Astrazioni polimorfe

Perché il polimorfismo

- ✓ non vogliamo definire versioni differenti dell'astrazione corrispondente ad una collezione di elementi
  - quando cambia il tipo degli elementi
  - insiemi di stringhe, insiemi di interi, insiemi di caratteri, etc.
- ✓ possiamo usare astrazioni polimorfe
  - che funzionano per diversi tipi
- ✓ un'astrazione di dato può essere polimorfa
  - rispetto al tipo degli elementi contenuti nei suoi oggetti
  - l'astrazione `Vector` è polimorfa rispetto al tipo dei suoi elementi
- ✓ una procedura o un iteratore possono essere polimorfi
  - rispetto ai tipi di uno o piú dei loro argomenti
  - il metodo per rimuovere un elemento di tipo arbitrario da un vettore

Polimorfismo “vero”?

- ✓ esistono tipi “parametrici”
  - parametri di tipo “tipo” (à la ML)
  - `t set`, `t stack`,
  - il parametro `t` può essere istanziato ad un tipo qualunque
    - producendo una versione del tipo come `int set` o `int stack stack`
- ✓ nelle versioni attuali di Java esistono le classi generiche
  - i tipi sono classi
  - le classi possono avere parametri
    - in particolare parametri di tipo classe
- ✓ questo meccanismo è presente solo a tempo di compilazione
  - il compilatore produce codice analogo a quello che si sarebbe in Java originale, inclusi i controlli

Polimorfismo di Java

- non useremo il meccanismo dei generici
  - non utilizzato nel testo della Liskov
  - dal punto di vista metodologico, può essere interessante vedere come il polimorfismo si può realizzare con le gerarchie di tipi

Polimorfismo in Java

- ✓ espresso attraverso la gerarchia di tipi
- ✓ gli argomenti e le variabili di istanza
  - rispetto ai quali si vuole essere polimorfi
- ✓ vengono dichiarati appartenere ad un supertipo (tipo apparente!)
- ✓ i valori effettivi potranno appartenere ad un qualunque sottotipo
- ✓ il polimorfismo di Java è molto più debole di quello offerto da linguaggi in cui esistono davvero “i tipi polimorfi” (ML)
  - poco supporto da parte del compilatore
  - possibili eccezioni a tempo di esecuzione (casting)

Scelta del supertipo in una astrazione polimorfa

- molto spesso è `Object`
  - come nel caso di `Vector`
  - i metodi dell'astrazione polimorfa devono poter essere definiti utilizzando soltanto i metodi di `Object`
- talvolta è necessario utilizzare altri metodi
  - il supertipo è definito da una apposita interface
    - che prevede tali metodi
    - che definisce i reali vincoli sul tipo degli elementi
- nell'approccio più comune (element subtype)
  - gli elementi sono sottotipi di tale interface
- in un approccio alternativo (related subtype)
  - bisogna definire un sottotipo dell'interface per ogni tipo potenziale di elementi

Sommario

- ✓ astrazioni di dati polimorfe come collezioni di `Object`
  - `Set`: specifica e implementazione
  - problemi relativi all'uguaglianza e contenitori
- ✓ utilizzazione delle astrazioni di dati polimorfe, casting
- ✓ interfacce nell'approccio element subtype
  - `Comparable` e `OrderedList`
- ✓ l' approccio related subtype
  - `Adder` e `SumSet`
- ✓ la combinazione dei due approcci

Astrazioni di dati polimorfe come collezioni di `Object`: `Set`

- ✓ astraiamo in `IntSet` dal tipo degli elementi
- ✓ specifica simile a quella di `IntSet`
  - i metodi accettano oggetti come argomenti e restituiscono oggetti
- ✓ l'overview ci dice che
  - per confrontare oggetti i metodi usano il metodo `equals`
  - l'oggetto `null` non è mai contenuto in `this`

