Architetture software

Laura Semini

Note per il corso di Ingegneria del Software

Corso di Laurea in Informatica
Corso di Laurea in Informatica Applicata

Dipartimento di Informatica
Università di Pisa

1 Architettura software

Le seguenti definizioni, diverse tra loro, riassumono le principali posizioni sul significato di architettura.

- La progettazione e la descrizione della struttura complessiva del sistema risultano essere un nuovo tipo di problema. Questi aspetti strutturali includono l'organizzazione di massima e la struttura del controllo; i protocolli di comunicazione, sincronizzazione e accesso ai dati, [...]

  [Garlan & Shaw, 1993]

- L'architettura software è l'organizzazione di base di un sistema, espressa dai suoi componenti, dalle relazioni tra di loro e con l'ambiente, e i principi che ne guidano il progetto e l'evoluzione.

  [IEEE/ANSI 1471–2000]

- L'architettura software è l'insieme delle strutture del sistema, costituite dai componenti software, le loro proprietà visibili e le relazioni tra di loro.

  [Bass, Clemens & Kazman, 1998]

La terza ha ispirato la definizione che segue, che viene fornita e usata in queste note:

> L'architettura di un sistema software (in breve architettura software) è la struttura del sistema, costituita dalle parti del sistema, dalle relazioni tra le parti e dalle loro proprietà visibili.

La struttura definisce, tra l'altro, la scomposizione del sistema in sottosistemi dotati di un'interfaccia e le interazioni tra essi, che avvengono attraverso le interfacce.

Le proprietà visibili di un sottosistema definiscono le assunzioni che gli altri sottosistemi possono fare su di esso, come servizi forniti, prestazioni, uso di risorse condivise, trattamento di malfunzionamenti, ecc.

La precisazione di considerare solo le proprietà visibili aiuta a chiarire la differenza tra progettazione architettonica e progettazione di dettaglio: solo in quest'ultima, infatti, ci si occupa degli aspetti "non visibili" dei sottosistemi, quali ad esempio strutture dati o algoritmi utilizzati per la loro realizzazione.

Un'architettura viene identificata da molti autori con la sua descrizione. In queste note, invece, i due concetti saranno tenuti distinti[1]. Vero è che un'architettura è un'entità astratta documentata solo da una descrizione. Ma, data un'architettura, possono essere fornite diverse descrizioni, che differiscono, per esempio, per il livello di dettaglio. Identificando un'architettura con la sua descrizione si dovrebbe parlare di architetture diverse.

Nel processo di sviluppo software, il progettista (o un gruppo di progettisti), dati i requisiti del sistema e i requisiti del software, definisce un'architettura e la descrive attraverso documenti di progetto. Essendo l'insieme di questi documenti l'unica descrizione dell'architettura immaginata dal progettista, si tende a far coincidere queste descrizioni con l'architettura stessa. In queste note tale insieme sarà invece chiamato *disegno di progetto architettonico*.

L'importanza di distinguere tra architettura e sua descrizione si coglie ragionando a posteriori: dato un sistema completamente realizzato, la sua architettura è la sua struttura, e non la descrizione della struttura. Infatti potrebbero esserci più descrizioni, a livelli di dettaglio diversi, o focalizzate su aspetti diversi. L'architettura rispetta l'architettura se il disegno di progetto architettonico ne è una descrizione.

Una distinzione simile si ha in edilizia, come riassunto dalla seguente presentazione di un corso di disegno dell'architettura:

> Il Disegno di Progetto permette di rappresentare agli altri l'architettura immaginata dal progettista: si realizza graficamente tramite piante prospetti e sezioni, [...]

2 Scopo dell'architettura

L'architettura di un sistema software viene definita nella prima fase di progettazione, quella architettonica. Lo scopo primario è la *scomposizione del sistema in sottosistemi*: la realizzazione di più componenti distinti è meno complessa della realizzazione di un sistema come monolito.

Ridurre la complessità di realizzazione non è l'unico scopo di un'architettura. Un altro fine è il miglioramento le caratteristiche di qualità di un sistema, in particolare per quanto riguarda i seguenti aspetti.

Modificabilità. In caso di modifiche nei requisiti, è possibile circoscrivere le modifiche da apportare a un sistema alle sole componenti ove i requisiti in questione sono realizzati. A tal fine è importante costruire, in fase di definizione

[1] Per analogia possiamo dire che l'architettura di un edificio è l'insieme delle parti dell'edificio (mattoni, travi, finestre, strati di cemento, etc,) con la loro collocazione e non l'insieme di disegni (normalmente chiamato progetto o disegno di progetto) che ne descrivono pianta, spaccato, prospetto.

dell'architettura, una matrice di *tracciabilità* che memorizzi la relazione di soddisfacimento/realizzazione tra requisiti e componenti.

Portabilità e interoperabilità. Per migrare un sistema su una piattaforma differente è sufficiente intervenire sulle componenti di interfaccia con la piattaforma sottostante. L'aver definito l'architettura di un sistema permette di individuare tali componenti[2].

L'interoperabilità può essere vista come sottocaso della portabilità: non solo si può migrare un'applicazione su una piattaforma distinta, ma le singole componenti dell'applicazione possono essere distribuite su varie piattaforme, in modo trasparente allo sviluppatore. Favorire l'interoperabilità significa, tra le altre cose, fornire l'infrastruttura di comunicazione usata dalle componenti[3].

Riuso. In questo contesto il riuso si riferisce a:

- Uso di componenti prefabbricate.

  L'idea che il software potesse essere scomposto in componenti, e che per la sua costruzione si potessero usare componenti prefabbricate è stata presentata da Douglas McIlroy's nel 1968. Con componente prefabbricato si intende il corrispondente di ciò che in edilizia rappresentano un bullone, una putrella o un caminetto prefabbricato. La prima implementazione di questa idea fu l'uso di componenti prefabbricati nel sistema operativo Unix. Un altro esempio è .NET che include un insieme di componenti base riutilizzabili che forniscono varie funzionalità (gestione della rete, sicurezza, etc.).

- Riuso di componenti realizzati in precedenti progetti.

  Il riuso di componenti esistenti mira a sfruttare in un nuovo progetto, e quindi in un nuovo contesto, un componente già realizzato, senza modificarlo. Si può anche progettare un componente in vista di un suo riuso futuro. Poiché i componenti non possono essere modificati ma possono essere estesi, se si progetta in vista di un riuso conviene definire componenti generici.

- Riuso di architetture.

  L'architettura di un sistema può essere riutilizzata per progettare sistemi con requisiti simili.

  Più in generale, è possibile, data l'architettura di un sistema e astraendo dalle peculiarità del sistema, definire un'astrazione dell'architettura. Questa astrazione può essere riusata.

Molte aziende software si configurano come produttori di *linee di prodotto*, piuttosto che di singoli prodotti. Una linea di prodotto consiste in un insieme di prodotti simili per funzionalità, tecnologia, possibili utilizzatori. I vantaggi

[2]Per esempio, la Java Virtual Machine (JVM) specifica una macchina astratta per la quale il compilatore Java genera il codice. Specifiche implementazioni della JVM per piattaforme hardware e software specifiche realizzano le componenti che permettono di portare codice Java su piattaforme diverse.

[3]Modelli e tecnologie che si prefiggono lo scopo di favorire il riuso di componenti e l'interoperabilità tra applicazioni sono ad esempio CORBA (OMG), RMI (Sun–Java), COM+ e .NET (Microsoft)

delle linee di prodotto sono duplici: da un lato si possono sfruttare sinergie di mercato fra prodotti, dall'altro si può praticare un efficace riuso di componenti e di architetture. Spesso, infatti, le linee di prodotto sono basate su un'architettura comune e i sistemi vengono derivati a partire da tale *architettura (di riferimento)*.

Soddisfacimento di requisiti sull'hardware e sulla piattaforma fisica di comunicazione tra nodi hardware distinti. L'architettura comprende la dislocazione del software sui nodi hardware. L'analisi dell'architettura permette di verificare se i requisiti sull'hardware o sul mezzo di comunicazione sono soddisfatti.

Dimensionamento e allocazione del lavoro. Anche la valutazione delle dimensioni del sistema beneficia della definizione di un'architettura, in quanto la misura delle dimensioni di un insieme di componenti risulta più precisa di una misura che si basi solo sulla specifica dell'intero sistema come monolito. Sulla base delle dimensioni dei componenti del sistema si basa l'allocazione del lavoro.

