde a un evento disponibile all’utilizzatore) si interfaccia con il dispositivo in modo da ordinare il rilascio fisico della caramella, mentre riempi() corrisponde al riempimento del deposito delle caramelle (e può essere chiamato solo dal gestore).

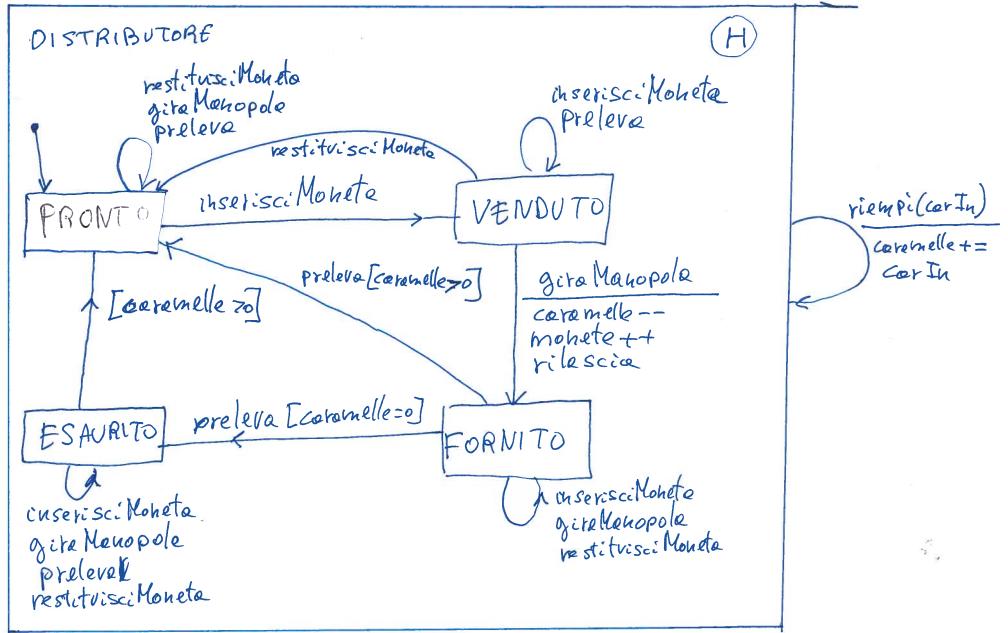
Il distributore può trovarsi nelle seguenti situazioni:

- esaurito, quando non vi è alcuna caramella;
- pronto, quando vi sono caramelle e il sistema è pronto a ricevere una moneta;
- venduto, quando l’acquirente ha inserito la moneta (ma non ha ancora girato la manopola);
- fornito, quando la caramella è stata rilasciata, in attesa di essere prelevata;

1. Descrivere questo sistema con un diagramma a stati UML, prevedendo opportune condizioni e azioni sulle transizioni.

Soluzione: Si noti che ciascuno dei metodi inserisciMoneta(), restituisciMoneta(), giraManopola() e preleva() rappresenta un’azione esterna dell’utente (un evento) e pertanto non ha precondizione: se il metodo viene chiamato in uno stato errato, semplicemente non ha alcun effetto. Ad es. una chiamata di restituisciMoneta() a partire da uno stato diverso da VENDUTO, viene semplicemente ignorata. Questo è rappresentato dagli autoanelli nei vari stati. Supponiamo che l’aggiornamento del numero di monete e del numero di caramelle avvenga solo al momento di girare la manopola. Naturalmente altre ipotesi sono possibili. Infine, usiamo lo stato con storia per rappresentare la possibilità che riempi() possa essere chiamato in qualunque stato, in modo da incrementare le caramelleIniziali senza modificare lo stato. Il passaggio da ESAURITO a PRONTO avviene solo in base alla condizione su

caramelle.



2. Completare la specifica della classe Distributore, definendo in JML la specifica dei metodi inserisciMoneta() e giraManopola(). La specifica deve soddisfare il vincolo che il numero di monete è pari alla somma delle caramelle erogate. Definire opportune abbreviazioni, se comodo.

Soluzione: Riportiamo qui, per completezza, la specifica dei metodi inserisciMoneta(), restituisciMoneta(), giraManopola() e preleva(). La specifica è ricavabile immediatamente dal diagramma a stati precedente.

Per brevità, definiamo come abbreviazione “inv(A)” l’espressione booleana:

```

A ==\old(A)

public class Distributore {
    ...
    //@ensures (\old(stato())==PRONTO) ==> stato()==VENDUTO;
    //@ensures \old(stato() !=PRONTO) ==> inv(stato());
    //@ensures inv(numMonete()) && inv(caramelle());
    public void inserisciMoneta();

    //@ensures (\old(stato())==VENDUTO) ==> stato()==PRONTO;
    //@ensures \old(stato() !=VENDUTO) ==> inv(stato());
    //@ensures inv(numMonete()) && inv(caramelle());
    public int restituisciMoneta();

    //@ensures (\old(stato())==VENDUTO) ==> stato()==FORNITO &&
    //@
    //@                                         numMonete() ==\old(numMonete())+1 &&
    //@                                         caramelle() ==\old(caramelle()-1);
    //@ensures (\old(stato() !=VENDUTO)==> inv(stato()) && inv(numMonete()) && inv(caramelle()
    public void giraManopola();

    //@ensures inv(numMonete()) && inv(caramelle());
    //@ensures (\old(stato())==FORNITO) && caramelle() >0 ==> stato()==PRONTO;
    //@ensures (\old(stato())==FORNITO) && caramelle() ==0 ==> stato()==ESAURITO;
}

```

```

//@ensures (\old(stato()) != FORNITO ==> inv(stato()));
public void preleva();

}

```

3. Come potreste rappresentare in JML il vincolo precedente sul numero di monete e di caramelle: “il numero di monete deve essere pari alla somma delle caramelle erogate”?

Soluzione: La proprietà è un tipico invarianto pubblico, ma per potere esser scritto come tale richiede di introdurre un nuovo metodo pubblico puro che restituisce il numero di caramelle all’inizio;

```
...public /*@ pure @*/ int caramelleInizio() ...
```

(Non si può introdurre una costante pubblica in quanto il metodo riempi() modifica il valore di caramelleInizio()). Occorre aggiungere alla postcondizione del costruttore e del metodo riempi() rispettivamente le condizioni:

```

//@ensures caramelleInizio() == caramelleIniziali;
//@ensures caramelleInizio() == nuoveCaramelle;

```

Ad ogni altro metodo modificatore occorre aggiungere la postcondizione:

```
//@ensures inv(caramelleInizio());
```

Infine, l’invariante è:

```
public invariant caramelleInizio()-caramelle() == numMonete();
```

4. Si consideri una possibile estensione del comportamento del Distributore, detta DistributoreConVincita, in cui si prevede un nuovo stato della macchina:

- vincita, quando la macchina (dopo che l’acquirente ha inserito la moneta) decide (tramite un generatore di numeri causali) di fornire due caramelle anzichè una.

È possibile definire questa estensione con una sottoclasse di Distributore? Giustificare la risposta.

Soluzione: La risposta dipende dalla specifica JML precedente, ma se questa soddisfa l’invariante previsto sopra, allora sarà senz’altro violata almeno la regola delle proprietà. Per la particolare specifica data sopra, tuttavia, si viola anche la regola dei metodi, in quanto il metodo giraManopola() prevede una postcondizione chiaramente incompatibile con il caso di vincita.

Esercizio 3

Si consideri una soluzione alternativa in Java per l'esempio dell'esercizio 2. La classe Distributore (nella versione senza vincita) ha ancora un attributo "stato" che memorizza lo stato in cui si trova il distributore. Questo stato, non è un intero o un enumerativo, ma è un oggetto appartenente a una delle seguenti classi: Esaurito, Pronto, Venduto, Fornito. Queste classi forniscono gli stessi metodi inserisciMoneta(), restituisciMoneta(), giraManopola(), e preleva() della classe Distributore, ma specializzati per lo stato corrispondente.

In particolare, ciascun metodo di ogni classe restituisce il nuovo stato in cui si porta il sistema dopo l'esecuzione dell'azione. Inoltre, ciascun oggetto delle classi Esaurito, Pronto, Venduto, Fornito deve contenere anche un riferimento "distributore" all'oggetto della classe Distributore di cui rappresenta uno stato.

Ad es. il metodo restituisciMoneta() della classe Pronto si rifiuta di fornire una moneta all'utilizzatore, restituendo al chiamante ancora uno stato di tipo Pronto, mentre il metodo inserisciMoneta() è in grado di accettare una moneta (restituendo in tal caso uno stato di tipo Venduto).

1. Descrivere con un diagramma delle classi UML la soluzione appena prospettata, identificando, classi, metodi e, se opportuno, interfacce.

Soluzione: NB: Quello proposto è un esempio di un pattern, noto in letteratura con il nome di State. Occorre definire una classe astratta StatoDistributore, di cui le classi Esaurito, Pronto, ecc. sono estensioni (in alternativa, va bene anche un'interfaccia, implementata da Esaurito, Pronto, ecc.).

```
abstract public class StatoDistributore {  
    protected Distributore distributore;  
    protected StatoDistributore(Distributore d) {distributore=d;};  
    abstract public StatoDistributore inserisciMoneta();  
    abstract public StatoDistributore restituisciMoneta();  
    abstract public StatoDistributore giraManopola();  
    abstract public StatoDistributore preleva();  
}
```

Occorre prevedere che la classe StatoDistributore abbia un'associazione 1-1 con il Distributore (che si trova esattamente in uno stato), che può essere rappresentata dicendo che ogni Distributore contiene una, e una sola, istanza di StatoDistributore.

2. Implementare (anche parzialmente) la rappresentazione (rep) e il costruttore di Distributore.

Soluzione: Un rep di Distributore:

```
private StatoDistributore stato;  
private int conta;  
private int monete;
```

Il costruttore:

```
public Distributore(int caramelleIniziali) {  
    monete = 0;  
    conta = caramelleIniziali;  
    if (conta <= 0) stato = new Esaurito();  
    else stato = new Pronto();  
}
```

Scrivere in JML un invarianto di rappresentazione per Distributore. **Soluzione:**

```
private invariant stato !=null && monete>=0 &&  
(conta <= 0 ==> stato instanceof Esaurito);
```

Implementare infine il metodo giraManopola() delle classi Pronto e Distributore.

Soluzione: Per completezza, riportiamo l'implementazione di giraManopola() anche per Fornito e Venduto.

La classe Pronto deve rifiutarsi di dare la caramella:

```

public StatoDistributore giraManopola(){
    System.out.println("Non hai inserito nessuna moneta!");
    return this; //lo stato non cambia;
}

```

La classe Venduto deve chiamare rilascia() del distributore:

```

public StatoDistributore giraManopola(){
    distributore.rilascia();
    return new Fornito();
}

```

La classe Fornito deve rifiutarsi di dare la caramella:

```

public StatoDistributore giraManopola(){
    System.out.println("Hai gi\'a preso la caramella!");
    return this; //lo stato non cambia;
}

```

La classe Distributore delega semplicemente allo stato:

```

public void giraManopola(){
    stato = stato.giraManopola();
}

```

3. Come potreste aggiungere il caso della Vincita, minimizzando le modifiche al codice definito in precedenza (classi Distributore, Esaurito, Pronto, Venduto e Fornito)? Valutare brevemente la vostra soluzione.

Soluzione: Definire un erede di Distributore non è di alcuna utilità, in quanto tale classe non considera in quale stato effettivo si trova il distributore. Conviene invece estendere StatoDistributore con una nuova classe Vincita. La classe Vincita deve chiamare due volte rilascia() del distributore:

```

public StatoDistributore giraManopola(){
    System.out.println("Hai vinto una caramella!");
    distributore.rilascia();
    distributore.rilascia();
    return new Fornito();
}

```

Essenzialmente, occorre anche aggiungere nel metodo giraManopola dello stato Venduto, un generatore di numeri casuali per decidere se passare allo stato Vincita o restare in Venduto

```

public StatoDistributore giraManopola(){
if ("random test") return (new Vincita()).giraManopola();
...qui come prima...
}

```

Questa soluzione è concettualmente “pulita” ma 1) vi costringe a modificare un metodo esistente; 2) vi porta a codice duplicato (tutti gli altri metodi di Vincita sono uguali a quelli di Venduto). Un’alternativa è introdurre, al posto della classe Vincita, una sottoclassificazione di Venduto, VendutoConPossibileVincita, che ridefinisce giraManopola() con la nuova versione randomizzata:

```

public StatoDistributore giraManopola(){
    if ("random test") {
        System.out.println("Hai vinto una caramella!");
        distributore.rilascia();
    }
    return super.giraManopola();
}

```

Occorrerà però modificare comunque il codice esistente, per introdurre (nel metodo inserisciMoneta() dello stato Pronto) la chiamata al costruttore di VendutoConPossibileVincita al posto di quello di Venduto. Questa soluzione evita le duplicazioni e riduce ulteriormente le modifiche del codice esistente, ma può essere concettualmente meno chiara.

Esercizio 4

Si consideri il metodo seguente e si generi un insieme “minimo” di casi di test che soddisfi il criterio di copertura delle diramazioni (*branch coverage*). Si trovi poi un insieme minimo per soddisfare il criterio di copertura delle condizioni.

```
00 public static void mystery(int[] a) {  
01     int i, j, n;  
02     for (i = 1; i < a.length; i++) {  
03         n = a[i];  
04         j = i;  
05         while (j > 0 && a[j - 1] > n) {  
06             a[j] = a[j - 1];  
07             j--;  
08         }  
09         a[j] = n;  
10     }  
11 }
```

Soluzione: Un qualunque array di due elementi in cui $a[0] > a[1]$ (corrispondente alla condizione $a[j - 1] > n$ nel while alla prima iterazione del for) permette di coprire tutte le diramazioni. Non basta a coprire tutte le condizioni, in quanto con due elementi il ciclo for viene eseguito una sola volta e si esce dal ciclo while dopo una sola iterazione.

Per coprire tutte le condizioni (e quindi anche tutte le diramazioni), basta che l'array sia di 3 elementi, non ordinato, in modo da potere uscire dal while una volta con $j \leq 0$ e una volta con $j > 0 \&\& a[j - 1] \leq n$. E' quindi sufficiente un singolo caso di test, come $a = [3 1 2]$. NB: l'algoritmo e' un semplice insertion sort.



Politecnico di Milano

Anno accademico 2010-2011

Ingegneria del Software – Appello del 4 luglio 2011

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione.
È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Il back-end di un sistema per la vendita on-line di biglietti per concerti, rappresentazioni teatrali ed eventi sportivi definisce una classe `Evento`:

```
public class Evento {  
    ...  
    public ArrayList<Posto> selezionaPosti(int num, int tipo)  
    public boolean acquistaPosti(ArrayList<Posto> p)  
    public int numFile()  
    public Posto[] fila(int n)  
    public int postiDisponibili()  
}
```

- `selezionaPosti` propone i migliori `num` posti disponibili per l'evento scelto. Per semplicità, si supponga che i posti siano divisi per tipo: 0 se il posto non è numerato e 1 se lo è. Per i posti di tipo 0 non c'è alcun controllo. Per i posti di tipo 1, il sistema deve invece sempre restituire la soluzione migliore possibile. I posti sono tutti ordinati per file e `p1` è migliore di `p2` se `p1.fila() < p2.fila()` (il metodo `fila()` è definito per la classe `Posto`, come si vedrà più avanti). Il metodo (`selezionaPosti`) non solleva eccezioni, ma è invocabile solo se la disponibilità di posti per l'evento non è esaurita.
- `acquistaPosti` consente l'effettivo acquisto dei biglietti richiesti. Il metodo deve controllare che quanto richiesto sia disponibile prima dell'esecuzione del metodo, mentre al termine dell'esecuzione i posti richiesti devono risultare occupati. Il metodo fornisce un risultato booleano che indica l'esito dell'operazione.
- `numFile` è un metodo puro che restituisce il numero di file di posti disponibili per l'evento.
- `fila` è un metodo puro che ritorna i posti della fila selezionata.
- `postiDisponibili` è un metodo puro che restituisce i posti disponibili per l'evento in quel momento.

La classe `Posto` offre il metodo `boolean disponibile()` per sapere se il posto richiesto è disponibile. Se il posto richiesto non dovesse esistere, la risposta sarebbe comunque `false`. `Posto` offre anche il metodo `int fila()` che restituisce la fila a cui si trova il posto.

Si scrivano in JML le pre- e post-condizioni dei metodi `selezionaPosti` e `acquistaPosti`.

Soluzione: una specifica non eseguibile e' la seguente:

```
// @ assignable / nothing;
// @ requires (tipo == 0 || tipo == 1) &&
//             (0 < num <= postiDisponibili());
// @ ensures \result.size() == num &&
//           (\forall Posto p; \result.contains(p); p.disponibile()) &&
//           (tipo == 1) =>
//             !(\exists Posto p;
//                \result.contains(p);
//                (\exists int f;
//                  0 < f && f < p.fila());
//                (\exists Posto q;
//                  fila(f).contains(q);
//                  q.disponibile() && !\result.contains(q)))
//             )
//           );
ArrayList<Posto> selezionaPosti(int num, int tipo);

//@ requires p != null &&
//@           (\forall Posto posto; p.contains(posto); posto != null);
//@ ensures  (\result <==>
//             (\forall Posto posto;
//               p.contains(posto);
//               \old(posto.disponibile) && !posto.disponibile()
//             )
//           ) &&
//             (\forall Posto posto;
//               !p.contains(posto);
//               posto.disponibile() <==> \old(posto.disponibile())
//             )
boolean acquistaPosti(ArrayList<Posto> p);
```

Esercizio 2

Un `Orto` contiene diversi ortaggi (definiti dalla classe `Ortaggio`). Normalmente gli ortaggi sono disponibili tutto l'anno (ortaggi “generici”), ma alcuni sono stagionali (`OrtaggioStagionale`). Le due tipologie d’ortaggio definiscono un metodo `raccogli()` senza parametri per raccogliere l’ortaggio destinatario del messaggio di invocazione. Si vuole discutere se, rispettando il principio di sostituzione, `OrtaggioStagionale` debba essere definito come sotto-classe di `Ortaggio` o vice-versa. Nella discussione, si può introdurre se necessario un metodo pubblico e puro `pronto()` che restituisce vero se l’ortaggio è pronto per essere raccolto.

Si considerino, in particolare, le seguenti alternative per l’operazione `raccogli()`:

1. Se `raccogli` dovesse essere sempre definita come funzione “totale”, ovvero venisse restituita un’eccezione qualora l’ortaggio non fosse pronto per il raccolto, quale gerarchia di ereditarietà potremmo immaginare? Perchè? Come si esprimerebbe in JML la specifica di `raccogli` nelle due classi?
2. Se si preferisse invece definire `raccogli` come una funzione parziale nel caso si volesse raccogliere un `OrtaggioStagionale` fuori stagione, quale gerarchia di ereditarietà potremmo immaginare? Perchè? Come si scriverebbe in JML la specifica di `raccogli` nelle due classi?

Soluzione Punto 1

La specifica di Ortaggio deve essere

```
public class Ortaggio{  
    //@ ensures (* this e' raccolto *);  
    public void raccogli();  
}
```

mentre quella di OrtaggioStagionale sarebbe:

```
public class OrtaggioStagionale{  
    //@ ensures pronto() && (* this e' raccolto *);  
    //@ signals (NonProntoException e) !pronto() && !(* this e' raccolto *);  
    public void raccogli() throws NonProntoException;  
  
    //@ ensures (* pronto() e' true sse this puo' essere raccolto *);  
    public boolean pronto();  
  
}
```

Con queste definizioni, OrtaggioStagionale non puo' essere sottoclasse di Ortaggio, perche' sarebbe violata la regola delle segnature (che non consente l'aggiunta di eccezioni a raccogli()). E' sintatticamente possibile che Ortaggio derivi da OrtaggioStagionale, ma apparentemente si violerebbe la regola dei metodi, in quanto la ensures di Ortaggio().raccogli() e' in generale piu' debole di quella di OrtaggioStagionale.raccogli() (perche' elimina la pronto()). E' possibile pero' rendere le due classi compatibili, definendo pronto(), ereditato anche in Ortaggio, in modo da valere sempre true. Basta quindi definire Ortaggio come segue:

```
public class Ortaggio extends OrtaggioStagionale{  
    //@ ensures true;  
    public void pronto();  
}
```

Quindi, questa versione di Ortaggio non ridefinisce raccogli() (e quindi non puo' violare la regola dei metodi).

Si noti che la ridefinizione di pronto() in Ortaggio non viola la regola dei metodi, in quanto ogni ortaggio stagionale ha sempre la possibilita' di essere pronto().

Soluzione Punto 2

```
public class OrtaggioStagionale{  
    //@ requires pronto();  
    //@ ensures raccolto();  
    public void raccogli();  
}
```

Se OrtaggioStagionale fosse erede di Ortaggio, allora il principio di sostituzione sarebbe violato, in quanto OrtaggioStagionale.raccogli ha una precondizione piu' forte di quella di Ortaggio.raccogli. Come nel caso precedente, tuttavia, Ortaggio puo' essere definito come erede di OrtaggioStagionale, sempre ridefinendo il solo metodo pronto() (come true):

```
public class Ortaggio extends OrtaggioStagionale{  
    //@ ensures true;  
    public void pronto();  
}
```

Esercizio 3

Si consideri il metodo seguente e si generi un insieme di casi di test che soddisfi il criterio di copertura delle diramazioni (*branch coverage*)

```
0  public int proc (int x , int y ) {
1      int n;
2      if ( x==y ) {
3          return 1;
4      } else {
5          if ( x>y ) {
6              y=x*x;
7              n=0;
8          } else {
9              n=1;
10         }
11         while ( x<y ) {
12             n=n+1;
13             x=x+2;
14         }
15     }
16     return n;
17 }
```

Soluzione: bastano tre casi di test, in modo da attraversare le diramazioni 3, 4, 6 9, 12, 15
Ad esempio: (1,1) (percorre 0,1,2,3), (2,1) (percorre 0,1,2,4,5,6,7,11,15,16), e (1,2) (percorre 0,1,2,4,5,8,9,10,11,12,13,14,11,15,16)

Esercizio 4

Un'applicazione per ascoltare musica dalla rete consente a ogni utente (account) di caricare i propri brani e di ascoltarli da un qualsiasi dispositivo. Un brano è identificato da un nome, una durata, un artista, un album, un genere e un contenuto (il brano vero e proprio). Ogni utente ha un nome, una password e un livello di servizio (base, avanzato, professionale). Ogni utente può registrare un numero massimo di 5 dispositivi con cui ascoltare la propria musica. Inoltre, ad ogni utente è sempre associata una e una sola collezione di brani, mentre l'utente può decidere di visualizzarla e usarla in modi diversi, ordinando i diversi brani per artista, per genere, per album, oppure random. L'utente deve poter scegliere la politica d'accesso a piacimento e deve anche poterla cambiare liberamente mentre ascolta la propria musica.

Si definisca un diagramma delle classi UML per rappresentare la specifica sopra riportata. Si consiglia di adottare opportuni design pattern, motivandone l'uso.

