Linguaggi

L'analisi e il design a oggetti sono indispensabili per qualsiasi progetto che richieda l'uso di un linguaggio object-oriented. Nel presente articolo viene mostrato come, accanto ai pattern, anche la programmazione generica, ossia l'uso dei template C++, possa aiutarci nella progettazione del codice

Pattern e template nei progetti C++: un case study

gdestri@programmers.net di Giulio Destri

È laureato in Ingegneria Elettronica e specializzato in Ingegneria Informatica. È Internet Solution Architect presso lo Studio Associato WebCon e Web Project Manager con Sinfo Pragma S.p.A.. Si occupa anche di formazione aziendale per progettisti di sistemi e ingegneri del software.

Per poter struttare appieno la potenza e la flessibilità dei linguaggi evoluti come il C++, oltre a conoscerne bene la sintassi, occorre impostare bene l'analisi a oggetti di un progetto sotware. Nel presente articolo, attraverso l'analisi dettagliata di un case study, lo sviluppo di un programma di clustering statistico, viene mostrato come l'uso combinato di design pattern e template, combinato con metodologie rigorose, può rendere più rapide sia l'analisi sia lo sviluppo di programmi complessi in C++.

L'uso dei pattern e delle caratteristiche avanzate nella programmazione C++

Un *pattern per architetture software* (si veda [1]) descrive un problema che ricorre in specifici contesti di design e propone un generico, ma ben dimostrato schema per la sua soluzione. Lo schema deve descrivere i componenti, le loro responsabilità e interrelazioni, chiarendo nel dettaglio il modo in cui essi collaborano. In pratica quindi un pattern è una soluzione ad un determinato problema in un ben definito contesto [3].

L'uso dei pattern rappresenta un'evoluzione quasi naturale del modello della programmazione ad oggetti, in quanto definendo le interazioni fra le classi, contribuisce validamente alla definizione architetturale, specialmente in casi complessi. Attraverso l'uso dei pattern diviene più facile definire i messaggi che le classi si scambiano e attribuire alle classi le responsabilità entro il sistema che si sta progettando. È però da osservare che di rado un pattern può essere trasferito pari pari entro un sistema così come è definito nella sua formulazione, ma spesso deve subire degli adattamenti.

Il C++ supporta anche il polimorfismo parametrico e tramite i *template* si possono definire famiglie di classi o funzioni che dipendono da uno o più parametri. Una class template è un modo per dichiarare, attraverso una dichiarazione parametrica, una famiglia di classi, dipendente da un parametro generico T. Per esempio, un template "lista" definisce una lista di entità generiche "lista di T", dove T può essere di qualisiasi tipo. Attraverso i template si giunge alla *programmazione generica*. Ad esempio, considerando proprio il contenitore "lista", è difficile che un programmatore abbia bisogno proprio di

una lista di stringhe, il concetto di lista è più generale e non dipende dal particolare dato che ne forma il "componente elementare". Se un algoritmo può essere espresso indipendentemente dai dettagli della rappresentazione in modo affidabile e senza "contorsioni" logiche, dovrebbe essere espresso in termini di programmazione generica. Il paradigma della programmazione generica può essere enunciato nel seguente modo, come definito in [2]:

1. decidere quale algoritmo si vuole;
2. parametrizzarlo, in modo tale che operi su un insieme di tipi e strutture dati e non su un singolo tipo.

Uno *smart pointer* del C++ è un oggetto che si comporta come un puntatore "normale", ma che, in più, svolge alcune azioni ogni volta che un oggetto viene acceduto attraverso di esso, per esempio provvedendo, in modo trasparente verso l'entità esterna che richiede il "puntamento", a "ricreare" l'oggetto, caricandone gli attributi da un sistema di persistenza come un DB, qualora tale oggetto non esista nel sistema al momento della chiamata. Oltre al caricamento in memoria di dati, possono essere inserite entro lo smart pointer altre azioni, come il tener conto, in modo automatico e trasparente verso l'esterno, delle referenze ad un oggetto, in modo tale da verificare sempre che l'oggetto puntato non sia *null*, evitando a priori il conseguente pericolo di crash del programma per *segmentation fault*.

