

Architetture software e progettazione di dettaglio - riassunto

Luigi

May 2020

1 Architettura software

Definizione: L'architettura di un sistema software (o più brevemente architettura software) è costituita dalle parti del sistema, dalle relazioni tra le parti e dalle loro proprietà visibili. La struttura definisce anche la scomposizione del sistema in sottosistemi che interagiscono tra loro mediante interfacce, e le **proprietà visibili** sono le proprietà per le quali è possibile fare certe assunzioni (servizi, prestazioni, uso di risorse..), ed è importante tenere a mente che nell'architettura software si considerano solo queste.

Il **disegno di progetto architettonico** è l'insieme di documenti di progetto prodotti durante il processo di sviluppo software, relativo alla descrizione dell'architettura immaginata dal progettista, dati i requisiti del sistema e i requisiti software. In sostanza l'architettura è la struttura di un sistema completamente realizzato, e non la descrizione di questo che coincide invece con il disegno di progetto architettonico.

2 Scopo dell'architettura

Lo scopo dell'architettura è quello di **ridurre la complessità del sistema scomponendolo in sottosistemi** e di migliorare le caratteristiche di qualità di un sistema seguendo questi aspetti:

- **Modificabilità.** Riguarda le modifiche dei requisiti, pertanto è importante costruire una matrice di tracciabilità che memorizzi le relazioni di soddisfacimento/realizzazione tra requisiti e componenti.
- **Portabilità e interoperabilità.** Con portabilità si intende la possibilità di spostarsi tra una piattaforma e l'altra operando sull'interfaccia dell'architettura con la piattaforma sottostante. Per interoperabilità invece si intende che anche le singole componenti dell'applicazione possono essere distribuite tra diverse piattaforme all'insaputa dello sviluppatore, quindi devono avere un'opportuna infrastruttura di comunicazione tra loro. Tramite l'architettura si definiscono le componenti.
- **Riuso.** Per riuso si intende l'uso di *componenti prefabbricate*, come per esempio librerie, API o framework come .NET, oppure il riuso di *componenti realizzate in progetti precedenti*, ossia riutilizzare componenti create in modo generico così da poter essere estese (in quanto non possono essere modificate). Oppure per riuso si può anche intendere il riuso di *architetture*, ovvero riutilizzare un'astrazione dell'architettura indipendentemente dalle peculiarità del sistema, per progettare sistemi con requisiti simili. Di fatto, molte aziende si concentrano sullo sviluppare linee di prodotto, ovvero insieme di prodotti simili pensati per diversi utilizzatori, piuttosto che prodotti singoli.

- **Soddisfacimento dei requisiti sull'hardware** e sulla piattaforma fisica di comunicazioni tra nodi hardware distinti; l'architettura comprende la dislocazione del software sui nodi hardware.
- **Dimensionamento e allocazione del lavoro.** La misura di un insieme di componenti è più precisa rispetto alla misura di un sistema monolitico, e su questo si basa anche l'allocazione del lavoro.
- **Prestazioni.** L'architettura permette la valutazione delle prestazioni in termini di volume di comunicazione tra componenti.
- **Sicurezza.** L'architettura permette di controllare le comunicazioni tra le parti e di controllare parti vulnerabili all'esterno, e di conseguenza di fornire anche componenti di protezione (i.e. firewall).
- **Rilascio incrementale.** Nel modello incrementale si ha un'iniziale identificazione dei requisiti e una definizione di architettura con le rispettive componenti da realizzare per prime. Magari sono le componenti più urgenti e sulle quali ci si aspetta un feedback dal committente.
- **Verifica.** La definizione di architettura consente anche una verifica di tipo incrementale sulle singole componenti a insiemi che diventano via via più grandi fino a verificare l'intero sistema.

3 Perché e come descrivere un'architettura

Linguaggio grafico, deve rispondere a:

- **Comunicazioni tra le parti interessate** (progettista e sviluppatori).
- **Comprensione** (più facile ragionare con buona documentazione).
- **Documentazione** (una documentazione scritta può essere consultata nel tempo).

3.1 Vista e tipi di vista, stili

Vista: È una proiezione di un'architettura secondo un criterio specifico.