Traccia della soluzione: Definire le classi Account, Dispositivo, CollezioneBrani, Brano, e un’interfaccia LivelloDiServizio, implementata dalle tre classi LivelloBase, LivelloAvanzato, LivelloProfessionale. L’implementazione di queste tre classi potrebbe utilizzare un singleton. Si completa con opportune relazioni di associazione (possiede), usa, DispositiviRegistrati, contiene, ecc, e relative cardinalità’.

Per quanto riguarda l’accesso, si potrebbe utilizzare il pattern Strategy, definendo una classe astratta VistaBrani e quattro classi concrete (VistaPerArtista, VistaPerGenere, VistaPerAlbum, VistaRandom). La VistaBrani e’ in una relazione “mostra” con la classe Brano, ed e’ usata dalla classe Account per stabilire la VistaInUso (che puo’ quindi cambiare dinamicamente).

Appello del 14 luglio 2005

Cognome

Nome

Matricola

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere **in matita**. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.). Non è possibile lasciare l'aula conservando il tema della prova in corso.
3. Tempo a disposizione: 2h., Punteggio totale a disposizione: 27/30.

Esercizio 1 (punti 10)

Il metodo statico *permutazione* riceve in ingresso un array *num* e restituisce un booleano. Il valore restituito vale true se, e solo se, l'array num contiene una permutazione dei valori 0, 1, ..., num.length-1.

Ad esempio, per gli array [0 3 4 2 1] e [1 3 2] il metodo restituisce true, mentre per [0 1 2 3 5] o [1 2 2 3 4 5] il metodo restituisce false..

- a) Si scriva la specifica formale in JML del metodo, senza utilizzare commenti.

```
//@ requires num != null;  
.....  
//@ ensures  
//@ \result ==(\forallall int i; 0<= i && i <num.length;  
//@           (\existsint int j; 0<=j && j<num.length; num[j] == i));  
  
public static boolean permutazione(int [] num)
```

- b) Si elenchi il minimo numero di casi di test funzionali per il metodo che ritenete sufficientemente adeguati per il suo test, specificando sia i dati che il valore ritornato atteso.

(utilizzare il retro del foglio se necessario)

- [] true (cond. ai limiti)
- [0] true (cond. ai limiti)
- [1] false (cond. ai limiti)
- [0 3 4 2 1] true (combinazioni prop.)
- [0 3 4 2 2] false (combinazioni prop.)

Esercizio 2 (punti 17)

Si consideri la seguente classe **Numerolllimitato** che rappresenta numeri interi che possono crescere illimitatamente (cioe' senza essere limitati dal numero di bit della loro rappresentazione). Si supponga che la classe abbia la seguente specifica:

```
public class NumeroIlliimitato {  
    //OVERVIEW: Rappresenta numeri interi, non negativi, in base 10 e la  
    //cui dimensione è qualunque. Tipo Immutabile.  
    //@ensures (* \result è il numero di cifre di this *)  
    public int size();  
    //@ensures (*costruisce NumeroIlliimitato inizializzato all'intero  
    n*);  
    public NumeroIlliimitato(long n),  
        //@ensures (* \result è un iteratore per scorrere le cifre dalla  
        //meno significativa alla più significativa. Ogni cifra è un Integer.  
    *)  
    public Iterator cifre();  
  
    //@ensures (* \result è la somma di this e n *)  
    public NumeroIlliimitato somma(Numerolllimitato n)  
  
    //@requires 1<=n && n<=9;  
    //@ensures (* \result è ottenuto da this aggiungendo la cifra n in  
    //base 10 nella posizione più significativa *);  
    public addCifra(int n);  
    ... più altri metodi che ignoriamo...  
}
```

Parte (a).

Utilizzando solamente i metodi pubblici di Numerolllimitato, si implementi, sotto l'ipotesi semplificativa che this e n abbiano lo stesso numero di cifre, il metodo **somma**:

```
Numerolllimitato res = new Numerolllimitato(0);  
int riporto =0; int c = 0;  
for(Iterator i = this.cifre(), Iterator j=n.cifre(); i.hasNext();) {  
    int c1= ((Integer) i.next().intValue());  
    int c2= ((Integer) j.next().intValue());  
    c = c1 + c2 + riporto;  
    if (c>=10) {riporto =1; c = c-10;}  
    res.addCifra(c);  
}  
if (riporto>0) res.addCifra(1);  
return res;
```

Parte (b)

Si supponga che vi siano due implementazioni possibili per la classe Numerolllimitato:

La classe **NumerolllimitatoString** memorizza un numero in un oggetto di tipo String, utilizzando la consueta rappresentazione testuale in base 10 (la cifra 0 è memorizzata con il carattere "0", ecc.).

La classe **NumerolllimitatoBCD** utilizza un array per memorizzare la sequenza di cifre. Ogni elemento dell'array è a sua volta un array di 4 bit, che codifica una cifra decimale ([0,0,0,0] = 0, [0,0,0,1] = 1, [0,0,1,0] = 2, ..., [1,0,0,1] = 9, mentre le codifiche corrispondenti ai numeri binari 1010, 1011, 1100, 1101, 1110, e 1111 non sono utilizzate).

Si mostri il rep e si tratteggi l'implementazione del metodo addCifra nei due casi.

NumerolllimitatoString:

Il rep e':

```
private String num;
```

addCifra:

```
Numerolllimitato nuovo = new NumerolllimitatoString(0);
if (n>0 && n<=9) {
    String cifra = String.valueOf(n); //converte n in stringa
    nuovo.num = c+this.num;
    return nuovo;
}
```

NumerolllimitatoBCD:

Il rep e'

```
private boolean num[][];
```

addCifra:

```
Numerolllimitato nuovo = new NumerolllimitatoBCD(0);
if (n>0 && n<=9) {
    boolean cifra[] = new boolean[4];
    "converte n nell'array cifra[]";
    nuovo.num = new boolean [this.length][4];
    "copia in nuovo.num ogni riga di this.num";
    nuovo.num[nuovo.num.length-1] = cifra;
    return nuovo;
}
```

Parte (c) Si discutano brevemente vantaggi e svantaggi di ciascuna delle due implementazioni precedenti e si scriva l'invariante di rappresentazione come private invariant in JML per entrambi i casi:

L'implementazione con le stringhe e' piu' semplice da realizzare. L'implementazione con i BCD e' piu' macchinosa, ma, se i boolean sono effettivamente implementati con un singolo bit di memoria, puo' occupare molto meno spazio, prezioso per numeri di dimensioni illimitate.

Ri per NumerolllimitatoString:

```
num !=null && (!forall int i; 0<= i<num.size(); num.charAt[i] >= "0" &&
num.charAt[i]<="9");
```

Ri per NumerolllimitatoBCD:

```
num != null && (!forall int i; 0<= i && i <num.size(); num[i].size() == 4 &&
(!num[0] && !num[1] && !num[2] && !num[3] || NB: 0000
!num[0] && !num[1] && !num[2] && num[3] || NB: 0001
...
num[0] && !num[1] && !num[2] && num[3]) NB: 1001
```

Le ultime clausole formalizzano il fatto che ogni elemento di num[i] deve corrispondere a una rappresentazione BCD.

Parte (d)

Si realizzi per la classe NumerolllimitatoString anche il generatore (cioe' la classe che implementa l'iteratore) per poter implementare il metodo **Iterator cifre()**

In NumerolllimitatoString si dichiara la seguente classe:

```
private class NumlllimitatoStringGen implements Iterator {  
    private int pos;  
    private NumlllimitatoString n;  
    public NumlllimitatoStringGen(NumlllimitatoString t) {n=t; pos ==0;}  
    public boolean hasNext() {pos < n.size();}  
    public Object next() throw NoSuchElementException{  
        if (pos>= n.size()) throw new NoSuchElementException();  
        return (n.charAt(pos).parseInt());  
    }  
}
```

Il metodo cifre() è implementato come:

```
return new NumlllimitatoStringGen(this);
```

NB: le soluzioni sono solo indicative e non corrispondono a codice Java immediatamente compilabile e eseguibile.

Parte (e)

Discutere brevemente, aiutandosi anche con diagrammi UML, come devono essere organizzate le classi in modo che:

- 1) gli utilizzatori della classe Numerolllimitato non conoscano l'esistenza delle due classi NumlllimitatoString e NumlllimitatoBCD,
- 2) la scelta della rappresentazione possa essere realizzata da parte dei metodi produttori.

Si possono proporre anche modifiche alle classi presentate, qualora necessarie per rispettare (1) e (2).

La classe Numerolllimitato deve essere astratta e senza costruttori. L'implementazione e' realizzata dalle due classi eredi di Numerolllimitato: NumlllimitatoString e NumlllimitatoBCD,

La classe Numerolllimitato non fornisce costruttori, ma un metodo statico del tipo:

```
//result è un Numerolllimitato pari a n  
public static Numerolllimitato costruisci(int n) {  
}
```

In base ad esempio alle dimensioni di n, il metodo stabilisce se chiamare il costruttore di NumlllimitatoString o NumlllimitatoBCD.

L'implementazione dei produttori come somma dovrà includere anche la decisione del tipo effettivo dell'oggetto ritornato. Ad esempio, la NumlllimitatoString.summa() potrebbe restituire un NumlllimitatoString se anche il suo argomento è un NumlllimitatoString, ma altrimenti restituire un NumlllimitatoBCD.



Politecnico di Milano

Anno accademico 2012-2013

Ingegneria del Software – Appello del 27 febbraio 2014

Cognome:

Nome:

Matricola:

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 1

Si scrivano le pre- e post-condizioni del metodo `static boolean confronto(ArrayList<Integer> a, ArrayList<Integer> b)`, nei tre casi seguenti:

1. Il metodo ritorna `true` se gli `arrayList a` e `b` contengono gli stessi elementi, magari in ordine diverso, senza ammettere elementi ripetuti;
2. Il metodo ritorna `true` se gli `arrayList a` e `b` contengono gli stessi elementi, magari in ordine diverso, ammettendo elementi ripetuti;
3. Il metodo ritorna `true` se gli `arrayList a` e `b` contengono gli stessi elementi, magari in ordine diverso, ammettendo sia elementi ripetuti sia il caso in cui il numero di occorrenze di un certo elemento `n` in `a` possa essere diverso dal numero di `n` in `b`.

Esercizio 2

Il Guardaroba di una persona contiene abiti indossabili sempre (istanze di Abito) e abiti stagionali (AbitoStagionale), adatti solo per una o più stagioni:

1. Volendo definire un'opportuna gerarchia di abiti, e supponendo che ogni abito offra un metodo `public boolean indossabile(Stagione s)`, sarebbe possibile definire una gerarchia di ereditarietà che rispetti il principio di sostituibilità di Liskov? Motivare brevemente la risposta.
2. Volendo definire un iteratore `abitiEstivi()` per la classe Guardaroba che restituisca i soli abiti adatti alla stagione estiva, quali classi definiresti per le diverse tipologie di abito e per l'iteratore e come implementeresti i suoi metodi?
3. L'uso di una classe *astratta* semplificherebbe la realizzazione dell'iteratore?

Esercizio 3

Si consideri il seguente metodo Java:

```
public static int test(int x, int y) {  
    int z = x;  
  
    while (z >= 0) {  
        if (z < y || x == -1) break;  
        else x = x / z;  
        z--;  
    }  
    return z;  
}
```

1. Si disegni il diagramma del flusso di controllo;
2. Si identifichi, se esiste, un insieme di test (minimo) per coprire tutte le istruzioni e le decisioni (branch) del metodo.
3. Si identifichi, se esiste, un insieme di test (minimo) per coprire tutte le istruzioni e le condizioni del metodo.
4. Si spieghi il comportamento del metodo per $x = 4$ e $y = -1$.

Esercizio 4

Si progetti il diagramma delle classi UML per realizzare una semplice applicazione che opera su array di interi. Oltre alle solite operazioni di somma degli elementi, ricerca del valore minimo, medio e massimo e di calcolo della mediana, l'applicazione deve consentire la visualizzazione degli array in modi diversi. Ad esempio, sarebbe possibile usare il modo compatto (3, 45, -8, 56), oppure quello standard ($a[0] = 3, a[1] = 45, a[2] = -8, a[3] = 56$). Si vorrebbe anche poter aggiungere ulteriori formati di stampa personalizzati senza stravolgere le classi esistenti. Come si potrebbe realizzare la struttura delle classi per essere certi di avere un solo motore di calcolo e per poter variare le opzioni di stampa in modo semplice? Quali *design pattern* utilizzereste?



Politecnico di Milano

Anno accademico 2013-2014

Ingegneria del Software – Appello del 10 luglio 2014

Cognome:

Nome:

Matricola:

Sezione (segnarne una): Baresi

Ghezzi

San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h.
6. Se il compito risultasse insufficiente, non sarà possibile partecipare al prossimo appello.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 4:

Esercizio 5:

Esercizio 1

È data la specifica della classe `IntSet` (come presentata in aula), con i seguenti metodi:

- osservatori puri `isIn`, `size`;
- costruttori: `IntSet`;
- modificatori: `insert`, `remove`.

Si vuole aggiungere un nuovo metodo, chiamato `extract`, che restituisce un `IntSet` e che ha parametro `x` intero. Il metodo solleva un'eccezione `IllegalArgumentException` se `x` non appartiene all'insieme corrente, altrimenti restituisce il sottoinsieme dei valori contenuti nell'insieme corrente inferiori a `x`.

1. Fornire una specifica formale del metodo `extract`, ipotizzando che si tratti di un osservatore puro.
2. Definire un insieme di casi di test per il metodo utilizzando un approccio *black-box*
3. Supporre che il metodo `extract` abbia l'effetto collaterale di eliminare dall'insieme corrente il sottoinsieme calcolato. Come cambia la specifica?
4. Come cambia l'insieme dei dati di test per la nuova versione del metodo?

SOLUZIONE

Punto 1

Il metodo `extract` deve essere dichiarato puro e deve avere questa clausola ensures

```
//@ ensures (forall int i;;(result.isIn(i) <==> this.isIn(i) && i<x));
//@signals (IllegalArgumentException e) (!isIn(x));
public /*@ pure */ IntSet extract(int x);
```

Punto 2

1. caso in cui `x` non appartiene e insieme non vuoto, p.es. $(3, \{1, 5, 2\})$, risultato atteso: $\{1, 5, 2\}$.
2. caso in cui insieme vuoto, p.es. $(3, \{\})$, risultato atteso: $\{\}$
3. caso in cui `x` appartiene (e ovviamente insieme non vuoto), p.es. $(3, \{1, 3, 2\})$, risultato atteso: $\{1, 2\}$
4. caso in cui il risultato è vuoto, p.es. $(3, \{3, 5, 6\})$, risultato atteso: $\{\}$
5. caso in cui il risultato è il massimo sottoinsieme che può essere ritornato, cioè è l'intero insieme tranne il valore massimo, p.es. $(3, \{1, 3, 2\})$, risultato atteso: $\{1, 2\}$

Punto 3

Il metodo non è più puro. Nella specifica di ensures cui al punto 1 a `this` bisogna sostituire `old(this)` e aggiungere la condizione che tutti e soli gli elementi diversi da `x` sono ancora nell'insieme. Anche la signals deve modificata: il metodo non è puro quindi occorre imporre che `this` non cambi.

```
\begin{verbatim}
//@ ensures (forall int i;;result.isIn(i) <==> this.isIn(i) && i<x) &&
//@   (forall int i;;this.isIn(i) <==> \old(this.isIn(i)) && i>=x);
//@signals (IllegalArgumentException e) (!isIn(x)) &&
//@   (forall int i;;isIn(i) <==> \old(isIn(i)));
public IntSet extract(int x);
```

Punto 4

Se non era presente nei test definiti per la versione pura del metodo (ma nel nostro caso lo era), aggiungere un test in cui vengono eliminati zero elementi e uno in cui vengono eliminati tutti gli elementi. Il metodo non è puro, quindi i test precedenti devono anche verificare il valore di `this` al termine della chiamata (i valori di `this` sono rispettivamente: $\{1, 5, 2\}, \{\}, \{\}, \{5, 6\}, \{\}$).

Esercizio 2

Si considerino le seguenti classi Java:

```
public class Frutto {  
    public String toString(){return "Frutto";}  
    public void macedonia(Frutto f){System.out.println(this + " - " + f);}  
    public void confronta(Frutto f){System.out.println("Sono uguali");}  
}  
  
public class Mela extends Frutto {  
    public String toString(){return "Mela";}  
    public void macedonia(Mela f){System.out.println(this + " - " + f);}  
    public void confronta(Mela m){System.out.println("Sono diversi");}  
}  
  
public class Arancia extends Frutto {  
    public String toString(){return "Arancia";}  
    public void macedonia(Arancia f){System.out.println(this + " - " + f);}  
    public void confronta(Arancia a){System.out.println("Non ho idea");}  
}
```

e si spieghi cosa stamperebbe il seguente metodo main, motivando brevemente le risposte:

```
public class Test {  
    public static void main(String args[]) {  
        Frutto f1, f2;  
        Mela m;  
        Arancia a;  
  
        f1 = new Mela();  
        f2 = new Arancia();  
        m = new Mela();  
        a = new Arancia();  
  
        f1.macedonia(f2);  
        f1.confronta(f2);  
  
        a.confronta(f2);  
        a.confronta(m);  
        f2 = m;  
  
        f1.macedonia(f2);  
        a.macedonia(m);  
        a.confronta(a);  
    }  
}
```

SOLUZIONE

Le stampe sono
Mela-Arancia
Sono uguali
Sono uguali
Sono uguali
Mela-Mela
Arancia-Mela
non ho idea

Esercizio 3

È data la specifica della classe Punto con i seguenti metodi:

- osservatori puri (getter) float getX(), float getY() per ottenere le coordinate X e Y in un piano;
- costruttore Punto(float x, float y).

Si vuole specificare una classe NuvolaPunti con i seguenti metodi:

- osservatori puri Punto getBaricentro(), float getPrecisione(), int numPunti(), boolean isIn(Punto p);
- costruttore NuvolaPunti(float baricentro, float precisione);
- modificatori inserisciPunto(Punto p) throws violaInvariante, eliminaPunto(Punto p) throws violaInvariante.

Sostanzialmente la classe rappresenta un insieme di punti (numPunti fornisce la cardinalità dell'insieme) che devono rispettare un vincolo dato da un invariante (vedi punto successivo), la cui potenziale violazione viene segnalata da una eccezione da parte dei metodi modificatori.

1. Si vuole che la classe soddisfi l'invariante (astratto) che il valore medio delle coordinate X e il valore medio delle coordinate Y approssimino i valori delle coordinate del valore ottenuto mediante getBaricentro() entro il valore di precisione dato da getPrecisione(). Specificare formalmente tale invariante astratto.
2. Si specificino i metodi inserisciPunto e eliminaPunto in modo tale da assicurare che la proprietà invariante sia soddisfatta.
3. Si ipotizzi che l'invariante astratto venga ulteriormente rafforzato richiedendo che i punti della nuvola debbano avere coordinate non negative (non è richiesto scrivere questa nuova versione dell'invariante). Si ipotizzi anche che l'implementazione di NuvolaPunti venga realizzata mediante un ArrayList. Quale rep invariant deve essere rispettato dagli oggetti concreti della classe NuvolaPunti?

SOLUZIONE

Denotiamo con sommaX l'espressione $(\sum \text{Punto pun; } \text{isIn(pun); } \text{pun.getX()})$ e con sommaY l'espressione $(\sum \text{Punto pun; } \text{isIn(pun); } \text{pun.getY()})$.
1)

```
//@ public invariant numPunti()>0 ==>
//@ Math.abs(sommaX/numPunti() -getBaricentro.getX()) <=getPrecisione() &&
//@ Math.abs(sommaY/numPunti() -getBaricentro.getY()) <=getPrecisione();
```

2) La specifica di inserisciPunto deve garantire l'inserimento di p e la proprietà invariante. Per garantire che il metodo non costruisca una nuvola che viola l'invariante, occorre specificare nella signals che l'insieme dei punti attuali uniti al nuovo punto p violerebbe l'invariante.

```
//@ requires p != null;
//@ ensures
//@   (\forallall Punto pun;; \text{isIn(pun)} <==> \old{\text{isIn}(pun)} || pun == p) &&
//@   Math.abs(sommaX/numPunti() -getBaricentro.getX()) <=getPrecisione() &&
//@   Math.abs(sommaY/numPunti() -getBaricentro.getY()) <=getPrecisione();
//@ signals !\old{\text{isIn}(p)} && (\forallall Punto pun;; \text{isIn(pun)} <==> \old{\text{isIn}(pun)}) ) &&
//@   Math.abs((p.getX() + sommaX)/(numPunti() + 1) -getBaricentro.getX()) > getPrecisione() &&
//@   Math.abs((p.getY() + sommaY)/(numPunti() + 1) -getBaricentro.getY()) > getPrecisione();
public void inserisciPunto(Punto p);
```

La estraiPunto è analoga, ma la signals deve considerare la violazione dell'invariante nel caso in cui p sia eliminato.

```

//@ requires p != null;
//@ ensures
//@  (\forall Punto pun;; isIn(pun) <=> \old(isIn(pun)) && pun != p) &&
//@  Math.abs(sommaX/numPunti()-getBaricentro.getX()) <=getPrecisione() &&
//@  Math.abs(sommaY/numPunti() -getBaricentro.getY()) <=getPrecisione();
//@ signals \old(isIn(p) && (\forall Punto pun;; isIn(pun) <=> \old(isIn(pun)) ) &&
//@  Math.abs((-p.getX()+ sommaX)/(numPunti()-1)-getBaricentro.getX())>getPrecisione() &&
//@  Math.abs((-p.getY()+ sommaY)/(numPunti()-1)-getBaricentro.getY())>getPrecisione();
public void estraiPunto(Punto p);

3)
//rep:
private ArrayList<Punto> punti;
//@ private invariant punti!=null && !punti.contains(null) &&
//@  (forall int i; 0<=i && i<punti.size(); punti.getX()>=0 && punti.getY()>=0);

```

Esercizio 4

Si consideri il seguente metodo Java:

```
public static void testMethod(int x, String s) {
    int l;

    if (x < 0) return;
    l = s.length();

    while (x >= 2) {
        if (x >= 0 || l > 0) x = x % l;
        else break;
    }
    return;
}
```

1. Si disegni il diagramma del flusso di controllo;
2. Si identifichi, se esiste, un insieme di test (minimo) per coprire tutte le istruzioni e le decisioni (branch) del metodo.
3. Si identifichi, se esiste, un insieme di test (minimo) per coprire tutte le istruzioni e le condizioni del metodo.
4. In generale un insieme di test che copre tutte le decisioni compre anche tutte le condizioni? Nel caso non fosse vero, potremmo dire che invece un insieme di test che copre tutte le condizioni compre anche tutte le decisioni?

1) ...

2) Non esiste un insieme che possa coprire tutte le istruzioni: il break (e il relativo branch) non è raggiungibile (infatti, il resto della divisione di un numero positivo x per un intero l non può che essere non negativo). Un insieme che copre tutti le rimanenti istruzioni e branch è costituito da due dati di test:

Primo dato: per uscire dal primo return basta prendere $x < 0$, s qualsiasi;

Secondo dato: per entrare nel loop e poi uscire dal secondo return, occorre che: $x \geq 2$, con s diverso da null (per evitare eccezione all'istruzione $l = s.length$) e di lunghezza tale per cui $x \% s.length < 2$ (p.es. $x = 2$, $s.length = 2$, e quindi il resto è zero.) Infatti, se si scegliesse un s per cui $x \% s.length \geq 2$, si entrerebbe in un loop infinito (non raggiungendo mai il return).

