Azzolini Riccardo 2019-10-03

Software Design

1 Architettura

L'obiettivo dell'attività di design è produrre l'**architettura** (o progetto) del software.

La progettazione richiede più passaggi: in genere, si procede da una visione ad alto livello (high-level design o progettazione in grande) verso il basso, fino a considerare le singole unità (low-level design o progettazione in piccolo).

In particolare, l'architettura di un sistema software riguarda l'high-level design: essa *definisce il sistema* in termini di **componenti computazionali** (*moduli*) e **interazioni** tra di essi.

2 Componenti e interazioni

Componenti e interazioni possono essere definiti a due livelli di astrazione diversi:

1. meccanismi;
2. stili.

2.1 Meccanismi

Al livello dei meccanismi, si descrive quali sono i costituenti del sistema, e come essi sono aggregati e correlati. A tale scopo, bisogna rispondere alle seguenti domande:

- Quali sono i moduli?
- Quali sono le loro interfacce?
- Quali sono le relazioni utili tra moduli?

Quest'aspetto della progettazione introduce alcune problematiche:

- Problema metodologico: quali sono i criteri per scomporre il sistema in moduli?
- Documentazione: come documentare il catalogo di moduli e relazioni?

2.2 Stili

In architetture diverse, gli stessi componenti (ad esempio client, server, database, filtri, e livelli di un sistema gerarchico) possono avere forme diverse, e/o essere "messi insieme" in modi diversi (con interazioni semplici, quali chiamate di procedura/metodo e accesso a variabili condivise, oppure più complesse, come protocolli client-server, multicast asincrono di eventi, ecc.[1]). Lo stile è quindi ciò che caratterizza una specifica architettura rispetto alle altre.

2.3 Confronto tra i due livelli

Come analogia, al livello dei meccanismi la carrozzeria di un'auto è vista come un sistema composto da portiere, cofano, ecc., mentre lo stile riguarda la differenza tra una coupé, una station wagon, ecc.

Il livello dei meccanismi e quello degli stili sono due *punti di vista* diversi relativi allo stesso "mondo". Per questo, la distinzione non è sempre chiara.

Questi due livelli hanno alcune caratteristiche in comune:

- a entrambi i livelli, deve essere possibile "ragionare" sull'architettura e sulle proprietà del sistema;
- entrambi questi livelli forniscono una descrizione prevalentemente *statica* (topologica) dell'architettura (mentre una rappresentazione dinamica sarebbe costituita da degli esempi di esecuzione del software, come i casi di test).

3 Moduli, servizi e interfacce

Un **modulo** è una parte di un sistema che fornisce un insieme di servizi ad altri moduli. I **servizi** sono elementi computazionali che possono essere usati da altri moduli.

L'insieme dei servizi forniti (*esportati*) da un modulo costituisce l'interfaccia del modulo stesso.

- Un modulo è costituito dalla sua interfaccia e dal suo corpo (implementazione, segreti).
- L'interfaccia può essere in parte esplicita (ad esempio, i parametri richiesti e il tipo restituito da un metodo) e in parte implicita (ad esempio, le aree di memoria globali su cui lavora un metodo).

[1] A basso livello, però, tutte le interazioni sono implementate mediante i concetti fondamentali della programmazione, che sono appunto chiamate di procedura e variabili condivise.

- L'interfaccia definisce un contratto tra il modulo e i suoi utenti (gli altri moduli che lo usano).
- Gli utenti conoscono il modulo solo attraverso la sua interfaccia. Di conseguenza, l'implementazione del modulo può variare liberamente, ma l'interfaccia deve essere "congelata" una volta definita: se essa subisse modifiche, andrebbero cambiati anche tutti i moduli utenti.
- L'interfaccia deve essere:
  - il più grande possibile, per mettere a disposizione tutte le funzionalità necessarie;
  - il più piccola possibile, per non esporre i segreti del modulo.

  Bisogna quindi trovare un equilibrio.