La specifica di `Set`

```
public class Set {
 // OVERVIEW: un Set è un insieme modificabile di Objects. con un numero qualunque
 // di elementi. null non può mai essere elemento di un Set. Si usa equals per
 // determinare l'uguaglianza degli elementi
 // costruttori
 public Set ()
  // EFFECTS: inizializza this all'insieme vuoto
 // metodi
 public void insert (Object x) throws NullPointerException
  // MODIFIES: this
  // EFFECTS: se x è null solleva NullPointerException, altrimenti
  // aggiunge x agli elementi di this
 public void remove (Object x)
  // MODIFIES: this
  // EFFECTS: se x è in this lo rimuove, altrimenti non fa nulla
 public boolean isIn (Object x)
 // EFFECTS: ritorna true se x appartiene a this, altrimenti ritorna false
 public boolean subset (Set s)
 // EFFECTS: ritorna true se tutti gli elementi di this appartengono a s,
 // altrimenti ritorna false
 // specifica di size e elements
}
```

Implementazione di `Set`

```
public class Set {
 private Vector els;
 public Set ( ) { els = new Vector( );  }
 private Set (Vector x) { els = x;  }
 public void insert (Object x) throws NuIlPointerException {
   if (getIndex(x) < 0) els.add(x); }
 private int getIndex (Object x) {
   for (int i = 0; i < els.size(); i++)
        if (x.equals(els.get(i))) return i;
   return -1; }
 public boolean subset (Set s) {
   if (s == null) return false;
   for (int i = 0; i < els.size(); i++);
    if (!s.isIn(els.get(i))) return false;
   return true; }
 public Object clone ( ) { return new Set((Vector) els.clone( )); }
}
```

Implementazione di `Set`: commenti

- il metodo `insert` memorizza nell'insieme l'oggetto e non un `clone` dell'oggetto
  - indicato nella specifica
  - `x`, cioè l'oggetto, è aggiunto all'insieme
- il metodo `clone` non clona gli elementi dell'insieme ma clona il vettore `els`
  - l'insieme clonato condivide gli oggetti con l'insieme che viene clonato

Funzione di astrazione ed invariante di rappresentazione

- ancora molto simili a quelle di `IntSet`
- la funzione di astrazione produce ora gli oggetti in `c.els` invece degli interi

```
//α(c) = { c.els.get(i) | 0 <= i < c.els.size() }
```

- il rep invariant include la condizione che l'insieme non contenga `null` e dice anche che l'uguaglianza degli elementi è controllata dal metodo `equals`

```
// I(c) = c.els != null e
// per ogni intero i, tale che 0<=i<c.els.size())
//                  c.els.get(i) non è null,
// e  per tutti gli interi i,j, tali che
//          0 <= i < j < c. els.size(),
// ! c.els.get(i).equals(c.els.get(j))
```

Uguaglianza 1

- una collezione come `Set` determina se un elemento è membro della collezione usando il metodo `equals`
  - il contenuto di un oggetto del tipo della collezione dipende da come è implementato `equals` per gli elementi della collezione
- esempio: insiemi di `Vector`
  - il metodo `equals` per `Vector` restituisce `true` se i due vettori hanno lo stesso stato
  - può essere complesso fare in modo che vettori distinti vengano comunque visti come elementi distinti dell'insieme

Uguaglianza 2

```
Set s = new Set();
Vector x = new Vector();
Vector y = new Vector();
s.insert(x);
s.insert (y); // y non viene aggiunto ad s perché risulta esserci già
x.add(new Integer(3));
if (s.isIn(y)) // non ci arriva!
```

- ✓ poiché `y` ha lo stesso stato di `x` quando è inserito in `s` non è aggiunto a `s`
- ✓ quando lo stato di `x` cambia, `y` non piú uguale a `x` e la chiamata a `isIn` restituisce `false`

Contenitori

- ✓ quando vogliamo distinguere oggetti distinti con lo stesso stato, possiamo avvolgere gli oggetti in `Containers`
- ✓ un `Container` è non modificabile
- ✓ due `Containers` sono uguali se contengono esattamente lo stesso oggetto
- ✓ anche `Container` è polimorfo

La classe Container

```
public class Container {
// OVERVIEW: un Container contiene un singolo oggetto; due
// Containers sono uguali se contengono lo stesso oggetto; i
// Containers non sono modificabili
 private Object el;
 // costruttore
 public Container (Object x)
 // EFFECTS: fa in modo che this contenga x
  { el = x; }
 // metodi
 public Object get (  )
 // EFFECTS: ritorna l'oggetto contenuto in this
  { return el; }
 public boolean equals (Object x)
 { if (! x instanceOf Container) return false;
   return (el == ((Container) x.el)); } }
```