3) E' impossibile rendere falsa la condizione $x \geq 0$: la condizione non può quindi essere esercitata. Di fatto, il valore della condizione $l > 0$ (a causa della valutazione corto-circuitata), non è preso in considerazione nella valutazione del valore dell'or. Si può esercitare il caso false della condizione $l > 0$, che corrisponde a $l = 0$, ossia un dato di test del tipo (2, ""), che causa tuttavia una DivisionByZeroException.

4) Naturalmente coprire tutte le decisioni non garantisce la copertura delle condizioni. Anche il viceversa, tuttavia, non vale in generale: si tratta di casi "patologici", in cui il valore di verita' di una condizione non cambia anche assegnando tutti i possibili valori di verita' alle condizioni elementari di cui e' composta. Esempi tipici sono i seguenti, dove x e y sono variabili booleane:

```
if (x && y && !x) ... //sempre falso
else ...
if (x&&y ? !x : false) ... //sempre falso
else ...
if (foo(x && y)) ... //dove foo restituisce sempre false
else ...
```

Pur esercitando tutte le condizioni per x e per y , i branch else non possono essere eseguiti.

Esercizio 5

Si consideri la classe Stack che implementa la seguente interfaccia:

```
interface Stack_interface {  
    void push(int n);  
    int pop(); //estrae e ritorna l'elemento in cima allo stack  
    boolean is_empty();  
}
```

Si ipotizzi che oggetti della classe Stack possano essere manipolati concorrentemente e che quindi si debba garantire un accesso corretto. Se un task cerca di effettuare l'operazione pop quando l'oggetto stack è vuoto, esso dovrà essere sospeso fino a che un altro task non avrà effettuato un'operazione push (a meno che non intervenga una InterruptedException).

Si completi la parziale implementazione dei metodi in modo da garantire appunto un accesso corretto.

```
class Stack implements Stack_interface {  
    private class Element {  
        int info;  
        Element next;  
        Element(int n, Element e) { info = n; next = e; }  
    }  
    private Element first;  
    public Stack() { first = null; }  
  
    public void push(int n) {  
  
    }  
    public int pop() {  
  
    }  
    public boolean is_empty() {  
  
    }  
}
```

SOLUZIONE Una soluzione, senza particolari requisiti di efficienza, consiste nel dichiarare synchronized i tre metodi. La push deve notificare i thread che sono in attesa che la pila non sia vuota per completare un pop. Ignoriamo anche la gestione di InterruptedException.

```
public synchronized void push(int n){  
    first = new Element(n,first);  
    notifyAll();  
}  
public synchronized int pop(){  
    while (first == null) wait();  
    Element elm = first;  
    first = first.next;  
    return elm.info;  
}  
public synchronized boolean is_empty(){  
    return (first == null);  
}
```



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da

Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Il Prova di Ingegneria del Software

28 Giugno 2004

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h30m.
6. Punteggio totale a disposizione: 13/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 1 (punti 4)

Si consideri la seguente specifica informale sotto riportata che definisce le regole riguardanti il superamento dell'esame di Ingegneria del Software.

“Verranno svolte due prove intermedie. Ogni prova intermedia assegna un massimo di 13 punti ed è considerata valida se lo studente ottiene almeno un punteggio minimo di 6 punti per la I prova e di 6 punti per la II prova; chi ottiene un punteggio inferiore a quello minimo in una prova è obbligato a ripeterla nelle prove di recupero. La ripetizione riguarda comunque entrambe le prove; pertanto il mancato raggiungimento della soglia minima in una delle due prove richiede la partecipazione a una prova di recupero che riguarda l'intero programma del corso.

L'attività svolta in laboratorio permette di ottenere un massimo di 4 punti, ed è considerata sufficiente se lo studente ottiene almeno 2 punti. Non è previsto il recupero del laboratorio. Pertanto in caso di valutazione insufficiente lo studente dovrà ripetere il corso nell'anno accademico successivo: saranno annullati gli eventuali risultati ottenuti durante le prove in itinere e non sarà possibile partecipare agli appelli.

Per superare l'esame è inoltre necessario che la somma dei punteggi delle due prove in itinere sia almeno di 16 punti sui 26 disponibili e che il risultato complessivo (che comprende anche il voto relativo all'attività di laboratorio) sia almeno 18; lo studente che non soddisfa le precedenti condizioni dovrà recuperare entrambe le prove in un appello a propria scelta.”

1. Utilizzando opportune formule che definiscono le pre e post-condizioni, si fornisca la specifica di un'astrazione procedurale che in input riceve tre valori: P1, P2, L, che rappresentano i voti ottenuti nella I e nella II prova intermedia e nel laboratorio, rispettivamente. In output viene fornito un valore intero che rappresenta il voto conseguito (se sufficiente), oppure 0, se l'allievo deve sostenere il recupero, o -1 se deve ripetere;
2. In base alla specifica fornita, si definiscano dati a supporto del test funzionale (black-box testing) in base al criterio di copertura delle combinazioni proposizionali e i risultati attesi per ciascun dato.
3. Si identifichino almeno 5 dati di test corrispondenti a *valori limite* dei parametri dell'astrazione.

Esercizio 2 (punti 6)

La classe astratta Coda specifica le operazioni per la manipolazioni di una coda FIFO di Object: insert (inserisce in coda un Object), remove (estrae un Object dalla coda, eliminandolo, e lo restituisce come risultato), length (lunghezza della coda). L'iteratore Elements restituisce un generatore che consente di iterare su tutti gli oggetti in coda. Le operazioni ritornano un'eccezione (NullPointerException) nel caso in cui l'oggetto a cui si applicano sia null. L'operazione di inserimento può, in alcuni casi che non vengono dettagliati, sollevare un'eccezione di tipo OggettoNonValido anziché inserire l'oggetto. Non vengono generate altre eccezioni dalle operazioni (tranne l'eccezione CodaVuota nel caso si cerchi di estrarre un oggetto da una coda vuota); i metodi che implementano le operazioni definiscono inoltre funzioni totali.

1. Considerando la seguente implementazione parziale di Coda, si completino le operazioni con la loro specifica (OVERVIEW, REQUIRES, EFFECTS, MODIFIES)

```
public abstract class Coda {  
    .....  
    protected int tot;  
    public Coda() {tot = 0;}  
    .....  
    public abstract void insert (Object x) throws OggettoNonValido;  
    .....  
    public abstract Object remove() throws CodaVuota;  
    .....  
    public abstract Iterator Elements();  
    .....  
    public int length() {return tot;}  
    .....  
}
```

(continua esercizio 2)

2. Si vogliono definire due sottoclassi concrete di Coda, chiamate rispettivamente Codallimitata e CodaFinita, che rappresentano, rispettivamente, le code che possono crescere arbitrariamente di lunghezza e le code di lunghezza finita e predeterminata. Nel secondo caso, si assume che la lunghezza massima delle code sia definita da una costante statica privata e che il tentativo di inserimento di un Object in una coda piena generi un'eccezione checked CodaPiena. Codallimitata e CodaFinita rispettano il principio di sostituibilità? Motivare la risposta facendo riferimento a una precisa descrizione della specifica di CodaFinita e di Codallimitata.

```
public class CodaFinita extends Coda {  
    .....  
}  
  
public class Codallimitata extends Coda {  
    .....  
}
```

3. Si consideri la classe AltraCodaFinita, che ha la stessa definizione della classe CodaFinita ma con la seguente modifica: quando la coda è piena, il metodo insert genera un'eccezione OggettoNonValido (anziché l'eccezione CodaPiena). La classe AltraCodaFinita soddisfa il principio

di sostituibilità? Anche in questo caso giustificare la risposta, aiutandosi con una precisa descrizione della specifica di CodaFinita.

4. Si consideri la classe TerzaCodaFinita, la cui specifica è identica a quella di CodaFinita, con la seguente modifica: quando la coda è piena, il metodo insert non fa nulla. Dire, anche in questo caso, se TerzaCodaFinita rispetta o non rispetta il principio di sostituibilità, giustificando come al solito la risposta.

(continua esercizio 2)

5. Si tratteggi l'implementazione della classe Codalllimitata volendo che essa sia utilizzabile per definire un'astrazione polimorfa, usabile per definire code non limitate in dimensioni di elementi di un qualunque tipo. Si vuole verificare che gli elementi inseriti in una coda illimitata siano tutti dello stesso tipo. A tale scopo si adotti l'approccio *element subtype*, definendo un'interfaccia chiamata Accodabile con un metodo confrontaClasse; il metodo insert, se la coda in cui avviene l'inserzione non è vuota, invoca il metodo confrontaClasse su uno degli elementi già presenti in coda, passando come parametro l'oggetto di tipo Accodabile che si sta tentando di inserire. Nel caso che l'oggetto abbia tipo scorretto, questo non viene inserito nella coda, e viene sollevata un'eccezione di tipo OggettoNonValido. Definire una classe StringaAccodabile, che contiene un dato di tipo String e implementa Accodabile, codificando in dettaglio il suo metodo confrontaClasse.

```
public class Codalllimitata extends Coda {  
    .....  
    public void insert .....  
    .....  
    .....  
    .....  
    .....  
}  
  
public interface Accodabile {  
    public boolean confrontaClasse(Object x);  
}  
  
public class StringaAccodabile implements Accodabile {  
    private String s;  
    .....  
}
```

Esercizio 3 (punti 3)

Si descriva un diagramma delle classi UML per la seguente situazione. In una società che sviluppa software, quando si scopre un errore in un modulo software, viene generata una SegnalazioneDiMalfunctionamento, che contiene la descrizione del malfunzionamento e la data in cui esso si è manifestato. Ogni progettista può segnalare malfunzionamenti e ogni malfunzionamento ha associato il progettista che l'ha segnalato. Un malfunzionamento ha un attributo che ne indica lo stato. Quando viene segnalato, il suo stato viene considerato "aperto". Per eliminare un malfunzionamento, gli viene assegnato un progettista responsabile della sua correzione. Una volta corretto, lo stato del malfunzionamento diventa "potenzialmente chiuso". Al malfunzionamento sono associati uno o più dati di test. Se questi vengono eseguiti con successo, lo stato del malfunzionamento diventa "chiuso".

1. Si descriva un Class Diagram che illustra le entità in gioco e le relative associazioni.
2. Si descriva mediante un Sequence Diagram uno scenario in cui un particolare progettista esamina un malfunzionamento che si trova nello stato "potenzialmente chiuso", esegue i test associati al malfunzionamento e, nel caso questi vengano eseguiti con successo, dichiari il malfunzionamento "chiuso".

Politecnico di Milano

Ingegneria del Software – a.a. 2008/09

Appello del 09 Settembre 2009

Cognome

Nome

Matricola

Sezione (segnarne una) Baresi Ghezzi San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non verranno in nessun caso** presi in considerazione. È possibile scrivere **in matita**.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 2h15m.
6. Punteggio totale a disposizione: 25/30. La sufficienza si raggiunge con 16 punti, a cui sommare il risultato delle prove di laboratorio.

Esercizio 1:

Esercizio 2:

Esercizio 3:

Totale

Esercizio 1 (punti 6)

Si consideri il seguente metodo statico:

public static String[] permutazioni (String s)

Il metodo riceve come parametro una stringa non vuota di caratteri tutti diversi tra loro. Il metodo restituisce l'insieme di tutte e sole le permutazioni dei caratteri che compongono la stringa in ingresso, questa compresa, senza ripetizioni e in ordine casuale.

Ad esempio, se fosse "ABC", il metodo dovrebbe restituire: {"ABC", "ACB", "BAC", "BCA", "CAB", "CBA"}, includendo quindi tutte le $3!=6$ permutazioni di "ABC".

Specificare **permutazioni(...)** in JML.

Suggerimento: Si consiglia di specificare dapprima un metodo puro della classe String, denominato public /*@pure@*/ boolean permut(String x), che restituisce true se x è una permutazione di this, false in ogni altro caso. Utilizzare poi il metodo permut() nella specifica di permutazioni(..).

Per accedere ai singoli caratteri di una stringa, si può utilizzare il metodo della classe String "char charAt(int index)", che restituisce il carattere in posizione index di this, con index che va da 0 a this.length()-1.

SOLUZIONE:

permut può essere definita in modo generale, anche per casi con caratteri ripetuti, come segue, segnalando che il numero di occorrenze di ogni carattere in x è lo stesso del numero di occorrenze di quel carattere in this:

```
//@ensures \result == (x!= null && x.size()==this.0 &&
//@ (\forall int i; 0<=i && i<x.length();
//@   (num_of int j; 0<=j && j<x.length(); x.charAt(j)==x.charAt(i))
//@   ==
//@   (num_of int k; 0<=j && j<this.size(); this.charAt(k)==x.charAt(i)))
public /*@pure@*/ boolean permut(String x)
```

Permutazioni può a questo punto essere definito dicendo che ogni elemento in \result è una permutazione di s, che ogni elemento di \result è distinto, e che in tutto \result ha esattamente s.length()! elementi:

```
//@requires x!=null &&
//@ (\forall int i; 0<=i && i<s.length(); (num_of int j; 0<=j && j<s.length();
//@   s.charAt(i)==s.charAt(j))==1) //no ripetizioni in s
//@ensures (\forall int i; 0<=i && i<\result.length;
//@   s.permut(result[i])) &&
//@   (num_of int j; 0<=j && j<\result.length; \result[i].equals(result[j])==1) &&
//@   \result.length== Math факт(s.length())
public static String[] permutazioni (String s)
```

Esercizio 2 (punti 14)

Si consideri un'applicazione java di supporto al gestore di un social network. L'applicazione include una classe Java "Membro", che descrive un partecipante al network.

La classe Membro ha, fra gli altri, i seguenti metodi modificatori, correlati alla possibilità per un membro di diventare "amico" di altri membri:

1. `public void invita (Membro m)`

il cui effetto è di invitare un altro membro del social network a diventare un amico del membro this, purché non si cerchi di invitare se stessi oppure un membro che in passato abbia già risposto **positivamente** all'invito.

2. `public void accettaInvito (Membro m)`

il cui effetto è di accettare l'invito di un altro membro, purché questi abbia effettivamente invitato il membro this, rendendo quindi "amici" this e m.

3. `public void declinaInvito (Membro m)`

il cui effetto è di rifiutare l'invito di un altro membro a this, rimuovendo il relativo invito di m.

Si osservi che, in base a questi metodi, le possibili relazioni fra due membri A e B sono solo le seguenti:

- A e B sono amici (ossia uno ha invitato l'altro, che ha accettato);
- A e B non sono amici, ma uno ha invitato l'altro, che non ha ancora risposto;
- A e B non sono amici, e non vi sono inviti pendenti (perché nessuno dei due ha mai invitato l'altro oppure perché uno ha invitato l'altro che però ha rifiutato).

a) *Si specifichino i precedenti metodi 1,2 e 3 della classe Membro in JML. Si ipotizzi a tale scopo che Membro fornisca anche i seguenti osservatori puri:*

```
//restituisce l'elenco dei membri invitati da this, che non hanno risposto:  
public /*@ pure */ ArrayList<Membro> invitati()
```

```
//restituisce l'elenco degli amici di this:  
public /*@ pure */ ArrayList<Membro> amici()
```

e il seguente costruttore, di cui si presenta una specifica parziale, "depurata" dei vari argomenti come nome, email, ecc., che sono per semplicità ignorati in questa specifica:

```
//@ensures amici().size()==0 &&...altri condizioni...;  
public Membro(...)
```

SOLUZIONE:

Si utilizzano per semplicità i metodi contains e containsAll delle Collection (che verificano rispettivamente l'appartenza di un oggetto e l'inclusione di una collezione nell'ArrayList this).

Al solo scopo di abbreviare le specifiche, definiamo un ulteriore metodo statico puro:

```
//@requires a1!= null && a2!=null && x!=null  
//@ensures \result == (a1.containsAll(a2) && !a2.contains(x) &&  
//@ a1.contains(x) && a1.size() == 1+a2.size()  
public /*@pure@*/ static boolean differenza(ArrayList<T> a1, ArrayList<T>  
a2, <T> x)
```

il cui significato è quindi che a1 contiene tutti gli elementi di a2 (non importa in che ordine) e inoltre il solo x, ossia x è l'unico elemento di differenza fra a1 e a2.

```
//@ requires !m == null && !m==this && !amici().contains(m) &&  
//@ //non ci sono inviti pendenti fra m e this:  
//@!invitati().contains(m) &&!m.invitati().contains(this);  
//@ ensures differenza(invitati(),\old(invitati()), m) && //m è nuovo invitato  
//@ amici().equals(\old(amici()));  
//@assignable this.*;//solo this è modificato, m è inalterato.  
public void invita (Membro m)
```

```
//@ requires !m == null && m.invitati().contains(this);  
//@ ensures differenza(\old(m.invitati()), m.invitati, this) &&  
//@ differenza(amici(),\old(amici()), m) &&  
/@ differenza(m.amici(),\old(m.amici()),this) &&  
/@ invitati().equals(\old(invitati()));  
//@ assignable this.*, m.*;//sia this che m sono modificati.  
public void accettaInvito (Membro m)
```

```
//@ requires !m == null && && m.invitati().contains(this);  
//@ ensures differenza(old(m.invitati()),m.invitati(),this)  
//@ m.amici().equals(\old(m.amici()));  
//@ assignable m.*;//solo m è modificato, this è inalterato.  
public void declinaInvito (Membro m)
```

- b) Si considerino le seguenti proprietà, che sono parte del contratto di Membro:
- I) "Un membro non può invitare un amico"
 - II) "La relazione di amicizia fra membri è simmetrica, ma non riflessiva".
 - III) "Se due membri A e B sono amici, allora in passato A ha invitato B oppure B ha invitato A, e l'invito è stato accettato dall'altro membro"

b.1) Di che tipo di proprietà si tratta?

SOLUZIONE: Le prime due sono invarianti pubblici, la terza è una proprietà evolutiva. La prima potrebbe in realtà anche essere considerata una proprietà evolutiva, in quanto predica su un cambiamento di stato; tuttavia, come nel punto successivo, può essere facilmente espressa con un invariante pubblico.

b.2) Descrivelle in JML, se possibile. Nel caso non sia possibile farlo, spiegare come mai accade questo.

SOLUZIONE: I e II) sono:

- I) `public invariant (\forall Membro m; amici().contains(m); !invitati().contains(m));`
- II) `public invariant (\forall Membro m; amici().contains(m); m.amici().contains(this)) && !this.contains(this);`
- III) Non può direttamente essere espressa in JML, salvo che ricorrendo a qualche "trucco" (come inserire la proprietà come postcondizione dei metodi pubblici modificatori).

b.3) Le proprietà sono verificate dalla specifica data?

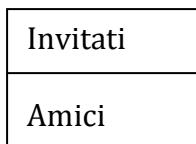
Se la soluzione e' quella data SI, in quanto:

- I) L'invito ad un amico comporta la violazione della requires di `invita(..)`;
- II) La postcondizione di `accettaInvito` include "`amici().contains(m) && m.amici().contains(this)`" garantendo la simmetria. L'invito a se stessi comporta la violazione della requires di `invita(..)`;
- III) Ogni membro all'inizio (v. il costruttore) non ha né amici né inviti, e l'unico modo disponibile per definire amici è tramite `accettaInvito`, che richiede che in precedenza sia stato inserito un invito per `this`.

c) Si consideri la seguente implementazione parziale di Membro:

```
//rep:  
private ArrayList<Membro> altriMembri;  
private int inizioInvitati;  
public Membro(...)//vari parametri ignorati qui{  
    altriMembri= new ArrayList<Membro>;  
    inizioInvitati=0;  
}
```

L'ArrayList `altriMembri` è suddiviso in DUE porzioni. La prima porzione, nelle posizioni da 0 a `inizioInvitati-1`, include tutti e soli gli amici di `this`; la seconda porzione, nelle posizioni da `inizioInvitati` alla fine di `altriMembri` (ossia da `inizio Invitati` ad `altriMembri.size()-1`) include tutti e soli gli invitati da `this` che non hanno ancora risposto. Se `inizioInvitati` vale `altriMembri.size()`, allora non vi sono invitati, mentre se `inizioInvitati` vale 0 allora non vi sono amici.



c1) Scrivere in JML l'invariante di rappresentazione della classe Membro.

SOLUZIONE:

```
private invariant altriMembri!=null && 0<= inizioInvitati && inizioInvitati<=altriMembri.size() &&  
(\forallall int i; 0<= i && i<altriMembri.size(); altriMembri.get(i) !=null)
```

c2) Descrivere, tramite un private invariant di JML, che metta in relazione i metodi osservatori puri e le parti private, una possibile funzione di astrazione della classe Membro, ossia una formula JML che descriva come un oggetto concreto, ossia definito dal rep, corrisponda ad un oggetto astratto, ossia definito dagli osservatori.

SOLUZIONE:

```
private invariant  
altriMembri.size()=amici().size()+invitati().size() &&  
(\forallall int i; 0<= i && i< altriMembri.size();  
    altriMembri.get(i).equals( i<inizioInvitati ? amici().get(i) :  
                                invitati().get(i)));
```

c3) Implementare il metodo declinaInvito.