Prestazioni. L'architettura permette di valutare il carico di ogni componente, il volume di comunicazione tra componenti o, per esempio, il numero di accessi a una base di dati.

Sicurezza. L'architettura permette un controllo sulle comunicazioni tra le parti, l'identificazione delle parti vulnerabili ad attacchi esterni e l'introduzione di componenti di protezione (ad esempio, firewall).

Rilascio incrementale. Il modello incrementale di ciclo di vita del software prevede un'iniziale identificazione dei requisiti, seguita dalla definizione dell'architettura e dall'individuazione dei componenti che devono essere realizzati per primi. Questi possono essere, per esempio, i componenti che forniscono le funzionalità più urgenti per il cliente, o i componenti per i quali è utile avere un feedback (per interventi correttivi) prima del completamento del progetto.

Verifica. La definizione dell'architettura del sistema consente in fase di verifica, di seguire un approccio incrementale: verificare prima i singoli componenti, quindi insiemi sempre più ampi di componenti, per giungere, in modo incrementale, alla verifica dell'intero sistema.

3 Perché e come descrivere un'architettura

Un'architettura software è descritta da diagrammi in un linguaggio grafico e risponde alle seguenti necessità:

Comunicazione tra le parti interessate: in particolare tra il progettista, che ha definito l'architettura, e gli sviluppatori, che devono realizzare il sistema.

Comprensione: per l'ovvio motivo che è molto più facile ragionare su qualcosa di descritto e documentato che solo a parole.

Documentazione: la descrizione in forma scritta può essere conservata e consultata in eventuali successivi interventi di modifica al sistema.

In questa sezione definiamo i concetti di: *tipo di vista* su un'architettura software; *vista*; *stile architettonico* e sue relazioni con viste e tipi di viste. Introduciamo quindi i tipi di vista e le viste di cui parleremo nel seguito.

3.1 Descrizione di un'architettura: viste e tipi di vista. Stili

Definiamo il concetto di vista su un'architettura e classifichiamo le viste in tre tipologie. Definiamo anche il concetto, indipendente e ortogonale, di stile architettonico.

Una vista su un'architettura software è una proiezione dell'architettura secondo un criterio. Secondo questa definizione, una vista considera solo alcuni sottosistemi, per esempio considera solo la strutturazione del sistema in componenti, o solo alcune relazioni tra sottosistemi[4].

Un tipo di vista caratterizza un insieme di viste. Seguendo [2] le viste sono così classificate:

- **viste di tipo strutturale:** descrivono la struttura del software in termini di unità di realizzazione. Un'unità di realizzazione può essere una classe, un package Java, un livello, etc.
- **viste di tipo comportamentale** (componenti e connettori): descrivono l'architettura in termini di unità di esecuzione, con comportamenti e interazioni. Un componente può essere un oggetto, un processo, una collezione di oggetti, etc.
- **viste di tipo logistico:** descrivono le relazioni con altre strutture, tipo hardware o organigramma aziendale. Per esempio, l'allocazione dei componenti su nodi hardware.

Stile. Uno stile architettonico è una particolare proprietà di un'architettura. Gli stili architettonici noti sono proprietà che valgono su grandi insiemi di architetture. Per esempio, lo stile a strati è una proprietà delle architetture di tutti i sistemi strutturati in livelli di macchine virtuali[5].

3.2 Organizzazione: elementi, relazioni, proprietà, usi

In queste note la definizione di ciascun tipo di vista e di ciascuna vista è strutturata come segue:

Elementi e Relazioni. Ad ogni tipo di vista (e ad ogni vista) corrisponde un insieme di elementi e di relazioni. Ad esempio, nelle viste di tipo strutturale gli elementi possono essere moduli software, classi, collezioni di classi, e le relazioni quelle di inclusione tra moduli, ereditarietà tra classi.

[4] Una vista su un'architettura software corrisponde a una vista sull'architettura di un edificio (tipo pianta, prospetto, sezione, impianto elettrico, etc). Se si descrive la pianta di una casa è inutile utilizzare la rappresentazione per finestre e persiane che si userebbe per descrivere la facciata.

[5] Similmente, in edilizia lo stile architettonico è una proprietà, tipicamente morfologica, espressa cioè in termini di forma, tecniche di costruzione, materiali, etc. Giusto per fare un esempio, si pensi allo stile gotico, che caratterizza tutte le architetture che soddisfano determinate proprietà, ad esempio avere strutture slanciate o prevedere archi a sesto acuto e di grandi vetrate.

Proprietà. Si specificano informazioni aggiuntive su elementi e relazioni. Ad esempio, in una vista strutturale si suggerisce di indicare l'autore di una classe o modulo.

Usi. Si descrive l'utilità del tipo di vista (o singola vista) nell'economia della descrizione di un'architettura.

Notazioni. I diagrammi che usiamo per descrivere le architetture sono diagrammi UML2 [1]. Per ogni vista vengono elencati i costrutti UML utilizzati.

Per ogni tipo di vista presenteremo prima gli aspetti comuni a tutte le viste del tipo, poi le singole viste. In entrambi i casi definiamo quali possono essere gli elementi, le relazioni, etc.

La definizione di una singola vista si ottiene con una restrizione dell'insieme degli elementi e delle relazioni, rispetto a quanto introdotto per presentare il tipo di vista. Per quanto riguarda i diagrammi, questa specializzazione corrisponde a una restrizione sui costrutti grafici che si possono usare.

3.3 Descrizione di un'architettura

Il disegno di progetto architettonico di un sistema software consiste di una parte introduttiva e della descrizione di ciascuna vista rilevante e di eventuali viste ibride (si veda Sezione 7). Inoltre può contenere la definizione delle relazioni tra le viste, le motivazioni delle scelte operate e i vincoli globali.
La documentazione di ogni vista consiste di

- visione complessiva, spesso grafica
- catalogo degli elementi
- specifica degli elementi: interfacce e comportamento.

4 Viste di tipo strutturale

Le viste di tipo strutturale considerano la struttura di un sistema in termini di unità di realizzazione e sono caratterizzate dai seguenti tipi per elementi, relazioni, etc.

Elementi. Un elemento può essere

- un modulo: unità software che realizza un insieme coerente di funzionalità, ad esempio:
  - una classe;
  - un package: una collezione di moduli, una collezione di classi, un livello;

Relazioni. Una relazione tra elementi può essere:

- parte di
- eredita da
- usa (dipende da)
- può usare

Proprietà. Le proprietà specificano soprattutto caratteristiche degli elementi, e in particolare ne definiscono:

- nome, che deve rispettare le usuali regole dello spazio dei nomi (per esempio non ci possono essere due moduli con lo stesso nome in uno stesso package);
- responsabilità del modulo: funzionalità realizzate;
- visibilità, autore;
- informazioni sulla realizzazione, tipo codice associato, informazioni sul test, informazioni gestionali, vincoli sulla realizzazione;

Usi. Le viste di tipo strutturale sono utili per:

- costruzione: la vista può fornire la struttura del codice che definisce la struttura di directory e file sorgente;
- analisi: tracciabilità dei requisiti (requisiti del sistema soddisfatti dalle funzionalità del modulo) e analisi d'impatto (per predire gli effetti di una modifica nel sistema);
- comunicazione: per esempio, permette di descrivere la suddivisione delle responsabilità nel sistema agli sviluppatori coinvolti in un progetto già avviato o in una fase di manutenzione.

Notazione. Per descrivere i moduli si usa la notazione UML per classi e package. Per quanto riguarda le relazioni, la notazione usata sarà discussa alla fine della descrizione delle singole viste.

Figura 1: Viste di tipo strutturale: notazione UML per gli elementi.

4.1 Vista strutturale di decomposizione

Una vista strutturale in cui le relazioni sono unicamente di tipo **parte di** viene chiamata **vista strutturale di decomposizione**. Il vincolo sul tipo di relazioni corrisponde a restringersi a considerare le relazioni tra moduli di tipo modulopadre–sottomodulo. Si usa questa vista per mostrare come un modulo ad alto livello è raffinato in sottomoduli e come le responsalilità di un modulo sono ripartite tra i moduli figli.