Tipo di vista: Caratterizza un insieme di viste:

- **Viste di tipo strutturale:** struttura del software in termini di unità di realizzazione (classe, package...).
- **Viste di tipo comportamentale** (componenti e connettori): descrivono l'architettura in termini di unità di esecuzione con comportamenti e interazioni.
- **Viste di tipo logistico:** descrivono le relazioni con altre strutture (i.e. hardware, organigramma aziendale...).

Stile: Particolare proprietà di un'architettura; gli stili architettonici noti valgono su grandi insiemi di architetture (i.e. stile a strati).

3.2 Organizzazione: elementi, relazioni, proprietà, usi

Elementi e relazioni. Ad ogni tipo di vista e vista corrispondono degli elementi (i.e. architettura strutturale: classi, moduli..) e relazioni tra essi (inclusione, ereditarietà...).

Proprietà. Informazioni aggiuntive su elementi e relazioni.

Usi. Utilità del tipo di vista nell'economia della descrizione di un'architettura.

Notazioni. UML.

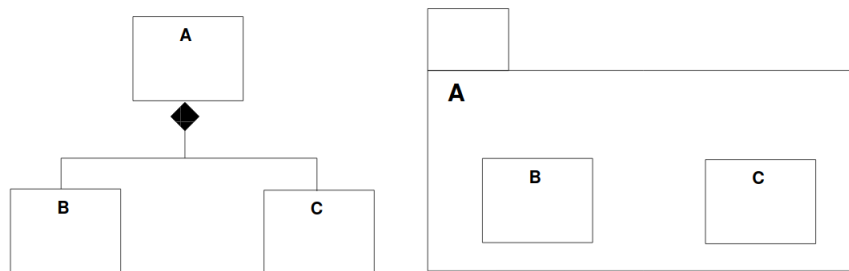
3.3 Descrizione di un'architettura

Il disegno di progetto architettonico consiste in una parte introduttiva e nella descrizione di ciascuna delle viste, e può contenere anche la definizione di relazioni tra le viste, motivazioni scelte e vincoli globali. La documentazione di ogni vista consiste di: **Visione complessiva (grafica)**, **catalogo degli elementi**, **interfacce e comportamento**.

4 Viste di tipo strutturale di un'architettura software

In questa vista, un **elemento (o modulo)** può essere un modulo, una classe o un package. Una **relazione** può essere parte di, eredita da, usa, può usare. Una **proprietà** specifica le caratteristiche soprattutto degli elementi quali nome (con opportune regole dello spazio dei nomi), funzionalità realizzate, visibilità, informazioni sulla realizzazione. Gli **usi** nelle viste di tipo strutturale possono essere di costruzione (possono fornire la struttura del codice che definisce la struttura o directory), di analisi (per la tracciabilità dei requisiti e l'analisi di impatto a fronte delle modifiche) o di comunicazione (ossia descrivere la suddivisione delle responsabilità del sistema agli sviluppatori). La **notazione** è quella UML.

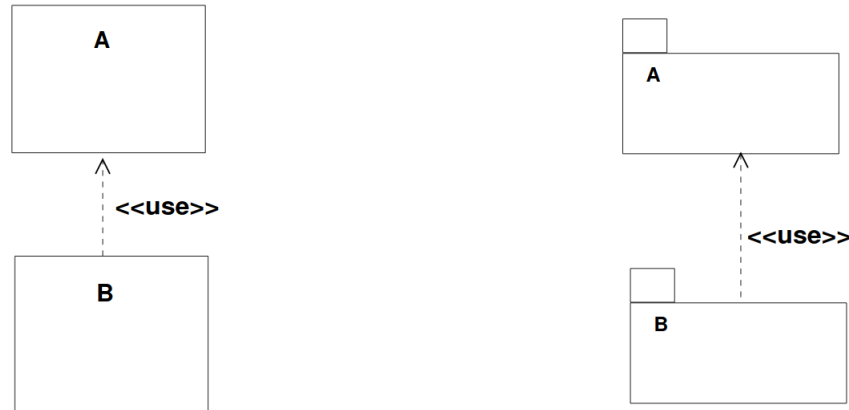
4.1 Vista strutturale di decomposizione



In una vista strutturale con relazioni di tipo **parte di** è detta **vista strutturale**

di decomposizione, si usa per mostrare come un modulo ad alto livello è raffinato in sotto moduli e come le responsabilità di un modulo siano spartite tra i figli. **N.B.:** A lezione, invece del rombo nero, come notazione ha usato un \oplus .