Nei paragrafi seguenti viene mostrato come questi strumenti vengono impiegati nell'analisi e stesura del codice.

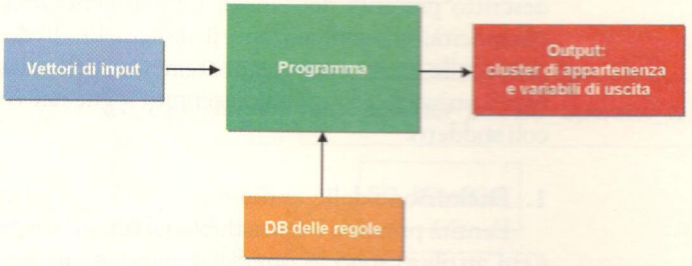
Il case-study: il clustering o suddivisione in classi di una popolazione

Il problema usato come case study rappresenta un caso molto frequente nell'ambito dell'analisi statistica, applicata sia in campo scientifico-tecnico sia in campo economico. Scrivere software per il clustering è descrivibile nel seguente modo: realizzare uno strumento che, dato un vettore di parametri in ingresso, potenzialmente composto di un grande numero di variabili, ne compia un'analisi, associandolo ad una classe (intesa come gruppo), scelta fra un insieme di classi possibili in uscita, ed eventualmente associandogli anche un vettore di parametri di uscita.

In tal modo lo spazio dei vettori in ingresso, aventi un insieme di valori possibili per i propri parametri interni, risulta suddiviso in gruppi o classi (*cluster*), ciascuno dei quali corrisponde ad una determinata caratteristica (l'operazione svolta è presentata in **Figura 1**).

L'obiettivo presentato in questo articolo è realizzare un sistema programmabile di classificazione, entro cui la suddivisione in classi viene compiuta attraverso un insieme di operatori parametrici,

FIGURA 1 La struttura "esterna" del programma: una serie di vettori di input vengono presentati in ingresso e devono essere classificati producendo anche il corrispondente vettore di uscita, applicando le regole contenute nel DB.



combinati in un albero con varie strade possibili. I percorsi attraverso cui "passano" i vettori campione da classificare dipendono sia dal valore delle singole variabili interne al vettore, sia dal valore dei parametri che definiscono il comportamento degli operatori. Il ramo finale in cui il vettore termina il proprio passaggio entro il sistema è associato al cluster in cui il vettore stesso viene classificato.

In **Figura 2** è riportato un esempio semplice della struttura interna di un sistema di classificazione del tipo descritto, dove con **1** sono indicati gli operatori che hanno solo lo scopo di fare calcoli, generando risultati intermedi a partire dal valore delle variabili che compongono il vettore o da risultati di calcoli parziali precedenti, con **2** gli operatori a suddivisione semplice (paragonabili a degli "IF") e con **3** gli operatori di selezione multipla (paragonabili a dei "CASE").

Le quattro classi di uscita hanno anche associato un vettore di variabili d'uscita, non necessariamente composto dalle stesse variabili in ogni classe, ottenute applicando operatori alle variabili del vettore d'ingresso o ai risultati intermedi di calcoli precedenti. È da notare, nella parte inferiore, la presenza di un "anello", dove i risultati dei tre operatori di tipo **1** attraversano poi comunque il medesimo operatore di selezione.

Il sistema deve essere completamente programmabile: sia le posizioni degli operatori, sia le regole associate agli operatori stessi non sono fisse, ma espresse come insiemi di parametri, che devono essere letti da un database o da un file alla partenza del programma stesso, in modo sia da comporre insieme gli operatori a formare l'albero dei percorsi, sia da definire il comportamento interno degli operatori stessi, ossia le funzioni da essi compiute sulle variabili. Inoltre il programma deve essere disegnato in modo tale che aggiunte di nuovi operatori in fasi successive non conducano allo stravolgimento del programma stesso.

La struttura può anche essere vista come un flowchart girato di 90° in senso antiorario: il percorso di ciascun vettore rappresenta il "cammino" compiuto entro un programma, "guidato" dai valori contenuti entro il vettore stesso stesso.

