Architetture software
e
Progettazione di dettaglio

Carlo Montangero *Laura Semini*

Note per il corso di Ingegneria del Software

Corso di Laurea in Informatica

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 dalle sue 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 dalle 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 distinte 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.

[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.

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 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 sotto caso 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 prefabbricata 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 prefabbricate 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 realizzate in precedenti progetti.

  Il riuso di componenti esistenti mira a sfruttare in un nuovo progetto, e quindi in un nuovo contesto, una componente già realizzata, senza modificarla. Si può anche progettare una componente in vista di un suo riuso futuro. Poiché le componenti non possono essere modificate ma possono essere estese, se si progetta in vista di un riuso conviene definire componenti generiche.

- 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.

[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)

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 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 delle 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 delle componenti che devono essere realizzate per prime. Queste possono essere, per esempio, le componenti che forniscono le funzionalità più urgenti per il cliente, o le componenti per le 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 le singole 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. Una 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 delle 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 di una architettura sw

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 modulo padre–sotto modulo. Si usa questa vista per mostrare come un modulo ad alto livello è raffinato in sotto moduli e come le responsalilità di un modulo sono ripartite tra i moduli figli.

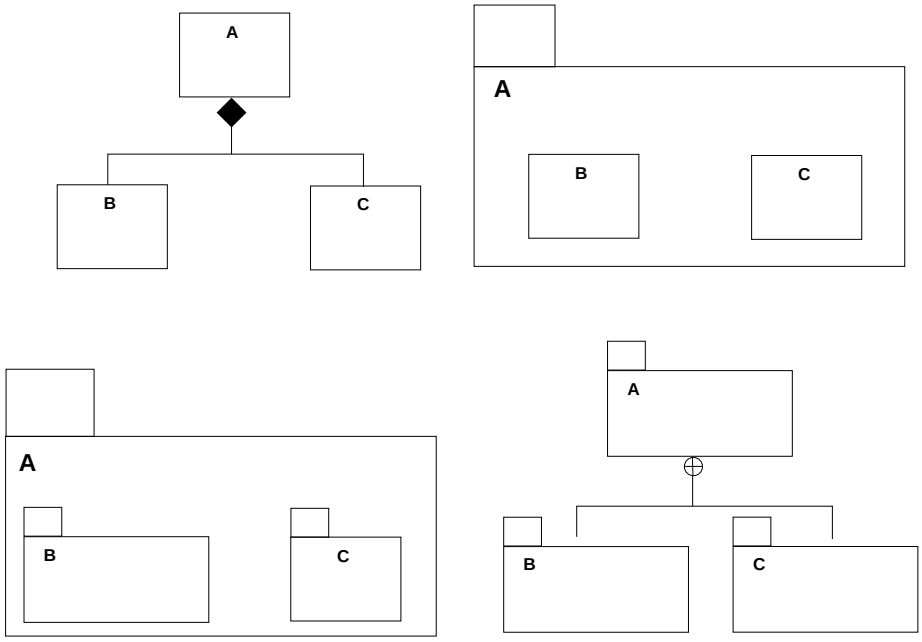

Figura 2: Vista strutturale di decomposizione: per rappresentare la relazione "parte di" si usa la composizione tra classi e l'inclusione tra package.

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.

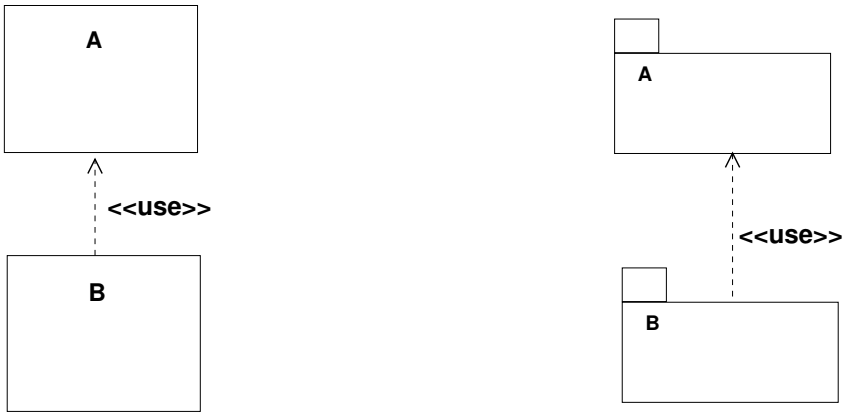

Figura 3: Vista strutturale d'uso: per rappresentare la relazione "usa (dipende da)" si utilizza la freccia che indica dipendenza, etichettata con $\langle\langle use \rangle\rangle$.

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.