4.2 Vista strutturale d'uso



In una vista strutturale con relazioni di tipo **usa** è detta **vista strutturale d'uso**. Questa vista favorisce la **pianificazione di uno sviluppo incrementale del sistema**, dove vengono definite delle priorità tra moduli sulla base di **aspetti di natura funzionale** (relativi alle funzionalità volute dai committenti), **aspetti tecnologici** (relativi alla difficoltà di realizzazione dei moduli, con priorità dipendenti dalla complessità dei moduli o dalla stabilità per la loro realizzazione), **aspetti di natura architettonica** (se A usa B, B ha priorità maggiore). Favorisce inoltre anche il **test incrementale del sistema** agevolando la progettazione di stub e driver e l'**analisi d'impatto**, poiché una vista strutturale d'uso fa capire bene che la modifica di un modulo A può richiedere modifiche nei moduli che usano A.

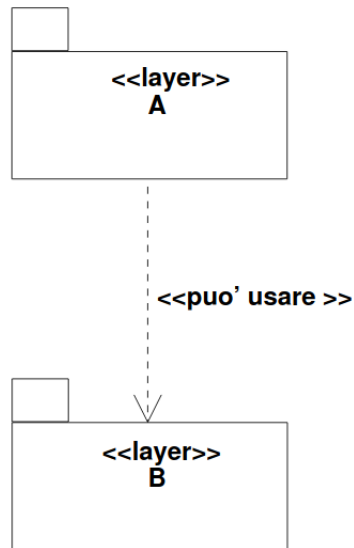
4.3 Vista strutturale di generalizzazione



In una vista strutturale con relazioni di tipo **eredita da** è detta **vista strutturale di generalizzazione**, utile per descrivere architetture derivanti dall'istanziamento di un framework, insieme di classi astratte relazionate tra esse. Quando si eredita da un framework e si implementano le classi astratte si ereditano le loro

relazioni, e si ottiene quindi un insieme di classi concrete e relazionate. Una vista strutturale di questo tipo evidenzia la relazione tra framework e istanza. Se si usa una libreria invece che un framework, non si ereditano le relazioni tra classi, quindi per ogni classe bisogna decidere quali saranno gli oggetti invocati dai metodi della classe che si sta realizzando.

4.4 Vista strutturale a strati



In una vista strutturale con relazioni di tipo **può usare** è detta **vista strutturale a strati**, ed è lo stile più usato, secondo il quale un sistema è suddiviso in livelli di macchine virtuali. L'elemento di interesse è uno strato, ovvero un **insieme di moduli** che mette a disposizione un'interfaccia per i suoi servizi. A può usare B vuol dire che l'implementazione di A può usare qualsiasi servizio messo a disposizione da B. Con **servizio** si intende un insieme di funzionalità con un'interfaccia comune.

5 Vista comportamentale di un'architettura software (C&C)

È chiamata anche vista a componenti e connettori, considera la struttura in termini di unità di esecuzione con comportamenti e interazioni. È caratterizzata da: **Elementi (o entità)**: In questa vista gli elementi possono essere **componenti**, cioè ad esempio un processo, oggetto, server o deposito dati. A run-time, potrebbero esserci più istanze della stessa classe, e nella vista di run-time potrebbero apparire tutte. I **connettori** invece sono i canali di comunicazione tra le componenti quali protocolli, flussi di informazione, modi per accedere ad una

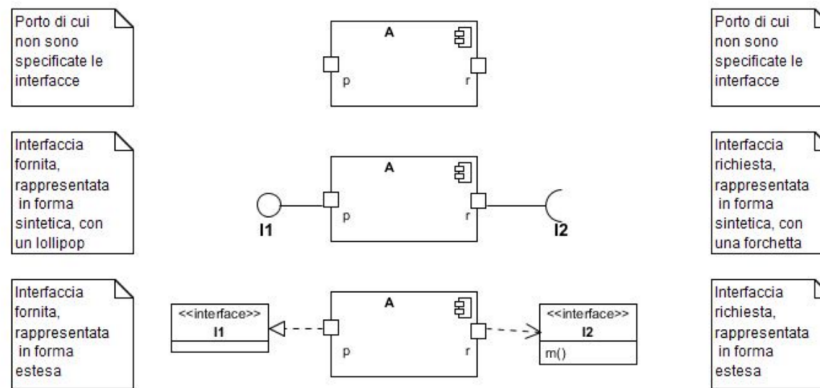
base di dati. Un connettore può collegare 2 o più componenti.