Uguaglianza 3

- ✓ avvolgendo i vettori nei contenitori, possiamo inserirli nell'insieme anche quando hanno lo stesso stato

```
Set s = new Set ( );
Vector x = new Vector(  );
Vector y = new Vector(  );
s.insert(new Container(x));
s.insert(new Container(y));
x.add(new Integer(3));
if (s.isIn(new Container(y))) // arriva qui
```

- ✓ `s` contiene due elementi, uno per `x` e l'altro per `y`
- ✓ anche se `x` viene modificato, continuiamo a trovare `y` nell'insieme
  - notare che ora passiamo oggetti di tipo `Container` come argomenti ai metodi di `Set`

Utilizzazione delle astrazioni polimorfe

- ✓ nella collezione possono essere messi solo oggetti
  - i valori primitivi devono essere avviluppati nel loro corrispondente tipo oggetto
- ✓ osservatori che restituiscono elementi della collezione restituiscono `Object`
  - occorrerà usare il casting al valore primitivo

```
Set s = new Set();
s.insert(new Integer(3));
…
Iterator g = s.elements();
while (g.hasNext()) {
   int i = ((Integer) g.next()).intValue();
… }
```

Utilizzazione delle astrazioni polimorfe: compilazione e casting

- ✓ tre diversi modi di fare insiemi (omogenei) di interi
  - la classe `IntSet` in Java
    - i metodi prendono come argomenti e ritornano solo interi
    - il tutto è controllato staticamente dal compilatore
  - inserendo `Integer`s nella classe `Set` in Java
    - i metodi devono fare il casting e controllare che la collezione sia omogenea
    - il compilatore non può aiutare
    - gli "errori di tipo" si rilevano come Eccezioni di Cast a tempo di esecuzione
  - istanziando il tipo parametrico `t set` a `int set` in ML
    - il compilatore tratta realmente il tipo parametrico e le sue istanze
    - è in gradi di rilevare staticamente errori di tipo come se avessi `IntSet`

Interfacce nell'approccio element subtype

- ✓ il tipo `Set` e molte altre astrazioni di dati polimorfe applicano ai loro parametri solo metodi di `Object`
- ✓ alcune astrazioni richiedono metodi aggiuntivi
  - supponiamo di voler definire un tipo `OrderedList`
    - versione polimorfa di `OrderedIntList`
  - abbiamo bisogno di ordinare gli elementi
    - `Object` non ha associata nessuna relazione di ordinamento
  - ci serve un supertipo i cui sottotipi abbiano tutti un metodo per il confronto (relazione di ordinamento totale)
    - esiste
    - si chiama `Comparable`
    - è definito in `java.util`

L'interfaccia Comparable

```
public interface Comparable {
// OVERVIEW: i sottotipi di Comparable forniscono un metodo
// per determinare la relazione di ordinamento fra i loro
// oggetti; l'ordinamento deve essere totale e, ovviamente,
// transitivo e simmetrico; infine
// x. compareTo (y) == 0 implica x. equals (y)
public int compareTo (Object x) throws ClassCastException,
   NullPointerException;
// EFFECTS: se x è null, lancia NullPointerException;
// se this e x non sono confrontabili, solleva ClassCastException;
// altrimenti, se this è minore di x ritorna -1;
// se this = x ritorna 0; se this è maggiore di x, ritorna 1
}
```

Sottotipi di `Comparable` ed eccezioni

- ✓ nell'implementazione di `CompareTo` in tutte le classi che implementano `Comparable`, bisogna analizzare un po' di casi eccezionali
  - l'argomento è `null`
  - l'argomento ha un tipo che non è un sottotipo di `Comparable`
  - l'argomento ha un tipo che è un sottotipo di `Comparable`, ma il tipo di `this` e quello dell'argomento sono incompatibili tra loro
    - sia `Integer` che `String` sono sottotipi di `Comparable`
    - `x.compareTo(s)`, con `x Integer` e `s String` non ha senso
- ✓ in tutti questi casi, salvo il primo, `compareTo` deve sollevare `ClassCastException`
- ✓ altra situazioni in cui "errori di tipo" non possono essere scoperti dal compilatore e diventano Eccezioni a run time