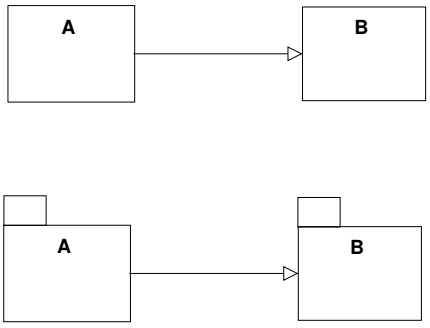

Figura 4: Vista strutturale di generalizzazione: La relazione "eredita da" è resa come una generalizzazione.

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 strutturale d'**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.

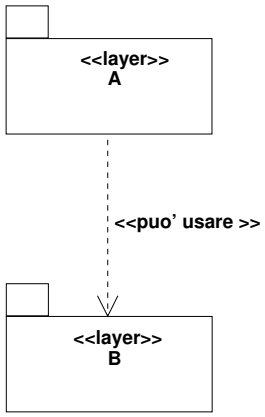

Figura 5: Vista strutturale a strati: la relazione "può usare" è resa con una dipendenza etichettata con $\langle\langle puo' \ usare \rangle\rangle$.

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 di una architettura sw

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.

  Una 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

- una 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 una 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 delle 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 delle singole 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 le componenti è un rettangolo con, oltre al nome della componente, un'icona e lo stereotipo $\langle\langle component \rangle\rangle$. Alcuni editor mettono sia l'icona che lo stereotipo.

I porti sono associati alle componenti e rappresentati con quadratini sul bordo del rettangolo. Le interfacce che caratterizzano un porto sono rappresentate:

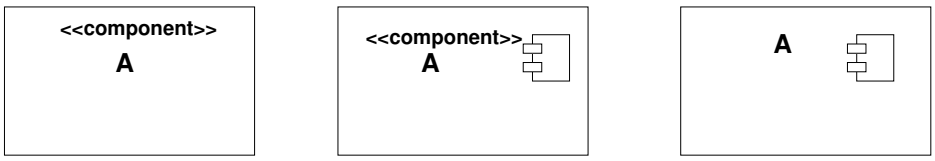

Figura 6: Rappresentazione di una componente.

- con il simbolo della classe stereotipata $\langle\langle interface\rangle\rangle$ e collegate al porto con una relazione di implementazione o di dipendenza, oppure
- con un'icona, usando i *lollipop* e le *forchette* (anche chiamata notazione *ball–and–socket*).

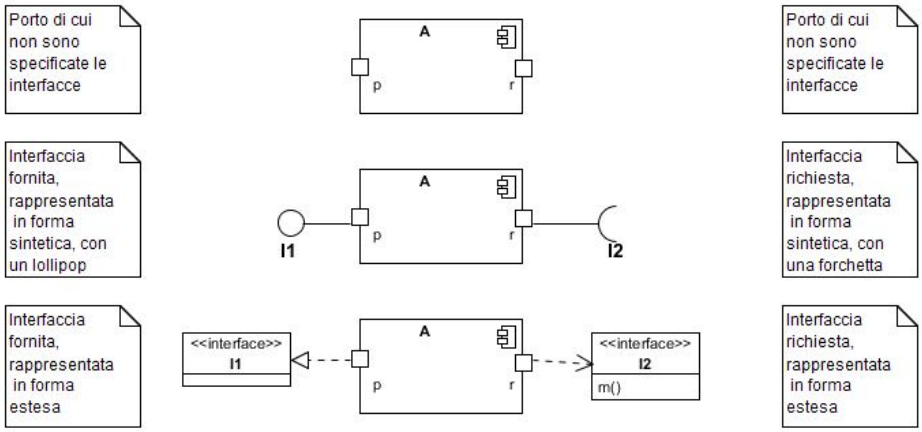

Figura 7: Rappresentazione di porti e interfacce.

In UML un connettore tra porti è rappresentato con una associazione tra i porti. Inoltre è possibile:

- indicare i ruoli delle componenti agli estremi di un connettore,
- etichettare i connettori per esprimerne, se non la semantica, almeno il tipo (si usa uno stereotipo);
- indicare il flusso delle informazioni usando una freccia a un estremo del connettore (la freccia che si usa nei diagrammi delle classi per indicare la navigabilità).

Alcuni autori descrivono i connettori usando, ad esempio, classi UML, questo perché il connettore UML non permette di associare informazioni sul suo comportamento, se non con stereotipi. In effetti, 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 rappresentato con un'associazione tra porti e indica un canale di comunicazione, senza la possibilità di esprimere comportamenti specifici.

