Corso di Linguaggi di Programmazione

Lezione 16

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Informatica
Università degli Studi di Milano

30 Aprile 2013

Università degli Studi di Milano

Programming in the large

- Progetto ed analisi di algoritmi: problemi piccoli, ma complessi, da risolvere con programmi compatti
- Situazioni reali: problemi grandi, composti da molti sottoproblemi, molti programmatori coinvolti, ecc.
- ingegneria del software per *risolvere* i problemi (natura tecnica e manageriale)
- i linguaggi di programmazione devono soddisfare certi requisiti per fornire un supporto all'ingegneria del software.
- Obiettivi: manutenibilità, riuso, portabilità ...
- Principali caratteristiche coinvolte: *factoring*, *locality*
- Principi fondamentali: *astrazione* e *modularità*

Process abstraction

L'esempio più semplice di astrazione è il sottoprogramma:

sortInt(list, listLen)

è un'astrazione del processo di ordinamento.

Data abstraction

L'esempio più semplice di data abstraction sono i tipi built-in (es. floating point):

il programmatore può operare su floating point tramite i costrutti del linguaggio (+, / ...) e dimenticarsi cosa succede a livello di bit.

questo tra l'altro assicura la *portabilità* del codice.

User–defined ADT

Definition (User–defined Abstract Data Types)

L'ADT è un *tipo di dati* che soddisfa i seguenti requisiti:

- La dichiarazione del *tipo* e delle *operazioni* sugli elementi del tipo sono contenute in una *singola unità sintattica*.
- La rappresentazione di elementi del tipo è *nascosta* dalle unità che usano il tipo, quindi le *uniche* operazioni possibili sugli elementi del tipo sono quelle fornite nella dichiarazione.

digressione: unità sintattiche

- Domanda: in che senso "unità sintattica"?
- Esempio: 'package' ADA
- Esempio (non del tutto preciso): 'Classi' Java, C++ o C#
- Il problema delle unità di compilazione verrà affrontato più avanti nel corso ...
- ... per ora ci concentriamo sul concetto di astrazione di dati (tipi) e processo (operazioni)

Caratteristiche degli ADT

- l'ADT è descritto da un'*interfaccia*
- L' *implementazione* (definizione) può essere nella stessa unità di compilazione (file) o in unità diverse.
- Anche altre unità devono poter creare variabili di quel tipo.

- **Information hiding:** la rappresentazione interna (implementazione) deve essere ininfluente per il resto del programma.
- **Encapsulation:** le uniche operazioni accessibili da codice esterno all'unità di definizione sono quelle dichiarate nell'interfaccia.

Esempio di dichiarazione (interfaccia) di un ADT

Tipo: stack
Operazioni:

- create() $\rightarrow$ stack
- destroy(stack)
- empty(stack) $\rightarrow$ bool
- push(stack, element) $\rightarrow$ stack
- pop(stack) $\rightarrow$ stack
- top(stack) $\rightarrow$ element

Esempio di utilizzo di un ADT

```
...
s = create();
s = push(s, 21);
s = push(s, 3);
if ( ! empty(s) ) write top(s);
...
```

Esempio di implementazione di un ADT

- stack $\rightarrow$ array ed un intero 'puntatore' alla prima componente libera
- create(): alloca l'array e ne restituisce il puntatore alla prima componente
- destroy(stack): rilascia la memoria riservata per l'array
- empty(stack): il puntatore è 0?
- ...

Esempio di implementazione alternativa di un ADT

- stack $\rightarrow$ lista a puntatori
- create(): restituisce un puntatore NULL
- destroy(stack): dealloca tutti gli elementi della lista
- empty(stack): il puntatore è NULL?
- push: crea nuovo elemento della lista

N.B. Il codice 'client' non cambia!
N.B. Ogni sottoprogramma potrebbe essere implementato secondo paradigmi diversi ("fundamental problems").

Alcuni esempi su linguaggi imperativi (CODICE).

Encapsulation ed Information Hiding

- solo attraverso utilizzo di file separati
- minima (dato che sempre possibile modificare all'esterno i campi del record!)
- nessun modificatore di scope
- l'implementazione è solo parzialmente nascosta
- ...

Encapsulation ed Information Hiding, es.2

- solo attraverso utilizzo di file separati
- modificatore di scope `static`
- al prezzo di avere una sola istanza dello stack
- non c'è name encapsulation
- implementazione nascosta
- ...