La classe `OrderedList`

- ✓ `Comparable` è un supertipo che si assume definito prima dei sottotipi che lo implementano (elementi di `OrderedList`)
- ✓ specifica e implementazione simili a quelle di `OrderedIntList`
  - argomenti e risultati sono `Comparable` invece che `int`
  - il confronto è fatto usando `compareTo`
- ✓ `OrderedList` assicura che gli elementi della lista siano omogenei
  - necessario, perché `compareTo` solleva un'eccezione se gli oggetti non sono confrontabili
- ✓ il tipo degli elementi nella lista è determinato dall'inserimento del primo elemento
  - se la lista diventa vuota il tipo può cambiare con l'aggiunta di un nuovo elemento
- ✓ il metodo `addEl` assicura che il primo elemento sia comparabile rigettando il tentativo di aggiungere alla lista `null`

Specifica e implementazione di `OrderedList` 1

```
public class OrderedList {
// OVERVIEW: `e una lista modificabile ordinata di oggetti di tipo Comparable
// Oggetto tipico [x1, . . . , xn] con xi < xj se i < j
// Il confronto fra gli elementi è effettuato con il loro metodo compareTo
 private boolean empty;
 private OrderedList left, right;
 private Comparable val;
 //  costruttore
 public OrderedList (  )
  // EFFECTS: inizializza this alla lista ordinata vuota
  { empty = true; }
```

Specifica e implementazione di OrderedList 2

```
public class OrderedList {
// OVERVIEW: `e una lista modificabile ordinata di oggetti di tipo Comparable
// Oggetto tipico [x1, . . . , xn] con xi < xj se i < j
// Il confronto fra gli elementi è effettuato con il loro metodo compareTo
 private boolean empty;
 private OrderedList left, right;
 private Comparable val;
// metodi
 public void addEl (Comparable el) throws NullPointerException,
   DuplicateException, ClassCastException
    // MODIFIES: this
    // EFFECTS: se el è in this, solleva DuplicateException; se el è null
    // solleva NullPointerException; se el non è confrontabile con gli altri
    // elementi in this solleva ClassCastException; altrimenti, aggiunge el a
    // this
  {if (el == null) throw new NullPointerException("OrderedList.addEl”);
  if (empty) { left = new OrderedList(); right = new OrderedList ();
    val = el; empty = false; return; }
  int n = el.compareTo(val);
  if (n = 0) throw new DuplicateException("OrderedList.addEl");
  if (n < 0) left.addEl(el); else right.addEl(el) ; }
```

Specifica e implementazione di OrderedList 3

```
public class OrderedList {
// OVERVIEW: `e una lista modificabile ordinata di oggetti di tipo Comparable
// Oggetto tipico [x1, . . . , xn] con xi < xj se i < j
// Il confronto fra gli elementi è effettuato con il loro metodo compareTo
 private boolean empty;
 private OrderedList left, right;
 private Comparable val;
// metodi
 public void remEl (Comparable el) throws NotFoundException
// MODIFIES: this
// EFFECTS: se el non è in this, solleva NotFoundException;
// altrimenti, rimuove el da this

 public boolean isIn (Comparable el)
// EFFECTS: se el è in this ritorna true altrimenti ritorna false
}
```

Interfacce nell'approccio related subtype

- ✓ nell'approccio element subtype
  - definiamo l'interfaccia che definisce le proprietà del tipo polimorfo
  - realizziamo gli oggetti come istanze di sottotipi di tale interfaccia
    - i tipi vanno progettati "a priori"
- ✓ talvolta un tipo polimorfo collezione è definito dopo che già esistono i tipi per gli elementi desiderati
  - abbiamo bisogno di un diverso modo per accedere i metodi usati nella collezione
- ✓ nell'approccio related subtype
  - definiamo un'interfaccia i cui oggetti hanno i metodi richiesti
  - gli oggetti non sono istanze di sottotipi dell'interfaccia
  - i tipi degli oggetti possono essere definiti prima dell'interfaccia
  - per ogni tipo di elementi "preesistente", definiamo un opportuno sottotipo dell'interfaccia "a posteriori"