In Figura 8 il primo esempio mostra un connettore generico tra i porti p di A e r di B. Il secondo esempio mostra come rappresentare indicare con uno stereotipo le caratteristiche del connettore, in questo caso $\langle\langle client - server\rangle\rangle$, in cui conviene indicare quale componente gioca il ruolo di server e quale quella di client. Nel terzo esempio, un connettore di tipo $\langle\langle pipe\rangle\rangle$ rappresenta una generica *condotta* (*pipe*) tra una coppia di componenti *filtro*). La freccia indica che A è la sorgente dei dati e B il destinatario.

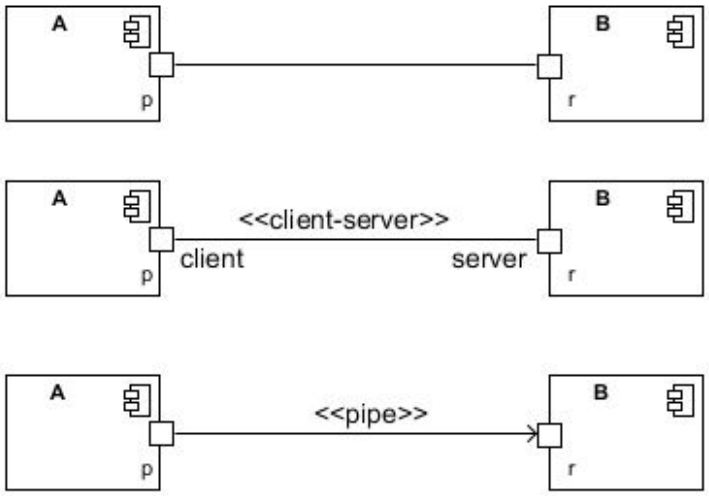

Figura 8: Rappresentazione dei connettori.

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 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 le 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.

5.1.2 Dati condivisi (Shared data)

E' uno stile focalizzato sull'accesso a dati condivisi tra le varie componenti. Prevede una 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 le componenti.

Un connettore che colleghi la base di dati con le 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

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

Le componenti sono caratterizzate da un'interfaccia che pubblica e/o sottoscrive eventi. Un connettore di tipo publish–subscribe è un bus di eventi: le componenti, pubblicando gli eventi, li consegnano al bus, il quale li consegna alle componenti (consumatori) appropriate.

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 le 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) le 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 di una architettura sw

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 descrivere 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.

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.

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

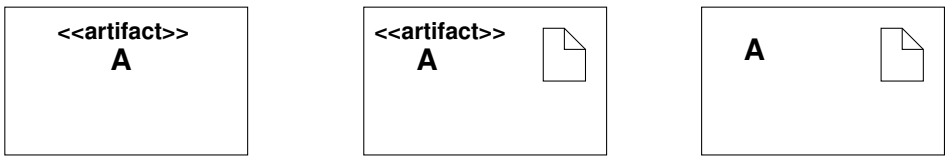

Figura 9: Rappresentazione di un artefatto.

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).

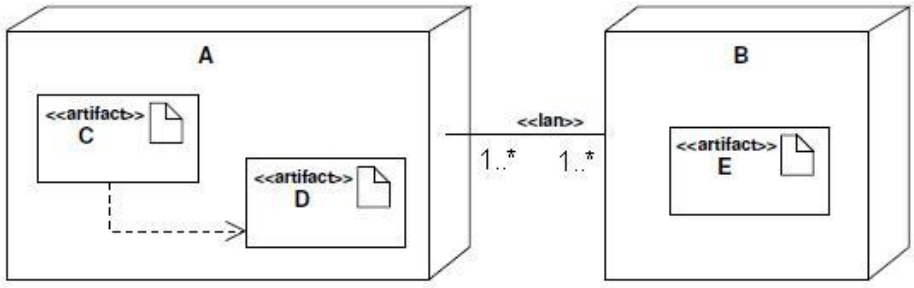

Figura 10: Vista logistica di dislocazione.

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 11 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.