Proprietà: Le proprietà di un componente sono nome, tipo (numero e tipo di porti), informazioni di affidabilità, prestazioni,... I **porti** identificano i punti di interazione di una componente e sono caratterizzati dalle interfacce che forniscono e/o richiedono. Le proprietà di un connettore sono invece nome, tipo (numero e tipo di ruoli), caratteristiche del protocollo,... I **ruoli** identificano i ruoli dei partecipanti in un'interazione.

Relazioni: Una relazione in questa vista collega i porti delle componenti con i ruoli dei connettori.

Usi: La vista comportamentale è utile per l'analisi delle caratteristiche e di qualità a run-time, come funzionalità, prestazioni, affidabilità, disponibilità, sicurezza. È utile anche per la documentazione e la comunicazione della struttura del sistema in esecuzione nonché per descrivere stili architetturali noti.

In sintesi, in notazione UML:

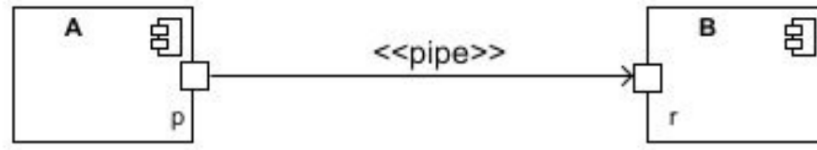


5.1 Vista comportamentale per descrivere gli stili

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

5.1.1 Pipe & Filters (condotte e filtri)

Le componenti in questo stile sono di tipo *filtro* e i connettori di tipo *condotta*. La caratteristica peculiare di un filtro è che può ricevere una sequenza di dati in input e produrre una sequenza di dati in output. In più, due o più filtri possono operare in parallelo: il filtro 2 può elaborare i primi output del filtro 1 mentre quest'ultimo continua ad elaborare i suoi input per fornire via via gli output.



5.1.2 Shared data (dati condivisi)

È uno stile focalizzato sull'accesso a dei dati condivisi tra le componenti, ad esempio un database. Un connettore può descrivere ad esempio un protocollo di interazione magari con una fase di autenticazione.

5.1.3 Publish-subscribe

Una componente si abbona a classi di eventi rilevanti per il suo scopo. Sono caratterizzati da un'interfaccia che pubblica e/o sottoscrive eventi. Un connettore publish-subscribe è come un bus che consegna ai consumatori "iscritti" (componenti) gli eventi. In un'architettura di questo tipo non c'è un'associazione diretta tra publisher e subscriber, quindi il produttore non conosce il numero di subscriber né la loro identità, e questo semplifica le modifiche dinamiche al sistema. Mittenti e destinatari in questo stile dialogano tramite un dispatcher (o broker), per questo il publisher non sa le identità dei subscriber, perché egli invia al dispatcher il materiale da condividere e quest'ultimo lo inoltra a tutti gli iscritti. Questo non conoscere il numero e l'identità degli iscritti da parte del publisher potrebbe favorire la scalabilità del sistema. Un esempio potrebbe essere il Data Distribution Service for Real Time Systems (DDS), che è uno standard emanato dall'Object Management Group (OMG) che definisce un **middleware** per la distribuzione di dati in tempo reale secondo il paradigma publish/subscribe.

5.1.4 Client-server

In questo stile le componenti sono di tipo client o server. Le interfacce dei server descrivono i servizi o funzionalità offerti, mentre quelle dei clienti descrivono i servizi usati. L'avvio di interazione è data da un client che attende risposta dal server, quindi il client deve conoscere l'identità del server ma non viceversa, poiché l'identità del client è comunicata insieme alla richiesta di servizio. Potrebbero tuttavia esserci anche dei vincoli.

5.1.5 Master-slave

È un caso particolare di client-server, in quanto il server serve solo uno specifico client (il master). Si usa di solito per la replica di database, dove i database master sono considerati come fonte autorevole e i database slave sono sincronizzati con esso.

