Azzolini Riccardo 2019-10-07

Software Design

1 Cambiamenti

Uno degli obiettivi della progettazione (e anche delle altre fasi dello sviluppo, fin dallo studio di fattibilità) è ridurre il rischio.

Un importante fattore di rischio sono i cambiamenti. Perciò, il software deve essere progettato in modo da facilitare i cambiamenti ("design for change"), e in generale la manutenzione. A tale scopo, bisogna anticipare i probabili cambiamenti futuri,[1] evitando di concentrarsi solo sulle esigenze attuali, e pensando invece anche all'evoluzione. Una tecnica basata su tale principio è la *prototipazione evolutiva*: il prodotto è "sempre un prototipo", che continua a evolversi.

Spesso, inoltre, è utile vedere il programma come membro di una famiglia di programmi simili, e progettare tutta la famiglia (sviluppando, ad esempio, un framework riutilizzabile), non ciascun membro separatamente. Così, in futuro, sarà molto più semplice realizzare questi altri programmi.

1.1 Cambiamenti più comuni

- Cambiamenti di algoritmi.
- Cambiamenti di strutture dati, che costituiscono il 17 % dei costi di manutenzione.
- Cambiamenti nella macchina astratta sottostante (a tutti i livelli: periferiche hardware, sistema operativo, DBMS, ecc.). Questi possono essere gestiti facilmente se si introduce uno strato software che fa da *adattatore* tra il programma e la macchina sottostante: se cambia la macchina, deve essere modificato solo l'adattatore.
- Cambiamenti nell'ambiente (ad esempio nelle leggi).
- Cambiamenti imposti dalla strategia di sviluppo (soprattutto per i progetti gestiti con cicli di vita iterativi).

[1] Alcuni possibili cambiamenti si evidenziano naturalmente in fase di progettazione: se si individuano più modi per realizzare un determinato aspetto del software, sarà probabile che in futuro si voglia passare a una soluzione diversa. Ciò vale soprattutto quando, per vari motivi, non si può adottare subito la soluzione ottimale: in questo caso, si sviluppa un'alternativa temporanea, che andrà sostituita in seguito (*debito tecnico*).

2 Principi di design

- Come selezionare i moduli?
- Come definire le interfacce dei moduli?
- Come definire le relazioni USES?

2.1 Moduli e USES

- Un modulo deve essere un'unità autocontenuta: tutto ciò che serve per la sua funzionalità deve essere contenuto nel modulo stesso.
- Le relazioni USES devono essere ridotte al minimo.

Principio: **massimizzare la coesione** e **minimizzare l'accoppiamento**.

- Massimizzare la coesione interna: gli elementi di un modulo devono essere logicamente correlati (e non svolgere funzioni completamente diverse).
- Minimizzare l'accoppiamento tra moduli: limitare la quantità di effetti collaterali che si avrebbero se ci fosse un problema nel modulo (con un elevato grado di accoppiamento, i problemi tendono a propagarsi ad altri moduli).

2.2 Interfacce

- Distinguere tra *cosa* un modulo fa per gli altri e *come* lo fa (i suoi segreti).
- Minimizzare il flusso informativo verso i clienti, fornendo primitive di accesso ai dati controllate dal modulo stesso.
- Garantire la stabilità dell'interfaccia (tranne, eventualmente, in caso di eventi eccezionali), in quanto contratto tra il modulo e i suoi clienti.

Principio fondamentale: **information hiding**: definire cosa si vuole nascondere e costruirvi intorno un modulo.

Osservazione: Un'interfaccia più ampia permette di fornire più funzionalità, ma favorisce l'accoppiamento.

2.3 Conclusioni

- Un modulo è un'unità logica.
- I suoi segreti sono incapsulati e protetti al suo interno.
- Esso filtra l'accesso ai suoi segreti tramite la sua interfaccia.
- Se si effettuano modifiche alla parte segreta, i cambiamenti non hanno effetto sui clienti.

2.4 Esempio

Si consideri, come esempio, un modulo che implementa una tabella, la quale permette di inserire dati, eliminarli, e stamparli in un qualche ordine (ad esempio alfabetico).

Mettendo nell'interfaccia solo le operazioni INSERT, DELETE e PRINT, è possibile cambiare liberamente:

- la struttura dati;
- la politica di gestione (tenere i dati sempre in ordine, oppure ordinarli prima della stampa).

2.5 Concetti e principi chiave

Alcuni concetti e principi chiave della progettazione sono:

- scomposizione;
- astrazione;
- information hiding;
- modularità;
- estendibilità;
- struttura a macchine virtuali;
- gerarchia;
- famiglie di programmi.