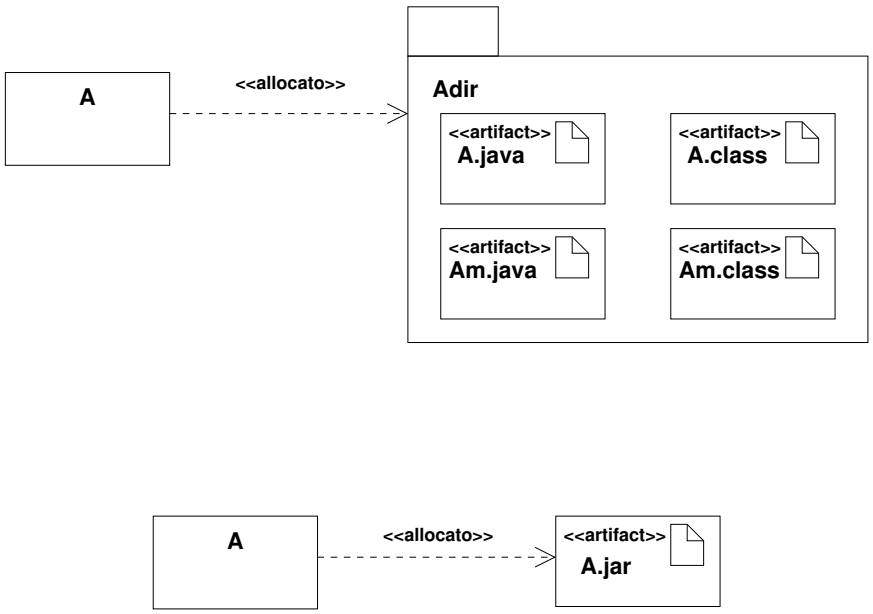

Figura 11: Vista logistica di realizzazione: due esempi.

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 di una architettura sw

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 le componenti sugli artefatti.

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

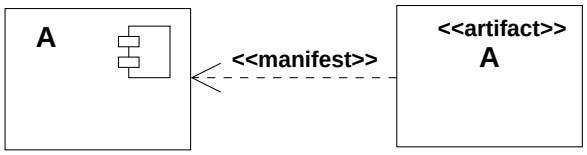

Figura 12: Relazione $\langle\langle manifest \rangle\rangle$.

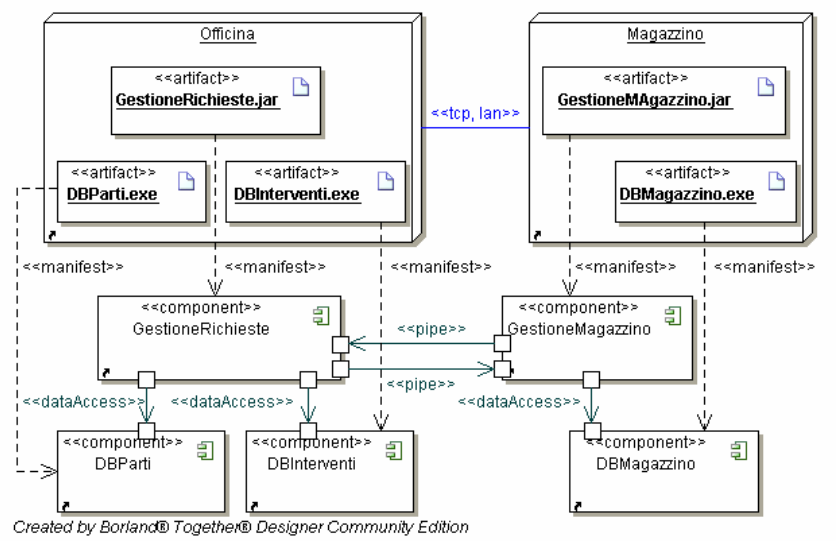

Figura 13: Un esempio di vista ibrida: dislocazione con componenti.

In Figura 13 si mostrano: una vista logistica di dislocazione in forma istanza nella parte alta; una vista comportamentale nella parte bassa; le relazioni *manifest* 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 una componente può essere necessario il codice di più di un modulo. Una libreria di Input/Output può corrispondere a un connettore.