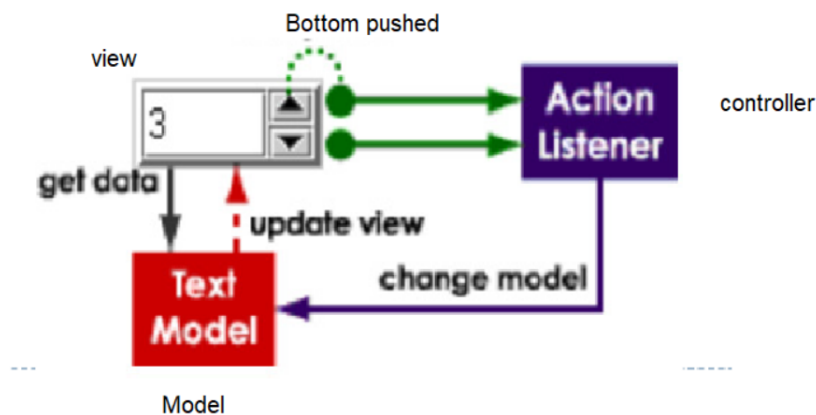
5.1.6 Peer-to-peer (pari a pari)

Nello stile P2P le componenti sono sia server che client e interagiscono tra loro alla pari per scambiarsi servizi. Le richieste quindi di interazione-risposta non sono asimmetriche come in client-server ma possono essere iniziate da entrambi i lati.

5.1.7 Model-View Controller (MVC)

In questo stile si isola invece la logica di business dal controllo sull'input e dalla vista sui dati (presentazione), consentendo così uno sviluppo indipendente, un test e una manutenzione su ciascuno di essi. Il **modello** consiste nella rappresentazione del dominio dei dati su cui opera l'applicazione; quando un modello cambia il suo stato, le sue viste associate vengono notificate così da potersi aggiornare. La **vista** mette a disposizione il modello in una forma adatta all'interazione, come un'interfaccia utente, e vi possono essere più viste per un singolo modello per scopi distinti. Il **controllore** invece riceve l'input ed effettua chiamate agli oggetti del modello. I passi che vengono compiuti sono:

- l'utente interagisce con la vista.
- il controller riceve le azioni e le interpreta
- il controller chiede al modello di cambiare il suo stato
- il modello notifica la vista del suo cambiamento di stato
- la vista chiede lo stato al modello



5.1.8 Coordinatore di processi

È uno stile comportamentale utilizzato per realizzare processi complessi, ove si conosce la sequenza di passi per realizzarli. Di solito, il coordinatore di processi riceve una richiesta, chiama i server secondo l'ordine prefissato e fornisce infine una risposta. Tuttavia i server non conoscono il loro ruolo nel complesso o il numero/ordine di passi, ma solamente il proprio servizio da fornire, e la comunicazione tra essi è flessibile, infatti può essere sincrona o asincrona.

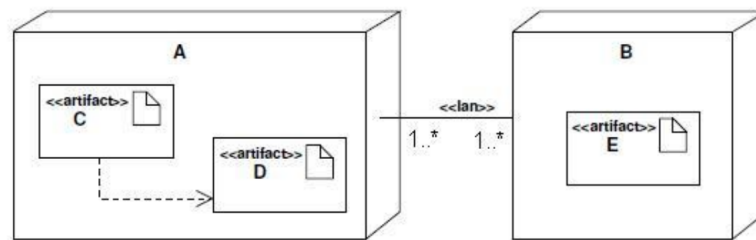
6 Viste di tipo logistico di un'architettura software

6.1 Vista logistica di dislocazione (deployment)

Questa vista considera la mappatura degli artefatti 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 all'inizio del processo di sviluppo per descrivere l'ambiente su cui dovrà essere eseguito il sistema da sviluppare.

