Politecnico di Milano

Ingegneria del Software – a.a. 2006/07

Appello del 6 febbraio 2008

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 15)

Stack è una struttura di dati gestita con una strategia last-in-first-out implementata in Java dalla classe generica **Stack<E>**. Ne assumiamo una sua variante, le cui operazioni sono:

- **Stack()**, costruttore: genera uno stack vuoto.
- **boolean empty()**: ritorna vero se lo stack è vuoto, falso altrimenti
- **E top()**: restituisce l'oggetto in cima allo stack (quello inserito più recentemente).
- **E pop()**: elimina dallo stack l'oggetto in cima, restituendo tale oggetto.
- **E push(E item)**: **i**nserisce un oggetto in cima allo stack e ritorna l'oggetto inserito.
- **int size():** ritorna la dimensione dello stack (numero di oggetti presenti).
- **int search(Object o):** restituisce la posizione nello stack dell'oggetto identico (secondo l'operatore '==') all'oggetto o. La posizione dell'elemento alla base della pila è 0, quella dell'elemento successivo è 1 e così via ; se l'elemento non appartiene allo stack viene restituito -1.

Per semplicità, ipotizziamo che la pila contenga elementi tutti diversi tra loro (pertanto, la *search* trova sempre al più un oggetto identico a quello passato come parametro).

Domanda a

Fornire la specifica JML dei metodi seguenti, aggiungendo opportune eccezioni ove ritenuto necessario. Si supponga che esista un metodo osservatore puro **ArrayList<E> getAbstractRep()**, che fornisce una rappresentazione astratta dello stato corrente della pila: gli elementi della pila sono restituiti in un array-List, con l'elemento alla base della pila in posizione 0 e l'elemento sulla cima in ultima posizione.

```
//@ public invariant this.getAbstractRep() != null &&
        (\forall int i; 0 <= i < this.getAbstractRep().size();
                (\forall int j; i < j <  this.getAbstractRep().size();
                        this.getAbstractRep().get(i) != this.getAbstractRep().get(j) ));

//@ensures this.empty()
public Stack()

//@ensure \result <==> this.size() == 0
public /*@ pure @*/ boolean empty()

//@ requires !this.empty()
//@ ensures \result != null && \result == this.getAbstractRep().get(this.size() - 1)
public /*@ pure @*/ E top()

//@ requires !this.empty()
//@ ensures \result == \old(this.getAbstractRep().get(size() - 1) ) &&
        this.size() ==  \old(this.size()) - 1 &&
        (\forall int i; 0 <= i <  this.getAbstractRep().size();
                \old(this.getAbstractRep().get(i)) == this.getAbstractRep().get(i) );
public E pop()

//@ requires item != null && this.search(item) == -1
//@ ensures this.size() ==  \old(this.size() + 1) && item == this.getAbstractRep().get(size() - 1) &&
        (\forall int i; 0 <= i <  \old(this.size());
                \old(this.getAbstractRep().get(i)) == this.getAbstractRep().get(i) );
public E push(E item)

//@ ensures \result == this.getAbstractRep().size()
public /*@ pure @*/ int size()


\\@ requires o != null
```

```
\\@ ensures result >= -1 &&
        ( \result >= 0 ==> this.getAbstractRep().get(\result) == o ) &&
        ( \result == -1 ==> (\forall int i; 0 < i < this.size(); this.getAbstractRep().get(i) != o) )
public /*@ pure @*/ int search(Object o)
```

Domanda b

Specificare la sottoclasse **TraversableStack<E>** in cui, oltre all'elemento corrente che sta in cima allo stack, è possibile denotarne un altro, tramite il metodo puro **public E current()** che restituisce, in assenza di precedenti chiamate agli operatori up e down (definiti nel seguito) l'elemento che è inserito da più tempo nello stack (l'elemento alla base della pila). Altre due operazioni, **up()** e **down()** consentono di spostare il valore restituito da current. Il metodo **up()** sposta il current di una posizione verso la cima dello stack (ossia, verso il top); il metodo **down()** lo sposta invece verso il basso. L'esecuzione delle operazioni **up()** e **down()** genera un'eccezione quando, al momento della loro chiamata, current restituisce, rispettivamente, l'elemento top o l'elemento inserito da più tempo, oppure quando la pila è vuota. Si noti che anche l'operazione **pop** modifica il riferimento all'elemento current, quando questo, al momento della chiamata di pop, è proprio l'elemento in cima alla pila.