Vantaggi e svantaggi

Articolato, poco naturale e per niente sicuro: praticamente, non un ADT.

Encapsulation

- `package` in due parti (specification e body)
- tre modificatori di scope: `public`, `private` (continua ad avere assegnamento e confronti) e `limited private` (anche assegnamento e confronti devono essere "ridefiniti")
- `with` e `use`: naming encapsulation
- si utilizza anche la notazione Package.metodo
- esistono anche packages senza body (solo tipi e costanti)
- un programma che ha accesso al package ha accesso a tutte le sue entità `public`

Information Hiding

- specification e body condividono lo stesso nome, ma possono essere in files separati
- `body` discrimina tra body e specification
- posso compilare separatamente, ma specification prima

Vantaggi e svantaggi

- flessibile, potente e completo
- codice prolisso
- molte clausole diverse, difficile da padroneggiare

Encapsulation

- sia variabili che funzioni (data members e member functions, o properties e methods o . . .)
- stack o heap dynamic, anche con heap-dynamic data members
- tre modificatori di scope: `public`, `private` (e `protected`)
- gestione tramite puntatori, allocazione / deallocazione esplicita
- utilizza `::` per (parziale) naming encapsulation
- struct / class è un *tipo*: un programma che ha accesso alla unit può accedere alle sue entità pubbliche solo *dopo aver istanziato*.
- utilizza `namespace` e `using`.

Information Hiding

- il corpo di ogni metodo può apparire con il prototipo, *nel qual caso diviene 'inline'*
- oppure separatamente (nello stesso o in un altro file).
- ha *costruttori* e *distruttori*

Vantaggi e svantaggi

- supporto completo per l'astrazione
- codice compatto, leggibile ed efficiente
- le struct sono sempre tipi di dati, mentre i package Ada sono entità più generiche
- poco ortogonale rispetto ad altri aspetti del linguaggio ...

Encapsulation

- Molto simile a C++
- tre modificatori di scope: `public`, `private` (e `protected`)
- differenza di 'design': non ci sono sezioni public e private, ma ogni proprietà e metodo ha la sua clausola
- naming encapsulation tramite `import` e sistema gerarchico dei packages

Information Hiding

- Java ha *solo* classi (nessun sottoprogramma al di fuori dei metodi della classe)
- a meno di ricorrere a `interface`, ogni metodo deve essere definito al momento della dichiarazione (non è possibile definire il corpo altrove)
- niente *distruttori* (garbage collection)

Vantaggi e svantaggi

- Più ortogonale ed essenziale del C++
- codice più compatto (garbage collection) (vant o svant?)
- gestione naming più articolata
- meno potente (niente inline, definizioni e compilazioni separate ...)

Encapsulation ed Information Hiding

- cinque modificatori di scope: `private` , `public` , `protected`, `internal`, `protected internal`
- come Java, ogni classe è heap dynamic (ma ogni struct stack dynamic)
- codice compatto come in Java (still garbage collection)
- ha distruttori (usati raramente)
- accessors e mutators get() e set() per gestire le proprietà nell'assegnamento (l-value e r-value)
- per il resto analogo a Java ...

```
public class Weather
    public int DegreeDays {
        get {
            return degreeDays;
        }
        set {
            degreeDays = value;
        }
    }
    private int degreeDays;
    ...

...
w.DegreeDays = 10;
i = w.DegreeDays
...
```

Encapsulation ed Information Hiding

- modificatori di scope impliciti *nei nomi degli identificatori*
- supporto completo per encapsulation
- naming encapsulation tramite utilizzo di moduli separati e parole chiave `from` e `import`
- codice molto compatto (scripting)
- information hiding "parziale" (non ho strumenti per definire interfacce)

Riassumendo:

Encapsulation:

- catturare più errori in fase di compilazione – analisi semantica (anche compilazioni separate!)
- aumentare la facilità di manutenzione del software (locality)
- fornire strumenti per condividere packages e librerie in modo "sicuro" (protetto da accessi esterni, soprattutto accidentali)

Modularità:

- Avere ADT in files diversi consente unità di compilazione separate
- Avere interfaccia ed implementazione in files diversi consente di avere le specifiche descritte a parte