I principali obiettivi di questi concetti e principi sono:

- gestire la complessità dei sistemi software;
- migliorare la qualità del software;
- facilitare il riuso sistematico del software sviluppato.

3 Tipi di moduli

Alcuni dei principali tipi di moduli sono:

- **Operazione astratta**: non è associata a dei dati, e quindi non c'è bisogno di un oggetto.

  Un modulo di questo tipo potrebbe essere, ad esempio, una funzione in un linguaggio procedurale, oppure un metodo statico in Java.

- **Oggetto astratto** (o *macchina a stati astratta*): incapsula un struttura dati ed esporta un insieme di operazioni che ne cambiano lo stato.

  In Java, tutti gli oggetti devono essere istanze di classi, ma altri linguaggi permettono la creazione diretta di un singolo oggetto.

- **Tipo di dato astratto**: permette di istanziare oggetti astratti. Definendo un tipo di dato astratto, rispetto all'utilizzo diretto di oggetti astratti, si guadagna la possibilità di:

  – effettuare assegnamenti tra variabili di questo tipo;

  – determinare l'uguaglianza tra istanze di questo tipo;

  – ecc.

  In Java, questo tipo di modulo corrisponde a una classe.

4 Possibile notazione di design

Per descrivere il design dei moduli è possibile usare, ad esempio, la seguente notazione testuale:

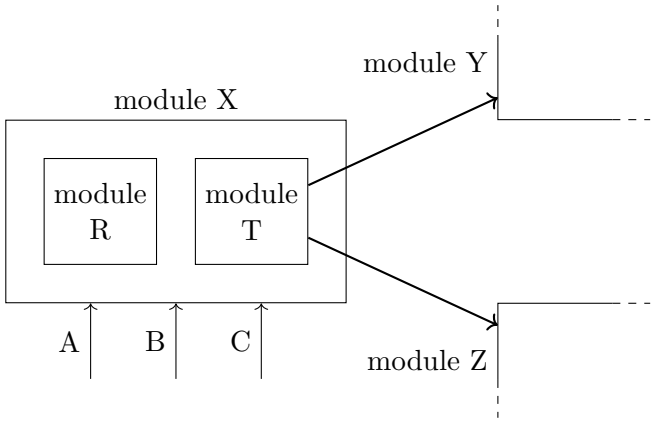
```
module X
uses Y, Z
exports
    var A: integer;
    type B: array (1..10) of real;
    procedure C (D: in out B; E: in integer; F: in real);
    Descrizione opzionale in linguaggio naturale di A, B e C.
implementation
    Se necessario, commenti su modularizzazione, implementazione, ecc.
    is composed of R, T;
end X

module R
```

```
uses Y
exports
    var K: record ... end;
    type B: array (1..10) of real;
    procedure C (D: in out B; E: in integer; F: in real);
implementation
    ...
end R

module T
uses Y, Z, R
exports var A: integer;
implementation
    ...
end T
```

In alternativa, può essere usata una notazione grafica, come:



5 Design orientato agli oggetti

- Il modulo è una classe, ed è esso stesso una risorsa: viene usato da altri moduli per creare istanze.
- Si introduce la relazione di ereditarietà, che permette di estrarre una parte comune a più componenti, ma aggiunge delle interdipendenze molto forti tra i moduli.
- Le variazioni possono essere realizzati ridefinendo nei sottocomponenti solo le parti che cambiano.

6 Progettazione per contratto

La **progettazione per contratto** (**design by contract**) è un principio particolarmente adatto per il design orientato agli oggetti.

L'interfaccia di un modulo (classe) definisce un **contratto**: esso è un accordo tra il *cliente* e il *contraente* (fornitore di servizi), che definisce *obbligazioni* e *benefici* per entrambi.

6.1 Precondizioni e postcondizioni

Il contratto per un modulo orientato agli oggetti definisce:

precondizioni: cosa è richiesto da ciascun metodo (cioè le obbligazioni del cliente);

postcondizioni: cosa è fornito da ciascun metodo (ovvero le obbligazioni del contraente).

La precondizione di un metodo deve essere garantita dal cliente, mediante un controllo esplicito (`if` / `else`) prima dell'invocazione oppure ragionando sul programma (ma, in questo caso, è possibile sbagliare). La postcondizione deve invece essere garantita dal contraente, mediante l'implementazione del metodo.