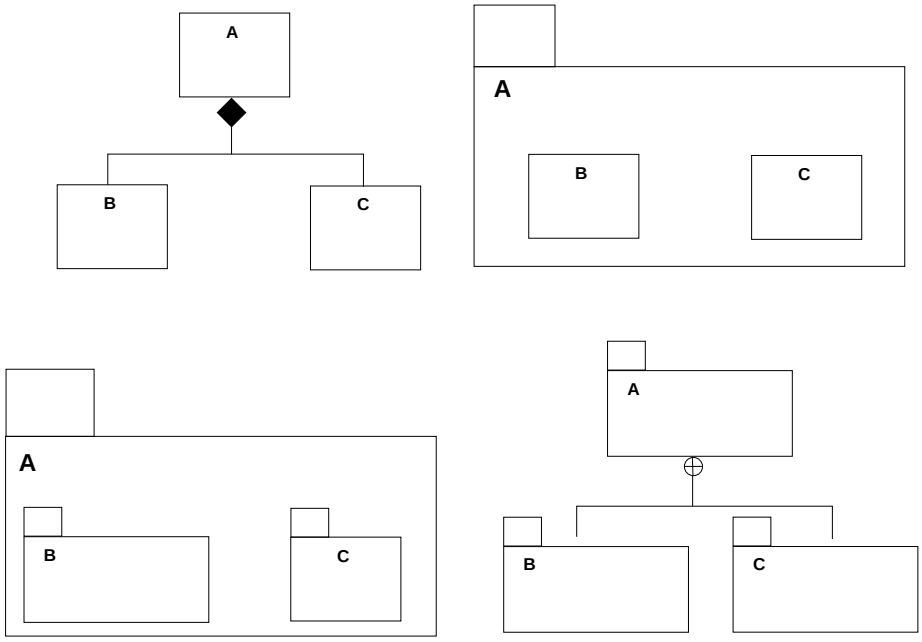

Figura 2: Vista strutturale di decomposizione: per rappresentare la relazione "parte di" si usa la composizione tra classi e l'inclusione tra package.

4.2 Vista strutturale d'uso

Una vista strutturale in cui le relazioni sono unicamente di tipo **usa** viene chiamata **vista strutturale d'uso**. Il vincolo sul tipo di relazioni corrisponde a restringersi a considerare le dipendenze d'uso tra moduli: il modulo A usa ($\langle\langle use \rangle\rangle$) il modulo B se dipende dalla presenza di B (funzionante correttamente) per soddisfare i suoi requisiti.

Questa vista, mettendo in luce le dipendenze funzionali tra i moduli, favorisce:

la pianificazione di uno sviluppo incrementale del sistema: secondo questo modello di sviluppo, durante la fase di progettazione architettonica, vengono definite delle priorità tra moduli, sulla base dei seguenti aspetti:

- aspetti di natura funzionale, relativi cioè alle esigenze dei committenti e delle parti interessate: una priorità alta caratterizza i moduli più urgenti.
- aspetti tecnologici, relativi alla difficoltà di realizzazione di un modulo, e alla stabilità dei linguaggi e degli strumenti che si prevede di usare per la realizzazione del modulo. I moduli ritenuti più complessi e problematici vengono ritenuti a priorità alta. La priorità alta viene invece normalmente assegnata ai moduli con tecnologie di realizzazione più stabili, mentre si rimandano, ad esempio, moduli che devono essere realizzati in un linguaggio di cui sta per uscire una nuove versione.
- aspetti di natura architettonica: se il modulo A usa il modulo B, allora B ha una priorità maggiore.

Sulla base delle priorità definite, la realizzazione del sistema avviene per passi incrementali: partendo dalla realizzazione del modulo o insieme di moduli a priorità maggiore, per quindi aggiungere in un secondo tempo i moduli a priorità bassa.

il test incrementale del sistema, agevolando la progettazione di stub e driver.

l'analisi d'impatto. Questa analisi valuta le ripercussioni di una modifica nei requisiti o nella realizzazione di un modulo sull'intero sistema. Una descrizione secondo la vista strutturale d'uso favorisce questo tipo di analisi: la modifica di un modulo A può richiedere modifiche nei moduli che usano A.

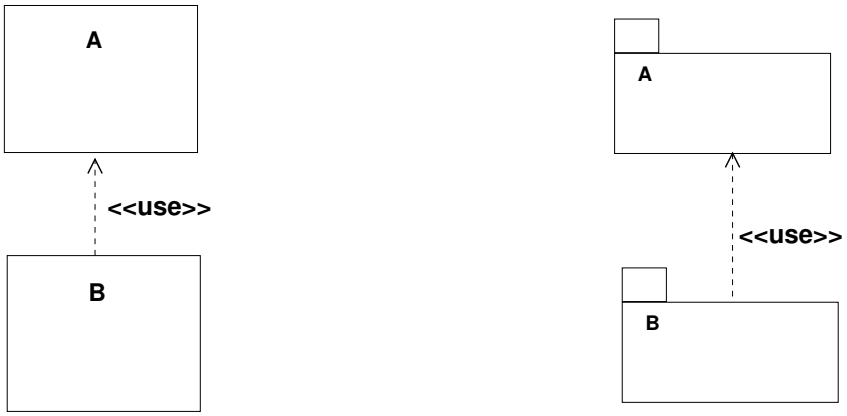

Figura 3: Vista strutturale d'uso: per rappresentare la relazione "usa (dipende da)" si utilizza la freccia che indica dipendenza, etichettata con $\langle\langle use \rangle\rangle$.

4.3 Vista strutturale di generalizzazione

Una vista strutturale in cui le relazioni sono unicamente di tipo **eredita da** viene chiamata **vista strutturale di generalizzazione**. Questa vista è utile soprattutto per descrivere architetture ottenute per istanziazione di un framework.

Un framework è definito da un insieme di classi astratte e dalle relazioni tra esse. Istanziare un framework significa fornire un'implementazione delle classi astratte. L'insieme delle classi concrete, definite ereditando il framework, eredita le relazioni tra le classi. Si ottiene in questo modo un insieme di classi concrete, con un insieme di relazioni tra classi. La descrizione di un'architettura secondo la vista di generalizzazione mostra la relazione tra il framework e l'istanza.

Vi è una notevole differenza tra l'uso di un framework e l'uso di una libreria per la realizzazione di un'applicazione. Usando una libreria, per ogni classe che si realizza, si deve decidere quali saranno gli oggetti, istanza delle classi della libreria, invocati dai metodi della classe che si sta realizzando. Quando si usa un framework e si realizza una classe come istanza di una classe astratta del framework, si eredita anche un insieme di relazioni con altre classi, che vincolano e guidano la realizzazione dei metodi della classe.

La figura 4 illustra la notazione usata per descrivere le generalizzazioni. La generalizzazione tra package è di fatto un abuso di notazione, con il significato che alcune delle classi contenute nel package A ereditano da alcune classi del package B.

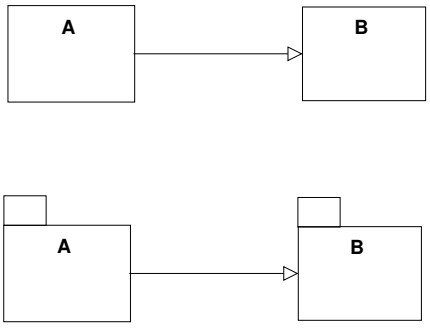

Figura 4: Vista strutturale di generalizzazione: La relazione "eredita da" è resa come una generalizzazione.

4.4 Vista strutturale a strati

Una vista strutturale in cui le relazioni sono unicamente di tipo **può usare** viene chiamata **vista strutturale a strati**.

Lo stile a strati[6] è da sempre uno dei più usati nelle architetture: questa vista ha volutamente lo stesso nome.

Secondo lo stile a strati il sistema è strutturato in livelli di macchine virtuali: l'elemento architettonico di interesse è uno strato, cioè un insieme di moduli, che mette a disposizione un'interfaccia per i suoi servizi. Dire che "A può usare B" significa che l'implementazione di A può usare qualsiasi servizio messo a disposizione

[6]Sinonimi sono stile a layer e stile a macchine virtuali.

da B. Con servizio si intende un insieme di funzionalità con un'interfaccia comune. Il nome "a strati" è legato al fatto che tradizionalmente le architetture a macchine virtuali vengono modellate come una pila di rettangoli bassi e larghi, dove ogni strato usa quello sottostante.