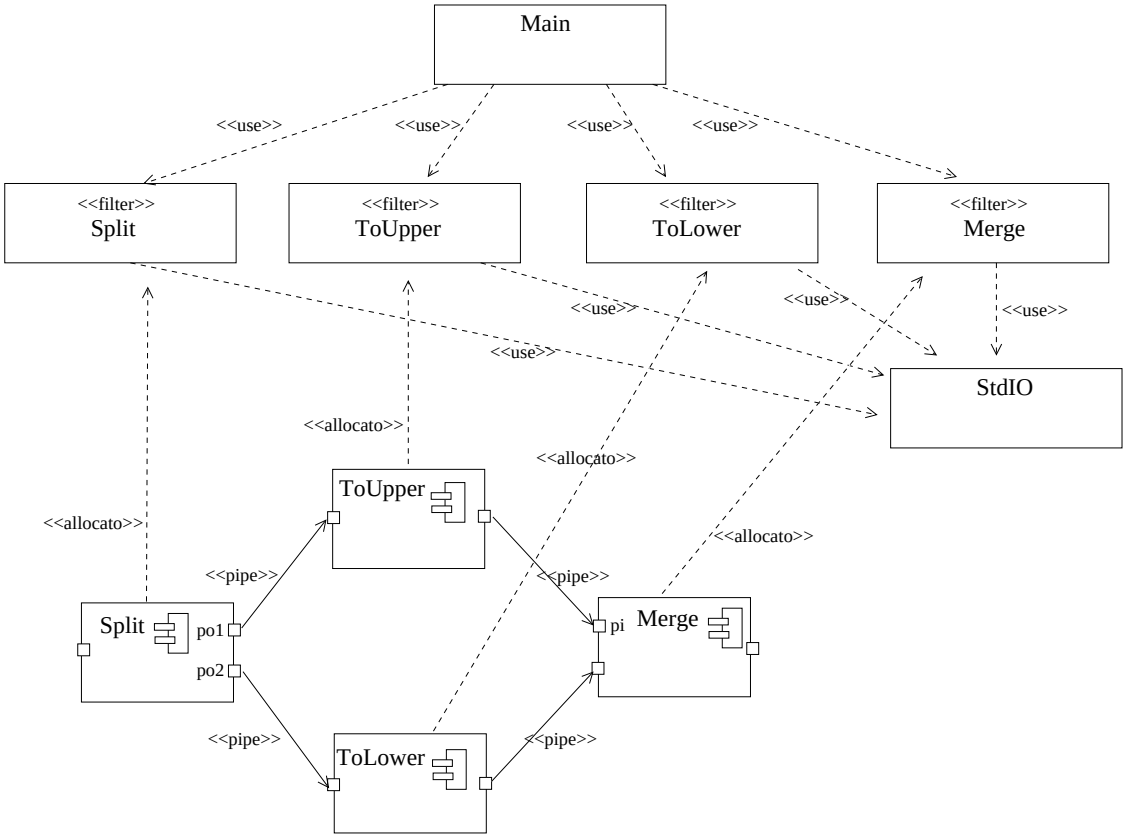

Figura 14: Un esempio di vista ibrida: strutturale con componenti.

La Figura 14 dà nella parte bassa una vista comportamentale dell'architettura di un sistema, che possiamo chiamare Alternatore, che presa una sequenza 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:

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 alle 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 Progettazione di dettaglio di una componente software

Per descrivere la struttura interna di una componente usiamo i diagrammi di struttura composita.

Classificatore strutturato

È un classificatore (di solito una componente, ma anche una classe) di cui si mostra la struttura di dettaglio (o struttura interna) a tempo di esecuzione, data in termini di

- parti (eventualmente con molteplicità),
- porti (eventualmente con interfacce fornite e/o richieste) per mostrare il comportamento visibile all'esterno,
- connettori, che esplicitano le interazioni fra le parti e punti di interazione (porti) con l'esterno

Un classificatore strutturato definisce l'implementazione di un classificatore: così come le interfacce del classificatore definiscono cosa deve fare, la sua struttura interna definisce come viene fatto il lavoro.

I tipi che definiscono le parti contenute in un classificatore strutturato, possono a loro volta essere strutturati, ricorsivamente.

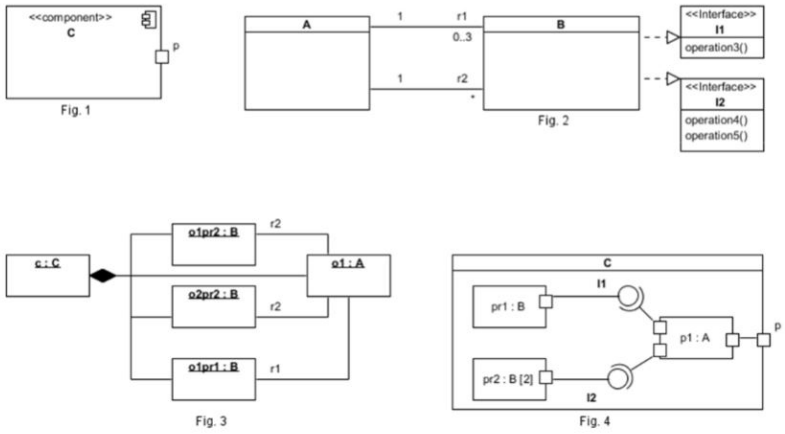

Figura 15: In Fig. 1 si mostra la componente C. In Fig. 2 si specificano le classi A e B, con associazioni, ruoli e interfacce. Si assume che un'istanza di A usi l'interfaccia I1 (I2) di B quando connessa con unistanza di B in ruolo r1 (r2 risp.). In Fig. 3 si mostra un diagramma degli oggetti conforme al diagramma in Fig. 2. In Fig. 4 si mostra la struttura di C, che fornisce una astrazione del diagramma in Fig. 3. Introduce vincoli rispetto al diagramma delle classi: dice che uso istanze di A e di B, collegate in un certo modo, con dato ruolo nel contesto di C, e date molteplicità.