Le fasi di analisi

Sebbene in apparenza semplice, il sistema sopra descritto presenta dei vincoli forti di generalità e dinamicità, che influenzano il design. L'indivduazione delle entità e, conseguentemente, delle classi non è univoca, e la scelta presentata segue dai vincoli suddetti.

1. Definizione delle entità

L'entità primaria è senza dubbio il *vettore di input*, i cui attributi sono le *variabili di ingresso*, ma a cui dovranno essere associate anche le *variabili intermedie*, contententi i risultati dei calcoli degli operatori "attraversati" dal vettore stesso. Le altre entità fondamentali sono invece i risultati che il programma deve produrre, ovvero le *classi di uscita* e i *vettori di uscita* (si ricordi che tali vettori possono essere diversi secondo le classi) ad esse associati, contententi le *variabili di uscita*. L'*albero di classificazione* è esso stesso un'entità, ed è composto da *nodi*, ossia i singoli operatori, appartenenti a differenti tipi, che internamente contengono le *operazioni* (appartenenti anch'esse a differenti tipi), formate dalle *operazioni elementari*, ed i *parametri* che le governano.

2. Architettura ed interazioni fra entità

In base a quanto detto in precedenza l'architettura deve prevedere un raggruppamento dinamico di nodi all'interno dell'albero, dove ogni nodo deve conoscere il suo successore (eventualmente costituito da una classe di uscita) o i suoi successori (la configurazione dell'albero si forma a run-time in base a quanto letto dal DB di configurazione del programma stesso). Il vettore da analizzare "si muove" poi attraverso gli operatori, arricchendosi via via di variabili intermedie (ovvero dei vari risultati degli operatori attraversati), seguendo il percorso determinato dalle regole associate alle operazioni compiute negli operatori sulle variabili del vettore stesso. È importante notare che alcune delle strutture (ad esempio le operazioni interne a ciascun nodo) non necessariamente "si formano" (ovvero gli oggetti che le compongono vengono creati) sin dall'inizio del programma, ma potrebbero formarsi via via che le condizioni lo richiedono. Ciò significa che le strutture dati del programma devono essere in grado di gestire la creazione di oggetti "on demand".

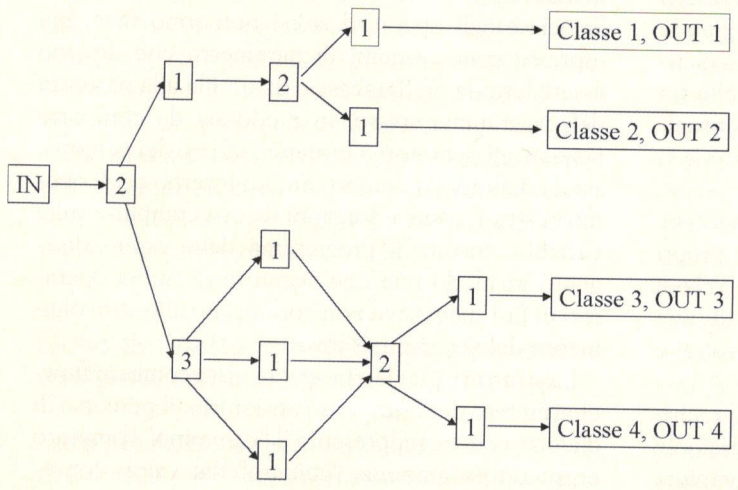
3. Le classi: suddivisione fra Classi Entità e Classi di Servizio

In base alla metodologia qui seguita, quando si passa alla definizione delle classi, accanto alle classi Entità, che derivano, più o meno direttamente, dalla proiezione nel mondo degli oggetti di una entità appartenente al problema in esame, occorre definire anche le classi di Servizio, ossia classi che rappresentano strutture "di supporto" associate al programma, come un input, un output, un database. A livello di stesura di codice è opportuno scegliere un prefisso diverso da porre davanti agli identificatori delle classi (ad esempio EC_ per classe Entità e SC_ per classe di Servizio).