SOLUZIONE:

```
public void declinaInvito(Membro m) {  
    for int i =m.inizioInvitati; i<m.altriMembri.size(); i++)  
        if (m.altriMembri.get(i).equals(this)) {  
            //ultimo elemento copiato in pos. i:  
            m.altriMembri.set(i, m.altriMembri.lastElement());  
            //rimuove ultimo elemento:  
            m.altriMembri.remove(m.altriMembri.size()-1);  
            return;  
        }  
}
```

- d) Si consideri una nuova classe MembroPrivilegiato, erede della classe Membro. Un membro privilegiato del social network può seguire la consueta procedura ad inviti per diventare amico di un altro membro. Tuttavia, ha la facoltà di potere diventare amico di un membro non privilegiato anche se questi non l'ha invitato o non ha accettato un suo invito.
A tale scopo, la classe fornisce un metodo public void diventa Amico(Membro m),

- 1) *Specificare tale metodo*, inserendo anche eventuali precondizioni o eccezioni se ritenute opportune:

SOLUZIONE:

```
//@requires m!= null && !m instanceof MembroPrivilegiato;
//@ ensures (\forall Membro m1; m!=m1 && m1!=this;
//@   (amici().contains(m1) <==>\old(amici().contains(m1))) &&
//@   (m.amici().contains(m1) <==>\old(amici().contains(m1)))) &&
//@ amici().contains(m) &&
//@ m.amici().contains(this) &&
//@ invitati().equals(\old(invitati()))&&
//@ m.invitati().equals(\old(m.invitati()));
//@ assignable this.* , m.*;
```

public void diventa Amico(Membro m)

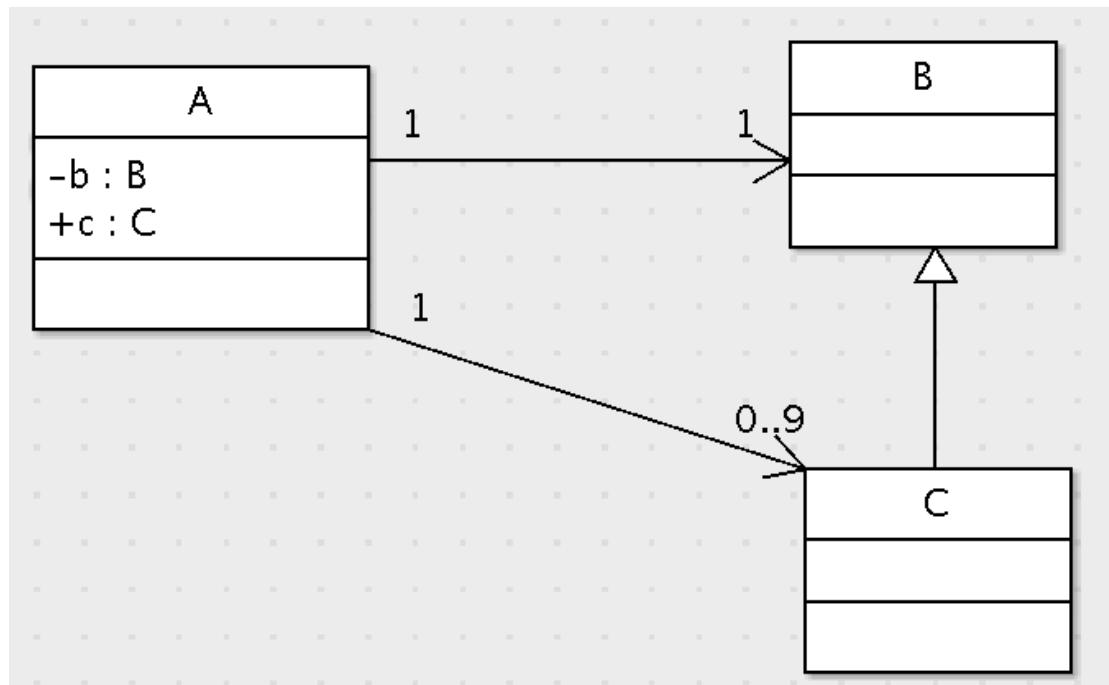
- 2) Commentare sinteticamente, ma efficacemente, sulla possibilità che MembroPrivilegiato rispetti, come erede di Membro, il principio di sostituzione di Liskov.

SOLUZIONE: le regole delle segnature e dei metodi sono verificate in quanto MembroPrivilegiato ha esattamente gli stessi metodi di Membro, con le stesse specifiche, oltre a un metodo addizionale. Tuttavia, non vale più la proprietà evolutiva B.III, che era da considerarsi parte integrante del contratto (anche se non esprimibile direttamente in JML). Pertanto, il principio di sostituzione NON è verificato.

Esercizio 3 (punti 5)

(A) Si consideri una classe A che ha come attributo pubblico un elemento di classe B e come attributo privato un array di 10 elementi di classe C. Si supponga anche che C erediti da B. Si disegni un diagramma UML che rappresenta questo frammento di programma.

Soluzione



(B) Si considerino i seguenti frammenti Java

```
interface X {  
    void x1();  
    float x2 (Yy);  
}  
class Y {  
    public void y1() {...}  
    public float y2 (X x) {...}  
}  
class Z implements X extends Y {  
    public void y1(float a) {...}  
    public float y2 (X x) {...}  
    public void x1() {...}  
    public float x2 (Y y) {...}  
}
```

e le dichiarazioni seguenti: float i, j; X a, Y b; Z c;

A fianco di ciascuna delle seguenti istruzioni si scriva SI o NO a seconda che l'istruzione sia corretta o no dal punto di vista del "type checking" e si scriva una sintetica motivazione, in particolare specificando quale metodo viene chiamato.

a. x1(); **(SI, nella situazione attuale l'oggetto a cui a riferisce puo' solo essere un elemento di classe Z, per cui verra' chiamata x1 implementata da Z)**

i = a. x2(c); **(SI, v. sopra, si chiama la x2 di Z, che come parametro si aspetta un Y, e Z e' sostituibile a Y)**

a = c; **[SI]**

i = b. y2(a); **(SI, si chiama la y2 di Y o se il tipo dinamico dell'oggetto dovesse essere Z, la y2 ridefinita in Z)**

j = b. y2(c); **(SI, c.s., essendo Z sostituibile a X)**

c = a; **NO**

c = (Z) a; **(SI, ma possibile errore a run time se il tipo concreto di a non dovesse essere Z)**

c. y1(); **(SI, viene chiamata la y1 ereditata da Y)**

c. y1(3.77); **(SI, viene chiamata la y1 definita da Z, che genera un overloading di quella ereditata)**



Dipartimento di Elettronica e Informazione

Politecnico di Milano

prof. Carlo Ghezzi

prof. Angelo Morzenti

prof. Pierluigi San Pietro

20133 Milano (Italia)

Piazza Leonardo da Vinci, 32

Tel. (39) 02-2399.3400

Fax (39) 02-2399.3411

Appello di Ingegneria del Software

9 Febbraio 2005

Cognome

Nome

Matricola

Sezione (segnarne una) Ghezzi , Morzenti , SanPietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare **solo i fogli distribuiti** utilizzando il **retro** delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. **non** verranno **in nessun caso** presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È **proibito** l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, ecc.).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. Tempo a disposizione: 1h30m.
6. Punteggio totale a disposizione: 26/30.

Valutazione:

Esercizio 1:

Esercizio 2:

Esercizio 3:

Esercizio 1 (punti 10)

Il metodo statico suGiuOgiuSu riceve come parametro un array a contenente tre numeri interi e restituisce al chiamante un valore di tipo *boolean*; se $a[0]>a[1]$ allora se $a[1]<a[2]$ suGiuOgiuSu restituisce true, altrimenti restituisce false; se $a[0]<a[1]$ allora se $a[1]>a[2]$ restituisce true, altrimenti false.

- a) Considerando il paragrafo precedente come una specifica del metodo suGiuOgiuSu *evidenziare un suo difetto* particolarmente rilevante, motivando adeguatamente la risposta.

R. La specifica è incompleta, perché non è definito il valore restituito nel caso in cui $a[0]$ è uguale ad $a[1]$.

- b) Sempre considerando il primo paragrafo come una specifica del metodo, *fornire 4 casi di test funzionali* per tale metodo, scegliendoli, e motivando tale scelta, sulla base del criterio delle combinazioni proposizionali.

R. Si possono scegliere 4 casi che rendano vere o false le due proposizioni $a[0]>a[1]$ e $a[1]>a[2]$. Per esempio, si possono quindi ottenere

a[0]	a[1]	a[2]	Val. Restituito
1	2	1	true
1	2	3	false
2	1	3	true
2	1	0	false

- c) Si consideri la seguente implementazione del metodo, non necessariamente corretta rispetto alla specifica.

```
static boolean suGiuOgiuSu (int [] a) {
    if ( a[0]>a[1] && a[1]<a[2] )
        return true;
    else    if ( a[0]<a[1] )
        if ( a[1]>a[2] )
            return true;
        else return false;
    else return false;
}
```

Indicare, motivando la risposta, *quanti dati di test occorrono* per effettuare un test strutturale che copra tutte le diramazioni. Fornire un esempio di tali dati di test.

R. Bastano 4 dati di test, che corrispondono ai 4 cammini che portano alle 4 istruzioni *return*.

- 1) $a[0]>a[1]$ e $a[1]<a[2]$ e.g. $a[0]=2, a[1]=1, a[2]=3$
- 2) $a[0]<a[1]$ e $a[1]>a[2]$ e.g. $a[0]=2, a[1]=3, a[2]=1$
- 3) $a[0]<a[1]$ e $a[1]<=a[2]$ e.g. $a[0]=2, a[1]=3, a[2]=4$
- 4) $a[0]>=a[1]$ e $a[1]>=a[2]$ e.g. $a[0]=3, a[1]=2, a[2]=1$

d) Per ottenere la copertura rispetto al criterio delle condizioni è *necessario aggiungere altri dati di test oltre a quelli indicati* al punto (c)? Se sì, indicare quali, se no, spiegare perché.

R. No, perché quelli sopra riportati permettono di rendere vere e false ognuna delle tre condizioni delle istruzioni if e anche ognuna delle due componenti della prima di esse ($a[0]>a[1]$ e $a[1]<a[2]$).

e) Con riferimento al punto (c), *definire un'eccezione TuttiUgualiException di tipo checked, che possieda solo due costruttori, uno senza argomenti e uno con un argomento di tipo stringa. Modificare il codice del metodo suGiuOgiuSu in modo che esso lanci un'eccezione TuttiUgualiException nel caso in cui i tre elementi dell'array siano tutti uguali.*

R.

```
class TuttiUgualiException extends Exception {  
    public TuttiUgualiException () { super(); }  
    public TuttiUgualiException (String s) { super(s); }  
}  
  
static boolean suGiuOgiuSu (int [] a) throws TuttiUgualiException {  
    if (a[0]==a[1] && a[1]==a[2])  
        throw new TuttiUgualiException ("suGiuOgiuSu ");  
    if ( a[0]>a[1] && a[1]<a[2] )  
        return true;  
    else    if ( a[0]<a[1] )  
        if ( a[1]>a[2] )  
            return true;  
        else return false;  
    else return false;  
}
```

Esercizio 2 (punti 8)

Si vuole definire la classe *ExtremelyLongInt* che permetta di rappresentare e manipolare numeri interi anche estremamente elevati. Allo scopo di facilitare l'effettuazione di calcoli algebrici (p.es., il calcolo del massimo comune divisore e del minimo comune multiplo tra due oggetti *ExtremelyLongInt*) si sceglie di rappresentare un qualsiasi valore *ExtremelyLongInt* con un REP che consista nella scomposizione del numero in fattori primi.

```
class ExtremelyLongInt {
    ...
    // il REP
    private int [] base; // contiene tutti i fattori primi dell'oggetto ExtremelyLongInt
    private int [] exp; // contiene gli esponenti nello stesso ordine dei fattori
    private int segno; // contiene +1 se il numero è positivo, -1 altrimenti
    ...
}
```

Per esempio, il valore astratto 360 di tipo *ExtremelyLongInt* ha una rappresentazione in cui segno =1, base = {2, 3, 5} ed exp = {3, 2, 1}, dal momento che $360 = 2^3 \cdot 3^2 \cdot 5^1$.

a) Scrivere (senza implementare) l'*invariante di rappresentazione* per tale astrazione, usando formule aritmetiche commentate con frasi in linguaggio naturale.

R.

(segno = -1 || segno =1)
 $\&\& \text{base.length} = \text{exp.length} \quad // \text{i due array hanno la stessa lunghezza}$
 $\quad // (\text{tante basi quanti esponenti})$
 $\&\& \forall i \ (0 \leq i < \text{exp.length} \rightarrow \text{exp}[i] > 0) \ // \text{gli esponenti sono tutti positivi}$
 $\&\& \forall i \forall j \ (0 \leq i, j < \text{exp.length} \ \& \& i \neq j \rightarrow \text{base}[i] \neq \text{base}[j]) \ // \text{basi tutte diverse}$
 $\&\& \forall i \ (0 \leq i < \text{base.length} \rightarrow \text{primo}(\text{base}[i])) \ // \text{i fattori sono tutti primi.}$
dove $\text{primo}(j)$ vale true sse j è un numero primo.
Formalmente, $\text{primo}(j) \Leftrightarrow \neg \exists h > 1, k > 1 : h * k = j$

b) Scrivere la *funzione di astrazione*.

R. La funzione di astrazione prende come argomenti il REP, cioè la coppia di array base e exp, restituisce il valore del numero rappresentato ed è definita come segue

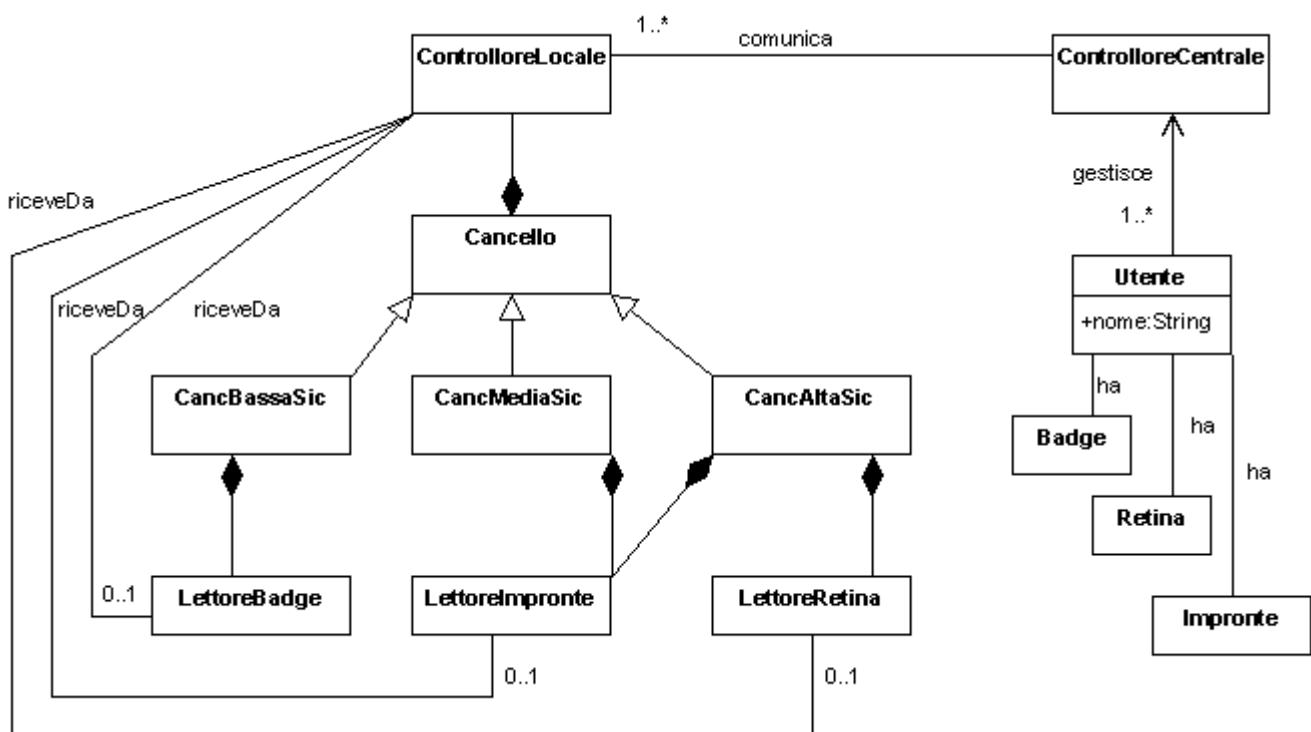
$$\text{AF(c)} = \text{segno} * \prod_{0 \leq i < \text{base.length}} \text{base}[i]^{\text{exp}[i]}$$

Esercizio 3 (punti 8)

1. Si descriva mediante un class diagram in UML i dati utilizzati dal seguente sistema di controllo degli accessi a un edificio.

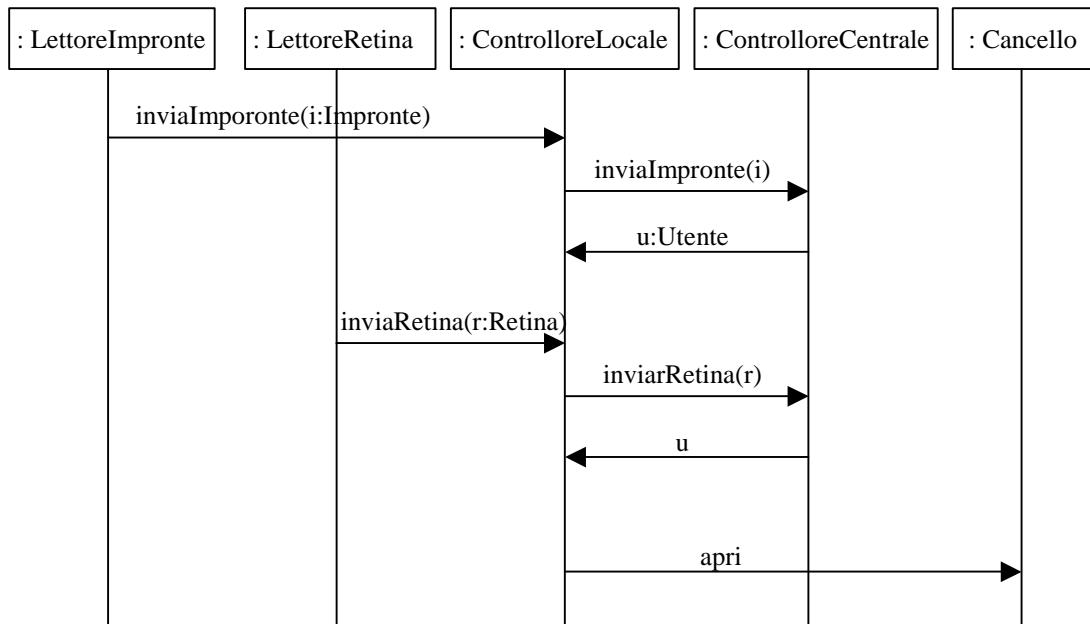
Il sistema si compone di un controllore centrale e di una serie di cancelli agli accessi dell'edificio. Il controllore centrale mantiene anche un database con i dati degli utenti che possono accedere all'edificio. Ci sono 3 tipi di cancelli: a bassa, media, ed alta sicurezza. I cancelli a bassa sicurezza verificano l'identità degli utenti solo mediante un lettore di badge. I cancelli a media sicurezza, invece, verificano l'identità mediante un lettore di impronte digitali. Infine, i cancelli ad alta sicurezza verificano l'identità sia mediante un lettore di impronte digitali, che mediante un lettore di retina.

Ogni cancello ha un controllore locale, il quale riceve i dati dai vari lettori e comunica con il controllore centrale per verificare l'identità degli utenti. Ogni utente e' caratterizzato da un nome, da un badge, da delle impronte digitali, e dai dati della sua retina.



2. Si descriva mediante un sequence diagram il seguente caso di funzionamento del sistema.

Un utente arriva ad un cancello ad alta sicurezza, e fa leggere prima le impronte digitali, poi la retina all'apposito lettore. Ogni lettore spedisce i dati al controllore locale, il quale li rimanda al controllore centrale, e ne riceve indietro un oggetto con i dati dell'utente corrispondente. Se l'utente ricevuto dal controllore centrale e' lo stesso entrambe le volte, il controllore locale invia un segnale di apertura al cancello.



Ingegneria del Software — SOLUZIONE DEL TEMA 16 luglio 2020

Esercizio 1

Si consideri la seguente specifica di una classe mutabile `Buffer`, che rappresenta un buffer di messaggi. Un messaggio è un elemento della classe `Message`, specificata più avanti. Il buffer permette l'inserimento di un messaggio in ultima posizione e la rimozione del messaggio in prima posizione (spostando di conseguenza tutti i messaggi rimanenti di una posizione). Il buffer può contenere al massimo `maxBytes()` byte.

```
public class Buffer {
    // Costruisce un buffer vuoto che puo' contenere al massimo maxBytes byte
    public Buffer(int maxBytes);

    //@ensures (* \result e' il numero di messaggi nel buffer *)
    public /*@ pure @*/ int size();

    //@ensures (* \result e' il messaggio i-esimo nel buffer se 0<=i<size() *).
    //@signals (IndexOutOfBoundsException e) (i<0 || i>=size());
    public /*@ pure @*/ Message get(int i) throws IndexOutOfBoundsException;

    //@ensures (* \result e' il numero massimo di byte che il buffer puo' contenere *)
    public /*@ pure @*/ int maxBytes();

    //@ensures (* inserisce un nuovo messaggio in ultima posizione *)
    //@signals (NullPointerException e) (* se il messaggio e' null *)
    //@signals (BufferFullException e)
    //@(* se l'inserimento del messaggio farebbe superare il massimo numero di byte *)
    public void push(Message m) throws NullPointerException, BufferFullException;

    //@ensures (* restituisce e rimuove il messaggio contenuto in prima posizione *)
    //@signals (BufferEmptyException e) (* se il buffer e' vuoto *);
    public Message pull() throws BufferEmptyException;
}
```

La classe immutabile `Message` è così specificata:

```
public /*@ pure @*/ class Message {
    // costruisce un messaggio
    public Message(String content);

    //@ restituisce il contenuto del messaggio
    public String getContent();

    //@ restituisce la dimensione (in byte) del messaggio
    public int numBytes();
}
```

Domanda a)

Si scriva la specifica JML del metodo `push`.

Soluzione

```
//@ensures m != null &&
//@(\sum int i; 0<=i && i<\old(size); \old(get(i).numBytes()) + m.getBytes() <= maxBytes() &&
//@\size() == \old(size()) + 1 && get(size()-1) == m &&
//@(\forall int i; 0<=i && i<\old(size()); get(i) == \old(get(i)));
//@

//@signals (NullPointerException e) m == null && size() == \old(size()) &&
//@(\forall int i; 0<=i && i<\old(size()); get(i) == \old(get(i)));
//@
```

```

// @signals (BufferFullException e) m != null &&
// @(\sum int i; 0<=i && i<size(); get(i).numBytes() + m.getBytes() > maxBytes() &&
// @size() == \old(size()) &&
// @(\forall int i; 0<=i && i<\old(size()); get(i) == \old(get(i)));
public void push(Message m) throws NullPointerException, BufferFullException;;

```

Domanda b)

Si scriva la specifica JML del metodo pull.

Soluzione

```

// @ensures size() > 0 && size() == \old(size()-1) &&
// @\result == \old(get(0)) &&
// @(\forall int i; 0<=i && i<size()-1; get(i) == \old(get(i+1)));
// @
// @signals (BufferEmptyException e) \old(size()) == 0 &&
// @size() == \old(size());
public Message pull() throws BufferEmptyException;

```

Domanda c)

Si consideri una variante `NullBuffer` che permette l'inserimento di messaggi nulli (ovvero, il metodo `push` non lancia `NullPointerException`).

È possibile definire `NullBuffer` come derivata da `Buffer`, rispettando il principio di sostituzione? Il viceversa? Motivare la risposta.

Soluzione

Come noto, Java permette di ridefinire un metodo rimuovendo eccezioni, ma non aggiungendone nel caso di eccezioni checked. Quindi è sintatticamente possibile aggiungere la `NullPointerException` (unchecked). Quindi dal punto di vista puramente sintattico sarebbe corretto sia che `NullBuffer` estenda `Buffer` che il viceversa.

Tuttavia, dal punto di vista del modello di Liskov, la postcondizione eccezionale è parte della post-condizione complessiva del metodo.

Quindi nessuna delle due classi può ereditare dall'altra senza violare la regola dei metodi: nella stessa situazione (`m==null`) la `push` lancia un'eccezione oppure no a seconda del tipo dinamico dell'oggetto, quindi le due postcondizioni non si possono implicare né in un senso né nell'altro.

Si noti che anche ignorando la questione del lancio di un'eccezione la regola è violata::

`Buffer.push(null)` non inserisce nulla e non cambia la dimensione del buffer, mentre `Buffer.push(null)` inserisce un valore e cambia quindi la dimensione del buffer: ancora le due postcondizioni non si possono implicare né in un senso né nell'altro.