4 Relazioni

Le principali relazioni tra moduli sono:

USES: un modulo usa i servizi esportati da una altro;

IS_COMPONENT_OF: descrive l'aggregazione di moduli in altri moduli di livello più alto (perché i moduli sono raramente unitari);

INHERITS_FROM: per i sistemi orientati agli oggetti.

4.1 Definizione e proprietà matematiche

Sia $S = \{M_1, M_2, \ldots, M_n\}$ un insieme di moduli. Una *relazione binaria* r su S è un sottoinsieme di $S \times S$.

Notazione: Se M_i e M_j sono moduli in S, $\langle M_i, M_j \rangle \in r$ si può scrivere come $M_i \; r \; M_j$.

La *chiusura transitiva* r^+ di r è una relazione binaria tale che $M_i \; r^+ \; M_j$ se e solo se:

- $M_i \; r \; M_j$, oppure
- $\exists M_k \in S$ tale che $M_i \; r \; M_k$ e $M_k \; r^+ \; M_j$.

r è una *gerarchia* se e solo se

$$\forall M_i, M_j, \quad M_i \; r^+ \; M_j \implies \neg(M_j \; r^+ \; M_i)$$

cioè se e solo se non ha cicli.

Le relazioni possono essere rappresentate come grafi diretti (cioè orientati). In particolare, una gerarchia è un grafo diretto aciclico (DAG). In quest'ultimo caso, i nodi non

raggiungibili da altri si dicono *radici* della gerarchia (siccome una gerarchia non è per forza un albero, può esisterne più di una).

5 Relazione USES

A USES B significa che:

- A può accedere ai servizi esportati da B tramite la sua interfaccia;
- A dipende da B per fornire a sua volta i propri servizi: se B non è corretto, allora anche A non è corretto.

Si dice anche che A è un *cliente* di B.

Questa relazione è definita staticamente (ad esempio, scrivendo nel codice di A delle chiamate a metodi di B).

Quando si definiscono le relazioni USES, bisogna cercare di formare una gerarchia, evitando, per quanto possibile, gli usi ciclici. In questo modo, si ottiene una struttura a livelli di astrazione:

- è più semplice comprendere come funziona il sistema;
- è più facile sviluppare e testare il software in modo incrementale, perché i moduli di ciascun livello della gerarchia dipendono solo da quelli dei livelli inferiori (mentre, in presenza di cicli, si rischia di avere un sistema in cui niente funziona finché non sono stati completati tutti i componenti);

6 Relazione IS_COMPONENT_OF

Questa relazione è usata per descrivere un modulo di livello più alto come insieme di moduli di livello più basso (ad esempio, in Java, un metodo è un componente di una classe, una classe è componente di un package, ecc.). A IS_COMPONENT_OF B significa infatti che B è costituito da vari moduli, uno dei quali è A.

Si dice anche che:

- se A IS_COMPONENT_OF B, allora B COMPRISES A;
- se $M_Z = \{M_k \mid M_k \in S \wedge M_k \text{ IS\_COMPONENT\_OF } Z\}$ è l'insieme di tutti i moduli che sono componenti di Z, allora M_Z IMPLEMENTS Z.

Questa relazione forma una gerarchia (per la precisione, un albero). Un modulo che non è una foglia in questa gerarchia non è altro che un contenitore di moduli di livello più basso (ad esempio, una classe Java è "semplicemente" un contenitore di variabili è metodi), quindi i moduli foglia sono quelli che implementano effettivamente il sistema.

7 Relazione INHERITS_FROM

In un sistema orientato agli oggetti, la relazione INHERITS_FROM permette a un componente di estenderne un altro.

In pratica, questa relazione è una forma più forte di USES, perché l'erede può accedere ad (alcuni) dei segreti del componente da cui eredita.

Osservazione: Anche la relazione `friend` esistente in C++ è una forma più forte di USES, ma è meno forte di INHERITS_FROM, perché permette a una classe di specificare esattamente quali funzioni/metodi possono accedere ai suoi segreti, invece di consentire l'accesso a tutte le sottoclassi.