Le classi Entità sono legate fra loro da rapporti di ereditarietà identificabili in base a quanto detto in precedenza. Anzitutto vediamo la *variabile generica (EC_Var)*, da cui derivano la *variabile di ingresso (EC_InputVar)*, la *variabile intermedia (EC_InternalVar)* e la *variabile di output (EC_OutputVar)*, ognuna delle quali agisce come contenitore per un proprio attributo (il valore), di tipo numerico, stringa o composito. Per generalizzare, lo stesso *valore* diviene un oggetto *(EC_Value)*, fungendo da contenitore per questo tipo generico di struttra dati (in pratica applicando la funzionalità di Wrapper del pattern *Adapter* [1]). La differenza principale sta nel fatto che le variabili di input sono popolate con i valori di input all'inizio di ogni ciclo di calcolo e, analogamente a dei contenitori che vengono riutilizzati, servono per tutta la fase di azione del programma, mentre le altre possono essere create e distrutte all'interno dello stesso ciclo. Le variabili sono attributi del *vettore campione (EC_SampleVector)* che deve essere analizzato dal sistema.

L'*albero (EC_Tree)* contiene i *nodi (EC_Node)*, che si specializzano nei vari *operatori (EC_Operator_1...)*. All'interno di ogni operatore sta un insieme di *operazioni (EC_Operation)* composte dalle *operazioni elementari (EC_MicroOp)* e dai *parametri (EC_Parameter)* che le governano (come visibile in **Figura 3**).

FIGURA 2 Esempio di percorso di selezione: il vettore di input, in base al valore dei propri componenti ed ai parametri associati agli operatori percorre uno dei cammini possibili entro l'albero e termina in una delle classi.



Accanto a queste classi devono essere definite le classi di servizio del *canale di input (SC_Input Channel)*, che deve "mascherare" al programma come i dati entrano per andare a popolare gli attributi, l'analogo per l'output *(SC_OutputChannel)*, la sorgente dei *parametri* che governano le operazioni *(SC_ParameterSource)*, ecc.

4. Le "strutture portanti" del sistema: raggruppamenti e Template

Dalla descrizione del paragrafo precedente appare evidente che occorrono raggruppamenti dei vari attributi, ovvero "vettori" o "liste" di attributi entro ciascun oggetto "contenitore". L'identificazione delle variabili o dei nodi dovrà essere univoca. Per esempio potrebbe avvenire tramite una chiave di identificazione, che definisca univocamente il "nome" (e, di conseguenza il ruolo) della variabile o del nodo entro il programma. Questo "nome", svolge esplicitamente, a livello logico, il ruolo dell'OID, ed ha come corrispondente diretto la chiave nel DB relazionale dove i dati sono memorizzati. Una struttura dati appropriata è quindi un'hashtable dinamica, dove vengono inseriti gli elementi con la propria chiave associativa che ne consente un richiamo immediato. Nel dettaglio, indicando con "lista" ciascuno di questi raggruppamenti, troveremo quindi:

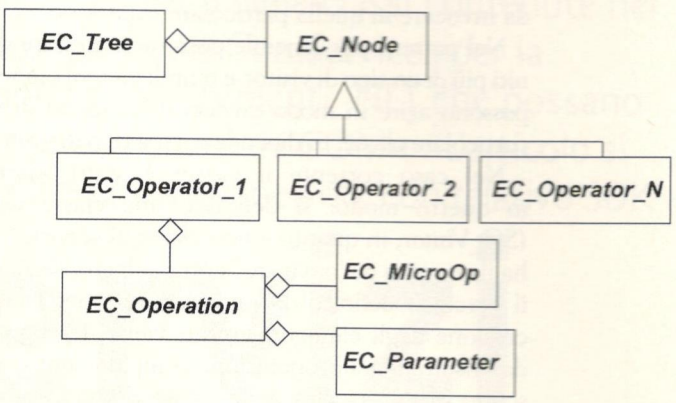
- una lista di variabili (suddivisa fra variabili di input e variabili risultanti da calcoli intermedi e quindi dinamica) entro il vettore di dati da analizzare;
- una lista di nodi entro l'albero, con i "link" che li congiungono, definendo i possibili percorsi che il vettore dati da analizzare "compie" entro l'albero stesso;
- una lista di operazioni entro ciascun nodo;
- una lista di operazioni elementari (eventualmente ridotta ad un singolo elemento) e parametri componenti ciascuna operazione;
- una lista di variabili di uscita componenti ciascun vettore di uscita.