Parte

Una parte ha un nome, un tipo e una molteplicità (anche se sono tutti facoltativi):

$$nomeParte \ : \ Tipo \ [molt]$$

Una parte $p : T$ descrive il ruolo che una istanza di T gioca allinterno dellistanza del classificatore la cui struttura contiene p. La molteplicità indica quante istanze possono esserci in quel ruolo. Un'istanza di p è un'istanza di T (e quindi di un qualunque sottotipo).

Porto

Un porto è rappresentato con un quadratino, come nei diagrammi di componenti.

La struttura composita che mostra la struttura di dettaglio del componente C ha tutti i porti di C sul proprio bordo, più i porti associati alle parti, che permettono le interazioni tra loro e con l'esterno.

Connettore

Nei diagrammi di struttura composita ci sono due tipi di connettori:

- di assemblaggio (assembly connector): esprime un legame che permette una comunicazione tra due istanze di parti, nei ruoli specificati dalla struttura.

  Notazione: Si usa la notazione lollipop, e unisce due porti delle parti che comunicano[7].

- di delega (delegation connector): identifica l'istanza che realizza le comunicazioni attribuite a un porto della struttura composita.

  Notazione: Si usa una semplice linea tra il porto della parte e il porto (sul bordo) della struttura composita.

Nei connettori di assemblaggio, il verso del lollipop non ha alcun legame con il verso in cui viaggiano i dati. Ha solo a che vedere con chi ha il controllo e con chi, interrogato, risponde, come mostrato in Figura 16.

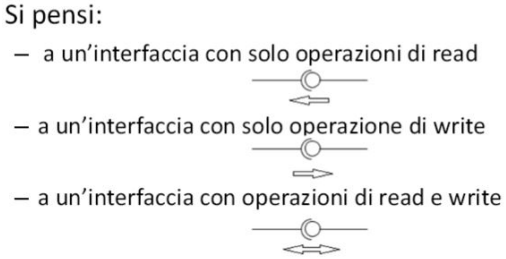

Figura 16: Verso del lollipop e verso dell'informazione.

[7] La dispensa di esercizi è stata scritta prima dell'ultimo rilascio di UML. Troverete una notazione leggermente diversa, tra le altre cose, connettori tra parti collegati alle parti invece che ai loro porti.

Metodo: come si struttura una componente

Un modo conveniente di strutturare una componente prevede di separare gli aspetti di comunicazione da quelli di realizzazione delle funzionalità richieste. Ciò favorisce infatti modificabilità e comprensibilità della componente[8].

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.

e due connettori

- di delega tra $driverP$ e P
- di assemblaggio tra $driverP$ e *logica* (il verso del lollipop non è fissato a priori)

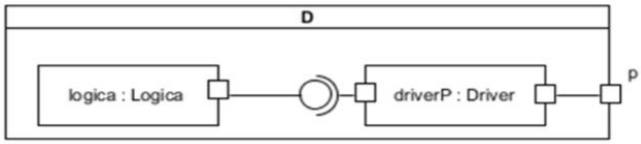

Figura 17: Struttura minima di una componente.

In generale, data una componente con n porti, la sua struttura minima dovrà prevedere:

- n parti col ruolo di driver

  collegate ognuna a un porto, con un connettore di delega
- una parte che realizza la logica della componente

  collegata a tutti i driver con connettori di assemblaggio

I nomi *driver* e *logica* per le parti di una componente non sono standard di UML. Sono usati per scopi didattici, per aiutare a distinguere le loro responsabilità.

È ovviamente possibile, quando necessario, dettagliare maggiormente la struttura di una componente:

- raffinando la *logica* in un insieme di parti interconnesse
  - Con connettori di assemblaggio
  - Con dipendenze, per esempio una $\langle\langle create \rangle\rangle$, quando una parte ha il ruolo di build per la componente: c'è una dipendenza dalla parte che fa la build verso le altre parti della componente.

[8]Per approfondimenti su questo tema si veda l'articolo in appendice.

- Introducendo esplicitamente le parti che servono per realizzare comunicazioni con sistemi remoti o chiamate al sistema operativo, che chiameremo *proxy*, per mediatore[9].

  I *proxy* sono connessi alle parti che descrivono la logica, con connettori di assemblaggio, ma non sono collegati ai porti, perché non realizzano la comunicazione con altre componenti del sistema che si sta progettando.

Esempio: MyAir

Come esempio di struttura di dettaglio di una componente, si considera l'esempio MyAir della dispensa di esercizi, in cui la componente GestionePunti realizza i casi d'uso AccumuloPunti e AggiornamentoAnnuale.