Esercizio 2

Si consideri la seguente implementazione di una versione semplificata della classe `Buffer` presentata nell'Esercizio 1, che contiene i soli metodi pubblici `push` e `pull`.

```
public class Buffer {
    private final List<Message> messages;
    private final int maxBytes;

    public Buffer(int maxBytes) {
        messages = new ArrayList<>();
        this.maxBytes = maxBytes;
    }

    public void push(Message m) throws NullPointerException, BufferFullException {
        if (m==null) throw new NullPointerException();
        if (currentBytes() + m.numBytes() > maxBytes) throw new BufferFullException();
        messages.add(m);
    }

    public Message pull() throws EmptyBufferException {
        if (messages.isEmpty()) throw new EmptyBufferException();
        return messages.remove(0);
    }

    private int currentBytes() {
        int sum = 0;
        for (final Message msg : messages) sum += msg.numBytes();
        return sum;
    }
}
```

Domanda a)

Modificare l'implementazione precedente per fare in modo che, in caso di buffer pieno, il metodo `push` non lanci una `BufferFullException` ma invece sospenda il thread chiamante in attesa che ci sia spazio disponibile.

Soluzione

```
public class Buffer {
    ...

    public synchronized void push(Message m) throws NullPointerException {
        if (m==null) throw new NullPointerException();
        while (currentBytes() + m.numBytes() > maxBytes) {
            try {
                wait();
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
        messages.add(m);
    }

    public synchronized Message pull() throws BufferEmptyException {
        if (messages.isEmpty()) {
            throw new BufferEmptyException();
        }
        notifyAll();
        return messages.remove(0);
    }

    ...
}
```

Domanda b)

Modificare ulteriormente l'implementazione per fare in modo che anche il metodo `pull` non lanci una `BufferEmptyException` in caso di buffer vuoto, ma sospenda il chiamante in attesa di messaggi.

Soluzione

```
public class Buffer {
    ...
    public synchronized void push(Message m) throws NullPointerException {
        if (m==null) throw new NullPointerException();
        while (currentBytes() + m.numBytes() > maxBytes) {
            try {
                wait();
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
        messages.add(m);
        notifyAll();
    }

    public synchronized Message pull() {
        while (messages.isEmpty()) {
            try {
                wait();
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
        notifyAll();
        return messages.remove(0);
    }
    ...
}
```

Esercizio 3

Si consideri il seguente metodo statico:

```
1 static int mystery(int u, int v) {
2     if (u<0 && v!=0) {
3         while (v>0) {
4             if (v<2)
5                 u+=v;
6             else u--;
7             v--;
8         }
9     }
10    return u;
```

Domanda a)

Si determini un insieme minimo di test per garantire la copertura delle decisioni (edge coverage).

Soluzione

Il primo `if` deve essere valutato positivamente e negativamente. Quando si entra nel primo `if` con $v \geq 2$ si valutano le successive decisioni sia positivamente che negativamente.

Bastano quindi i due casi: p.es. $(u = -1, v = 2)$ e $(u = 0, v \text{ qualunque})$.

Il primo caso permette di entrare nel `while`: alla prima iterazione si entra nel ramo `else`, alla seconda nel ramo `if`, infine si esce dal `while`.

Il secondo caso non fa entrare nel primo `if`.

I due casi permettono quindi di coprire tutte le decisioni.

Domanda b)

Si determini un insieme minimo di test per garantire la copertura delle decisioni e delle condizioni (edge and condition coverage).

Soluzione

Le uniche condizioni composte sono nel primo `if`. Servono quindi 3 casi. Vanno bene i due precedenti, a cui aggiungere un caso in cui $u < 0$ ma $v \neq 0$ è falsa, ossia $(u = -1, v = 0)$.

Domanda c)

Si determini la condizione logica sulle variabili u e v ("path condition") che permetta di percorrere il seguente cammino:

1 2 3 4 6 7 4 6 7 4 6 7 4 5 7 8 9 10

Sintetizzare quindi un dato di test che percorra lo stesso cammino.

Soluzione

Il valore di v deve essere 4 per potere percorrere il ciclo 4 volte (tre volte prendendo il ramo `then` dell'`if`, ossia 4 6 7 4 6 7 4 6 7, e una volta il ramo `else`, quando $v = 1$, ossia 4 5 7). Per u è sufficiente un valore negativo. La path condition quindi è: $u < 0 \& \& v == 4$.

Come dato di test, basta ad es. $(u = -1, v = 4)$.

18 febbraio 2019



Politecnico di Milano

Anno accademico 2018-2019

Ingegneria del Software

Cognome:	LAUREANDO <input type="checkbox"/>
Nome:	Matricola:
Sezione (segnarne una): <input type="checkbox"/> Cugola	<input type="checkbox"/> Margara
	<input type="checkbox"/> San Pietro

Istruzioni

1. La mancata indicazione dei dati anagrafici e della sezione comporta l'annullamento del compito.
2. Al termine, consegnare solo i fogli distribuiti utilizzando il retro delle pagine in caso di necessità. Non separare questi fogli. Eventuali fogli di brutta, ecc. non verranno in nessun caso presi in considerazione. È possibile scrivere in matita.
3. È possibile consultare liberamente libri, manuali o appunti. È proibito l'uso di ogni dispositivo elettronico (quali calcolatrici tascabili, telefoni cellulari, smartwatch, ...).
4. Non è possibile lasciare l'aula conservando il tema della prova in corso.
5. La consegna di un compito gravemente insufficiente comporta l'impossibilità a partecipare all'appello successivo.
6. Tempo a disposizione: 2h.

Esercizio 1:

Esercizio 2:

Esercizio 3:

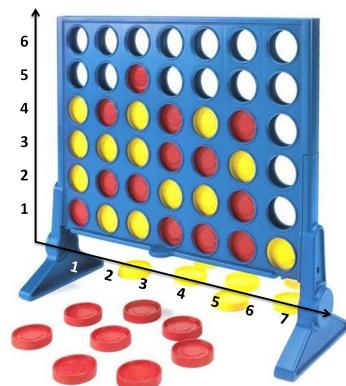
Esercizio 4:

Esercizio 1

Si consideri il tipo di dato astratto Connect4 che modella una partita al noto gioco “Forza 4” (vedi figura a lato).

Lo scopo del gioco è allineare quattro pedine dello stesso colore nella scacchiera di gioco sia in verticale, orizzontale o obliqua. A questo scopo due giocatori (uno dotato di 21 pedine gialle, l’altro di 21 pedine rosse) si alternano facendo scivolare la pedina nei binari prestabiliti dalla scacchiera.

Si supponga una dimensione della “griglia” come in figura (7 colonne per 6 righe). Una pedina viene depositata invocando il metodo `put` ed indicando la colonna in cui si vuole depositare la pedina (nell’intervallo 1–7) e l’identificativo del giocatore. I due giocatori sono convenzionalmente identificati dai valori interi 1 e 2. Il metodo `get` ritorna 0 se la cella indicata è vuota, oppure 1 o 2 a seconda della pedina presente nella posizione indicata (del giocatore 1 o del giocatore 2). La cella in basso a sinistra ha posizione 1,1 (vedi figura).



```
public class Connect4 {  
    // Crea una partita vuota.  
    public Connect4() ;  
  
    // Deposita la pedina del giocatore player (1 o 2) nella colonna indicata.  
    // Restituisce l’eccezione FullColumnException se la colonna è piena.  
    // Restituisce l’eccezione GameOverException se c’è già un vincitore.  
    public void put(int column, int player) throws FullColumnException, GameOverException;  
  
    // Restituisce 0 se la cella è libera, 1 o 2 se è occupata dalla pedina  
    // del giocatore corrispondente. La posizione indicata deve appartenere  
    // all’intervallo ammesso (1-7 x 1-6). La cella in basso a sinistra (vedi figura)  
    // ha posizione 1,1.  
    public /*@ pure @*/ int get(int column, int row) ;  
  
    // Restituisce true sse c’è un vincitore (4 pedine dello stesso giocatore allineate).  
    public /*@ pure @*/ boolean winner() ;  
  
    // Svuota la griglia dando inizio a una nuova partita.  
    public void clear() ;  
}
```

Domanda a)

Si specifichino in JML i metodi `put`, `winner` e `clear`.

Soluzione

Per semplicità (pur non essendo necessario) definiamo un’abbreviazione:

```
unchanged(row, column) =  
(\forallall int c; c>=1 && c<=7;  
 (\forallall int r; r>=1 && r<=6;  
  get(c, r)==\old(get(c, r)) || (r==row && c==column)))
```

che afferma che ogni cella diversa da (row,column) non è modificata. Una simile abbreviazione `unchanged` afferma che nessuna cella è modificata.

```
//@requires column>=1 && column<=7 and (player==1 || player==2)  
//@ensures \old(get(column, 6)==0) && \old(!winner()) &&  
//@ (\existsint int row; row>=1 && row<=6; \old(get(column, row)==0) && get(column, row)==player &&  
//@ (\forallall int r; r>=1 && r<row; \old(get(column, r)!=0)) &&
```

```

//@      unchanged(row,column)
//@signals(FullColumnException e) (\forall int r; r>=1 && r<=6; \old(get(column,r)!=0)) &&
//@                      unchanged;
//@signals(GameOverException e) \old(winner()) && unchanged;
public void put(int column, int player) throws FullColumnException, GameOverException;

//@ensures \result <=>
//@  (\exists int c; c>=1 && c<=7;
//@    (\exists int r; r>=1 && r<=6;
//@      (\exists int dx; dx>=-1 && dx<=1;
//@        (\exists int dy; dy>=-1 && dy<=1;
//@          !(dx==0 && dy==0) && c+3*dx>=1 && c+3*dx<=7 && r+3*dy>=1 && r+3*dy<=7 &&
//@            get(c,r)==get(c+dx,r+dy)==get(c+2*dx,r+2*dy)==get(c+3*dx,r+3*dy) &&
//@            get(c,r)!=0)))))
public /*@ pure @*/ boolean winner() ;

{@ensures (\forall int c; c>=1 && c<=7; (\forall int r; r>=1 && r<=6; get(c,r)==0));
public void clear() ;

```

Domanda b)

Si specifichi in JML l'invariante pubblico del tipo Connect4.

Soluzione

```

//@ public invariant (\forall int c; c>=1 && c<=7; (\forall int r; r>=1 && r<=6;
//@      ( get(c,r)==0 || get(c,r)==1 || get(c,r)==2 ) &&
//@      ( get(c,r)!=0 => (\forall int r1; r1>=1 && r1<=r; get(c,r1)!=0))))
```

Domanda c)

Si consideri il tipo Space4 che modella una variante del gioco Connect4 seconda la quale il metodo put posiziona la pedina del giocatore indicato non “in cima” alla colonna ma in una posizione libera casuale (della colonna indicata).

Il tipo Space4 può essere un sottotipo di Connect4 secondo il principio di sostituibilità di Liskov? Motivare la propria risposta. Indicare poi (motivando la risposta) se sia possibile il contrario.

Soluzione

Il tipo Space4 non può essere un sottotipo di Connect4 in quanto ne violerebbe l'invariante pubblico che stabilisce che le colonne siano riempite dal basso (senza spazi vuoti). Verrebbe violata anche la postcondizione del metodo put.

Il contrario è invece possibile poichè l'ordine con il quale le pedine vengono fatte “cadere” nel gioco originale rappresenta un caso particolare dell'ordine arbitrario (non deterministico) previsto per il gioco Space4.

Esercizio 2

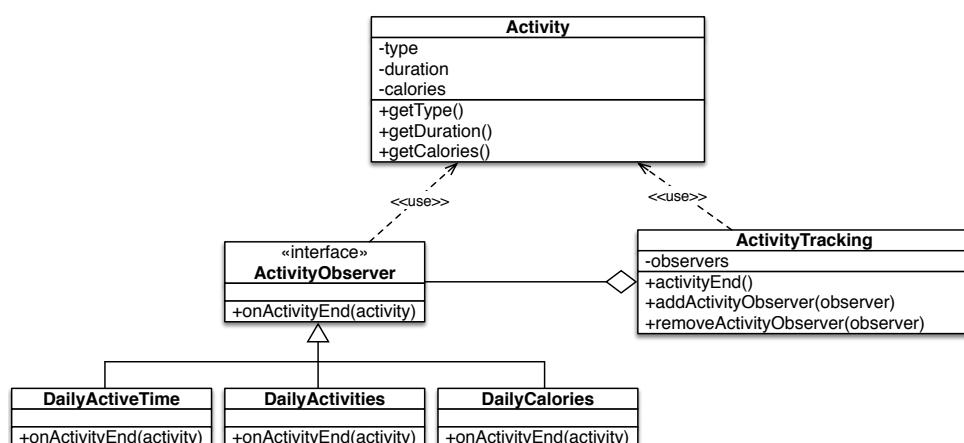
Un'app per smartwatch contiene un componente ActivityTracking che registra le attività dell'utente. Una attività (classe Activity) è caratterizzata da un tipo (nuoto, ciclismo, corsa, ...), una durata in minuti e una stima delle calorie bruciate. Il componente ActivityTracking contiene, tra gli altri, un metodo per terminare l'attività corrente. Quando un'attività termina, diversi altri componenti devono essere aggiornati: il componente DailyCalories, che salva il numero di calorie bruciate durante la giornata, il componente DailyActivities, che salva la lista dei tipi di attività completate durante la giornata, e il componente DailyActiveTime, che salva il numero totale di minuti di attività durante la giornata.

Si disegni una possibile architettura per questo sistema usando un diagramma delle classi UML e si fornisca una bozza di implementazione Java per le classi principali. Nello scrivere il codice Java si assuma che la classe Activity sia già implementata e si mostri l'implementazione della classe ActivityTracker, di una tra DailyCalories, DailyActivities e DailyActiveTime, e di ogni altra classe che possa essere rilevante.

Si favoriscano soluzioni che permettono future modifiche ed estensioni, considerando in particolare il caso in cui l'insieme dei componenti che devono essere aggiornati al termine di un'attività possa cambiare in futuro.

Soluzione

La soluzione consiste nell'adottare il pattern observer.



```
public class Activity {
    private final String type;
    private final int duration;
    private final int calories;

    public Activity(String type, int duration, int calories) {
        this.type = type;
        this.duration = duration;
        this.calories = calories;
    }

    public final String getType() { return type; }
    public final int getDuration() { return duration; }
    public final int getCalories() { return calories; }
}

public interface ActivityObserver {
    public void onActivityEnd(Activity activity);
}

public class ActivityTracking {
    private final List<ActivityObserver> observers = new ArrayList<>();

    public void addActivityObserver(ActivityObserver observer) {
        observers.add(observer);
    }

    public void removeActivityObserver(ActivityObserver observer) {
        observers.remove(observer);
    }

    public void activityEnd() {
        final Activity activity = new Activity(...);
        for (final ActivityObserver observer : observers) {
            observer.onActivityEnd(activity);
        }
    }
    ...
}

public class DailyActiveTime implements ActivityObserver {
    private int dailyActiveTime;

    @Override
    public void onActivityEnd(Activity activity) {
        dailyActiveTime += activity.getDuration();
    }
    ...
}

public class DailyActivities implements ActivityObserver {
    private final Set<String> dailyActivities = new HashSet<>();

    @Override
    public void onActivityEnd(Activity activity) {
        dailyActivities.add(activity.getType());
    }
    ...
}

public class DailyCalories implements ActivityObserver {
```

```

private int dailyCalories;

@Override
public void onActivityEnd(Activity activity) {
    dailyCalories += activity.getCalories();
}
...
}

```

Esercizio 3

In un'app per smartwatch, la classe Stats salva statistiche relative ad attività e salute dell'utente. In particolare, la classe contiene una lista activities con le attività completate e una lista nutrients con le sostanze nutritive assunte.

Domanda a)

Si implementino i seguenti metodi della classe Stats:

- void addActivity(Activity activity): aggiunge activity alla lista activities;
- void addNutrient(Nutrient nutrient): aggiunge nutrient alla lista nutrients;
- List<Activity> getActivities(): ritorna una copia della lista activities;
- List<Nutrient> getNutrients(): ritorna una copia della lista nutrients;

Si assuma che diversi thread possono invocare i metodi elencati sopra in maniera concorrente. Si favoriscano soluzioni che massimizzino le possibilità di esecuzione parallela: per esempio, due thread dovrebbero essere in grado di aggiungere una activity e un nutrient senza interferire tra loro.

Soluzione

Possiamo sincronizzare i metodi che accedono ad activities sull'oggetto activities e i metodi che accedono a nutrients sull'oggetto nutrients.

```

public class Stats {
    private final List<Activity> activities;
    private final List<Nutrient> nutrients;

    public Stats() {
        activities = new ArrayList<>();
        nutrients = new ArrayList<>();
    }

    public void addActivity(Activity activity) {
        synchronized (activities) { activities.add(activity); }
    }

    public void addNutrient(Nutrient nutrient) {
        synchronized (nutrients) { nutrients.add(nutrient); }
    }

    public List<Activity> getActivities() {
        final List<Activity> result = new ArrayList<>();
        synchronized (activities) { result.addAll(activities); }
        return result;
    }
}

```

```

public List<Nutrient> getNutrients() {
    final List<Nutrient> result = new ArrayList<>();
    synchronized (nutrients) { result.addAll(nutrients); }
    return result;
}
}

```

Domanda b)

Si assuma ora che si voglia anche salvare una variabile `caloriesBalance` che registra la somma delle calorie ottenute mediante sostanze nutritive meno la somma di tutte le calorie consumate durante attività. La variabile `caloriesBalance` viene aggiornata quando si aggiunge una sostanza nutritiva o un'attività, e può essere letta mediante il metodo `int getCaloriesBalance()`.

Si modifichi l'implementazione della classe `Stats` per soddisfare i nuovi requisiti. Si assume che sia la classe `Nutrient` che la classe `Activity` forniscano un metodo `getCalories()` che fornisce le calorie fornite dalla sostanza nutritiva o consumate dall'attività.

Soluzione

Ora i metodi `addActivity()` e `addNutrient()` aggiornano una variabile condivisa. Di conseguenza, tutti i metodi devono essere sincronizzati sullo stesso oggetto (ad esempio `this`).

```

public class Stats {
    private final List<Activity> activities;
    private final List<Nutrient> nutrients;
    private int caloriesBalance;

    public Stats() {
        activities = new ArrayList<>();
        nutrients = new ArrayList<>();
        caloriesBalance = 0;
    }

    public synchronized void addActivity(Activity activity) {
        activities.add(activity);
        caloriesBalance -= activity.getCalories();
    }

    public synchronized void addNutrient(Nutrient nutrient) {
        nutrients.add(nutrient);
        caloriesBalance += nutrient.getCalories();
    }

    public synchronized List<Activity> getActivities() {
        return new ArrayList<>(activities);
    }

    public synchronized List<Nutrient> getNutrients() {
        return new ArrayList<>(nutrients);
    }

    public synchronized int getCaloriesBalance() {
        return caloriesBalance;
    }
}

```

Esercizio 4

Si consideri il seguente metodo statico, che ha la precondizione che ciascun elemento di `nums` é una stringa corrispondente a una rappresentazione testuale di un numero intero.

```
public static List<Integer> addX(List<String> nums, int x) {  
    List<Integer> plusX = new LinkedList<>();  
    for(String numString : nums) {  
        number = Integer.valueOf(numString);  
        if (number>0)  
            plusX.add(number + x);  
    }  
    return plusX;  
}
```

Domanda a)

Si definisca un insieme minimale di test che copre tutte le condizioni e diramazioni (branches and conditions) del codice dato.

Soluzione

```
nums = {"1", "-1"}, x=1
```

Domanda b)

Quanti sono i cammini nel codice se il valore del parametro `nums` é una lista di 10 elementi che contiene esattamente 2 stringhe rappresentanti due interi positivi?

Soluzione

$9 + 8 + 7 + \dots + 1 = \frac{9*10}{2} = 45$. (ci sono 9 possibili cammini quando l'elemento in posizione 0 é positivo e un elemento fra le posizioni da 1 a 9 é positivo; altri 8 cammini quando l'elemento in posizione 0 é positivo e un elemento fra le posizioni da 1 a 9 é positivo, ecc.)

Domanda c)

Si riscriva il metodo utilizzando i costrutti della programmazione funzionale di Java 8.

Soluzione

```
public static List<Integer> addX(List<String> nums, int x) {  
    return nums.stream()  
        .map(Integer::valueOf);  
        .filter(number -> number>0)  
        .map(number -> number + x)  
        .collect(Collectors.toList());  
}
```

Ingegneria del Software — Soluzione del Tema 07/06/2023

Esercizio 1

Si consideri la seguente classe `LastHundredPosts` che memorizza i post e i commenti di un nuovo social network, che mantiene solo gli ultimi 100 post di ogni utente e solo i commenti relativi a tali post.

```
public class LastHundredPosts {
    // Ritorna l'insieme degli utenti del social network.
    public /*@ pure @*/ Set<User> getUsers() {...}

    // Ritorna la lista dei post effettuati dall'utente passato per parametro.
    // La lista contiene i post in ordine inverso di inserimento: dal piu' recente al meno recente.
    // Lancia una UnknownUserException se l'utente non e' parte del social network.
    public /*@ pure @*/ List<Post> getPosts(User user) throws UnknownUserException {...}

    // Aggiunge un post avente come autore l'utente passato per parametro. Se l'utente ha gia' effettuato
    // almeno 100 post, cancella il post meno recente tra quelli attualmente salvati per quell'utente.
    // Lancia una UnknownUserException se l'utente non e' parte del social network.
    public void addPost(Post post, User user) throws UnknownUserException {...}

    // Aggiunge un commento al post passato per parametro e avente come autore l'utente passato per parametro.
    // Lancia una UnknownUserException se l'utente non e' parte del social network.
    // Lancia una UnknownPostException se il post non e' presente nel social network.
    public void addComment(Comment comment, Post post, User user)
        throws UnknownUserException, UnknownPostException {...}

    // Ritorna i 10 commenti (di qualunque utente e associati a qualunque post) che hanno ricevuto piu' like.
    // I commenti sono ritornati in ordine inverso di numero di like (prima quelli con piu' like).
    // In caso di parita' l'ordine non e' specificato. In ogni caso, la lista contiene al piu'
    // 10 commenti (o meno, se nel social network non sono presenti almeno 10 commenti).
    public /*@ pure @*/ List<Comment> mostLikedComments() {...}
}

public class Post {
    // Ritorna i commenti associati a questo post.
    public /*@ pure @*/ List<Comment> getComments() {...}
    ...
}

public class Comment {
    // Ritorna il numero di like ricevuti da questo commento.
    public /*@ pure @*/ int numLikes() {...}
    ...
}
```

Domanda a)

Specificare in JML il metodo `addPost`.

Soluzione

Usiamo come soli metodi di base per specificare tutti gli altri `getUsers` e `getPosts`.