Una vista a strati non differisce molto da un caso particolare di una vista secondo lo stile **uso**: uno strato è un insieme coeso di moduli (a volte raggruppato in segmenti) e la divisione in stati costituisce una partizione dell'insieme dei moduli. Le relazioni d'uso sono ristrette a coppie di strati: è raro che una macchina virtuale faccia uso di servizi messi a disposizione dalla macchina virtuale che sta due livelli sotto, e solo in pochi casi eccezionali può accedere ai servizi della macchina sovrastante.

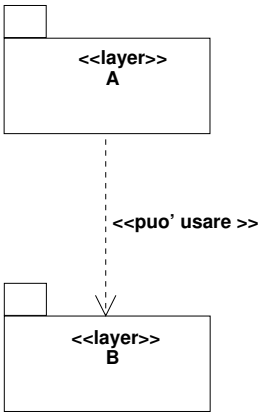

Figura 5: Vista strutturale a strati: la relazione "può usare" è resa con una dipendenza etichettata con $\langle\langle puo' \; usare \rangle\rangle$.

Un esempio familiare di architettura a strati è l'ambiente di esecuzione di programmi Java (Java platform): la macchina virtuale Java (Java Virtual Machine o JVM) rappresenta un'astrazione del sistema operativo e costituisce uno strato tra questo e l'applicazione nel bytecode generato dalla compilazione del codice sorgente.

5 Vista comportamentale

La vista comportamentale, anche chiamata a componenti e connettori (C&C), considera la struttura del sistema in termini di unità di esecuzione, con comportamenti e interazioni.

Data un'architettura, esiste un unico tipo di vista comportamentale (diversamente dagli altri due tipi di vista, strutturale e logistico). Questa vista è caratterizzata come segue:

Elementi. In questa vista gli elementi sono:

- componenti, cioè unità concettuali di decomposizione di un sistema a tempo di esecuzione, quali un processo, un oggetto, un deposito di dati, un servente.

  Un componente può non corrispondere esattamente ad alcun eseguibile.

Descrivendo la struttura del sistema a tempo di esecuzione, in una vista possono comparire due o più istanze di un tipo di componente. Basti pensare alla necessità di rappresentare, in un sistema costruito con tecnologia a oggetti, la compresenza di due o più oggetti istanza di una data classe.

- connettori, cioè canali di interazione tra componenti. Un connettore modella, ad esempio, un protocollo, un flusso d'informazione, un modo di accedere a un deposito dati.

  Per comprendere l'importanza della descrizione dei connettori, si pensi all'interazione tra un client e un server. Il connettore rappresenta un protocollo di interazione che può prevedere un'autenticazione iniziale del client, vincoli sull'ordine in cui possono essere fatte le richieste al server, una gestione dei fallimenti, una chiusura della sessione.

  Un connettore può collegare più di due componenti.

Proprietà. Le proprietà di

- un componente sono: il nome, il tipo (numero e tipo dei porti), altre informazioni tipo prestazioni, affidabilità, etc.

  I *porti* identificano i punti di interazione di un componente e sono caratterizzati dalle interfacce che forniscono e/o richiedono. Il tipo di un porto caratterizza le informazioni che possono fluire attraverso il porto.

- un connettore sono: il nome, il tipo (numero e tipo dei ruoli), caratteristiche del protocollo, prestazioni, etc.

  I *ruoli* identificano il ruolo dei partecipanti in un'interazione. Il tipo di un ruolo vincola il tipo dei partecipanti.

Relazioni. Una relazione tra elementi di questa vista collega i ruoli dei connettori con i porti dei componenti.

Usi. La vista comportamentale è utile per:

- analisi delle caratteristiche di qualità a tempo d'esecuzione, quali: funzionalità, prestazioni, affidabilità, disponibilità, sicurezza. Per esempio, la conoscenza del livello di sicurezza dei singoli componenti e dei canali di comunicazione permette di stimare il grado di sicurezza dell'intero sistema e di evidenziare eventuali componenti vulnerabili;
- documentazione e comunicazione della struttura del sistema in esecuzione. Si descrivono, ad esempio, flusso dei dati, dinamica, parallelismo, replicazioni;
- descrivere stili architetturali noti, quali condotte e filtri, client–server, etc.

Notazione. Il simbolo UML per i componenti è un rettangolo con, oltre al nome del componente, un'icona e/o la parola chiave $\langle\langle component \rangle\rangle$.

In UML un *artefatto (artifact)* rappresenta un pezzo di informazione fisica usato o prodotto durante il processo di sviluppo software o dalla dislocazione e operazione di un sistema. Esempi di artefatti sono gli eseguibili, i sorgenti e i file di dati.

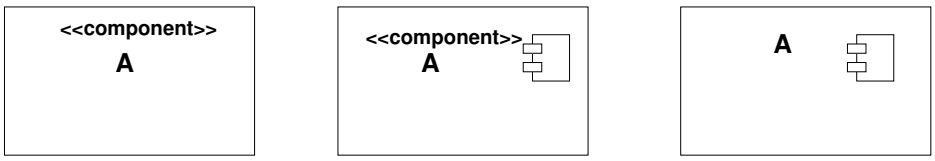

Figura 6: Rappresentazione di un componente.

La notazione UML per gli artefatti è un rettangolo con la parola chiave $\langle\langle artifact\rangle\rangle$ e/o l'icona del file.

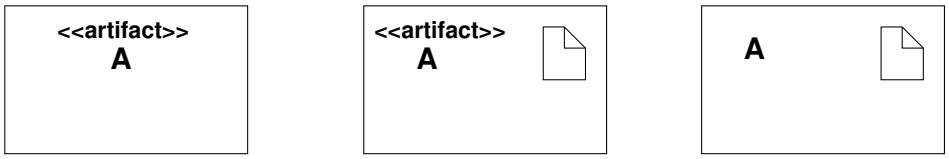

Figura 7: Rappresentazione di un artefatto.

La relazione tra un componente e i corrispondenti artefatti è una dipendenza etichettata $\langle\langle manifest\rangle\rangle$.

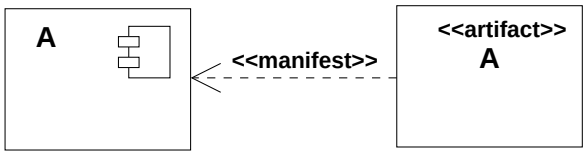

Figura 8: Relazione $\langle\langle manifest\rangle\rangle$.

I porti sono associati ai componenti e rappresentati con quadratini sul bordo del rettangolo. Le interfacce che caratterizzano un porto sono rappresentate usando i *lollipop* e le *forchette* (anche chiamata notazione *ball–and–socket*).

Alcuni autori descrivono i connettori usando, ad esempio, classi UML[7]. Noi invece li rappresentiamo in due modi:

- collegando un'interfaccia richiesta con una esportata;
- con una linea tra due (o più) porti.

La scelta dipende dalla quantità di dettagli che è necessario descrivere. Per esempio, è preferibile usare lollipop e forchette se si vuole evidenziare il fatto che le due componenti comunicano attraverso chiamate alle operazioni dell'interfaccia.

[7] C'è un'accesa discussione in atto nella comunità scientifica sull'adeguatezza di UML per descrivere architetture software, soprattutto dal punto di vista comportamentale. I punti di discussione riguardano in particolare il potere espressivo di UML nel descrivere i connettori. Nei più noti Architectural Description Languages (ADL) i connettori hanno una semantica che può essere anche complessa, per esempio per esprimere un protocollo di interazione, una coda o una politica di sicurezza. In UML, invece, un connettore è un'associazione tra componenti che rappresenta un canale di comunicazione, senza la possibilità di esprimere comportamenti specifici.

