Azzolini Riccardo 2019-10-10

Unified Modeling Language

1 Modelli

Il problema principale nella progettazione e nella realizzazione di un sistema è la complessità: ragionare su sistemi complessi e realizzarli è difficile.

Allora, per poter analizzare particolari caratteristiche di un sistema complesso, trascurando i dettagli non rilevanti, è necessario costruire dei **modelli**. Essi sono infatti il linguaggio dei progettisti: si usano per rappresentare un sistema da costruire o costruito, sia per analizzarne particolari caratteristiche, sia come strumento di comunicazione.

1.1 Astrazione

Con **astrazione** si intende la descrizione di un sistema (o di parte di esso) che ne riporta solo le caratteristiche rilevanti (per un determinato contesto).

La modellizzazione sfrutta proprio l'astrazione per dominare la complessità: un modello *può* e *deve* trascurare i dettagli irrilevanti.

2 Modularità

Un sistema è **modulare** se è diviso un parti che hanno

- una sostanziale autonomia individuale
- una ridotta interazione con le altre parti

cioè alta coesione interna e basso grado di accoppiamento.

2.1 Criteri

Scomponibilità modulare: scomporre un problema in sottoproblemi di minori dimensioni e complessità (in modo che la somma delle complessità dei sottoproblemi sia *minore* della complessità dell'intero problema).

Componibilità modulare: ri-aggregare moduli esistenti per risolvere nuovi problemi. Per rispettare questo criterio, in modo da favorire la riusabilità, già in fase di scomposizione del problema bisogna costruire i componenti in modo che siano riusabili.

Comprensione modulare: capire un modulo osservando solo quello e i "confinanti".

Continuità modulare: un piccolo cambiamento delle specifiche comporta cambiamenti solo in uno o pochi moduli. In pratica, non è sempre possibile raggiungere quest'obiettivo, anche perché cambiamenti apparentemente piccoli delle specifiche possono alterare notevolmente il funzionamento del sistema.

Protezione modulare: gli errori in un modulo influenzano solo il modulo stesso, o, al massimo, si propagano ai confinanti, quindi deve essere corretto un numero limitato di moduli.

2.2 Metodi

I metodi per realizzare un sistema modulare sono:

- utilizzare linguaggi che mettano a disposizione *unità linguistiche modulari* (ad esempio, linguaggi di programmazione con funzioni o classi);
- ridurre al *minimo* il *numero di comunicazioni* tra moduli: in un sistema con n moduli, il numero di comunicazioni deve essere vicino a $n-1$ (che è il minimo indispensabile per formare un unico sistema connesso) e lontano da $\frac{n(n-1)}{2}$ (che corrisponde al caso in cui ogni modulo comunica con tutti gli altri);
- *interfacce piccole*: ogni modulo deve scambiare il minor numero possibile di informazioni con gli altri moduli;
- *interfacce esplicite*: il fatto che due moduli A e B comunichino deve risultare evidente sia dal codice di A che dal codice di B,[1] evitando quindi il più possibile gli effetti collaterali;
- *information hiding*: il modulo deve distinguere tra informazioni pubbliche e informazioni private.

[1] Da questo punto di vista, l'interazione tra classi in Java è una "via di mezzo", perché il modulo chiamato non conosce il chiamante, mentre nel codice del chiamante sono esplicitamente scritte le invocazioni dei metodi dell'altra classe.

3 Viste

Un sistema complesso ha molti "aspetti" da descrivere:

- la struttura statica (quali sono i componenti);
- il comportamento dinamico (come si comporta il sistema durante l'esecuzione);
- l'organizzazione logica (come sono organizzati i componenti);
- la distribuzione fisica (dove vengono installati i componenti);
- ecc.

Siccome è difficile comprimere in un unico modello tutte queste informazioni di natura diversa, si realizzano invece *più modelli*, ciascuno specializzato per fornire un certo tipo di informazioni.

4 Linguaggi di descrizione

L'attività di sviluppo del software produce vari semilavorati:

- definizione del problema (requisiti);
- progetto del sistema;
- implementazione;
- documentazione;
- casi di test;
- ecc.

Ciascuno di questi è scritto in un qualche **linguaggio di descrizione**:

- linguaggi di programmazione e affini (C++, Java, HTML, ecc.);
- linguaggio naturale (inglese, italiano, ecc.);
- linguaggi specifici per la modellizzazione.