Occorre specificare che l'insieme degli utenti non cambia e i post degli altri utenti non cambiano (e non cambiano posizione). Il post viene inserito in prima posizione tra quelli dell'autore e tutti gli altri post si spostano di una posizione. Se la lista conteneva 100 elementi, l'ultimo elemento viene eliminato. In caso di eccezione, l'elenco degli utenti e dei post non cambia.

```
//@requires post!=null && user!=null
//@
//@ensures \old(getUsers()).contains(user) && getUsers().size() == \old(getUsers()).size() &&
//@ \old(getUsers()).containsAll(getUsers()) &&
//@ (\forall User u; getUsers().contains(u) && !u.equals(user));
//@     getPosts(u).size() == \old(getPosts(u)).size() &&
//@     (\forallall int i; i>0 && i<getPosts(u).size());
//@         getPosts(u).get(i) == \old(getPosts(u)).get(i) ) ) &&
//@ getPosts(user).size() == (\old(getPosts(user)).size() == 100) ? 100 : \old(getPosts(user)).size() + 1 &&
//@ getPosts(user).get(0) == post &&
//@ (\forallall int i; i>0 && i<getPosts(user).size());
//@     getPosts(user).get(i) == \old(getPosts(user)).get(i-1) )
//@

//@signals (UnknownUserException) !\old(getUsers()).contains(user) &&
//@ getUsers().size() == \old(getUsers()).size() && \old(getUsers()).containsAll(getUsers()) &&
//@ (\forall User u; getUsers().contains(u);
//@     getPosts(u).size() == \old(getPosts(u)).size() &&
```

```

//@      (\forall int i; i>0 && i<getPosts(u).size());
//@      getPosts(u).get(i)==\old(getPosts(u)).get(i) ) ) &&
public void addPost(Post post, User user) throws UnknownUserException {...}

```

Domanda b)

Specificare in JML il metodo mostLikedComments.

Soluzione

Il metodo è puro, quindi non occorre specificare che nulla cambia durante la sua esecuzione e non è necessario usare `old` per riferirsi allo stato precedente all'esecuzione del metodo.

Occorre specificare che tutti i commenti ritornati sono commenti relativi a qualche post presente nel social network. Se esistono almeno 10 commenti, allora la lista ritornata ha dimensione 10, altrimenti la lista contiene tutti i commenti presenti nel social network. I commenti sono in ordine inverso di numero di like. Per ogni posizione i della lista ritornata, non devono esistere commenti con un numero di like più alto che sono presenti nel social network ma non sono presenti nella lista in una posizione precedente a i . La lista ritornata non deve contenere duplicati.

Definiamo `numComments` come il numero totale di commenti presenti nel social network. Definiamo `commentInSocial` come un predicato che vale `true` se e solo se un commento è contenuto nel social network (ovvero è il commento di qualche post presente nel social network).

```

numComments == (\sum User u; getUsers().contains(u);
    (\sum Post p; getPosts(u).contains(p); p.getComments().size()) )

commentInSocial(Comment c) <==> (\exists User u; getUsers().contains(u);
    (\exists Post p; getPosts(u).contains(p); p.getComments().contains(c)) )

//@ensures (\forall Comment c; \result.contains(c); commentInSocial(c) &&
//@ \result.size() == (numComments<10) ? numComments : 10 &&
//@ (\forall int i; i>0 && i<\result.size();
//@     \result.get(i).numLikes()<=\result.get(i-1).numLikes() &&
//@     (\forall Comment c; commentInSocial(c) && c.numLikes()>\result.get(i).numLikes());
//@     \result.indexOf(c)>=0 && \result.indexOf(c)<i) ) &&
//@ (\forall int i; i>0 && i<\result.size()-1;
//@     (\forall int j; j>i && j<\result.size();
//@         ! \result.get(i).equals(\result.get(j))) )
public /*@ pure @*/ List<Comment> mostLikedComments() {...}

```

Domanda c)

Si consideri una implementazione della classe `LastHundredPosts` che utilizza una mappa per memorizzare i dati di ogni utente. Per ogni utente, la mappa contiene una lista con tutti i post di quell'utente.

```

public class LastHundredPosts {
    private Map<User, List<Post>> posts;
    ...
}

```

Per questa implementazione scrivere l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

L'invariante di rappresentazione deve dire che la mappa non è nulla, che non contiene chiavi o valori nulli, che nessuna lista contiene valori nulli o ripetuti, che nessuna lista contiene più di 100 post.

```

//@private invariant
//@ posts!=null && !posts.containsKey(null) && !posts.containsValue(null) &&
//@ (\forall User u; posts.containsKey(u); !posts.get(u).contains(null) &&
//@     posts.get(u).size()<=100 && (\forall int i; i>0 && i<posts.get(u).size();
//@         (\forall int j; j>0 && j<i; !posts.get(u).get(i).equals(posts.get(u).get(j))) ) ) )

```

La funzione di astrazione “mappa” gli attributi privati sui metodi base usati per la specifica: `getUsers` e `getPosts`.

```

//@private invariant
//@ getUsers().size()==posts.keySet().size() && getUsers().containsAll(posts.keySet()) &&
//@ (\forall User u; posts.containsKey(u); getPosts(u).size()==posts.get(u).size() &&
//@     (\forall int i; i>0 && i<getPosts(u).size(); getPosts(u).get(i)==posts.get(u).get(i)) )

```

Domanda d)

Si consideri una classe `LastHundredPostsOrdered` che ridefinisce il metodo `mostLikedComments` per fare in modo che, in caso di parità nel numero di like di due commenti, sia sempre ritornato prima il commento più recente.

`LastHundredPostsOrdered` può essere definita come sottoclasse di `LastHundredPosts` senza violare il principio di sostituzione di Liskov? Viceversa, `LastHundredPosts` può essere definita come sottoclasse di `LastHundredPostsOrdered`?

Soluzione

La classe `LastHundredPostsOrdered` può essere definita come sottoclasse di `LastHundredPosts` in quanto aggiunge un vincolo (rafforza) la post-condizione del metodo `mostLikedComments`, senza alterare le condizioni di altri metodi o le proprietà invarianti ed evolutive della classe. Viceversa, `LastHundredPosts` non può essere definita come sottoclasse di `LastHundredPostsOrdered` in quanto, togliendo l'ordine nei commenti, indebolirebbe la post-condizione del metodo `mostLikedComments`.

Esercizio 2

Si consideri la seguente classe DataBuffer che contiene dati numerici che possono essere scritti e letti a partire da una particolare posizione detta “cursoro”.

```
public class DataBuffer {
    private double[] data;
    private int cursor;

    public DataBuffer() { data = new double[100]; cursor = 0; }

    public double getDataAtCursor() {
        synchronized(data) { return data[cursor]; }
    }

    public void setDataAtCursor(double d) {
        synchronized(data) { data[cursor] = d; }
    }

    public void setCursorPos(int pos) {
        synchronized(cursor) { cursor = pos; }
    }
}
```

Domanda a)

Il codice è correttamente sincronizzato? Motivare la propria risposta e in caso di risposta negativa modificare un solo metodo della classe affinché la sincronizzazione risulti corretta.

Soluzione

I metodi che accedono all’attributo `cursor` acquisiscono lock su oggetti diversi. Inoltre, il metodo `setCursorPos` acquisisce il lock su un oggetto di un tipo base (`int`), il che non è consentito dal linguaggio. È possibile risolvere entrambi i problemi modificando il metodo come segue, garantendo che tutti i metodi acquisiscano il lock sul medesimo oggetto `data`:

```
public void setCursorPos(int pos) {
    synchronized(data) { cursor = pos; }
}
```

Domanda b)

Si modifichi il metodo `getDataAtCursor` in modo da sospendere il chiamante se il dato da ritornare risultasse minore di zero. Il chiamante verrà sbloccato quando il dato diventerà positivo.

Soluzione

```
public double getDataAtCursor1() throws InterruptedException {
    synchronized(data) {
        while(data[cursor]<0) data.wait();
        return data[cursor];
    }
}
```

Si devono anche modificare i metodi `setDataAtCursor` e `setCursorPos` come segue:

```
public void setDataAtCursor(double d) {
    synchronized(data) {
        data[cursor] = d;
        if(d>=0) data.notifyAll();
    }
}

public void setCursorPos(int pos) {
    synchronized(data) {
        cursor = pos;
        if(data[pos]>=0) data.notifyAll();
    }
}
```

Domanda c)

Si consideri il seguente metodo della classe `DataBuffer` che crea due thread e li esegue:

```

public void change() {
    new Thread(() -> {
        for(int i=0; i<100; i++) {
            setCursorPos(i);
            setDataAtCursor(i);
        }
    }).start();

    new Thread(() -> {
        for(int i=0; i<100; i++) {
            setCursorPos(i);
            setDataAtCursor(i*2);
        }
    }).start();
}

```

È corretto affermare che al termine dell'esecuzione dei due thread il contenuto dell'array `data` contiene in ogni posizione `i` il valore `i` oppure il valore `i*2`? Motivare la propria risposta e in caso di risposta negativa modificare i thread affinché la proprietà sia rispettata.

Soluzione

La proprietà non è rispettata perché uno dei due thread potrebbe impostare il cursore a una certa posizione per poi, prima di impostare il dato nella posizione del cursore, cedere il controllo all'altro thread che potrebbe a sua volta impostare il dato con un valore che non sarà più "sincronizzato" con la posizione del cursore. Il codice che segue risolve il problema:

```

public void change() {
    new Thread(() -> {
        for(int i=0; i<100; i++) {
            synchronized(data) {
                setCursorPos(i);
                setDataAtCursor(i);
            }
        }
    }).start();

    new Thread(() -> {
        for(int i=0; i<100; i++) {
            synchronized(data) {
                setCursorPos(i);
                setDataAtCursor(i*2);
            }
        }
    }).start();
}

```

Esercizio 3

Si consideri il seguente metodo:

```
1 public static int doSomething(int x, int y) {
2     if (x>=0 && y>1) {
3         while (y>0) {
4             if (y<3) {
5                 x+=y;
6             }
7             if (y>3 && x>0) {
8                 x--;
9             }
10            y--;
11        }
12    }
13    return x;
14 }
```

Domanda a)

Si determini un insieme minimo di casi di test per garantire la copertura delle istruzioni (statement coverage).

Soluzione

Il caso (1,4) è sufficiente: il loop viene eseguito quattro volte, alla prima iterazione copre l'istruzione 8, nelle terza (e nella quarta) l'istruzione 5.

Domanda b)

Si determini un insieme minimo di test per garantire la copertura delle decisioni (edge coverage).

Soluzione

Il caso precedente, durante l'esecuzione di diverse iterazioni del loop, copre, oltre alle decisioni vere degli if interni al ciclo, anche le decisioni false dei medesimi if. Alla prima iterazione copre il ramo else "nascosto" del primo if interno, alla seconda iterazione il ramo else "nascosto" del secondo if interno.

Manca soltanto la copertura della decisione false dell'if esterno al loop: basta aggiungere un caso di test, ad esempio (1, 1).

Domanda c)

Si determini un insieme minimo di test per garantire la copertura delle decisioni e delle condizioni (edge and condition coverage).

Soluzione

Usando i casi precedenti manca ancora la copertura di due condizioni:

Nel primo if, la condizione per cui $x \geq 0$ è falsa, quindi ad esempio (-1, qualunque).

Nel terzo if, la condizione per cui $y > 3$ è vera con $x > 0$ è falsa: sostituendo il caso (1, 4) con il caso (1, 5) si riesce a coprire anche questa condizione.

In tutto, sono quindi sufficienti 3 casi: (-1, 0), (1, 1), (1, 5).

Esercizio 4

Si considerino le seguenti dichiarazioni di classi Java.

```
1 abstract class Document {
2     public abstract void addTo(Channel c);
3     public void addTo(PublicChannel c) { System.out.println("Doc added to " + c); }
4 }
5 class Music extends Document {
6     public void addTo(Channel c) { System.out.println("Music added to " + c); }
7 }
8 class Podcast extends Document {
9     public void addTo(Channel c) { System.out.println("Podcast added to " + c); }
10    public void addTo(PublicChannel c) { System.out.println("Public Podcast added to " + c); }
11 }
12 class Channel {
13     public void insert(Document p) { p.addTo(this); }
14     public String toString() { return "a channel"; }
15 }
16 class PublicChannel extends Channel {
17     public String toString() { return "public Channel"; }
18 }
19
20 class Main {
21     public static void main(String[] args) {
22         Vehicle car = new Car();
23         HybridCar hybridCar = new HybridCar();
24         Car electricCar = new ElectricCar();
25
26         car.start();
27         car.start(true);
28         car.start(false);
29
30         hybridCar.start();
31         hybridCar.start(true);
32         hybridCar.start(false);
33
34         electricCar.start();
35         electricCar.start(true);
36         electricCar.start(false);
37
38
39     }
40 }
```

Domanda a)

Ci sono linee del programma non valide? In caso affermativo indicarle, motivando brevemente la risposta e precisando se causano errori a tempo di compilazione o di esecuzione.

Soluzione

Riga 3 genera errore di compilazione (non si può istanziare una classe astratta). Anche riga 7 genera errore di compilazione (Music non è sottoclasse di Podcast). Di conseguenza sono da eliminare le righe che usano le variabili d1 e m2, ossia le righe 11 e 18.

Domanda b)

Eliminate le righe che generano gli errori individuati al punto precedente, indicare cosa stampa il programma, motivando brevemente la risposta. Riportare sempre il numero di riga.

Soluzione

12. Podcast added to a channel
13. Music added to public Channel
14. Podcast added to public Channel
15. Music added to public Channel

16. Podcast added to public Channel

17. Music added to public Channel

Ingegneria del Software — Soluzione del Tema 10/07/2023

Esercizio 1

Si consideri la seguente classe immutabile Message. Ogni messaggio ha un contenuto, un mittente, un destinatario e un timestamp. Mittente e destinatario sono individuati univocamente da un identificatore, per semplicità un long. Ancora per semplicità, il timestamp è un long (anziché un Date).

```
public /*@ pure @*/ class Message {
    public Message(byte[] content, long recipientID, long senderID, long timestamp) {...}
    public byte[] content() {...}      // returns the content of this
    public long recipientID() {...}   // returns the recipientID of this
    public long senderID() {...}     // returns the senderID of this
    public long timestamp() {...}    // returns the timestamp of this
}
```

La seguente classe MessageManager descrive un gestore di messaggi. È possibile inviare ed estrarre messaggi, vedere elenchi dei messaggi e degli utenti; in particolare, è possibile fondere un altro gestore messaggi con quello corrente.

```
public class MessageManager {
    // Builds a new empty manager
    public MessageManager() {...}

    // Returns the set of all valid ids, i.e., ids of recipients or senders of some message in the past.
    public /*@ pure @*/ Set<Long> allIds() {...}

    // Returns all messages yet to be delivered to the recipient having the given recipientId,
    // from the oldest to the newest.
    // Returns an empty list if the recipient is unknown or if there is no message to be delivered it.
    public /*@ pure @*/ List<Message> toId(long recipientId) {...}

    // Returns all messages, yet to be delivered, sent from the sender having the given senderId.
    // Throws an UnknownIdException if the sender is unknown.
    public /*@ pure @*/ Set<Message> fromId(long senderId) throws UnknownIdException {...}

    // Inserts a new message in the manager, with the timestamp at the instant of the call.
    public void addMessage(byte[] content, long senderId, long recipientId) {...}

    // Returns and deletes the oldest message added for recipientId and not yet delivered.
    // Throws a NoMessageException if no message is present.
    // Throws an UnknownIdException if the id is unknown.
    public Message deliverTo(long recipientId) throws NoMessageException, UnknownIdException {...}

    // Copies all messages and ids in the manager other to this.
    // The manager other is unchanged.
    public void merge(MessageManager other) {...}
}
```

Domanda a)

Specificare in JML il metodo puro fromId.

Soluzione

```
/*@ ensures allIds().contains(senderId) && \result!=null &&
/*@ (\forall long recipientId; allIds().contains(recipientId);
/*@   (\forall Message m; toId(recipientId).contains(m) && m.senderID()==senderId;
/*@     \result.contains(m) ) ) &&
/*@ (\forall Message m; \result.contains(m);
/*@   m.senderID()==senderId && allIds.contains(m.recipientID()) &&
/*@   toId(m.recipientID()).contains(m) )
/*@

/*@ signals(UnknownIDException e) !allId().contains(senderId);
public /*@ pure @*/ Set<Message> fromId(long senderId) throws UnknownIdException {...}
```

Domanda b)

Specificare in JML metodo merge.

Soluzione

Occorre dire che `other` al termine dell'invocazione è immutato, mentre `this` può cambiare. Al termine dell'invocazione, `allIds` deve contenere tutti e soli gli id precedentemente presenti in `this` oppure in `other`. Le liste dei messaggi ritornata da `toId` devono contenere tutti e soli i messaggi precedentemente ritornati invocando `toId` in `this` o in `other`. Infine, le liste dei messaggi ritornate da `toId` devono essere in ordine di timestamp.

```
/*@ requires other!=null;
/*@ ensures
/*@ (\forall long id; ;
/*@   allIds().contains(id) <=> (\old(allIds()).contains(id) || \old(other.allIds()).contains(id)) ) &&
/*@ (\forall long recipientId; ;
```

```

//@ toId(recipientId).containsAll(\old(toId(recipientId))) &&
//@ toId(recipientId).containsAll(\old(other.toId(recipientId))) &&
//@ toId(recipientId).size() == \old(toId(recipientId)).size() + other.toId(recipientId).size() ) &&
//@ (\forall int i; 0<=i && i<toId(recipientId).size()-1;
//@     toId(recipientId).get(i).timestamp() <= toId(recipientId).get(i+1).timestamp() );
//@ other.allIds().containsAll(\old(other.allIds()) && other.allIds().size()==\old(other.allIds().size()) &&
//@ (\forall long id; 
//@     other.toId(id).size()==\old(other.toId(id).size()) ) &&
//@ (\forall int i; i>=0 && i<other.toId(id).size();
//@     other.toId(id).get(i)==\old(other.toId(id).get(i)) )
public void merge(MessageManager other) {...}

```

Si noti che non è necessario specificare come cambia il metodo `fromId()`, in quanto la sua specifica già lo descrive in funzione degli altri due metodi puri. Si noti inoltre che messaggi inviati tramite gestori diversi potrebbero avere lo stesso timestamp.

Soluzione c)

Si consideri la seguente semplice implementazione di `MessageManager`.

```

public class MessageManager {
    private Set<Long> validIds;
    private List<Message> pendingMessages;
    ...
}

```

I messaggi sono inseriti in ordine di arrivo nella `List`, aggiornando anche `validIds` se necessario. Una volta inviati (mediante invocazioni di `deliverTo`) i messaggi sono cancellati da `pendingMessages`. Scrivere in JML l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

Occorre dichiarare che i due attributi non sono nulli e non contengono valori nulli. Gli id dei sender e dei recipient dei messaggi in `pendingMessages` sono contenuti in `validIds`. Si noti che `validIds` potrebbe contenere degli id a cui non corrispondono messaggi (in quanto la specifica prevede di raccogliere tutti gli id per cui c'è stato un messaggio in passato). Occorre infine specificare che i messaggi in `pendingMessages` non sono duplicati e sono in ordine di timestamp.

RI:

```

//@ private invariant validIds!=null && pendingMessages!=null &&
//@ !validIds.contains(null) && !pendingMessages.contains(null) &&
//@ (\forall Message m; pendingMessages.contains(m);
//@     validIds.contains(m.senderID()) && validIds.contains(m.recipientID()) ) &&
//@ (\forall int i; 0<=i && i<pendingMessages.size()-1;
//@     pendingMessages.get(i).timestamp() <= pendingMessages.get(i+1).timestamp() &&
//@#   (\forall int j; i<j && j<pendingMessages.size();
//@     !pendingMessages.get(i).equals(pendingMessages.get(j)))

```

Per la funzione di astrazione occorre mettere in relazione il rep con i due metodi puri `allIds` e `toId` (il restante metodo puro `fromId` è specificato in funzione di questi).

AF:

```

//@ private invariant validIds.containsAll(allIds()) && validIds.size()==allIds.size() &&
//@ (\forall long recipientId; validIds.contains(recipientId);
//@     (\forall Message m; m.recipientId()==recipientId;
//@         toId(recipientId).contains(m) <=> pendingMessages.contains(m) )
//@     (\forall Message m1; toId(recipientId).contains(m1);
//@         (\forall Message m2; toId(recipientId).contains(m2);
//@             toId(recipientId).indexOf(m1) < toId(recipientId).indexOf(m2) <=>
//@             pendingMessages.indexOf(m1) < pendingMessages.indexOf(m2) ) ) )

```

Solanda d)

Si consideri una classe `ClearableMessageManager` che aggiunge alla classe `MessageManager` il metodo `clear` che cancella tutti gli id e tutti i messaggi pendenti. Può `ClearableMessageManager` essere definita come sottoclasse di `MessageManager` in accordo con il principio di sostituzione di Liskov? Si motivi la risposta.

Soluzione

`ClearableMessageManager` non può essere definita come sottoclasse di `MessageManager` in quanto violerebbe la proprietà evolutiva che richiede che gli id di tutti i mittenti e destinatari di messaggi passati siano conservati (ritornati da future chiamate di `allIds`).

Esercizio 2

Si consideri la seguente classe Point che rappresenta un punto nello spazio cartesiano.

```
public class Point {  
    private double myX;  
    private double myY;  
  
    public synchronized void setX(double x) { myX = x; }  
    public synchronized void setY(double y) { myY = y; }  
  
    public synchronized void setXY(double x, double y) throws InterruptedException {  
        while( (x+y)<0 ) wait();  
        notifyAll();  
        myX = x; myY = y;  
    }  
}
```

Domanda a)

Il metodo setXY è correttamente sincronizzato (ovvero non da luogo a conflitti nell'accesso alla memoria e non genera deadlock)? Motivare la propria risposta.

Soluzione

Il metodo da luogo ad un deadlock quando invocato con due valori la cui somma sia negativa. Il ciclo while (...) sospende infatti il chiamante fino a quando la condizione $(x+y) < 0$ è verificata, ma essendo la condizione data sul valore dei parametri, questi non possono mai essere cambiati da altri thread, con il risultato che il chiamante resta per sempre sospeso.

Domanda b)

Si scriva il metodo boolean compare(Point other) che ritorna true se e solo se il punto other ha le stesse coordinate del punto this. Si garantisca la correttezza dell'implementazione anche in presenza di thread multipli.

Soluzione

```
public boolean compare(Point other) {  
    double x,y;  
    synchronized(other) {  
        x = other.myX;  
        y = other.myY;  
    }  
    synchronized(this) {  
        return x==myX && y==myY;  
    }  
}
```

Domanda c)

Si scriva il metodo void printOncePerSecond(int numTimes) che, in un thread separato rispetto al chiamante, stampa per numTimes volte le coordinate del punto, una ogni secondo. Si faccia attenzione a garantire il corretto accesso alla memoria condivisa, senza bloccare i thread oltre il tempo minimo indispensabile.

Si ricorda l'esistenza del metodo statico Thread.sleep(millis) che sospende il thread corrente per il numero indicato di millisecondi.

Soluzione

```
public void printOncePerSecond(int numTimes) {  
    new Thread(() -> {  
        for(int i=0; i<numTimes; i++) {  
            synchronized(Point.this) {  
                System.out.println("X: "+myX+" Y: "+myY);  
            }  
            try { Thread.sleep(1024); }  
            catch(InterruptedException ex) { ex.printStackTrace(); }  
        }  
    }).start();  
}
```

Si noti la necessità di acquisire il lock su Point.this per evitare conflitti in presenza di chiamate di altri metodi della classe Point (tutti sincronizzati su this).