In particolare occorre notare che, entro ciascuna di queste hashtable i tipi di elementi sono potenzialmente diversi, anche se, come nel caso delle variabili, derivano dalla stessa classe padre. Per risolvere questo problema si è fatto uso dei template. Le hashtable e gli oggetti "contenitori" che le usano devono essere definite come "generiche", ossia come operanti non su classi (e quindi tipi di dati) predefiniti, ma generici.

La seconda importante funzionalità che la lista deve svolgere è la creazione "on demand" di oggetti descritta sopra: se un oggetto non è ancora presente nel sistema (ovvero internamente alla lista), la lista stessa, in modo trasparente, deve provvedere alla sua creazione nel sistema, svolgendo un ruolo analogo a quello del pattern *Factory* (si veda **[1]**).

Per quanto riguarda i legami, per esempio quelli che uniscono i percorsi di nodi entro l'albero, essi sono evidentemente dei puntatori, per cui dovranno essere smart pointer.

FIGURA 3 Una parte del class diagram del case-study



5. Interazione fra classi e soluzione applicando un pattern

Sino ad ora si è considerato soltanto lo sviluppo delle strutture dati in modo "statico", definiendo quindi le classi e i loro raggruppamenti. Ma non si è considerato come, effettivamente, l'operazione di "navigazione" dei dati entro l'albero deve avvenire, e quindi come devono interagire fra loro le classi. Da un punto di vista logico, il vettore di dati da analizzare deve compiere un percorso lungo l'albero, nei nodi del quale sono "contenute" le operazioni che devono agire sulle variabili contenute nel vettore stesso. Il cammino dipende dalle variabili iniziali e da quelle ottenute nei passi precedenti, ovvero dal percorso precedentemente compiuto entro l'albero stesso. L'interazione tra le classi è molto simile a quella espressa da un pattern: il *Visitor* (si veda **[4]**), che nasce con lo scopo di permettere la definizione di operazioni su elementi diversi di una gerarchia di oggetti senza cambiare le classi di elementi sulle quali si opera. Attraverso l'incapsulamento delle operazioni si arriva alla soluzione, che permette anche di estendere facilmente sia il numero di classi sia le operazioni su di esse definite. Se si deve cambiare l'operazione che viene eseguita sulla struttura dati non è necessario anche cambiare le classi su cui si opera. Il pattern visitor consente quindi di disaccoppiare le classi delle strutture dati e gli algoritmi usati su di esse.

Ogni elemento della struttura dati "accetta" il visitatore: in pratica il visitatore invoca il metodo *accept* dell'elemento, passandogli il riferimento a se stesso come argomento. All'interno del metodo accept è a sua volta invocato il metodo *visit* del visitatore specifico per l'elemento dato (per esempio *visitElementA*), contenente le operazioni previste per quella particolare classe di elemento. Il visitatore contiene quindi i metodi specifici per compiere le operazioni su ogni elemento, ma è l'elemento che sa effettiva-

mente, in base alla propria classe, quale è il metodo da invocare in quella particolare situazione.

Nel pattern più generale possono poi essere definiti più di un tipo di visitor, e quindi metodi omonimi possono agire in modo diverso su elementi di una particolare classe, in dipendenza dal tipo di visitor.

Nel caso corrente il visitor è stato adattato in questo modo: si definisce una classe *visitor* (*SC_Visitor*, in quanto è una classe di servizio) che ha il compito di "navigare" entro l'albero, seguendo il percorso definito da una elaborazione. La successione degli elementi visitati viene determinata dai risultati delle operazioni compiute entro ciascuno di essi (scelta delle uscite nel caso di elementi multiuscita). Poiché ogni elemento dell'albero deve intervenire con un accettazione del visitatore, deve essere definito nell'elemento generico (EC_Nodo) il metodo *Accept*, che viene poi specializzato effettivamente solo in ciascuno degli elementi effettivi, invocando al proprio interno il metodo *visit* del visitor, specifico per l'elemento.

