

Astrazione sui dati

Tipi di dati astratti

Varie forme di astrazione

Astrazione: nozione ricorrente in questo corso

- ▶ ignorare i dettagli per
 - ▶ mettere in luce gli aspetti significativi
 - ▶ gestire la complessità
- ▶ Astrazione sul controllo
 - ▶ nasconde l'implementazione nel corpo di procedure
- ▶ Astrazione sui dati
 - ▶ nasconde decisioni sulla rappresentazione delle strutture dati e sull'implementazione delle operazioni
 - ▶ nasconde rappresentazione ma anche codice
 - ▶ Esempio: una coda FIFO realizzata mediante
 - ▶ una lista
 - ▶ un vettore

Le parti di un'astrazione

- ▶ **Componente**: nome dell'oggetto che vogliamo astrarre,
 - ▶ es.: struttura dati, funzione, modulo
- ▶ **Interfaccia**
 - ▶ modi per interagire, dall'esterno, con il componente
- ▶ **Specifica** (formale o informale)
 - ▶ descrizione del comportamento atteso,
 - ▶ proprietà osservabili attraverso l'interfaccia
 - ▶ possibile descrizione formale: insieme di equazioni
 $\text{head} (\text{cons } x \text{ } xy) = x$
- ▶ **Implementazione**
 - ▶ definizione dei dati, codice delle funzioni
 - ▶ interni alla componente
 - ▶ invisibili all'esterno
 - ▶ dall'interfaccia non posso accedere all'implementazione
 - ▶ protetta da una capsula che la isola

Esempio di astrazione sui dati: coda di priorità

- ▶ Componente

- ▶ Coda a priorità: struttura dati che restituisce elementi in ordine decrescente di priorità

- ▶ Interfaccia

- ▶ Tipo `PrioQueue`

- ▶ Operazioni

```
empty      : PrioQueue
insert     : ElemType * PrioQueue → PrioQueue
getMax     : PrioQueue → ElemType * PrioQueue
```

- ▶ Specifica

- ▶ `insert` aggiunge un elemento alla coda
- ▶ `getMax` restituisce l'elemento di massima priorità e la coda degli elementi rimanenti

Esempio di astrazione sui dati: coda di priorità

- ▶ Implementazione
varie alternative possibili
 - ▶ un albero binario di ricerca
 - ▶ red and black tree
 - ▶ un vettore parzialmente ordinato

Secondo esempio: gli Interi

- ▶ **Componente**
 - ▶ tipo di dato `integer`
- ▶ **Interfaccia**
 - ▶ costanti, operazioni aritmetiche
- ▶ **Specifica**
 - ▶ le leggi dell'aritmetica
 - ▶ range di numeri rappresentabili
- ▶ **Implementazioni**
 - ▶ hardware: complemento a due, complemento a uno, lunghezza fissa
 - ▶ software: interi di dimensione arbitraria
- ▶ **Tipo di dato astratto?**
 - ▶ C: no, posso accedere all'implementazione (attraverso le operazioni booleane)
 - ▶ Scheme: sì, sugli interi solo operazioni aritmetiche

Tipi di dato astratti (Abstract Data Type)

- ▶ Le idee precedenti, definizione di “interfaccia, specifica, implementazione”
le posso applicare in altri ambiti
 - ▶ es. funzioni polimorfe
 - ▶ es. algoritmo di ordinamento,
 - ▶ specifico l'interfaccia degli oggetti su cui opera
 - ▶ nascondo l'implementazione
- ▶ applicate alle strutture dati definiscono gli ADT
- ▶ idee nate negli anni '70
- ▶ riassumendo:
 - ▶ si separa l'interfaccia dall'implementazione
- ▶ si usano meccanismi di **visibilità** per garantire la separazione
 - ▶ dall'esterno accesso alle sole operazioni dell'interfaccia
 - ▶ l'implementazione è incapsulata in opportuni costrutti opachi (ADT)

Un ulteriore esempio:

- ▶ Tipo di dato Set
con funzioni: `empty`, `insert`, `union`, `is_member?`, ...
- ▶ Set implementato come:
 - ▶ vettore
 - ▶ lista concatenata
 - ▶ tabella hash
 - ▶ ...