Esercizio 3

Si considerino le seguenti classi di cui viene qui omessa l'ovvia implementazione:

```
public class Drawing {  
    ...  
    public List<Shape> getShapes() { ... }  
    public String getName() { ... }  
    public int getDiagonal() { ... }  
}  
  
public class Shape {  
    ...  
    public Color getColor() { ... }  
}
```

Rispondere alle domande seguenti usando **esclusivamente** i costrutti della *programmazione funzionale* (*senza quindi usare while, if.. else, for, ecc.*)

Domanda a)

Completare l'implementazione del seguente metodo `printNameOfBiggest` che prende in ingresso una lista di dipinti (`Drawing`) e stampa il nome del dipinto avente la diagonale maggiore tra quelli nella lista. Se la lista è vuota, il metodo non stampa nulla. Se esistono più dipinti con la stessa diagonale, il metodo stampa comunque un solo nome (non viene specificato quale).

```
public static void printNameOfBiggest(List<Drawing> drawings) {  
    ...  
}
```

Soluzione

```
public static void printNameOfBiggest(List<Drawing> drawings) {  
    drawings.stream()  
        .max((d1, d2) -> Integer.compare(d1.getDiagonal(), d2.getDiagonal()))  
        .map(Drawing::getName)  
        .ifPresent(System.out::println);  
}
```

Domanda b)

Completare l'implementazione del seguente metodo `getAllColors` che prende in ingresso una lista di dipinti (`Drawing`) e ritorna una lista con tutti i colori utilizzati in almeno uno dei dipinti della lista. La lista ritornata non deve contenere duplicati.

```
public static List<Color> getAllColors(List<Drawing> drawings) {  
    return ...  
}
```

Soluzione

```
public static List<Color> getAllColors(List<Drawing> drawings) {  
    return drawings.stream()  
        .flatMap(d -> d.getShapes().stream())  
        .map(Shape::getColor)  
        .distinct()  
        .collect(Collectors.toList());  
}
```

Esercizio 4

Si consideri il seguente metodo statico:

```
public static int test(int x, int y) {  
    int k = 0;  
    if(x>0) {  
        if(y>0) {  
            k += y;  
        } else {  
            k += -y;  
        }  
    }  
    if(x==10) {  
        k += 10;  
    }  
    return k;  
}
```

Domanda a)

Si determini un insieme minimo di casi di test per garantire la copertura delle istruzioni (statement coverage).

Soluzione

Servono almeno due casi di test al fine di entrare sia nel ramo `if` che nel ramo `else` dello statement condizionale all'interno del primo `if`. Ad esempio i due casi di test: `(10, 1)` e `(10, 0)`

Domanda b)

Si determini un insieme minimo di casi di test per garantire la copertura delle decisioni (edge coverage).

Soluzione

Ai due casi precedenti si devono aggiungere i casi di test necessari a valutare false le due condizioni degli `if` “esterni”. Basta un caso di test, ad esempio `(0, qualsiasi)`. In definitiva servono tre casi di test: `(10, 1)`, `(10, 0)` e `(0, qualsiasi)`

Domanda c)

Si determini un insieme minimo di casi di test per garantire la copertura dei cammini (path coverage).

Soluzione

In totale ci sono 5 cammini possibili, esercitati dai seguenti casi di test: `(10, 1)`, `(5, 1)`, `(10, 0)`, `(5, 0)` e `(0, qualsiasi)`.

Ingegneria del Software — Soluzione del Tema 13/02/2023

Esercizio 1

Si consideri la seguente classe `Championship` che memorizza i risultati di un campionato tra diversi `Team`. Si noti che il campionato distingue tra andata e ritorno di ogni partita.

```
public class Championship {
    // Ritorna l'insieme dei team che fanno parte del campionato.
    public /*@ pure @*/ Set<Team> getTeams() {...}

    // Cancella tutti i risultati delle partite fin qui giocate.
    public void clear() {...}

    // Ritorna il team vincente nella partita giocata tra t1 (in casa) e t2 (fuori casa), oppure null
    // se non si e' giocata nessuna partita tra t1 e t2 (t1 e t2 devono essere squadre del campionato).
    // Si noti che il valore restituito potrebbe non coincidere esattamente con t1 o t2, ma deve essere
    // uguale ("equals") a uno dei due.
    public /*@ pure @*/ Team winner(Team t1, Team t2) {...}

    // Inserisce il risultato della partita tra t1 (in casa) e t2 (fuori casa) con vincitore winner
    // (riguardo t1, t2 e winner valgono le stesse note indicate per il precedente metodo).
    // Solleva l'eccezione MatchAlreadyPlayedException se la partita e' gia' stata giocata.
    public void addMatch(Team t1, Team t2, Team winner) throws MatchAlreadyPlayedException {...}

    // Ritorna l'elenco delle squadre con cui la squadra t ha vinto giocando in casa.
    public /*@ pure @*/ List<Team> defeatedAtHome(Team t) {...}

    // Ritorna il punteggio accumulato dalla squadra t (la squadra vincente, in casa o fuori casa,
    // prende un punto).
    public /*@ pure @*/ int numPoints(Team t) {...}
```

Domanda a)

Specificare in JML il metodo `addMatch`.

Soluzione

```
/*@requires t1!=null && t2!=null && getTeams().contains(t1) &&
/*@     getTeams().contains(t2) && !t1.equals(t2) && (winner.equals(t1) || winner.equals(t2))
/*@ensures \old(winner(t1,t2)==null) && winner(t1,t2).equals(winner) &&
/*@     getTeams().size()==\old(getTeams().size()) &&
/*@     getTeams().containsAll(\old(getTeams())) &&
/*@     (\forall Team tt1; getTeams().contains(tt1);
/*@         (\forall Team tt2; getTeams().contains(tt2);
/*@             !tt1.equals(t1) && !tt2.equals(t2) ==> winner(tt1,tt2)==\old(winner(tt1,tt2)))
/*@signals (MatchAlreadyPlayedException) \old(winner(t1,t2)!=null) &&
/*@     getTeams().size()==\old(getTeams().size()) &&
/*@     getTeams().containsAll(\old(getTeams())) &&
/*@     (\forall Team tt1; getTeams().contains(tt1;
/*@         (\forall Team tt2; getTeams().contains(tt2);
/*@             winner(tt1,tt2)==\old(winner(tt1,tt2)))
public void addMatch(Team t1, Team t2, Team winner) throws MatchAlreadyPlayedException {...}
```

Domanda b)

Specificare in JML i metodi `defeatedAtHome` e `numPoints`.

Soluzione

```
/*@requires t!=null && getTeams().contains(t)
/*@ensures (\forall Team t1; ; \result.contains(t1) <=>
/*@     (getTeams().contains(t1) && (winner(t,t1)!=null && winner(t,t1).equals(t))) ) &&
/*@     (\forall int i; i>=0 && i<\result.size();
/*@         (\forall int j; j>i && j<\result.size();
/*@             !\result.get(i).equals(\result.get(j)))
public /*@ pure @*/ List<Team> defeatedAtHome(Team t) {...}

/*@requires t!=null && getTeams().contains(t)
/*@ensures \result == (\numof Team t1; getTeams().contains(t1);
/*@             (winner(t,t1)!=null && winner(t,t1).equals(t)) ||
/*@             (winner(t1,t)!=null && winner(t1,t).equals(t)))
public /*@ pure @*/ int numPoints(Team t) {...}
```

Domanda c)

Si consideri una implementazione della classe `Championship` che usa una lista di `Team` per memorizzare le squadre del campionato e un array bidimensionale di `Team` per memorizzare le partite giocate.

```
public class Championship {  
    private List<Team> teams;  
    private Team[][] res;  
    ...
```

La posizione `res[i][j]` vale `null` se la partita tra l'i-esima squadra (in casa) e la j-esima squadra (fuori casa) nella lista `teams` non si è ancora giocata, mentre contiene un riferimento alla squadra vincente se la partita si è giocata.

Per questa implementazione scrivere l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

L'invariante di rappresentazione deve dire che i due attributi sono non nulli, che la lista `teams` contiene almeno due squadre, che la dimensione dell'array bidimensionale `res` è coerente con il numero di squadre nella lista `teams`, che non ci sono squadre duplicate nella lista `teams`, che la posizione `[i, j]` (per `i!=j`) della matrice `res` contiene il valore `null` o il riferimento ad una squadra che `equals`s l'i-esima o la j-esima squadra in `teams`, ed infine che la diagonale della matrice `res` contiene sempre `null`.

```
//@private invariant  
//@ teams!=null && res!=null && teams.size()>=2 &&  
//@ res.length==teams.size() &&  
//@ (\forall int i; i>=0 && i<teams.size(); res[i].length==teams.size()) &&  
//@ (\forall int i; i>=0 && i<teams.size());  
//@   (\forall int j; j>=0 && j<teams.size() && i!=j;  
//@     !teams.get(i).equals(teams.get(j)) &&  
//@     (res[i][j]==null || res[i][j].equals(teams.get(i)) || res[i][j].equals(teams.get(j)))) ) &&  
//@ (\forall int i; i>=0 && i<teams.size(); res[i][i]==null)
```

La funzione di astrazione “mappa” gli attributi privati sui metodi base (“ground”) usati per la specifica: `getTeams` e `winner`.

```
//@private invariant  
//@ getTeams().containsAll(teams) && teams.containsAll(getTeams()) &&  
//@ (\forall int i; i>=0 && i<teams.size();  
//@   (\forall int j; j>=0 && j<teams.size();  
//@     res[i][j]==winner(teams.get(i),teams.get(j)) ) )
```

Domanda d)

Si consideri una classe `ReplayableChampionship` che aggiunge alla classe `Championship` il metodo pubblico:
`void replayMatch(Team t1, Team t2, Team winner)`

`ReplayableChampionship` può essere definita come sottoclasse di `Championship` senza violare il principio di sostituzione di Liskov?

Soluzione

Si. La classe `ReplayableChampionship` aggiunge un metodo senza ridefinire metodi esistenti, quindi non viola le regole della signature e dei metodi. Inoltre il metodo aggiunto non viola alcuna proprietà invariante perché il suo comportamento è equivalente ad una opportuna sequenza di chiamate dei metodi originali della classe `Championship`, partendo con una chiamata del metodo `clear` e aggiungendo tutte le partite prima aggiunte con il medesimo risultato, tranne la nuova partita che si aggiunge con il nuovo risultato.

Esercizio 2

Si consideri la seguente classe `Products` che contiene informazioni su un insieme di prodotti. Ogni prodotto è caratterizzato da un identificativo e un nome univoci e un prezzo.

```
public class Products {
    private Map<Integer, String> idToName;
    private Map<String, Integer> nameToPrice;
    private int lastId;

    public Products() {
        idToName = new HashMap<>();
        nameToPrice = new HashMap<>();
        lastId = 0;
    }

    public int addProduct(String name, int price) {
        int assignedId = 0;
        synchronized (idToName) {
            assignedId = lastId++;
            idToName.put(assignedId, name);
        }
        synchronized (nameToPrice) { nameToPrice.put(name, price); }
        return assignedId;
    }

    // Ritorna il prezzo del prodotto avente l'id passato per parametro
    // Se tale prodotto non esiste, ritorna -1
    public int getPriceById(int id) {
        synchronized (idToName) {
            if (idToName.containsKey(id)) {
                String name = idToName.get(id);
                return nameToPrice.get(name);
            } else {
                return -1;
            }
        }
    }
}
```

Domanda a)

Si dica se la classe `Products` è opportunamente sincronizzata per permettere l'invocazione concorrente dei suoi metodi da parte di più thread. In caso negativo, si modifichi la classe per correggere la sua sincronizzazione.

Soluzione

La classe *non* è opportunamente sincronizzata, in quanto il metodo `getPriceById` accede alla variabile `nameToPrice` senza acquisire un lock su di essa: di conseguenza, la variabile potrebbe essere modificata durante un'esecuzione concorrente del metodo `addProduct`.

Si noti che proteggere l'accesso a `nameToPrice` nel metodo `getPriceById` con un blocco `synchronized` su di esso *non* sarebbe sufficiente. Occorre rendere atomico l'inserimento di un prodotto nelle due mappe per garantire che il nome del prodotto sia presente in entrambe o in nessuna durante l'invocazione del metodo `getPriceById`.

Entrambe le mappe devono quindi essere aggiornate all'interno del medesimo blocco `synchronized` nel metodo `addProduct`, e il metodo `getPriceById` deve sincronizzarsi sullo stesso oggetto. Nella soluzione proposta, si garantisce atomicità sincronizzando entrambi i metodi su `this`.

```
public class Products {
    private Map<Integer, String> idToName;
    private Map<String, Integer> nameToPrice;
    private int lastId;

    public Products() {
        idToName = new HashMap<>();
        nameToPrice = new HashMap<>();
        lastId = 0;
    }

    public synchronized void addProduct(String name, int price) {
        int assignedId = lastId++;
        idToName.put(assignedId, name);
    }

    // Ritorna il prezzo del prodotto avente l'id passato per parametro
    // Se tale prodotto non esiste, ritorna -1
    public int getPriceById(int id) {
        synchronized (this) {
            if (idToName.containsKey(id)) {
                String name = idToName.get(id);
                return nameToPrice.get(name);
            } else {
                return -1;
            }
        }
    }
}
```

```

        nameToPrice.put(name, price);
    return assignedId;
}

// Ritorna il prezzo del prodotto avente l'id passato per parametro
// Se tale prodotto non esiste, ritorna -1
public synchronized int getPriceById(int id) throws {
    if (idToName.containsKey(id)) {
        String name = idToName.get(id);
        return nameToPrice.get(name);
    } else {
        return -1;
    }
}
}

```

Domanda b)

A partire dalla soluzione data alla domanda (a), si modifichi il metodo getPriceById in modo che, invece di ritornare -1, sospenda il chiamante se il prodotto non esiste. Se necessario, si indichi come modificare altri metodi.

Soluzione

```

public class Products {
    private Map<Integer, String> idToName;
    private Map<String, Integer> nameToPrice;
    private int lastId;

    public Products() {
        idToName = new HashMap<>();
        nameToPrice = new HashMap<>();
        lastId = 0;
    }

    public synchronized void addProduct(String name, int price) {
        int assignedId = lastId++;
        idToName.put(assignedId, name);
        nameToPrice.put(name, price);
        notifyAll(); // Risveglia potenziali thread in attesa di questo id
        return assignedId;
    }

    public synchronized int getPriceById(int id) throws InterruptedException {
        while (!idToName.containsKey(id)) {
            wait(); // Sospende il chiamante se l'id non e' presente
        }
        String name = idToName.get(id);
        return nameToPrice.get(name);
    }
}

```

Domanda c)

Si scriva un metodo void copyPrices(Products other) che, per tutti i prodotti di this che sono anche contenuti in other sostituisce i prezzi contenuti in this con i prezzi contenuti in other. Si presti particolare attenzione a evitare il rischio di possibili deadlock.

Si ricordano alcuni metodi di Map che potrebbero risultare utili per l'implementazione: Map ha un costruttore che prende in ingresso un'altra mappa e ne copia il contenuto (chiavi e valori); il metodo keySet() ritorna un Set con l'insieme delle chiavi; il metodo put prende in ingresso una chiave e un valore e aggiunge o sovrascrive il valore associato alla chiave.

Soluzione

```

void copyPrices(Products other) {
    Map<String, Integer> copy = null;
    synchronized (other) {
        copy = new HashMap<>(other.nameToPrice);
    }
    synchronized (this) {
        for (String name : copy.keySet()) {
            if (nameToPrice.containsKey(name)) {
                nameToPrice.put(name, copy.get(name));
            }
        }
    }
}

```


Esercizio 3

Si considerino le seguenti classi Java

```
public class Shop {  
    public List<Product> getProducts();  
    ...  
}  
  
public class Product {  
    public int getPrice();  
    public String getName();  
    ...  
}
```

Si risponda alle seguenti domande facendo uso esclusivamente dei costrutti di programmazione funzionale di Java.

Domanda a)

Si completi il metodo `printExpensiveNames`, che stampa il nome di tutti i prodotti dello `Shop` `s` il cui prezzo è maggiore di 10.

```
public static void printExpensiveNames(Shop s) {  
    ...  
}
```

Soluzione

```
public static void printExpensiveNames(Shop s) {  
    s.getProducts().stream()  
        .filter(p -> p.getPrice() > 10)  
        .map(p -> p.getName())  
        .forEach(System.out::println);  
}
```

Domanda b)

Si completi il metodo `getAllProductsNames`, che ritorna una lista con tutti i prodotti in vendita in almeno uno `Shop` presente nella lista `shops`

```
public static List<String> getAllProductsNames(List<Shop> shops) {  
    return ...  
}
```

Soluzione

```
public static List<String> getAllProductsNames(List<Shop> shops) {  
    return shops.stream()  
        .flatMap(s -> s.getProducts().stream())  
        .map(p -> p.getName())  
        .collect(Collectors.toList());  
}
```

Domanda c)

Si aggiunga alla classe `Shop` un metodo `getBestProducts()`. Il metodo prende in ingresso una funzione `f`, la quale prende in ingresso un `Product` e ritorna un intero rappresentante l'indice di gradimento (rating) di quel prodotto. Il metodo stampa il nome di tutti i prodotti con rating maggiore di 10.

Si ricorda che in Java una funzione è rappresentata da un'interfaccia funzionale `Function<T, R>`, con un unico metodo: `R apply(T input)`.

Soluzione

```
public void getBestProducts(Function<Product, Integer> f) {  
    this.getProducts().stream()  
        .filter(p -> f.apply(p) > 10)  
        .map(p -> p.getName())  
        .forEach(System.out::println);  
}
```

Esercizio 4

Si considerino le seguenti dichiarazioni di classi Java.

```
abstract class Box {
    protected String secret;
    protected String content() { return secret; }
    public void show(Box p) {
        System.out.print("Box.show(Box)");
        System.out.println(p.content());
    }
    public Box(String param) {
        secret = param + " box ";
    }
}

class SteelBox extends Box {
    public void show(SteelBox p) {
        System.out.println(p.content());
    }
    public void show(Box p) {
        System.out.println(content() + "&& " + p.content());
    }
    public SteelBox(String param) {
        super(param + " steel ");
    }
}

class LockedBox extends SteelBox {
    protected String content() { return "secret! "; }
    public void show(LockedBox p) {
        System.out.println(content() + "and " + p.content());
    }
    public void show(Box p) {
        System.out.println(content() + "with " + p.content());
    }
    public LockedBox(String param) {
        super(param + " locked ");
    }
}
```

e il seguente frammento di codice, in cui le righe sono numerate per comodità:

```
1 Box b,s,l;
2 SteelBox sb;
3 LockedBox lb;
4 b = new Box(" b");
5 s = new SteelBox(" s");
6 l=s;
7 s.show(l);
8 lb=s;
9 l = new LockedBox(" l");
10 l.show(s);
11 sb = new SteelBox(" sb");
12 lb = new LockedBox(" lb");
13 sb.show(s);
14 sb.show(l);
15 sb.show(sb);
16 sb.show(lb);
17 lb.show(s);
18 lb.show(l);
19 lb.show(sb);
20 lb.show(lb);
```

Domanda a)

Si indichino i numeri di ogni riga del frammento sopra riportato che genera un errore, precisando se l'errore si manifesti in fase di compilazione o in fase di esecuzione. Motivare brevemente la risposta.

Soluzione

Riga 4: Compilazione, una classe astratta non può essere istanziata.

Riga 8: Compilazione, assegnamento polimorfo errato.

Domanda b)

Supponendo che tutte le righe che provocano errori in fase di compilazione o di esecuzione siano state rimosse, mostrare per ogni istruzione di stampa quale stringa viene prodotta in output, motivando brevemente la propria risposta. Riportare **sempre** il numero di riga di fianco alla risposta.

Soluzione

```
7  s steel  box &&  s steel  box
8
9
10 secret! with  s steel  box
11
12
13 sb steel  box &&  s steel  box
14 sb steel  box && secret!
15 sb steel  box
16 secret!
17 secret! with  s steel  box
18 secret! with secret!
19 sb steel  box
20 secret! and secret!
```

Ingegneria del Software — Soluzione del Tema 14/01/2023

Esercizio 1

Si consideri la seguente classe SpeedLimits per memorizzare i limiti di velocità per i vari segmenti (RoadSegments) di una strada. La strada inizia al km 0 e ha una lunghezza di `len()` km. Il costruttore assegna lo stesso limite di velocità all'intera strada. Un segmento (RoadSegment) è definito come un tratto di strada che inizia e termina con un cambio di limite di velocità e all'interno del quale il limite di velocità non cambia. Il primo segmento inizia al km 0 e l'ultimo segmento termina al km `len()` (escluso).

```
public class SpeedLimits {
    // Considera una strada di lunghezza len e imposta a speedLimit il limite
    // di velocita' per l'intera strada.
    public SpeedLimits(int speedLimit, int len);

    // Ritorna la lunghezza della strada in km.
    public /*@ pure @*/ int len();

    // Ritorna il limite di velocita' al km specificato.
    public /*@ pure @*/ int getSpeedLimitAt(int km);

    // Imposta il limite di velocita' a speedLimit per il tratto di strada che inizia
    // al km start (incluso) e finisce al km end (escluso).
    // Il limite di velocita' deve essere positivo e inferiore o uguale a 130 km orari.
    // Lancia una InvalidBoundariesException se gli estremi del tratto di strada non sono validi.
    public void changeSpeedLimit(int start, int end, int speedLimit) throws InvalidBoundariesException;

    // Ritorna la lista dei segmenti contenuti nella strada. I segmenti sono
    // ritornati in ordine (da quello che inizia per primo a quello che inizia per ultimo).
    public /*@ pure @*/ List<RoadSegment> getSegments();
}

public /*@ pure @*/ RoadSegment {
    // Ritorna il km di inizio del segmento (incluso nel segmento).
    public /*@ pure @*/ int getStart();

    // Ritorna il km di fine del segmento (escluso dal segmento).
    public /*@ pure @*/ int getEnd();

    // Ritorna il limite di velocita' definito per il segmento di strada.
    public /*@ pure @*/ int getSpeedLimit();
}
```

Domanda a)

Specificare in JML il metodo `changeSpeedLimit`.

Soluzione

```
//@ requires speedLimit > 0 && speedLimit <= 130
//@
//@ ensures start >= 0 && end <= len() && start < end &&
//@ (\forall int i; i>=0 && i<len(); getSpeedLimitAt(i) ==
//@      (i>=start && i<end) ? speedLimit : \old(getSpeedLimitAt(i)) )
//@
//@ signals (InvalidBoundariesException e) && (start < 0 || end > len() || start >= end) &&
//@ (\forall int i; i>=0 && i<len(); getSpeedLimitAt(i) == \old(getSpeedLimitAt(i)))
public void changeSpeedLimit(int start, int end, int speedLimit) throws InvalidBoundariesException;
```

Domanda b)

Specificare in JML il metodo `getSegments()`.