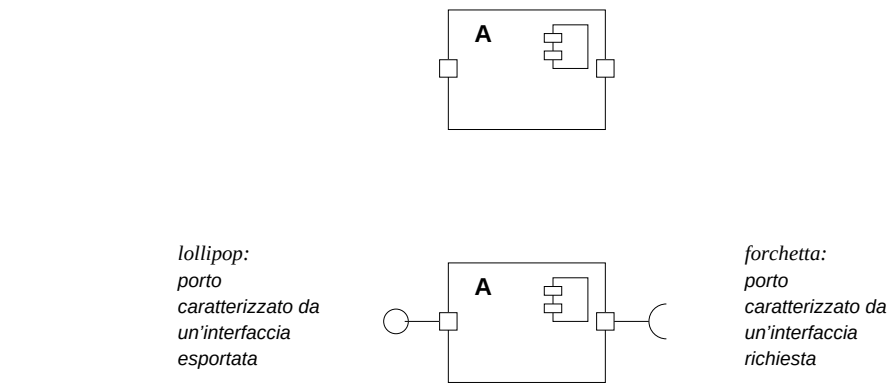

Figura 9: Rappresentazione di porti.

Inoltre è possibile:

- specificare i ruoli agli estremi del connettore;
- etichettare i connettori per esprimerne, se non la semantica, almeno il tipo (si usa uno stereotipo);
- indicare, nel caso di collegamento diretto tra porti, il flusso delle informazioni usando una freccia (la freccia che si usa nei diagrammi delle classi per indicare la navigabilità).

In Figura 10 il primo esempio mostra come rappresentare un connettore usando lollipop e forchette e come indicare i ruoli dei componenti (di fatto come i ruoli delle classi in un'associazione). Il secondo esempio mostra un generico connettore come linea tra porti. Nel terzo esempio, un connettore etichettato $\langle\langle pipe \rangle\rangle$ rappresenta una generica *condotta* (una *pipe* tra una coppia di componenti *filtro*). Se si indica il tipo dei porti agli estremi del connettore, una condotta collega un porto di tipo out con un porto di tipo in.

5.1 Vista comportamentale per descrivere gli stili

Molti tra i più comuni stili architettonici sono di fatto caratterizzati da aspetti comportamentali, e in particolare dal tipo dei componenti e dal tipo dei connettori.

Nei prossimi paragrafi ne presenteremo alcuni, mostrando come le loro peculiarità possano essere catturate e descritte in una vista comportamentale. Spesso uno stile architettonico viene caratterizzato vincolando elementi e relazioni della vista comportamentale.

5.1.1 Condotte e filtri (pipe & filters)

In questo stile i componenti sono di tipo *filtro* e i connettori di tipo *condotta*. Caratteristica di un filtro è quella di ricevere una sequenza di dati in input e produrre una sequenza di dati in output. Due o più filtri possono operare in parallelo: un filtro a valle può operare sui primi dati della sequenza di output di un filtro a monte, mentre questo continua a elaborare la sua sequenza di input.

Casi particolari sono la *pipeline* in cui si restringe la topologia a una sequenza lineare di filtri e i *bounded pipes* in cui si fissa una capienza massima per le condotte.

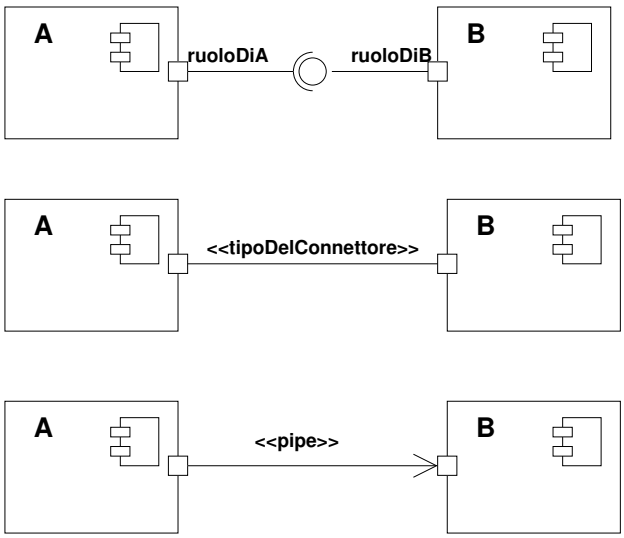

Figura 10: Rappresentazione dei connettori.

5.1.2 Dati condivisi (Shared data)

E' uno stile focalizzato sull'accesso a dati condivisi tra vari componenti. Prevede un componente che mantiene lo stato condiviso, per esempio una base di dati, e un insieme di componenti indipendenti che operano sui dati. Lo stato condiviso, almeno in un sistema a dati condivisi "puro", è l'unico mezzo di comunicazione tra i componenti.

Un connettore che colleghi la base di dati con i componenti che operano sui dati può descrivere, ad esempio, un protocollo di interazione che inizia con una fase di autenticazione.

5.1.3 Publish-subscribe

I componenti interagiscono annunciando eventi: ciascun componente si *abbona* a classi di eventi rilevanti per il suo scopo.

I componenti sono caratterizzati da un'interfaccia che pubblica e/o sottoscrive eventi. Un connettore di tipo publish–subscribe è un bus di eventi: i componenti, pubblicando gli eventi, li consegnano al bus, il quale li consegna ai componenti (consumatori) appropriati.

Un'architettura di questo tipo disaccoppia produttori e consumatori di eventi: un produttore invia un evento senza conoscere il numero né l'identità dei consumatori. In questo modo sono possibili modifiche dinamiche del sistema, in cui, ad esempio, varia l'insieme dei consumatori.

5.1.4 Cliente-servente (Client-server)

In questo stile i componenti sono di tipo cliente o di tipo servente. Le interfacce dei serventi descrivono i servizi (o in generale le funzionalità) offerti; le interfacce dei clienti descrivono i servizi usati.

Le comunicazioni sono iniziate dai clienti e prevedono una risposta da parte dei serventi. I clienti devono conoscere l'identità dei serventi, mentre il viceversa non vale, l'identità del cliente è comunicata assieme alla richiesta di servizio.

I connettori rappresentano un protocollo di interazione che prevede nel caso base una domanda e una risposta. Può però anche specificare, ad esempio, che i clienti inizino una sessione con il servizio, rispettino eventuali vincoli sull'ordine delle richieste, chiudano la sessione.

5.1.5 Da pari a pari (peer to peer)

Nello stile peer to peer (P2P) i componenti sono sia clienti sia serventi e interagiscono alla pari, per scambiarsi servizi. Le interazioni sono del tipo "richiesta–risposta", ma non sono asimmetriche come nel case cliente-servente: i connettori sono di tipo *invokes–procedure* e ammettono che l'interazione sia iniziata da entrambi i lati.

L'esempio classico di P2P sono le reti per la condivisione di file. In realtà, alcune reti usano interazioni di tipo client–server per alcuni compiti, ad esempio la ricerca di peer, e interazioni P2P per tutti gli altri. I problemi legali di Napster, per esempio, sono nati proprio dall'esistenza di server che mantenevano la lista dei sistemi connessi e dei file condivisi.

6 Viste di tipo logistico

Le viste di tipo logistico considerano le relazioni tra un sistema software e il suo contesto: file system, hardware e sviluppatori. Sono caratterizzate come segue:

Elementi. Gli elementi sono:

- Elementi software di altre viste: moduli, componenti e connettori;
- Elementi dell'ambiente: hardware, struttura di sviluppo, file system.

Relazioni. La relazione *allocato* permette di mappare un elemento software su un elemento dell'ambiente. Per esempio come mappare un eseguibile su un elemento hardware o un modulo sui file e le directory di un file system.

Altre relazioni sono caratteristiche delle singole viste.

Proprietà. Un elemento software richiede delle proprietà che devono essere fornite dall'ambiente.

Usi. Mappando un'architettura sull'hardware è possibile analizzare le prestazioni del sistema, la sua resistenza ai guasti (fault tolerance), le caratteristiche di sicurezza; la mappatura sul gruppo di sviluppatori permette di pianificare e gestire il processo di sviluppo; infine, la mappatura sul file system permette di gestire versioni e configurazioni.

6.1 Vista logistica di dislocazione (deployment)

Questa vista considera la mappatura degli artefatti, per esempio gli eseguibili, sugli elementi (processori, dischi, canali di comunicazione, sistemi operativi, ambienti di esecuzione, etc.) su cui viene eseguito il sistema (relazione *allocato*).

Un caso particolare è la *vista sull'hardware*, che considera solo gli elementi hardware e gli ambienti di esecuzione, senza mostrare la dislocazione degli artefatti. Questa vista può essere utile soprattutto nelle prime fasi del processo di sviluppo per descivere l'ambiente su cui dovrà essere eseguito il sistema da sviluppare.

