10.1 La specifica dei tipi astratti nella fase di concettualizzazione

La modularizzazione per tipo astratto in fase di concettualizzazione conduce il progettista alla individuazione dei tipi di dato che si ritengono interessanti nel problema in esame. L'interpretazione intuitiva di tipo di dato è quella che assegna a tale termine il significato di classe di oggetti su cui è possibile eseguire un insieme prefissato di operazioni. Il problema che affrontiamo in questo paragrafo riguarda la necessità di fornire, durante la fase di concettualizzazione, una specifica sufficientemente precisa di un tipo astratto di dato. Le considerazioni che abbiamo svolto sui princìpi di progettazione evidenziano che tale caratterizzazione deve soddisfare le seguenti condizioni:

- la descrizione del tipo sia chiara e non ambigua;
- le caratteristiche del tipo vengano specificate in modo completamente indipendente dalla rappresentazione dei valori e dalla realizzazione delle operazioni.

Ricordiamo che un tipo astratto di dato è un oggetto matematico costituito da: un insieme, detto il dominio del tipo, che raccoglie i valori del tipo, ed un insieme di funzioni, che corrispondono alle operazioni che si possono effettuare sui valori del tipo.

Un semplice esempio di tipo astratto è il tipo *Boolean*, in cui il dominio è formato dai due valori di verità (che possiamo chiamare "true" e "false") e le operazioni sono "or", "and" e "not", con il significato usuale.

Un esempio più complesso è il tipo astratto *Pila*, in cui il dominio, che chiameremo *Pile*, è formato da sequenze di elementi di un certo dominio T, e le operazioni sono:

- $PilaVuota$, che restituisce il valore corrispondente alla sequenza vuota.
- $EstVuota(p)$, che restituisce true se la pila p è il valore corrispondente a pila vuota, false altrimenti.
- $Push(p, e)$, che restituisce la pila ottenuta dalla pila p inserendo l'elemento e, che diventa il suo elemento affiorante.
- $Pop(p)$ restituisce la pila ottenuta dalla pila p eliminando l'elemento affiorante.
- $Top(p)$ restituisce l'elemento affiorante della pila p.

Quest'ultimo esempio ci chiarisce diversi aspetti che la definizione di tipo astratto presentata in precedenza tende a semplificare:

- in genere un tipo astratto è costituito da più domini, oltre al dominio del tipo (il cosiddetto dominio di interesse). Nel caso della pila, i domini in gioco sono: *Pile*, il dominio del tipo boolean, e T, il dominio degli elementi che formano le le pile;
- le funzioni possono essere definite su diversi domini, ossia la loro applicazione può richiedere zero, uno o più argomenti, e restituiscono un valore di un certo dominio. Inoltre, la definizione delle funzioni richiede una descrizione della legge con cui la funzione calcola il risultato in dipendenza degli argomenti.

12.2 Incapsulamento, information hiding, parametrizzazione e overloading

In questo paragrafo richiamiamo alcuni degli strumenti introdotti nel capitolo 4 ed analizziamo come il C++ permetta di superare le limitazioni di cui abbiamo parlato a proposito di realizzazioni di tipi astratti in PASCAL.

La migliore qualità delle realizzazioni di tipi astratti in C++ rispetto a linguaggi come il PASCAL viene ottenuta attraverso quattro meccanismi fondamentali:

- *incapsulamento* del tipo astratto in una classe;
- *information hiding* per nascondere all'esterno la rappresentazione dei valori del tipo astratto;

- *parametrizzazione* della definizione della classe rispetto ad eventuali altri tipi di dato;
- *overloading degli operatori* al fine di consentire l'uso di operatori sugli oggetti della classe.

L'uso di questi meccanismi suggerisce un semplice schema nella progettazione di una classe C che realizzi un tipo astratto T. Supponiamo che il tipo astratto T sia definito nella specifica in modo parametrico rispetto ad un generico tipo E. La definizione della classe C avrà allora la forma:

```
// File C.h
#ifndef C_H
#define C_H

template <class E>
class C
{  public:
    // funzioni corrispondenti alle operazioni del tipo astratto

    // eventuali funzioni speciali, ovvero overloading di
    // operatori quali "=", "==", costruttori, distruttore, ...
   private:
    // campi per la rappresentazione dei valori del tipo astratto

    // funzioni di servizio
 };
#endif

// File C.cpp
#include "C.h"

// definizione delle funzioni della classe C
```

La classe viene definita, come al solito, in due file: uno con estensione `.h` con la dichiarazione della classe, ed uno con estensione `.cpp` con la definizione delle funzioni. I moduli clienti della classe dovranno semplicemente includere il file con estensione `.h`. Commentiamo lo schema di definizione della classe.

- La classe raccoglie tutte e sole le caratteristiche del tipo. Nella parte pubblica si definiscono tutte funzioni necessarie per realizzare le operazioni del tipo astratto. La parte privata viene usata per nascondere all'esterno:
  1. le strutture di dati necessarie per rappresentare i valori del tipo astratto, in modo che nessun cliente della classe possa accedervi;
  2. le funzioni di servizio che operano su tali strutture.
- La possibilità di eseguire l'overloading degli operatori viene sfruttata per associare alla classe le cosiddette funzioni speciali, cioè i costruttori, il distruttore, l'operatore di assegnazione, l'operatore di verifica di uguaglianza ecc. Ricordiamo a questo proposito che ad una classe C++ viene assegnato automaticamente un costruttore standard (senza argomenti), un costruttore di copia standard, un distruttore standard, e l'operatore di assegnazione. La ridefinizione di queste

funzioni, così come la definizione di ulteriori operatori, consente la loro personalizzazione rispetto al tipo T. Alcune volte le funzioni speciali sono esterne alla classe, ed in particolare `friend`.

- La possibilità di usare la clausola `template` nella definizione della classe viene sfruttata per realizzare la classe in modo parametrico rispetto al tipo E. Ovviamente, la clausola `template` mancherà nel caso in cui il tipo T non sia specificato in modo parametrico rispetto ad un altro tipo.

Si noti che la classe `collezione<T>`, introdotta nel paragrafo 4.22, ha una struttura che rispetta le caratteristiche appena definite.

Esemplificheremo questi aspetti nel seguito riferendoci alla realizzazione del tipo *Insieme*(*Elem*) mediante una classe `Insieme`, resa parametrica mediante l'uso di template rispetto a un generico tipo `Elem`. Nella realizzazione di tale classe utilizzeremo anche la possibilità di effettuare l'overloading di diversi operatori.