Soluzione

```
//@ ensures \result != null && !\result.isEmpty() &&
//@
//@ (\forall RoadSegment s; ; \result.contains(s) <=> (s.getStart() < s.getEnd() &&
//@      (s.getStart() == 0 || getSpeedLimitAt(s.getStart()) != getSpeedLimitAt(s.getStart())-1) &&
//@      (s.getEnd() == len() || getSpeedLimitAt(s.getEnd())-1 != getSpeedLimitAt(s.getEnd())) ) &&
//@      (\forall int i; i>=s.getStart() && i<s.getEnd()-1;
//@          getSpeedLimitAt(i) == getSpeedLimitAt(i+1) ) ) ) &&
//@ (\forall int i; 0 <= i < \result.size()-1; \result.get(i).getEnd() == \result.get(i+1).getStart())
public /*@ pure @*/ List<RoadSegment> getSegments();
```

Domanda c)

Si consideri la seguente implementazione della classe `SpeedLimits` che usa una mappa per definire i limiti di velocità e un intero per definire la lunghezza della strada. La mappa ha come chiave il km al quale inizia un segmento di strada e come valore il limite di velocità definito per quel segmento di strada.

```
public class SpeedLimits {  
    private Map<Integer, Integer> segments;  
    private int len;  
    ... }
```

Per tale implementazione, si forniscano l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

L'invariante di rappresentazione richiede che la lunghezza della strada sia positiva. La mappa non deve essere nulla, non deve contenere chiavi o valori nulli. Le chiavi devono essere tra 0 (incluso) e `len` (escluso). I valori devono essere tra 0 (escluso) e 130 (incluso). Deve esistere la chiave 0. Chiavi che definiscono segmenti consecutivi non possono essere associate allo stesso valore.

```
//@ private invariant  
//@ len > 0 && segments != null && segments.containsKey(0) &&  
//@  
//@ (\forall Integer k; segments.containsKey(k);  
//@     k != null && k >= 0 && k < len &&  
//@     segments.get(k) != null && segments.get(k) > 0 && segments.get(k) <= 130) &&  
//@  
//@ (\forall Integer k1; segments.containsKey(k1);  
//@     (\forall Integer k2; segments.containsKey(k2) &&  
//@         !(\exists Integer k3; segments.containsKey(k3) && k1 < k3 && k3 < k2);  
//@         segments.get(k1) != segments.get(k2) ) )
```

La funzione di astrazione definisce `len()` e `getSpeedLimitAt` in base a `len` e `segments`. Tutti gli altri metodi sono definiti a partire da questi.

```
//@ private invariant  
//@ len() == len &&  
//@ (\forall int i; i >= 0 && i < len;  
//@     (\max Integer j; segments.containsKey(j) && j <= i;  
//@         getSpeedLimitAt(i) == segments.get(j) ) )
```

Domanda d)

Si consideri una classe `GreenSpeedLimits` che aggiunge un metodo `reduceEmissions` che abbassa del 15% tutti i limiti di velocità salvati al fine di ridurre le emissioni. Può tale classe essere definita come sottoclasse di `SpeedLimits` in accordo con il principio di sostituzione?

Soluzione

La classe può essere definita come sottoclasse di `SpeedLimits` perché aggiunge un metodo senza alterare i metodi esistenti e senza violare alcun invariante pubblico di `SeriesStore`. Lo stesso risultato del metodo aggiunto si potrebbe ottenere con la classe originale, modificando la velocità individualmente per tutti i segmenti.

Esercizio 2

Si consideri la seguente classe Car che rappresenta una autovettura con la sua targa, la sua velocità e il suo colore.

```
public class Car {  
    private String plateNumber;  
    private int speed;  
    private String color;  
    public Car(String plateNumber) {  
        this.platenumber = plateNumber;  
        speed = 0;  
        color = "black";  
    }  
    public String getPlateNumber() { return plateNumber; }  
    public void accelerate() { speed++; }  
    public void decelerate() { speed--; }  
    public int getSpeed() { return this.speed; }  
    public void paint(String newColor) { this.color = newColor; }  
    public String getColor() { return color; }  
}
```

Domanda a)

Si sincronizzino opportunamente i metodi della classe in maniera da massimizzare il parallelismo.

Soluzione

Al fine di massimizzare il parallelismo, sincronizziamo separatamente i metodi che accedono a speed e quelli che accedono a color, usando due opportuni oggetti separati come “lock”. Si noti che il metodo getPlateNumber non deve essere sincronizzato perché il plateNumber non viene mai cambiato.

```
public class Car {  
    private String plateNumber;  
    private int speed;  
    private String color;  
    private Object speedLock, colorLock;  
    public Car(String plateNumber) {  
        this.platenumber = plateNumber;  
        speed = 0;  
        color = "black";  
        speedLock = new Object();  
        colorLock = new Object();  
    }  
    public String getPlateNumber() { return plateNumber; }  
    public void accelerate() { synchronized(speedLock) speed++; }  
    public void decelerate() { synchronized(speedLock) speed--; }  
    public int getSpeed() { synchronized(speedLock) return this.speed; }  
    public void paint(String newColor) { synchronized(colorLock) this.color = newColor; }  
    public String getColor() { synchronized(colorLock) return color; }  
}
```

Domanda b)

Si aggiunga alla classe Car un metodo public void limitSpeed() che crea ed avvia un thread parallelo rispetto al chiamante, il quale, in un ciclo infinito (senza bloccare indefinitamente l'oggetto), decelera l'auto (chiamando il metodo decelerate) senza mai andare al di sotto di 10 km/h.

Soluzione

All'interno del ciclo infinito del nuovo thread occorre rendere atomico il frammento di codice che decide se decelerare in funzione della velocità attuale (per non rischiare di decelerare al di sotto dei 10 km/h).

```
public void speedLimit() {  
    new Thread() {  
        public void run() {  
            while (true) {  
                synchronized(speedLock) {  
                    if (getSpeed() > 10) decelerate();  
                }  
            }  
        }.start();  
}
```

Domanda c)

Fare in modo che il metodo paint sospenda il chiamante in attesa che la macchina sia ferma e cambi il colore solo mentre l'auto è ferma.

Soluzione

```
public void paint(String newColor) {  
    synchronized(speedLock) {  
        while(speed!=0) speedLock.wait();  
        synchronized(colorLock) { this.color = newColor; }  
    }  
}
```

Occorre anche inserire speedLock.notifyAll() nei metodi che modificano il valore di speed.

Esercizio 3

Si considerino le seguenti dichiarazioni di classi Java.

```
abstract class LibraryItem {
    public abstract String show();
}

class Text extends LibraryItem {
    public String show() { return "Text"; }
    public String show(LibraryItem a) { return this.show() + " and " + a.show(); }
}

class AudioBook extends Text {
    public String show(Text a) { return "Audio with " + a.show(); }
}
```

e il seguente frammento di codice

```
1 AudioBook item1;
2 Text item2;
3 LibraryItem item3;
4 item3 = new LibraryItem();
5 item2 = new AudioBook();
6 item1 = item2;
7 item3 = new Text();
8 System.out.println(item3.show());
9 System.out.println(item3.show(item2));
10 System.out.println(item2.show(item3));
11 item2 = (AudioBook) item3;
12 item1 = (AudioBook) item2;
13 System.out.println(item1.show());
14 System.out.println(item1.show(item2));
15 System.out.println(item1.show(item3));
```

Domanda a)

Si indichino il numero di ciascuna riga del frammento sopra riportato che genera un errore in fase di compilazione o in fase di esecuzione, motivando concisamente la propria risposta.

Soluzione

Errori di compilazione:

- riga 4 (impossibile istanziare una classe astratta);
- riga 6 (item2 ha tipo statico Text, che è un sopratipo del tipo statico di item1, ossia AudioBook);
- riga 9 (item3 ha tipo statico LibraryItem, quindi è visibile solo show(), non show(Text)).
- Errori run-time: la riga 11 (item3 non ha tipo dinamico AudioBook, quindi il casting fallisce).

Domanda b)

Supponendo che tutte le righe che provocano errori in fase di compilazione o di esecuzione siano state rimosse, mostrare per ogni istruzione di stampa quale stringa viene prodotta in output, motivando brevemente la propria risposta. Riportare *sempre* il numero di riga di fianco alla risposta.

Soluzione

- Riga 8: Text
- Riga 10: Text and Text
- Riga 13: Text
- Riga 14: Audio with Text
- Riga 15: Text and Text

Esercizio 4

Si consideri il seguente metodo statico.

```
1 public static void doSomething(int a, int b) {  
2     if (a>0 && b>0) {  
3         while (a<=2 && b>0) {  
4             if (a==1 && b==2) {  
5                 a++;  
6             }  
7             b--;  
8         }  
9     }  
10 }
```

Domanda a)

Si definisca il minimo un numero di casi di test (per a, b) necessari e sufficienti per coprire tutte le istruzioni del metodo, se ciò è possibile.

Soluzione

Basta il caso $a==1, b==2$.

NB: in questo caso si percorre due volte il ciclo; la seconda iterazione del ciclo è percorsa con $a==2, b==1$, terminando con $b==0$ per uscire quindi dal while.

Domanda b)

Si definisca il minimo numero di casi di test necessari e sufficienti per coprire tutte le diramazioni (edge coverage) del metodo, se ciò è possibile.

Soluzione

Le diramazioni del while e del secondo if sono già coperte dal caso precedente. Occorre aggiungere un solo caso per coprire la diramazione else del primo if: ad esempio $a==0, b$ qualunque.

Domanda c)

Si definisca il minimo numero di casi di test necessari e sufficienti per coprire tutte le diramazioni e condizioni (edge and condition coverage) del metodo, se ciò è possibile.

Soluzione

Occorre considerare i seguenti requisiti.

La copertura delle condizioni del primo if: manca il caso in cui $b>0$ è false (ma serve $a>0$ true per potere esercitare la condizione), quindi ad esempio $a==1, b==0$.

La copertura delle condizioni del while: i casi precedenti non coprono il caso $a<=2$ false, quindi occorre aggiungere ad esempio il caso $a==3$, con $b>0$ per entrare nel primo if.

La copertura delle condizioni del secondo if: il primo caso di test $a==1, b==2$ copre anche il caso $a==1$ false (perché $a==2$ alla seconda iterazione del ciclo). Resta da coprire solo il caso $b==2$ false (che richiede $a==1$ true). Se si sceglie $a==1$ e $b==3$, questo può sostituire il primo caso di test, in quanto il valore delle variabili diventerà proprio $a==1, b==2$ dopo un'iterazione.

In conclusione, sono necessari 4 casi. Ad esempio: $a==1$ e $b==0$, $a==0$ e b qualunque, $a==3$ e b positivo, $a==1$ e $b==3$.

Ingegneria del Software — Soluzione del Tema 28/08/2023

Esercizio 1

Si consideri la seguente classe CityMap che memorizza i punti di interesse di una città (Site), organizzati per tipo (ristoranti, hotel, musei, etc.).

```
public class CityMap {
    // Ritorna l'insieme di tutti i punti di interesse memorizzati
    public /*@ pure @*/ Set<Site> getSites() { ... }

    // Aggiunge il punto di interesse passato per parametro
    public void addSite(Site site) { ... }

    // Ritorna tutti i punti di interesse del tipo passato per parametro,
    // ordinati per distanza crescente rispetto alla location loc indicata.
    // Lancia una NotFoundException se non esiste alcun punto di interesse del tipo indicato.
    public /*@ pure @*/ List<Site> getSitesByType(Type type, Location loc) throws NotFoundException { ... }

    // Ritorna i k punti di interesse piu' vicini alla location loc indicata
    // (per semplicita', si assuma che non possano esistere due punti equidistanti da loc).
    // Se i punti di interesse sono meno di k, vengono ritornati tutti.
    // La mappa ritornata contiene i punti di interesse come chiave e la loro distanza da loc come valore.
    public /*@ pure @*/ Map<Site, Double> getNearestSites(int k, Location loc) { ... }
}

public /*@ pure @*/ class Site {
    // Ritorna la posizione geografica del punto di interesse
    public Location getLocation() { ... }

    // Ritorna il tipo del punto di interesse (ristorante, hotel, museo, ...)
    public Type getType() { ... }
    ...
}

public /*@ pure @*/ class Location {
    // Calcola la distanza rispetto alla Location passata per parametro
    public double getDistance(Location other) { ... }
}
```

Domanda a)

Si specifichi in JML il metodo getSitesByType.

Soluzione

```
/*@ requires type!=null && loc!=null
*/
/*@ ensures (\exists Site s; getSites().contains(s); s.getType().equals(type)) && \result!=null &&
/*@ (\forall Site s; \result.contains(s) <==> getSites().contains(s) && s.getType().equals(type)) &&
/*@ (\forall int i; i>=0 && i<\result.size());
/*@ (\forall int j; j>=0 && j<i; !\result.get(j).equals(\result.get(i)) &&
/*@ \result.get(j).getLocation().getDistance(loc)<=\result.get(i).getLocation().getDistance(loc) )
*/
/*@ signals (NotFoundException e) !(\exists Site s; getSites().contains(s); s.getType().equals(type))
public /*@ pure @*/ List<Site> getSitesByType(Type type, Location loc) throws NotFoundException { ... }
```

Domanda b)

Si specifichi in JML il metodo getNearestSites.

Soluzione

```
/*@ requires k>0 && loc!=null
*/
/*@ ensures \result!=null && \result.size() == (k<getSites().size() ? k : getSites().size()) &&
/*@ (\forall Site s; \result.keySet().contains(s);
/*@ getsites().contains(s) && \result.get(s)==s.getLocation().getDistance(loc) ) &&
/*@ (\forall Site s; getsites().contains(s);
/*@ (\num Site s1; getsites().contains(s1) && !s1.equals(s);
/*@ s1.getLocation().getDistance(loc) < s.getLocation().getDistance(loc) ) < k
/*@ ==> \result.keySet().contains(s) )
public /*@ pure @*/ Map<Site, Double> getNearestSites(int k, Location loc) { ... }
```

Domanda c)

Si consideri un'implementazione che fa uso di una mappa per memorizzare i punti di interesse organizzati per tipo, come mostrato di seguito.

```
public class CityMap {  
    private Map<Type, Set<Site>> sites;  
    ... }
```

Per tale implementazione di forniscano l'invariante di rappresentazione e la funzione di astrazione.

Soluzione

L'invariante di rappresentazione (RI) deve specificare che la mappa non sia nulla, non contenga chiavi nulle, non contenga valori nulli o vuoti. Inoltre ogni punto di interesse deve essere non nullo e memorizzato nell'insieme corrispondente al proprio tipo.

RI:

```
//@ private invariant sites!=null && !sites.containsKey(null) &&  
//@ (\forall Type t; sites.containsKey(t); sites.get(t)!=null && !sites.get(t).isEmpty() &&  
//@   (\forall Site s; sites.get(t).contains(s); s!=null && s.getType().equals(t)) )
```

La funzione di astrazione (AF) si limita a definire `getSites` in funzione di `sites`, in quanto tutti gli altri metodi sono specificati a partire da `getSites`.

AF:

```
//@ private invariant (\forall Site s; ; getSites().contains(s) <==>  
//@   sites.containsKey(s.getType()) && sites.get(s.getType()).contains(s) )
```

Domanda d)

Si consideri una classe `MergeableCityMap`, che aggiunge un metodo `void merge(CityMap other)` che aggiunge tutti i punti di interesse presenti in `other`. Può `MergeableCityMap` essere definita come sottoclasse di `CityMap` in accordo con il principio di sostituzione di Liskov? Si motivi la risposta.

Soluzione

`MergeableCityMap` può essere definita come sottoclasse di `CityMap` in quanto aggiunge un nuovo metodo senza modificare metodi presenti. Il nuovo metodo non viola alcuna proprietà della classe `CityMap`: infatti, invocare il metodo `merge` è equivalente ad aggiungere i punti di interesse singolarmente mediante molteplici invocazioni del metodo `addSite`.

Esercizio 2

Si consideri la seguente classe SensorReadings per memorizzare le letture di un sensore ambientale:

```
public class SensorReadings {  
    private int id;  
    private List<Double> data;  
    private int daysInUse;  
    public SensorReadings(int id) {  
        this.id = id;  
        data = new ArrayList<Double>();  
        daysInUse = 0;  
    }  
    public void addReading(double d) {  
        synchronized(data) { data.add(d); }  
    }  
    public double getReading(int index) {  
        synchronized(data) { return data.get(index); }  
    }  
    public void changeAllReadings(List<Double> newData) {  
        synchronized(data) { data = newData; }  
    }  
    public synchronized void newDayStarted() {  
        daysInUse++;  
    }  
}
```

Domanda a)

La classe è correttamente sincronizzata? In caso negativo si spieghi quali siano i problemi e si proponga una soluzione a tali problemi che massimizzi il parallelismo tra i metodi.

Soluzione

L'accesso all'attributo privato `id` non va sincronizzato (l'attributo non cambia mai valore). Problematica è invece la sincronizzazione del metodo `changeAllReadings` che cambia il riferimento all'oggetto `data` su cui avviene la sincronizzazione del metodo stesso e dei metodi `addReading` e `getReading`. Volendo, come richiesto, mantenere la possibilità di accesso parallelo a tali metodi rispetto al metodo `newDayStarted`, si può risolvere introducendo un nuovo attributo `dataLock`, riferimento ad un oggetto che non cambia mai, per poi usare quello come punto di sincronizzazione dei tre metodi: `changeAllReadings`, `addReading` e `getReading`, come proposto nella seguente versione della classe:

```
public class SensorReadings {  
    private int id;  
    private List<Double> data;  
    private Object dataLock;  
    private int daysInUse;  
    public SensorReadings(int id) {  
        this.id = id;  
        data = new ArrayList<Double>();  
        dataLock = new Object();  
        daysInUse = 0;  
    }  
    public void addReading(double d) {  
        synchronized(dataLock) { data.add(d); }  
    }  
    public double getReading(int index) {  
        synchronized(dataLock) { return data.get(index); }  
    }  
    public void changeAllReadings(List<Double> newData) {  
        synchronized(dataLock) { data = newData; }  
    }  
    public synchronized void newDayStarted() {  
        daysInUse++;  
    }  
}
```

Domanda b)

Si aggiunga alla classe `SensorReadings` un nuovo metodo pubblico: `void waitFor30()` che sospende il chiamante se ci sono meno di 30 letture nella lista `data`.

Soluzione

Alla luce delle modifiche proposte nella soluzione alla precedente domanda, il metodo deve essere così formulato:

```
public void waitFor30() throws InterruptedException {
    synchronized(dataLock) {
        while(data.size()<30) dataLock.wait();
    }
}
```

Bisogna inoltre aggiungere la chiamata: `dataLock.notifyAll()` alla fine dei metodi `changeAllReadings` e `addReading`.

Domanda c)

Si aggiunga un nuovo metodo pubblico: `void readyToBeReplaced()` che sospende il chiamante fino a quando non ci siano almeno 10000 letture nella lista `data` oppure siano passati meno di 365 giorni di utilizzo del sensore. Si chiarisca se e come deve cambiare la sincronizzazione degli altri metodi.

Soluzione

La condizione di sospensione riguarda tanto l'attributo `data` quanto l'attributo `daysInUse`, il cui accesso è stato fin qui protetto usando due "lock" diversi (`dataLock` e `this`). Per come funziona la sincronizzazione di Java questo approccio non si può più seguire ed è quindi necessario usare un unico "lock" per tutto. Ad esempio, scegliendo `this`, si può giungere alla seguente soluzione:

```
class SensorReadings {
    private int id;
    private List<Double> data;
    private int daysInUse;
    public SensorReadings(int id) {
        this.id = id;
        data = new ArrayList<Double>();
        daysInUse = 0;
    }
    public synchronized void addReading(double d) {
        data.add(d); notifyAll();
    }
    public synchronized double getReading(int index) {
        return data.get(index);
    }
    public synchronized void changeAllReadings(List<Double> newData) {
        data = newData; notifyAll();
    }
    public synchronized void newDayStarted() {
        daysInUse++; notifyAll();
    }
    public synchronized void waitFor30() throws InterruptedException {
        while(data.size()<30) wait();
    }
    public synchronized void readyToBeReplaced() throws InterruptedException {
        while(data.size()<10000 || daysInUse<365) wait();
    }
}
```

Esercizio 3

Si consideri il seguente metodo:

```
1 public static void test(int x, int y) {  
2     if (x<=0 || y<=0) return;  
3     y=y+x;  
4     while (y>x && y/x>x) {  
5         if (y%2==0)  
6             x++;  
7             y++;  
8     }  
9 }
```

Domanda a)

Si determini un insieme minimo di casi di test per garantire la copertura delle istruzioni (statement coverage).

Soluzione

(0,0) per coprire il return, (1,1) per tutto il resto.

Domanda b)

Si determini un insieme minimo di test per garantire la copertura delle decisioni (edge coverage).

Soluzione

I due test precedenti non coprono il ramo false dell'if più interno. Fermo restando la necessità di mantenere il primo dei due casi precedenti per coprire il ramo true del primo if, si può formulare un unico caso di test che, sfruttando il ciclo, copra i due rami dell'if interno al ciclo, come ad esempio il test (1,2).

Domanda c)

Si determini un insieme minimo di test per garantire la copertura delle decisioni e delle condizioni (edge and condition coverage).

Soluzione

Usando i casi precedenti manca ancora la copertura di due condizioni: Per la prima, nell'if più esterno, basta (1,0); per la seconda, si dovrebbe coprire il caso false di $y > x$, ma questo è impossibile in quanto $y > x$ all'entrata nel ciclo e durante l'esecuzione y aumenta sempre almeno quanto x .

Esercizio 4

Si considerino le seguenti dichiarazioni di classi Java.

```
1 abstract class Vehicle {  
2     abstract void start();  
3 }  
  
4 class Car extends Vehicle {  
5     void start() {  
6         System.out.println("Car started");  
7     }  
8     void start(boolean remoteControl) {  
9         if (remoteControl) {  
10             System.out.print(" Car started with remote");  
11         } else { System.out.print("missing remote, so ");  
12             start();  
13         }  
14     }  
15 }  
  
16 class HybridCar extends Car {  
17     void start() {  
18         System.out.println("Hybrid started");  
19     }  
20 }  
  
21 class ElectricCar extends HybridCar {  
22     void start(boolean remoteControl) {  
23         if (!remoteControl) {  
24             System.out.println("starting normally: ");  
25             super.start();  
26         }  
27         super.start(remoteControl);  
28     }  
29     void start() {  
30         System.out.println("Electric started");  
31     }  
32 }  
  
33 class Main {  
34     public static void main(String[] args) {  
35         Vehicle car = new Car();  
36         HybridCar hybridCar = new HybridCar();  
37         Car electricCar = new ElectricCar();  
38         car.start();  
39         car.start(true);  
40         car.start(false);  
41         hybridCar.start();  
42         hybridCar.start(true);  
43         hybridCar.start(false);  
44         electricCar.start();  
45         electricCar.start(true);  
46         electricCar.start(false);  
47     }  
48 }
```

Domanda a)

Ci sono linee del programma non valide? In caso affermativo indicarle, motivando brevemente la risposta e precisando se causano errori a tempo di compilazione o di esecuzione.

Soluzione

Le righe 7 e 8 non sono valide, in quanto Car ha tipo statico Vehicle, che non ha il metodo start(boolean)

Domanda b)

Eliminate le righe che generano gli errori individuati al punto precedente, indicare cosa stampa il programma, motivando brevemente la risposta. Riportare sempre il numero di riga riportato nel testo.

Soluzione

6 Car started
9 Hybrid started
10 Car started with remote
11 Missing remote, so Hybrid started
12 Electric started
13 Car started with remote
14 Starting normally: Hybrid started
Missing remote, so Electric started