Se una routine non specificasse una precondizione, essa dovrebbe essere in grado di gestire tutti i possibili input. Al contrario, se la precondizione è troppo forte, la routine diventa difficile da invocare, perché il cliente deve effettuare controlli/ragionamenti complessi per essere certo che tali condizioni siano soddisfatte. Per questo, la scelta delle precondizioni è una scelta progettuale: non ci sono regole generali, ma solitamente si preferisce scrivere routine semplici che soddisfano un contratto ben definito, piuttosto che una routine che cerca di gestire ogni possibile situazione.

6.2 Proprietà interne di una classe

Oltre alle precondizioni e postcondizioni dei metodi, è possibile specificare una proprietà, chiamata **invariante**, che tutte le istanze di una classe devono soddisfare.

L'invariante deve essere vero a partire dalla creazione dell'istanza, e poi rimanere vero prima e dopo ciascuna operazione. Esso deve essere garantito dall'implementazione della classe.

6.3 Correttezza di una classe

- Operazione di creazione:

$$\{\text{pre}_\text{c}\}\ \text{costruttore}\ \{\text{INV}'\}$$

  - prima della creazione, deve essere soddisfatta la precondizione del costruttore (pre_c);
  - l'istanza creata deve soddisfare l'invariante della classe (con INV' si denota che l'invariante è soddisfatto dallo stato della classe al termine dell'operazione).

- Qualsiasi altra operazione:

$$\{\text{pre}_{\text{operazione}} \wedge \text{INV}\}\ \text{operazione}\ \{\text{post}_{\text{operazione}} \wedge \text{INV}'\}$$

  - prima dell'operazione, devono essere soddisfatte la precondizione dell'operazione stessa e l'invariante della classe;
  - dopo l'operazione, devono essere soddisfatte la postcondizione dell'operazione stessa e l'invariante della classe;
  - l'invariante *non deve necessariamente essere soddisfatto* durante l'esecuzione dell'operazione.

6.4 Esempio

Si consideri, come esempio, un modulo incapsula una tabella e permette di inserirvi degli elementi.

L'operazione insert(element) ha:

- precondizione: no_elements $<$ size, cioè la tabella non deve essere piena;
- postcondizione:
  - element è contenuto nella tabella;
  - no_elements$'$ = no_elements + 1, dove no_elements$'$ denota il valore di no_elements dopo l'operazione.

Un esempio di invariante della classe è $0 \leq$ no_elements $\leq$ size.

6.5 Eccezioni

La verifica delle precondizioni è responsabilità del cliente, ma:

- potrebbe essere impossibile determinare se la precondizione di un'operazione è soddisfatta senza prima eseguire l'operazione stessa (ad esempio, per le operazioni di input/output);
- alcune operazioni frequenti non falliscono quasi mai (come ad esempio l'allocazione di memoria, che fallisce solo se è già occupata tutta la memoria disponibile), quindi non è pratico controllare le precondizioni ogni volta;
- a causa di errori di codifica, possono essere effettuate delle invocazioni anche se la precondizione non è soddisfatta.

Per questo, ciascun metodo deve sollevare un'**eccezione** se si ha una violazione:

- della sua precondizione;
- della sua postcondizione;
- dell'invariante della classe.

In questo modo, quando termina l'esecuzione di un metodo, o sono soddisfatti la sua postcondizione e l'invariante, o altrimenti viene sollevata un'eccezione, che può poi essere gestita dal chiamante.

Le eccezioni che un metodo può sollevare sono parte della sua interfaccia.

6.6 Ereditarietà

Le sottoclassi possono aggiungere attributi e metodi, e possono ridefinire alcuni metodi. Tra le precondizioni/postcondizioni dei metodi della classe e quelle dei metodi ridefiniti nella sottoclasse devono esistere particolari legami:

- $\text{pre}_{\text{classe}} \rightarrow \text{pre}_{\text{sottoclasse}}$, cioè la precondizione di un metodo della classe deve implicare la precondizione del metodo ridefinito nella sottoclasse;
- $\text{post}_{\text{sottoclasse}} \rightarrow \text{post}_{\text{classe}}$, cioè la postcondizione di un metodo ridefinito deve implicare la postcondizione del metodo originale.

Ciò si riflette anche nei tipi dei parametri e nel tipo restituito: quando si ridefinisce un metodo in una sottoclasse,

- il tipo restituito può diventare più specializzato (cioè essere sostituito con un suo sottotipo): *covarianza dei risultati*;
- il tipo di ciascun parametro può diventare più generale (cioè essere sostituito con un suo supertipo): *controvarianza dei parametri*.

Questo è un principio generale, che però non viene implementato da tutti i linguaggi: alcuni, ad esempio, permettono di specificare tipi più specializzati anche per i parametri di un metodo ridefinito, nonostante ciò possa portare a errori in fase di esecuzione.