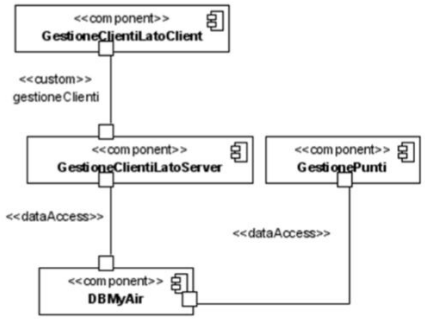

Figura 18: Vista C&C di MyAir.

La Figura 19 mostra la struttura di dettaglio della componente GestionePunti.

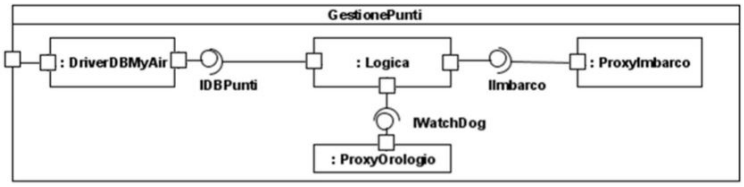

Figura 19: Struttura di dettaglio di GestionePunti.

Le parti introdotte hanno le responsabilità in Tabella 1

[9]L'uso dei termini *driver* e *proxy* in queste note si discosta dall'uso che se ne fa in contesti rilevanti per lo sviluppo software, come i Design Patterns e il middle-ware RMI. La scelta fatta è di assimilare i porti di una componente (che la mettono in comunicazione con le altri componenti del sistema) alle interfacce fisiche di un calcolatore, guidate appunto da un *driver*, nella terminologia standard, e di assimilare gli elementi esterni al sistema agli elementi remoti nella comunicazione in un sistema distribuito, che si raggiungono attraverso elementi locali che li rappresentano, *proxy* - mediatori, appunto.

Parte	Responsabilità
DriverDBMyAir	Realizza l'interfaccia IDBPunti che permette a Logica di accedere tramite il (solo) porto della componente al DBMyAir.
ProxyImbarco	Realizza la connessione con il sistema Imbarco (passando alla Logica la lista degli imbarcati).
ProxyOrologio	Sveglia la logica alla data e ora richiesta tramite IWatchDog
Logica	Realizza i casi d'uso, sfruttando le interfacce introdotte.

Tabella 1: Responsabilità delle parti

APPENDICE

A 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, le componenti, o più in generale gli elementi del modello. Possono esserci più artefatti associati a una componente, anche allocati su nodi distinti. Una componente può essere realizzata su nodi distinti. Solo gli artefatti risiedono (sono allocati) su nodi hardware, non le 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 Una componente è unità concettuale di decomposizione di un sistema a tempo di esecuzione. È caratterizzata dalla sua interfaccia, definita in termini di porti. Caratteristica delle componenti è la rimpiazzabilità: è possibile sostituire una componente di un sistema con un'altra 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 le componenti. Lo sviluppatore deve determinare le componenti che, aderendo alla logica collaborativa del framework, verranno coordinate da quest'ultima.

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

Interfaccia richiesta Un'interfaccia richiesta descrive le operazioni che una componente richiede ad altre componenti. Non è detto che tutte le operazioni richieste siano effettivamente usate, ma la componente è garantita funzionare se le 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 una componente, È caratterizzato dalle interfacce che fornisce e/o richiede.

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

B 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 20 dove una coppia di componenti (filters) sono collegate da una condotta (pipe). Questo schema architettonico rappresenta lo schema base di ogni architettura in stile "pipes & filters".

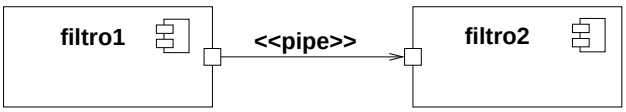

Figura 20: Esempio di schema architettonico.

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.

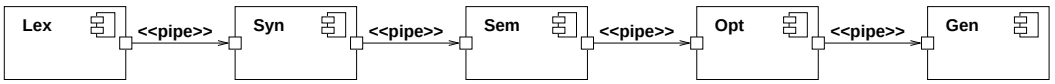

Figura 21: Esempio di architettura di riferimento

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 richieste 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.

C Stereotipi per architetture

C.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 verrà 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.

C.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 direttamente dalle sole librerie di comunicazione.

custom Un protocollo a livello applicativo di proprietà 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.

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.

A place for everything and everything in its place

Carlo Montangero
Dipartimento di Informatica, Università di Pisa
monta@di.unipi.it

Laura Semini
Dipartimento di Informatica, Università di Pisa
semini@di.unipi.it

ABSTRACT

We consider two of the best practices of software development, namely, the distributed proxy pattern and the documentation of the software architecture in multiple complementary views, and present a process to integrate them and generate the structure of the software design, organized in multiple views, too. The intent is to facilitate the achievement of the quality objectives of separation of concerns, portability, and modifiability that a good design should possess. The process alternates steps in which the views are enriched with containers for model elements, which are inserted in following steps. Hence, the title of the paper.