6.1.1 Notazione

In UML un diagramma di dislocazione è un grafo di nodi (parallelepipedi), dove ogni nodo rappresenta una risorsa computazionale. I nodi sono connessi da associazioni che descrivono canali fisici o protocolli di comunicazione. Le caratteristiche di un canale di comunicazione (ad esempio *wireless*) possono essere documentate con uno stereotipo. Esempi dei più comuni stereotipi sono descritti nell'appendice.

I nodi UML possono essere classificatori (tipi di nodo) o istanze. La notazione segue quella usata per classi e oggetti. Se si fornisce solo un nome, non sottolineato, si intende il tipo, mentre nel caso delle istanze si indicano il nome e/o il tipo, separati da ":" e sottolineati.

La relazione di allocazione è rappresentata disegnando gli artefatti (artifact) all'interno dei nodi. Una rappresentazione alternativa è una dipendenza etichettata $\langle\langle deploy \rangle\rangle$ dall' artefatto al nodo. Eventuali dipendenze tra artefatti sono rappresentate con frecce di dipendenza (tratteggiate).

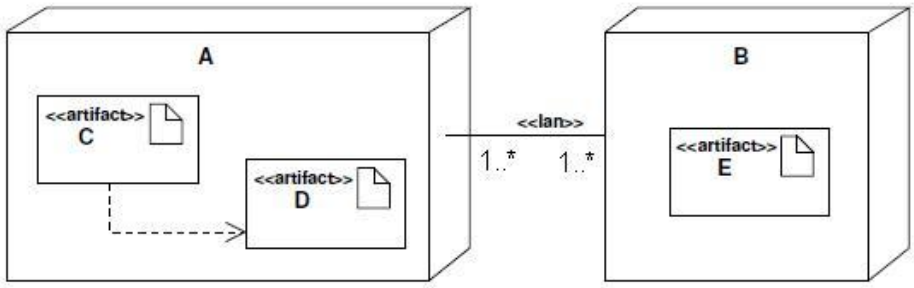

Figura 11: Vista logistica di dislocazione.

6.2 Vista logistica di realizzazione

Questa vista considera la mappatura di moduli su file e directory. Un modulo corrisponde a molti file: quelli che contengono i codice sorgente, file che contengono definizioni e che devono essere inclusi, file che descrivono come costruire un eseguibile (tipo un makefile), file risultato della compilazione.

Gli elementi di questa vista sono quindi da un lato i moduli software, dall'altro gli elementi di configurazione, tipo un file o una directory. Le relazioni sono *allocato* (tra un modulo e un elemento di configurazione) e *contiene* (tra una directory e una sotto–directory o file contenuti).

6.2.1 Notazione

Ci sono due possibili notazioni per questa vista: una testuale, per esempio usando un foglio Excel, e una grafica, che usa UML.

Nel caso grafico, rappresentiamo sia i moduli sia gli elementi dell'ambiente come package o rettangoli (classi) UML. La relazione *allocato* è una dipendenza etichettata $\langle\langle allocato \rangle\rangle$, con direzione dai moduli agli elementi dell'ambiente. La relazione *contiene* è rappresentata dall'inclusione tra package e dall'inclusione di file o classi in un package.

In Figura 12 si mostrano due esempi di relazione di allocazione, il modulo A nel primo caso è mappato su una directory, nel secondo caso su un file di tipo jar.

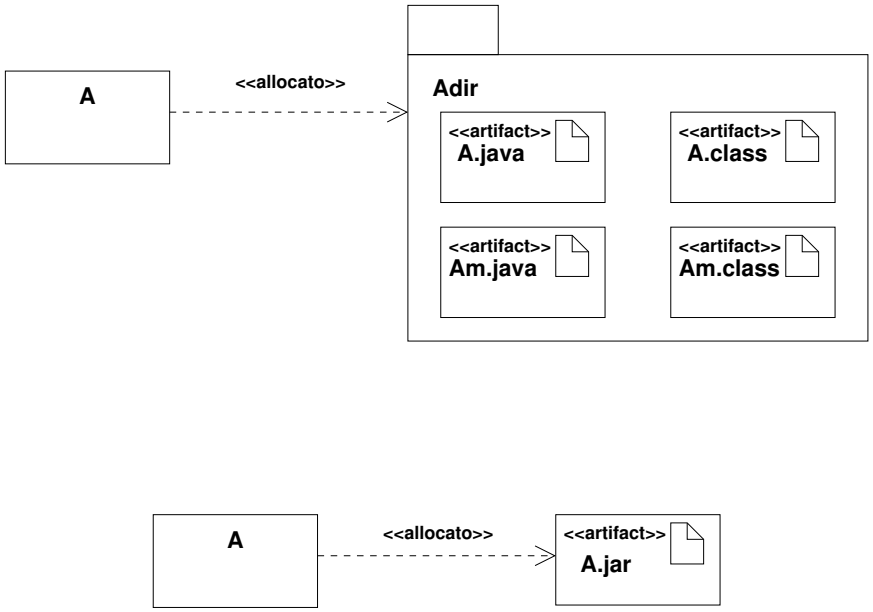

Figura 12: Vista logistica di realizzazione: due esempi.

6.3 Vista logistica di assegnamento del lavoro

Questa vista considera la mappatura di moduli su persone o gruppi di persone incaricate della loro realizzazione.

Gli elementi sono da un lato i moduli, dall'altro le singole persone, i gruppi, le divisioni, o ditte fornitrici esterne. La relazione è *allocato*, dai moduli alle persone.

6.3.1 Notazione

Non esiste un diagramma UML dedicato alla descrizione di questa vista. La scelta consigliata è mantenere l'usuale rappresentazione dei moduli software (rettangoli di classe e/o package) e di usare gli stessi elementi anche per gli elementi dell'ambiente. In particolare l'uso di package permette di descrivere strutture aziendali in cui le persone sono raggruppate in gruppi e insiemi di gruppi definiscono, ad esempio, una divisione. La relazione *allocato* è rappresentata con una dipendenza etichettata $\langle\langle allocato \rangle\rangle$.

Alternativamente, possono essere usate rappresentazioni testuali, ad esempio con fogli Excel.

7 Viste ibride

Talvolta è utile descrivere un'architettura secondo più di un punto di vista. In questi casi si dice che si usa una vista ibrida. Le più utilizzate sono la vista ibrida di dislocazione con componenti e la vista ibrida strutturale con componenti.

7.1 Vista ibrida: dislocazione con componenti

In questa vista, data una vista logistica di dislocazione, si mappano i componenti sugli artefatti. La relazione tra questi elementi si chiama *manifests*: un artefatto (o più) manifesta un componente.

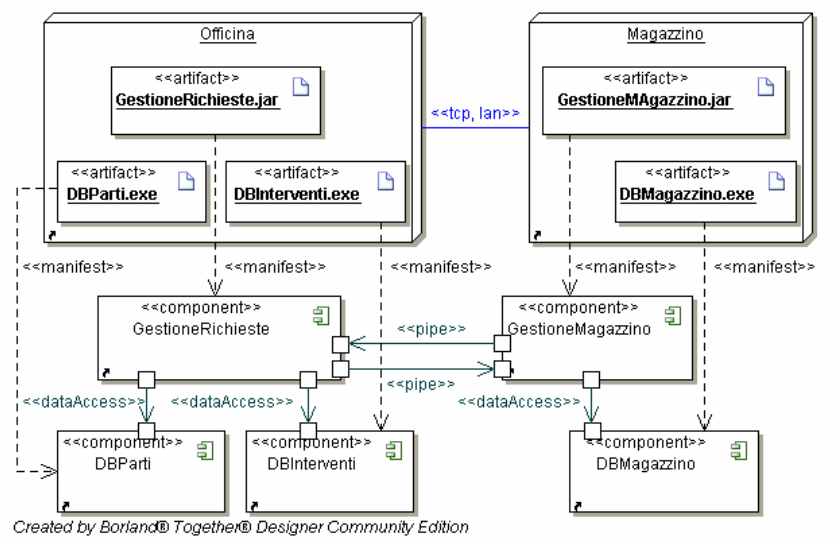

Figura 13: Un esempio di vista ibrida: dislocazione con componenti.