Interfacce nell'approccio related subtype

- nell'approccio related subtype
  - definiamo un'interfaccia i cui oggetti hanno i metodi richiesti
  - gli oggetti non sono istanze di sottotipi dell'interfaccia
  - i tipi degli oggetti possono essere definiti prima dell'interfaccia
  - per ogni tipo di elementi "preesistente", definiamo un opportuno sottotipo dell'interfaccia "a posteriori"
- esempio
  - supponiamo di voler definire un insieme (polimorfo) che mantiene l'informazione sulla somma degli elementi
    - per far questo il tipo polimorfo (operazioni `insert` e `remove`) deve poter accedere i metodi che il tipo degli elementi deve avere per sommare e sottrarre valori
  - il primo passo è la definizione di una interfaccia `Adder`, che ha due operazioni per sommare e sottrarre

L'interfaccia Adder

```
public interface Adder {
// OVERVIEW: tutti i sottotipi di Adder forniscono metodi per
// sommare e sottrarre gli elementi di un "tipo collegato"
 public Object add (Object x, Object y) throws
   NullPointerException, ClassCastException;
 // EFFECTS: se uno tra x o y è null, solleva
 // NullPointerException; se x e y non sono sommabili solleva
 // ClassCastException; altrimenti ritorna la somma di x e y
 public Object sub (Object x, Object y) throws
   NullPointerException, ClassCastException;
 // EFFECTS: se uno tra x o y è null, solleva
 // NullPointerException; se x e y non sono sommabili solleva
 // ClassCastException; altrimenti ritorna la differenza tra x e y
 public Object zero ( );
 // EFFECTS: ritorna l'oggetto che rappresenta lo zero per il
 // tipo collegato
}
```

L'approccio related subtype

- ✓ supponiamo di voler definire un insieme (polimorfo) che mantiene l'informazione sulla somma degli elementi
  - abbiamo definito un'interfaccia `Adder` i cui oggetti hanno i metodi richiesti
  - gli oggetti dell'insieme non sono istanze di sottotipi dell'interfaccia
  - ci interessa mettere nell'insieme oggetti di tipo `Poly`
  - definiamo un sottotipo di `Adder` collegato a `Poly`
    - che ha le operazioni per sommare e sottrarre `Poly`s
    - del sottotipo non occorre dare la specifica perché è un sottotipo di `Adder`

Il sottotipo di `Adder` collegato a `Poly`

```
public class PolyAdder implements Adder     {
 private Poly z;        // il Poly zero
 public PolyAdder ( ) {z = new Poly();  }
 public Object add (Object x, Object y) throws
   NullPointerException, ClassCastException {
  if (x == null || y == null)
    throw new NullPointerException ("PolyAdder.add");
  return ((Poly) x).add((Poly) y); }
public Object sub (Object x, Object y) throws
   NullPointerException, ClassCastException {
  if (x == null || y == null)
    throw new NullPointerException ("PolyAdder.sub");
  return ((Poly) x).sub((Poly) y); }
public Object zero ( )  { return z; }
}
```

✓ abbiamo messo lo zero nella rep

- potevamo generarne uno ogni volta che ci serviva lo zero

Relazione tra `PolyAdder` e `Poly`

- ✓ i metodi sono diversi (in questo caso solo nella segnatura)
  - `Object add (Object x, Object y)`
    - `Poly add (Poly x)`
  - `Object sub (Object x, Object y)`
    - `Poly sub (Poly x)`
- ✓ si può definire un `IntegerAdder` che aggiunge e sottrae `Integer`
  - anche se gli `Integer`s non hanno nessun metodo aritmetico

Il tipo `SumSet`

- insieme di oggetti che tiene traccia della somma degli oggetti che sono attualmente in esso contenuti
  - gli oggetti devono essere “sommabili”
  - esempio di uso dell’interfaccia `Adder`
  - il tipo degli elementi dell’insieme è determinato quando viene creato l’insieme mediante l’oggetto `Adder` che è un argomento del costruttore

Specifica e implementazione di SumSet 1

```
public class SumSet {
// OVERVIEW: un SumSet è un insieme modificabile di oggetti
   che mantiene la somma degli elementi nell'insieme. La somma
   è calcolata usando un oggetto Adder.
 private Vector els; // gli elementi
 private Object s; // la somma degli elementi
 private Adder a; // l'oggetto usato per fare i conti
// costruttore
 public SumSet (Adder p) throws NullPointerException
// EFFECTS: this diventa l'insieme vuoto di elementi del tipo
// collegato a p, valore iniziale della somma = p.zero()
   { els =new Vector( ); a = p; s = p.zero ( ); }
```