ADT per una pila di interi

```
abstype Int_Stack{
  type Int_Stack = struct{
      int P[100];
      int n;
      int top;
  }
  signature
  Int_Stack crea_pila();
  Int_Stack push(Int_Stack s, int k);
  int top(Int_Stack s);
  Int_Stack pop(Int_Stack s);
  bool empty(Int_Stack s);
  operations
  Int_Stack crea_pila(){
      Int_Stack s = new Int_Stack();
      s.n = 0;
      s.top = 0;
      return s;
  }
  Int_Stack push(Int_Stack s, int k){
      if (s.n == 100) errore;
      s.n = s.n + 1;
      s.P[s.top] = k;
      s.top = s.top + 1;
      return s;
  }
  int top(Int_Stack s){
      return s.P[s.top];
  }
  Int_Stack pop(Int_Stack s){
      s.n = s.n - 1;
      s.top = s.top - 1;
      return s;
  }
}
```

Un'altra implementazione per una pila di interi

```
abstype Int_Stack{
  type Int_Stack = struct{
      int info;
      Int_stack next;
  }
  signature
  Int_Stack crea_pila();
  Int_Stack push(Int_Stack s, int k);
  int top(Int_Stack s);
  Int_Stack pop(Int_Stack s);
  bool empty(Int_Stack s);
  operations
  Int_Stack crea_pila(){
      return null;
  }
  Int_Stack push(Int_Stack s, int k){
      Int_Stack tmp = new Int_Stack(); // nuovo elemento
      tmp.info = k;
      tmp.next = s; // concatenalo
      return tmp;
  }
  int top(Int_Stack s){
      return s.info;
  }
  Int_Stack pop(Int_Stack s){
      return s.next;
  }
  bool empty(Int_Stack s){
      return (s == null);
  }
}
```

Principio di incapsulamento

- ▶ Indipendenza dalla rappresentazione
 - ▶ due implementazioni corrette di un tipo (astratto) non sono distinguibili dal contesto (restante parte del programma)
- ▶ Le implementazioni sono modificabili senza con ciò interferire col contesto
 - ▶ i contesti non hanno alcun modo per accedere all'implementazione
 - ▶ le interfacce si simulano

Un linguaggio fornisce gli ADT se possiede meccanismi per realizzare questo **nascondimento** dell'informazione (information hiding)

Classi e oggetti.

Linguaggi a oggetti forniscono gli ADT:

- ▶ Il principio dell'incapsulamento (information hiding) è uno delle idee (ma non l'unica) alla base della programmazione OO.
- ▶ Attraverso le classi posso implementare un ADT.
 - ▶ variabili di istanza (rappresentazione interna) vengono tutte dichiarate private,
 - ▶ solo i metodi che implementano l'interfaccia sono pubblici,
 - ▶ rispetto ad altre implementazioni degli ADT una diversa sintassi per chiamare i metodi
 - ▶ `push(pila, elemento)`
 - ▶ `pila.push(elemento)`

- ▶ Costrutto generale per lo information hiding
 - ▶ disponibile in linguaggi imperativi, funzionali, ecc.
 - ▶ uso banale: evitare conflitti di nomi
- ▶ Supporta la programmazione in the large.
 - ▶ moduli distribuiti su file diversi
 - ▶ compilazione separata
- ▶ interfaccia, più sofisticata

- ▶ Dentro un modulo:
 - ▶ definiti i nomi visibili all'esterno, alcuni in maniera ristretta
 - ▶ variabili in solo lettura
 - ▶ tipi esportati in maniera *opaca*, usabili ma non esaminabili visibile il nome ma non la definizione
 - ▶ definito cosa voglio importare dagli altri moduli, ogni modulo, nel preambolo, specifica:
 - ▶ gli altri moduli che vuole importare, e nel dettaglio per ciascun modulo importato, seleziona
 - ▶ le parti dei moduli che vuole importare

- ▶ In quasi tutti i linguaggi, con nomi diversi:
 - ▶ module: in Haskell, Modula, uno dei primi linguaggi ad usarli, ...
 - ▶ package: in Ada, Java, Perl, ...
 - ▶ structure: in ML, ...

Moduli in Haskell

Definizione di modulo

```
module BinarySearchTree (Bst (Null, Bst), insert, empty, in
  where
data Bst = ...
insert = ...
empty  = ...
...
```

- ▶ la lista `(Bst (Null, Bst), insert, empty)` definisce in nomi esportati
- ▶ definizione che non appaiono nella lista non visibili all'esterno
- ▶ scelta migliore se voglio implementare un ADT
 - ▶ `Bst(Null)` costruttore `Bst` non visibile all'esterno

Per importare

```
module Main (main)
  where
import BinarySearchTree (Bst (Null), insert, in)
```

- ▶ le dichiarazioni di `import` vanno messe in testa
- ▶ posso selezionare i nomi da importare