Per esempio, nel caso dell'operatore OperatorOne, deve esserci un metodo:

```
void Accept(SC_Visitor &v) {
v.visitOperatorOne(this);
}
```

Ciascun metodo visit deve poi realizzare le operazioni effettive di calcolo, servendosi internamente dei metodi interni dei singoli oggetti nodo. A livello implementativo il vettore campione da analizzare (ossia l'oggetto di classe EC_SampleVector definito in precedenza) è un attributo del visitor (il visitor funge da "trasporto" per il vettore in questione). Gli operatori sono predisposti per agire su alcune variabili, chiaramente identificate tramite un "nome" (o meglio la chiave delle hashtable) e quindi si rivolgono al visitatore (o meglio allo smart pointer contenuto entro di esso che punta al Sample Vector) per accedere alla hashtable ed estrarre i valori delle variabili da usare nei propri calcoli interni. Al termine deve essere passato al visitatore anche il riferimento al successivo oggetto da visitare, cioè al nodo dell'albero che, nel percorso che si sta seguendo, segue il nodo appena visitato.

La successione delle operazioni diventa pertanto la seguente:

1. Il vettore campione viene riempito, ossia i valori provenienti dal canale di input sono scritti entro le variabili del vettore;
2. Il vettore viene associato all'oggetto visitor come suo attributo;
3. Il visitor viene applicato al primo nodo dell'albero;
4. In base ai risultati delle operazioni viene individuato, fra i nodi "linkati" dal nodo corrente, chi è il successivo;
5. Il visitor viene applicato al successivo nodo;
6. Il ciclo di visite termina quando il nodo è un nodo di uscita, dove vengono fatte le operazioni che determinano i valori di uscita;
7. I valori di uscita vengono inviati al canale di output;
8. Se nel canale di ingresso ci sono altri campioni il programma ricomincia dal punto 1.

Dall'analisi alla scrittura del codice: i problemi implementativi

Una volta completata l'analisi del sistema e definite compiutamente tutte le classi ed i metodi ad esse associati, i problemi sorgono dalle caratteristiche del compilatore che si usa. Anche se ormai la maggioranza dei compilatori C++ supportano l'uso di template e smart pointer, non tutti usano la Standard Template Library, ma librerie proprietarie che, in caso di cambio del compilatore, forzano la riscrittura di parte del codice. Come già detto in precedenza, tutti i riferimenti (ovvero i "link") che corrisponderebbero a puntatori devono essere implementati con degli smart pointer.

Conclusioni

Nell'articolo è stata presentata l'analisi di un case-study piuttosto complesso ed è stato mostrato come l'applicazione delle tecniche avanzate legate alla programmazione generica e ai pattern possa rendere più facili molti dei passi.

Con un approccio simile a quello mostrato in questo articolo è stato disegnato il cuore del Decision Support System StrategyOne di CRIF, sistema di messa in produzione di strategie di previsione del rischio in ambito bancario. Il programma, inizialmente implementato su piattaforma WinNT, è in seguito stato portato su sistemi IBM S/390 e AS/400, usando i compilatori C++ IBM di tali piattaforme.

Si ringraziano il Dr. Aldo Bruschi e l'Ing. Sergio Badini di CRIF, e Ivan Makale, Stefano Dolcini e Alessio Zatti per il buon lavoro svolto assieme.

BIBLIOGRAFIA

[1] E. Gamma et al., "*Elements of Reusable Object-Oriented Software* ", Addison-Wesley, 1995

[2] B. Stroustrup - "*C++ Programming Language, 3rd Edition*", Addison-Wesley, 1997

RIFERIMENTI

[3] G. Lo Russo "*Programmazione per Pattern*", Mokabyte N. 23, Ottobre 1998, http://www. mokabyte.it/1998/10/pattern.htm

[4] "*Subject-oriented programming and the visitor pattern*", http://www.research.ibm.com/sop/sopcvisp. htm