Specifica e implementazione di SumSet 2

```
private Vector els; // gli elementi
private Object s; // la somma degli elementi
private Adder a; // l'oggetto usato per fare i conti
public void insert (Object x) throws
  NullPointerException, ClassCastException
// MODIFIES: this
// EFFECTS: se x è null solleva NullPointerException, se x
// non è sommabile agli altri elementi di this, solleva
// ClassCastException; altrimenti aggiunge x a this e
// ricalcola la somma
{ Object z = a.add(s, x); int i = getIndex(x);
  if (i < 0) { els.add(x); s = z; }  }
public Object sum = ( )
// EFFECTS: ritorna la somma degli elementi di this
  { return s; }  }
```

Utilizzazione di `SumSet`

```
Adder a = new PolyAdder ( );
SumSet s = new Sumset (a);
s.insert(new Poly(3, 7));
s.insert(new Poly(4, 8));
Poly p = (Poly) s.sum ( );
```

- ✓ `SumSet` è scomodo da usare perché dobbiamo definire un sottotipo di `Adder` per ogni tipo di elementi
- ✓ può essere utile combinare l'approccio "related subtype" con quello "element subtype"
  - per esempio combinare `Adder` con un tipo come `Comparable`

La combinazione dei due approcci

```
public interface Adder {
 public Object add (Object x, Object y) throws NullPointerException,
   ClassCastException;
 public Object sub (Object x, Object y) throws NullPointerException,
   ClassCastException;
 public Object zero ( );}
public interface Addable {
 public Object add (Object x) throws NullPointerException,
   ClassCastException;
 public Object sub (Object x) throws NullPointerException,
   ClassCastException;
 public Object zero ( );}
```

- ✓ gli elementi di `SumSet` possono essere
  - sottotipi di `Addable`
    - definiti dopo `Addable`
    - forzati a implementare anche le operazioni di `Addable`
  - tipi per cui abbiamo definito un tipo collegato sottotipo di `Adder`
- ✓ due costruttori corrispondenti in `SumSet`

La combinazione dei due approcci nelle collezioni di `java.util`

- ✓ alcuni dei tipi polimorfi lì definiti usano insieme le due interfacce

```
public interface Comparator {
public int compare (Object x, Object y) throws
   ClassCastException, NullPointerException;
// EFFECTS: se x o y è null, lancia NullPointerException;
// se x e y non sono confrontabili, solleva ClassCastException;
// altrimenti, se x è minore di y ritorna -1;
// se x = y ritorna 0; se x è maggiore di y, ritorna 1
}

public interface Comparable {
public int compareTo (Object x) throws ClassCastException,
   NullPointerException;}
```

Procedure polimorfe

- ✓ stesse tecniche anche per rendere polimorfa l'astrazione procedurale
- ✓ si astrae dal tipo dei parametri formali
- ✓ per l'implementazione, stesse possibilità dell'astrazione sui dati
  - utilizza solo i metodi che tutti gli oggetti hanno, cioè quelli definiti da `Object`
  - usa un'interfaccia
    - supertipo del tipo dei parametri (element subtype)
    - supertipo di un tipo collegato al tipo dei parametri (related subtype)

Due `sort` polimorfi

- ✓ due metodi (specifiche solo!) `sort` che ordinano vettori
  - la prima funziona se gli elementi del vettore appartengono a sottotipi di `Comparable`
  - la seconda prende come argomento un `Comparator`

```
public static sort (Vector v) throws ClassCastException
 // MODIFIES: v
 // EFFECTS: se v non è null, lo ordina in modo crescente usando
 // il metodo compareTo di Comparable; se alcuni elementi di v sono
 // null o non confrontabili solleva ClassCastException
public static sort (Vector v, Comparator c) throws
   ClassCastException
 // MODIFIES: v
 // EFFECTS: se v non è null, lo ordina in modo crescente usando
 // il metodo compare di c; se alcuni elementi di v sono
 // null o non confrontabili solleva ClassCastException
```