Ricordando che esiste un metodo **ArrayList<E> getAbstractRep()**, che fornisce la rappresentazione astratta dello stato corrente della pila, fornire la specifica JML dei metodi seguenti. Si noti che current è un metodo puro e quindi può essere usato liberamente nelle specifiche che seguono.

```
// TraversableStack, oltre a garantire l'invariante pubblico di Stack, garantisce anche (viene messo in AND con il precedente invariante pubblico):
//@ public invariant !this.emtpy() ==> this.search(this.current()) >= 0

//@ensures this.empty()
public TraversableStack()

//@also
//@ requires !this.empty()
//@ ensures (\result == \old(this.current()) && !this.empty()) ==> this.current() == this.top()
public E pop()

//@ requires !this.empty()
//@ ensures \old(this.current()) != this.top() &&
//@       \old(this.search(this.current())) == this.search(this.current()) + 1 &&
//@       \old(this.size()) == this.size() &&
//@       (\forall int i; 0 <= i < this.size(); \old(this.getAbstractRef().get(i)) == this.getAbstractRef().get(i) )
//@ signals (IllegalStateException e) \old(this.current()) == this.top()
public void up()

//@ requires !this.empty()
//@ ensures \old(this.current()) != this.getAbstractRep().get(0) &&
//@       \old(this.search(this.current())) == this.search(this.current()) - 1 &&
//@       \old(this.size()) == this.size() &&
//@       (\forall int i; 0 <= i < this.size(); \old(this.getAbstractRef().get(i)) == this.getAbstractRef().get(i) )
//@ signals (IllegalStateException e) \old(this.current()) == this.getAbstractRep().get(0)
public void down()
```

Domanda c

Sarebbe possibile definire una sottoclasse di **TraversableStack,** chiamata **CircTraversableStack<E>**, che rispetti il principio di sostituzione, in cui il comportamento di **up()** e **down()** è ridefinito per far sì che, invece di generare eccezioni, lo stack venga attraversato in maniera circolare? Spiegare come definire tale sottoclasse o motivare perchè ciò non è possibile se si vuole rispettare il principio suddetto.

Tale implementazione non sarebbe corretta, secondo la regola dei metodi. Le postcondizioni di up() e down() definite in TraversableStack escludono esplicitamente la possibilità di avere un funzionamento circolare, imponendo che l'elemento corrente non sia rispettivamente il top o il primo elemento dello

stack. In particolare, in questi casi deve essere sollevata l'eccezione IllegalStateException, appunto per segnalare il caso particolare.
Siccome le postcondizioni di up() e down() definite in CircTraversableStack devono essere messe in AND con quelle della TraversableStack, non è possibile ottenere il comportamento desiderato nella classe figlia. Infatti il client della classe potrebbe fare affidamento sul fatto che, in quei casi particolari, venga lanciata un'eccezione. La classe che si vuole realizzare non rispetterebbe tale comportamento, e quindi non è compatibile con la classe TraversableStack.

Tuttavia il normale compilatore Java non solleverebbe alcun problema, in quanto la regola della segnatura è comunque rispettata (i parametri in ingresso / uscita delle funzioni non vengono modificati).

Esercizio 2 (punti 6)

Con riferimento all'esercizio 1, si consideri il seguente metodo statico

Set<int> elementsOf (Stack<int> x); // restituisce l'insieme di valori contenuti nello Stack x

Specificare l'operazione **elementsOf** in funzione delle operazioni dell'interfaccia della classe **Stack<E>**. Per la specifica, ipotizzare che Set abbia un'operazione pubblica boolean **contains()** che restituisce true se l'oggetto passato come parametro appartiene all'insieme; false altrimenti.

```
//@ requires x != null
//@ ensures \result != null && \result.size() == x.size() &&
        (\forall int i; 0 < i < x.size(); \result.contains(x.getAbstractRep().get(i)) )
```

Esercizio 3 (punti 4)

Descrivere con un diagramma delle classi UML la specifica qui riportata, aggiungendo anche tutti e soli gli attributi e metodi rilevanti per la specifica.

Un produttore di materiale informatico offre la possibilità di acquistare materiale via internet. Un cliente può selezionare il materiale desiderato sulla pagina web del produttore. I materiali sono classificati ad esempio in calcolatori, memorie, dischi rigidi, monitor, periferiche audio/video (CD, DVD, ecc). Alcuni componenti (ad es., dischi rigidi, perif. Audio/video e monitor) non sono configurabili, ma altri invece (a esempio i calcolatori) lo sono e richiedono al cliente di selezionare una configurazione standard o di definire interattivamente la configurazione desiderata. Per un calcolatore, la configurazione può includere monitor, dischi, memoria, perifiche A/V. Un calcolatore può essere venduto senza monitor o senza periferiche A/V, ma deve per forza essere venduto con almeno una memoria e un disco.