Una **descrizione** è un insieme di affermazioni riguardo una qualche realtà di interesse. I linguaggi di descrizione sono linguaggi adatti a scrivere descrizioni. Essi hanno varie caratteristiche:

Completezza: il linguaggio deve fornire strumenti per descrivere tutti gli aspetti *di interesse* (ma non necessariamente tutti gli aspetti esistenti).

Accuratezza: si deve poter costruire una descrizione precisa.

Consistenza: il linguaggio deve aiutare a evitare contraddizioni, sia in diverse rappresentazioni della stessa descrizione, che all'interno delle singole rappresentazioni.

Comprensibilità: la descrizione deve essere facilmente comprensibile da parte di chi la deve interpretare (per modificarla e/o per utilizzarla come riferimento).

Formalità: rigore con cui sono definite la sintassi e la semantica del linguaggio.

Stile: quale aspetto del sistema è più facile da descrivere utilizzando il linguaggio.

4.1 Gradi di formalità

Informale: non ha né sintassi né semantica formalmente definite. L'esempio tipico è il linguaggio naturale, spesso usato perché è facilmente comprensibile dal committente.

Semi-formale: ha sintassi ben definita, ma poca semantica (anche se, in generale, la sintassi porta a dare un'interpretazione relativamente uniforme, nonostante ne siano possibili diverse). Un linguaggio di questo tipo:

- è facilmente comprensibile;
- è una possibile fonte di ambiguità, anche se meno dei linguaggi informali.

Formale: ha sintassi e semantica rigorose. Un linguaggio formale:

- ha fondamenti logico-matematici, quindi elimina le ambiguità;
- consente di ragionare sulle proprietà di un sistema e verificarle (ma, comunque, non tutte le proprietà sono formalmente verificabili);
- è utilizzabile praticamente solo in ambito strettamente tecnico.[2]

4.2 Stili di descrizione

Lo stile di un linguaggio di descrizione può essere:

descrittivo: definisce le proprietà desiderate dell'oggetto che si sta descrivendo;

operazionale: definisce il comportamento desiderato dell'oggetto come sequenza di operazioni di una macchina astratta.

In realtà, lo stile di molti linguaggi è una via di mezzo tra questi due estremi.

[2] Nel caso di alcuni sistemi critici, per i quali è importante essere sicuri di aver definito bene il problema, si usano dei linguaggi formali anche in fasi che normalmente sarebbero basate su linguaggi informali, come ad esempio la specifica dei requisiti. Nella maggior parte dei progetti, però, ciò non si fa perché comporterebbe tempi e costi eccessivi.

4.2.1 Caratteristiche dello stile descrittivo

- La descrizione è generalmente più compatta.
- Si ha una maggiore astrazione, in quanto non viene descritta una possibile realizzazione ("lasciando aperta la strada" per la scelta dell'implementazione migliore).
- È più difficile capire il comportamento del sistema descritto.
- È più facile fare ragionamenti sulle proprietà del sistema:
  - dimostrare formalmente che vale una certa proprietà;
  - modificare la descrizione in modo da far valere una proprietà.

4.2.2 Caratteristiche dello stile operazionale

- Suggerisce in sé un metodo per la realizzazione, spingendo decisamente verso di esso, ma non è detto che questo sia il migliore
- È facilmente eseguibile (ad esempio, è facile costruire casi di test).
- È facile costruire un prototipo e verificarne l'aderenza alla descrizione (cioè che abbia il medesimo comportamento).
- È più difficile risalire alle proprietà del sistema e dimostrarle.

4.2.3 Esempio

La descrizione di un cerchio di raggio r potrebbe essere:

- *Stile descrittivo*: la curva soddisfa l'equazione $x^2 + y^2 - r^2 = 0$.
- *Stile operazionale*:
  1. prendere un compasso;
  2. piantare la punta nell'origine;
  3. aprire il compasso per avere ampiezza uguale a r;
  4. abbassare la mina fino al foglio nel punto $(0, r)$;
  5. tracciare in modo continuo in senso orario;
  6. terminare quando si ritorna al punto $(0, r)$.