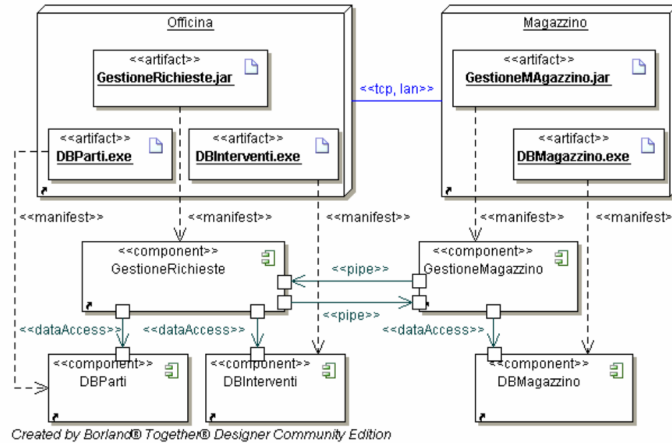
6.1.1 Notazione

In UML la notazione è quella di un grafo con i nodi che rappresentano una risorsa computazionale, e gli archi che rappresentano canali fisici o protocolli di comunicazione. I nodi possono essere sia classificatori che istanze, con la stessa notazione di classi e oggetti, ossia fornendo un nome non sottolineato (intendendo il tipo), oppure se si usano le istanze si usano nome e/o tipo separati dai ':' e sottolineati. Un **artefatto** è un pezzo di informazione fisica usato o prodotto durante un processo di sviluppo software o dalla dislocazione e operazione di un sistema (sorgenti, eseguibili,...).



7 Viste ibride di un'architettura software

7.1 Vista ibrida: dislocazione di componenti



In questa vista, data una logica di dislocazione, si mappano le componenti sugli artefatti, con una dipendenza etichettata `<<manifest>>`, ossia un artefatto manifesta una componente.



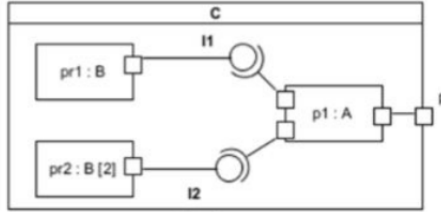
8 Progettazione di dettaglio di una componente software

Si parla di **progettazione di dettaglio** quando si ha il bisogno di definire un componente che non deve essere ulteriormente trasformato, poiché magari questa trasformazione sarebbe superflua o costosa, e magari è di nostro interesse solamente la descrizione di un'unità di realizzazione nella sua struttura interna. Per descrivere la struttura interna di una componente usiamo i **diagrammi di struttura composita**.

8.1 Classificatore strutturato

È un classificatore (di solito un componente o classe) di cui si mostra la struttura di dettaglio, a run-time, in termini di parti, porti e connettori. Un classificatore strutturato definisce l'implementazione di un classificatore, ovvero il come lavora

la componente.



8.2 Parte

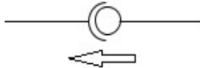
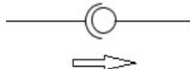

Una parte è definita come *nomeParte* : *Tipo*[*molt*], e descrive il ruolo che una istanza di Tipo gioca all'interno dell'istanza del classificatore la cui struttura contiene nomeParte. Un'istanza di nomeParte è un'istanza di Tipo (e quindi di un qualunque sottotipo).

8.3 Porto

Un porto è rappresentato da un quadratino come nel diagramma C&C, posto quindi sul bordo della componente per quanto riguarda i porti della componente e sulle varie parti per quanto riguarda i porti delle singole parti.

8.4 Connettore

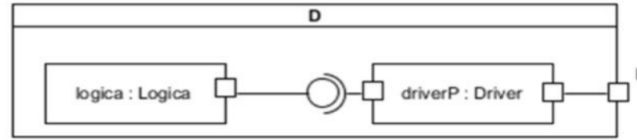
Ci sono i connettori di tipo **assemblaggio**, che esprimono una comunicazione tra due istanze di parti, con un lollipop; e i connettori di tipo **delega** identificano l'istanza che realizza le comunicazioni attribuite a un porto di struttura composita. **N.B.:** La direzione del lollipop dipende **esclusivamente** da chi ha il controllo su chi e chi risponde se interrogato, vedi esempio nell'immagine sotto:

- un'interfaccia con solo operazioni di read 
- un'interfaccia con solo operazione di write 
- un'interfaccia con operazioni di read e write 

8.5 Metodo: come si struttura una componente

Supponiamo di avere una componente D con un porto p. La struttura di D dovrà avere almeno due parti: **driverP**, che realizza la parte di comunicazione richiesta per implementare il porto; **logica**, che realizza la funzionalità richiesta alla componente. In più vi sono due connettori: di delega tra driverP e P e di

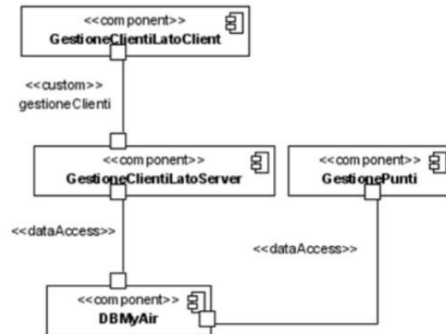
assemblaggio tra driverP e logica.