In Figura 13 si mostrano: una vista logistica di dislocazione in forma istanza nella parte alta; una vista comportamentale nella parte bassa; le relazioni *manifests* rappresentate come dipendenze di artefatti da componenti.

7.2 Vista ibrida: strutturale con componenti

In questa vista ibrida si mappa una vista di tipo strutturale su una vista di tipo comportamentale, mostrando quali componenti corrispondono agli elementi della vista strutturale.

Le relazioni tra componenti e moduli possono essere anche molto complesse: un modulo può servire per l'esecuzione di più componenti; simmetricamente, per eseguire un componente può essere necessario il codice di più di un modulo. Una libreria di Input/Output può corrispondere a un connettore.

La Figura 14 dà nella parte bassa una vista comportamentale dell'architettura di un sistema, che possiamo chiamare Alternatore, che presa una sequanza di caratteri la trasforma alternando maiuscole e minuscole: ad esempio "ABcDE" diventa "AbCdE". Il sistema è realizzato nelle stile "pipe and filter", e i suoi componenti sono descritti di seguito:

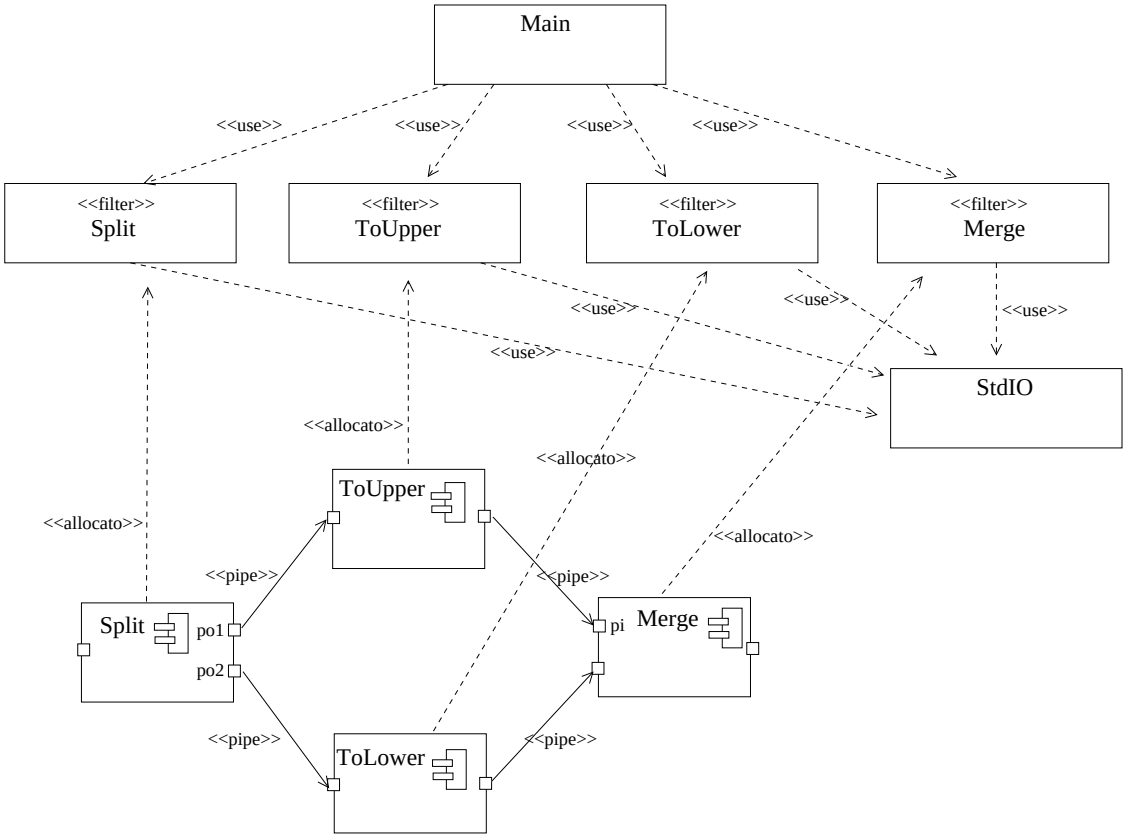

Figura 14: Un esempio di vista ibrida: strutturale con componenti.

Split: smista i caratteri nel flusso di ingresso alternativamente verso i porti di uscita *po1* e *po2*. Nell'esempio, abbiamo rispettivamente i flussi "AcE" e "BD".

ToUpper: trasforma i caratteri in maiuscole: "AcE" diventa "ACE".

ToLower: trasforma i caratteri in minuscole: "BD" diventa "bd".

Merge: fonde i due flussi in ingresso in un unico flusso in uscita, alternando i caratteri, a partire da flusso in *pi*.

La parte alta della figura presenta una vista d'uso dei moduli: oltre a quelli che corrispondono ai componenti della vista C&C, come mostrato dalla dipendenza $\langle\langle allocato \rangle\rangle$, abbiamo altro moduli, in particolare quello che viene usato per realizzare le pipe, StdIO, e il Main, che si occupa di creare e connettere i moduli come mostrato nella parte bassa. Per semplicità, non abbiamo mostrato la dipendenza $\langle\langle allocato \rangle\rangle$ tra i connettori e il modulo StdIO.

8 Glossario

Artefatto (artifact) Un artefatto (artifact) è un elemento concreto di informazione che è usato o prodotto da un processo di sviluppo software o durante l'esecuzione di un sistema. Esempi di artefatti sono: un file sorgente, un file binario eseguibile, un messaggio, un documento XML, un'immagine, uno script, un database (o anche una tabella di database), un jar. Un artefatto può essere composto da altri artefatti.

Gli artefatti manifestano, nel senso che realizzano, i componenti, o più in generale gli elementi del modello. Possono esserci più artefatti associati a un componente, anche allocati su nodi distinti. Un componente può essere realizzato su nodi distinti. Solo gli artefatti risiedono (sono allocati) su nodi hardware, non i componenti.

La realizzazione di un modulo software è memorizzato in artefatti. Per esempio una classe (pensata come modulo, e quindi come entità concettuale) è realizzata da un frammento di programma in un linguaggio di programmazione, memorizzato in un artefatto, il file sorgente. Il risultato della compilazione del sorgente è ancora un artefatto: un file che contiene la realizzazione (eseguibile) della classe.

Componente Un componente è unità concettuale di decomposizione di un sistema a tempo di esecuzione. È caratterizzata dalla sua interfaccia, definita in termini di porti. Caratteristica dei componenti è la rimpiazzabilità: posso sostituire un componente di un sistema con un'altro che abbia la stessa interfaccia, senza pregiudicare il funzionamento dell'intero sistema.

Connettore Un connettore è un canale di interazione tra componenti. Un connettore modella un protocollo, un flusso d'informazione, un modo di accedere a un deposito dati, etc.

Framework Un framework è una generalizzazione di un sistema software. È definito da un insieme di classi astratte e dalle relazioni tra esse. Istanziare un framework significa fornire un'implementazione delle classi astratte. L'insieme delle classi concrete, definite istanziando il framework, eredita le relazioni tra le classi. Si ottiene in questo modo un insieme di classi concrete, con un insieme di relazioni tra classi.

Il definire le interazioni tra le classi permette di classificare un framework come un *modello collaborativo*. Lo sviluppatore non scrive codice per coordinare i componenti. Lo sviluppatore deve determinare i componenti che, aderendo alla logica collaborativa del framework, verranno coordinate da quest'ultimo.

Interfaccia fornita Un'interfaccia fornita descrive le operazioni che un componente implementa e rende disponibili ad altri componenti.

Interfaccia richiesta Un'interfaccia richiesta descrive le operazioni che un componente richiede da altri componenti. Non è detto che tutte le operazioni richieste siano effettivamente usate, ma il componente è garantita funzionare se i componenti che usa forniscono almeno le operazioni richieste.

Modulo Un modulo è un'unità (concettuale) di software che realizza un insieme coerente di responsabilità. Esempi sono una classe, un insieme di classi, la specifica di una macchina astratta.

Porto Un porto identifica un punto di interazione di un componente, È caratterizzato dalle interfacce che fornisce e/o richiede.

Ruolo Un ruolo specifica un connettore identificando il ruolo dei partecipanti in un'interazione.