5 Unified Modeling Language

UML, **Unified Modeling Language**, è il linguaggio visuale standard (o, meglio, un insieme di linguaggi) per definire, progettare, realizzare,[3] documentare e testare i sistemi software, secondo un approccio object-oriented. Esso:

- riunisce molte proposte preesistenti;
- è sponsorizzato dalle maggiori industrie produttrici di software;
- è standardizzato e gestito dall'Object Management Group (OMG);
- copre l'intero processo di produzione del software, permettendo di descrivere/modellare tutti (o quasi) gli aspetti rilevanti di un sistema;
- può essere usato anche per analizzare sistemi già esistenti;
- è basato su una *notazione grafica*, costituita da vari tipi di diagrammi;
- è un linguaggio *semi-formale* (in particolare, alcuni diagrammi sono più formali, mentre altri meno);
- ha uno *stile misto*, parzialmente descrittivo e parzialmente operazionale (i diagrammi statici sono più descrittivi, mentre quelli dinamici sono più operazionali);
- è *indipendente da qualsiasi linguaggio di programmazione*;
- riunisce aspetti dell'ingegneria del software, delle basi di dati e della progettazione dei sistemi;
- è utilizzabile in domini applicativi diversi e per progetti di diverse dimensioni;
- è *estendibile*, per modellare meglio le diverse realtà.

6 Viste e diagrammi UML

UML definisce numerosi diagrammi, organizzati in "$4 + 1$" viste:

- **vista dei casi d'uso** (che si distingue dalle altre per il suo ruolo centrale, in quanto definisce i principali requisiti)
  - *use case diagram*
- **vista strutturale**
  - *class diagram*
  - *object diagram*

[3] Non è possibile realizzare un programma interamente in UML, ma esistono strumenti che permettono di generare uno "scheletro" di codice partendo da alcuni diagrammi UML.

  - *composite structure diagram*[4]
  - *package diagram*
- **vista comportamentale**
  - *sequence diagram*
  - *communication diagram*
  - *state diagram*
  - *activity diagram*
  - *interaction overview diagram*
  - *timing diagram*
- **vista implementativa**
  - *component diagram*
  - *composite structure diagram*[4]
- **vista ambientale**
  - *deployment diagram*

6.1 Vista dei casi d'uso

Descrive i casi d'uso che forniscono valore agli utenti (cioè, in pratica, i requisiti funzionali), che in UML vengono visualizzati mediante gli use case diagram.

I casi d'uso essenziali vengono usati come proof of concept per l'architettura dell'implementazione.

Ogni use case può avere più scenari possibili, ciascuno dei quali può essere descritto in modo testuale e/o attraverso vari diagrammi (sequence, communication e activity).

[4] Il *composite structure diagram* può essere considerato parte sia della vista strutturale che di quella implementativa.

6.2 Vista strutturale

Rappresenta gli elementi strutturali necessari per implementare la soluzione ai requisiti del sistema.

Definisce:

- elementi di analisi a oggetti e di progetto;
- il vocabolario del dominio e della soluzione;
- la decomposizione del sistema in strati e sottosistemi;
- le interfacce del sistema e i suoi componenti.

Non descrive, invece, l'evoluzione del sistema in fase di esecuzione.

È rappresentata da diagrammi statici UML:

- class diagram, con livelli di astrazione multipli;
- package diagram;
- composite structure diagram.

6.3 Vista comportamentale

Descrive l'interazione dinamica tra i componenti del sistema, permettendo di specificare i requisiti non funzionali (performance, manutenibilità, ecc.)

Essa mostra inoltre la distribuzione delle responsabilità, e permette di identificare colli di bottiglia di interazione e accoppiamento.

Tale vista è particolarmente importante per i sistemi distribuiti.

I diagrammi UML corrispondenti sono dinamici:

- sequence e/o communication diagram;
- activity diagram;
- state diagram;
- interaction overview diagram;
- timing diagram.

6.4 Vista implementativa

Descrive l'implementazione dei sottosistemi logici definiti nella vista strutturale (ad esempio, come deve essere effettivamente realizzata una classe definita a livello concettuale), e specifica le dipendenze dei componenti implementativi e delle loro connessioni dalle interfacce richieste e fornite.

È rappresentata da due tipi di diagrammi UML,

- component diagram
- composite structure diagram

e può comprendere anche documenti intermedi usati nella costruzione del sistema (file di codice, librerie, file di dati, ecc.).

6.5 Vista ambientale

Rappresenta la topologia hardware del sistema e definisce come vengono assegnati i componenti software ai nodi hardware.

Essa è utile sia per analizzare requisiti non funzionali (affidabilità, scalabilità, sicurezza, ecc.) che per fornire informazioni sull'installazione e sulla configurazione del sistema.

È rappresentata in UML dai deployment diagram.