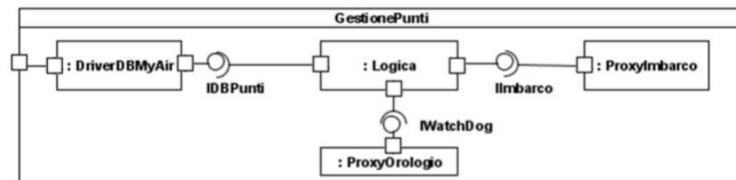
La logica si può inoltre raffinare introducendo connettori di assemblaggio, dipendenze (come ad esempio una `<<create>>` quando una parte ha il ruolo di build per la componente). Oppure si possono introdurre esplicitamente le parti che servono per realizzare comunicazioni con sistemi remoti o chiamate al sistema operativo, chiamate *proxy* (mediatori). I proxy sono collegati alla logica con connettori di assemblaggio ma non con i porti poiché non realizzano la comunicazione con le componenti esterne.

8.6 Esempio

Come esempio potremmo guardare MyAir, in cui la componente GestionePunti realizza i casi d'uso AccumuloPunti e AggiornamentoAnnuale.

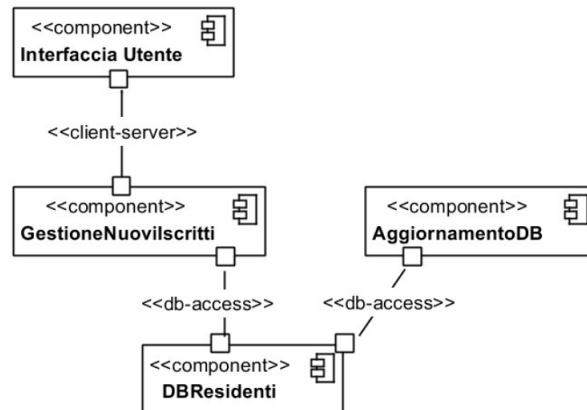


e il rispettivo diagramma di struttura composta:

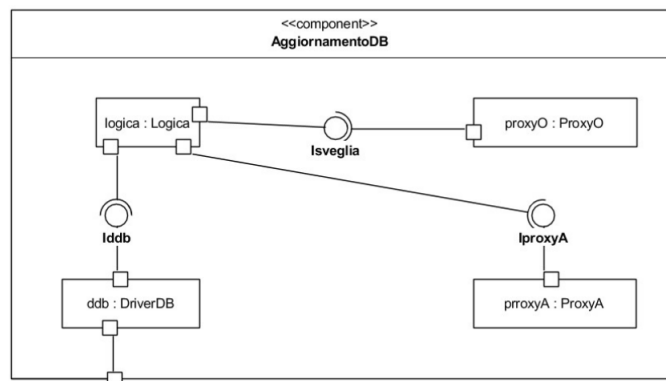


8.7 Esempio Pisa Mover

8.7.1 Diagramma C&C



8.7.2 Struttura composita



9 Glossario

- **API:** Application Programming Interface, ossia un'applicazione che, mediante modalità standard, espone le funzionalità di altre applicazioni, abilitando quindi il riutilizzo di servizi che possono essere a loro volta composti o scomposti a seconda delle esigenze.
- **Framework:** Architettura logica di supporto sul quale un software può essere sviluppato e realizzato, solitamente semplificando la vita del programmatore.

- **Package:** Solitamente in queste dispense si parla di package visto come insieme di classi o interfacce correlate tra esse.
- **Middleware:** Insieme di programmi che fungono da intermediari tra diverse applicazioni e componenti software, spesso usati come supporto per sistemi distribuiti con architetture **multitier**.
- **Architettura multitier:** Indica un'architettura software di tipo client-server per sistemi distribuiti, le cui funzionalità sono suddivise e distribuite tra più strati e livelli che comunicano tra loro.
- **Dispatcher:** È un thread che legge dalla rete delle richieste da elaborare.
- **Thread:** Un thread è un'istanza di sottoprocesso che può essere eseguito concorrentemente con altri.