9 Concetti e tecnologie legati alle architetture

Design Pattern. Un design pattern è, informalmente, la soluzione generale di un problema ricorrente. Creati in architettura dall'architetto Christopher Alexander, i design pattern hanno trovato grande successo in ambito informatico: nomi come Abstract Factory, Command, Proxy e molti altri sono facilmente riscontrabili nella documentazione tecnico-realizzativa dei prodotti più recenti.

Un pattern è una descrizione astratta di oggetti e classi cooperanti, che quando viene istanziata risolve un problema di progettazione. Il riferimento più noto sui patter è [3].

Schemi architettonici o architectural pattern: pattern a livello architettonico. È un'astrazione di un frammento di architettura, che mostra il tipo di uno o più elementi e le relazioni tra questi.

Un esempio è mostrato in Figura 15 dove una coppia di componenti (filters) sono collegati da una condotta (pipe). Questo schema architettonico rappresenta lo schema base di ogni architettura in stile "pipes & filters".

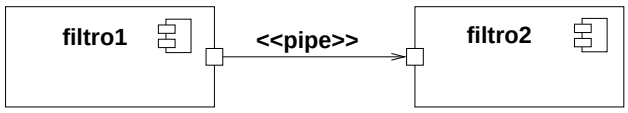

Figura 15: Esempio di schema architettonico.

Architetture di riferimento. Un'architettura di riferimento è la generalizzazione di più architetture (simili) di sistemi realizzati per uno stesso dominio applicativo.

Possono essere viste come schemi, o composizioni di schemi, istanziate su un dominio applicativo. Un ulteriore passo di istanziazione collega un'architettura di riferimento con l'architettura di un sistema specifico.

Un esempio noto è l'architettura tradizionale di un compilatore, una sequenza di "pipes & filters", dove i filtri sono, nell'ordine: un analizzatore lessicale (o scanner), un analizzatore sintattico (parser), un analizzatore semantico, un ottimizzatore di codice intermedio, un generatore di codice sorgente.

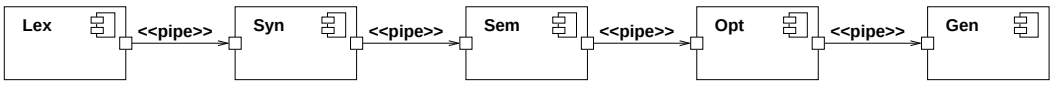

Figura 16: Esempio di architettura di riferimento

CORBA è l'insieme delle interfacce e dei modelli di riferimento che compongono la Object Management Architecture, un modello di architettura per lo sviluppo di applicazioni distribuite. Componente chiave di CORBA è l'Object Request Broker (ORB), che realizza tutte le funzionalità di serializzazione e di trasmissione delle richeste su rete.

COM+ è il middleware di Microsoft per lo sviluppo di applicazioni distribuite che integra COM (Component –o Common– Object Model, il modello a componenti derivato da OLE) e DCOM (l'estensione di COM per le applicazioni distribuite).

Java Remote Method Invocation (Java RMI) permette l'invocazione di oggetti Java da altre macchine virtuali, che possono risiedere su host distinti.

.NET è una tecnologia per lo sviluppo software di Microsoft caratterizzata da requisiti di interoperabilità e indipendenza dalla piattaforma hardware e software. .NET nasce come evoluzione di COM/COM+. Il Framework .NET è la parte centrale della tecnologia: è l'ambiente per la creazione, la distribuzione e l'esecuzione di tutti gli applicativi che supportano .NET. Il Common Language Runtime è il motore d'esecuzione della piattaforma .NET esegue cioè codice IL (Intermediate Language) compilato con compilatori che possono avere come target il CLR.

Marshalling & unmarshalling. Il marshalling permette di convertire un oggetto, o una struttura complessa dalla loro rappresentazione in memoria in una sequenza di byte o di caratteri (http) da trasmettere in rete. La conversione può essere necessaria anche per i tipi semplici, in quanto piattaforme diverse possono rappresentare i tipi in modo diverso (si pensi ai diversi modi di codificare i caratteri, tipo ASCII o Unicode). I parametri prima di essere spediti vengono convertiti in un formato opportuno. Il recupero dei dati da parte del ricevente viene detto unmarshalling.

Corba con Common Data Representation definisce un formato esterno di rappresentazione sia per i tipi semplici sia per quelli strutturati che può essere utilizzato da una varietà di linguaggi di programmazione.

Java usa la serializzazione e la comunicazione via Remote Method Invocation (RMI). La serializzazione definisce sia un formato esterno di rappresentazione sia un metodo per "appiattire" oggetti comunque complessi.

10 Stereotipi per architetture

10.1 Viste C&C

Riportiamo di seguito alcuni stereotipi di uso comune per indicare le proprietà delle relazioni tra componenti nella vista comportamentale.

Stereotipo	Padre
clientServer	
dataAccess	clientServer
masterSlave	
pipe	
peer2peer	
publish, subscribe	

Note

dataAccess Questo connettore normalmente comporta l'uso, da parte del cliente, di un driver ODBC (Open Database Connectivity), JDBC (Java Database Connectivity - utilizzabile solo se si sa che la componente verr realizzata in java, e quindi manifestata da un artefatto .jar o .class) o similari.

publish, subscribe Questi vengono usati nello stesso diagramma, spesso su cammini diversi, diretti verso una componente di gestione del protocollo.

10.2 Viste di dislocazione

Riportiamo di seguito alcuni stereotipi di uso comune per indicare le proprietà di elementi e relazioni delle viste logistiche. In alcuni casi si hanno delle gerarchie di specializzazione, definite dall'indicazione del *padre* (l'elemento più generale) nella relazione.

Natura dei nodi di elaborazione

Stereotipo	Padre
device	
ambienteEsecutivo	
browser	ambienteEsecutivo
webserver	ambienteEsecutivo
ftpclient	ambienteEsecutivo
ftpserver	ambienteEsecutivo
so	ambienteEsecutivo
windows	so
linux	so
os10	so
jvm	ambienteEsecutivo

Caratteristiche dei cammini di comunicazione

Natura del mezzo trasmissivo

Stereotipo	Padre
puntoApunto	
senzaFili	puntoApunto
cavo	puntoApunto
db9	cavo
rj45	cavo
usb	cavo
parallelo	cavo
diretta	senzaFili
accessPoint	senzaFili
lan	
man	
wan	

Natura del protocollo Si intende il protocollo di livello più alto disponibile, sui nodi connessi dal cammino di comunicazione, per l'applicazione da progettare.

Stereotipo	Padre	Nodi compatibili	Mezzi compatibili	Note
http		browser, webserver	wan, man, lan	Hyper Text Transport Protocol
ftp		ftpclient ftpserver	wan, man, lan	File Transport Protocol
ssl		so	wan, man, lan	Secure Socket Protocol
custom		so	wan, man, lan	
tcp		so	wan, man, lan	Transport Control Protocol
wifi		so	senzaFili	WIreless FIdelity
seriale		so	puntoApunto	
rs232	seriale	so	puntoApunto	
xonxoff	seriale	so	puntoApunto	
xonxoffchs	seriale	so	puntoApunto	Xon-Xoff con checksum

Note

so Un sistema operativo normalmente contiene adeguate librerie di comunicazione che supportano questi protocolli. In particolari situazioni (tipo schede con ridotte capacità di memoria e che non richiedono tutte le funzionalità di un so), un protocollo può aver supporto dirattemante dalle sole librerie di comunicazione.

custom Un protocollo a livello applicativo di propriet dello sviluppatore, normalmente basato (come i precedenti) su TCP (Transmission Control Protocol) o UDP (User Datagram Protocol).

tcp Anche se questo protocollo non è a livello applicativo nella gerarchia Internet, spesso rappresenta l'assunzione minima che conviene fare nelle prime fasi di analisi e progettazione.

Ringraziamenti Queste note si basano sul materiale didattico prodotto per il Corso di Ingegneria del Software in collaborazione con Vincenzo Ambriola e Carlo Montangero, che ringrazio per i molti e utili consigli sulle versioni preliminari.

Riferimenti bibliografici

[1] J. Arlow and I. Neustadt. *UML 2 e Unified Process, Seconda Edizione italiana*. McGraw-Hill, 2006.

[2] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.