1. INTRODUCTION

Software architecture (SA) and detailed design are two key elements in software development. SA is "a multidimensional reality, with several intertwined facets, and some facets – or views – of interest to only a few parties" [3]. A *view* is a projection of the SA according to a given criterion. A view considers only some concerns/aspects, e.g. it considers the structuring of the system in terms of components, or it considers some relationships between subsystems: The *module view* highlights the code structure, the *C&C view* a snapshot of the system in execution in terms of components and connectors, the *allocation view* the deployment of the system on the hardware.

The usual practice in software architecture (SA) documentation is to focus on the components and connectors (C&C) view, since it is in this perspective that the architect addresses system decomposition: it is natural to reason in terms of the behavior of sub-systems and of their interaction at run-time. In particular, it is more natural than reasoning in terms of the structure of the code. However, a good code structuring, mirroring the architectural C&C decisions, facilitates the achievement of the objectives of portability, modifiability, and separation of concerns of a good design.

We address the derivation of a complete design model from a view of the SA that collects the architect decisions on the logical structure of the system.

More precisely, we introduce a refinement process that results in a design structure of the code and keeps the implementation of the communication and of other non functional, e.g. security, requirements clearly separated from the implementation of the component functionalities. So, adaptations to a different communication/security context and changes in the functional requirements can be dealt with independently without any interference. Besides, it is particularly important, in distributed applications, to have a well structured model of the target run time architecture, to guide in the complex task of configuring and initialize the system, taking into account all the related requirements. The last step in our process achieves this goal too.

The process starts at an abstract level from an architectural model, where the C&C view is complete, in terms of black boxes with an associated specification, and the other views are empty.

The first step, that we call *introjection*, refines the C&C view (Figure 1, left), decomposing each component into parts, each dealing with a different responsibility, in the C&C view at the «design» level (Figure 1, center-left). This refined view is detailed enough to entail the gross structure of the code, in the Module view, and the implementation relations between the modules and the parts in the C&C view. So, the second refinement creates the «design» Module view, making (separate) space for the code of components and connectors and puts code for these parts in their place, and the «design» Deployment view, for allocation of the system at hand (Figure 1, center-right).

The focus on the SA allows enriching the model driven approach to software development with advantages also in dealing effectively with non functional requirements and design decisions regarding the run time platform, like the choice of the implementation language for a component, the libraries to be used in its implementation, the middle-ware supporting the communication between components, the security characteristics of a connection, the execution environment for a component.

Once the architect has introduced the relevant constraints on the appropriate elements in the «design» C&C view, the last refinement step, *constraints propagation*, propagates them to the Deployment view, to be exploited by the configuration engineer, and to the Module view, to inform the developers.

Overall, the refinements transform the C&C view into a complete design model, where the different views are pairwise related to insure the consistency of the design, as shown in Figure 1 (right). The intended meaning of the stereotypes in the figure is the following: The execution environments in the deployment view execute the artifacts that *manifest* the components in the C&C view, i.e., the artifacts behavior is the one specified for the components and connectors in the C&C view. On the other side, the executables are *built* from the code modules identified in the Module view, which in turn *implement* the components specified in the C&C view, that is, the code modules satisfy both the functional requirements ex-

.

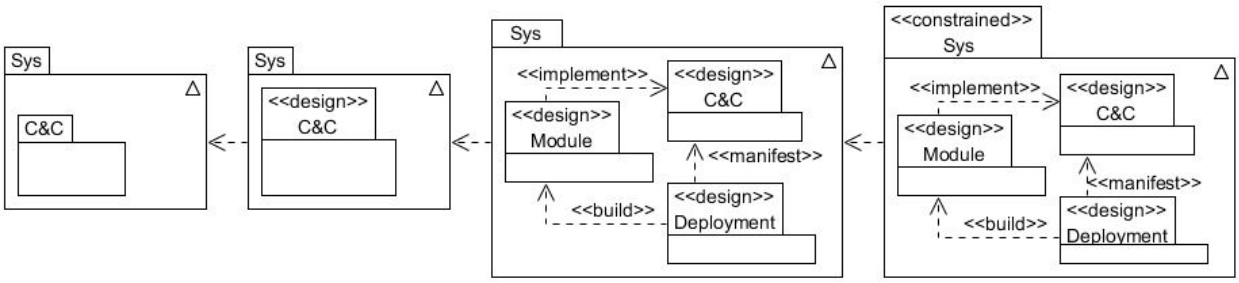

Figure 1: Refinement steps. All the dependencies between the models are refinements.

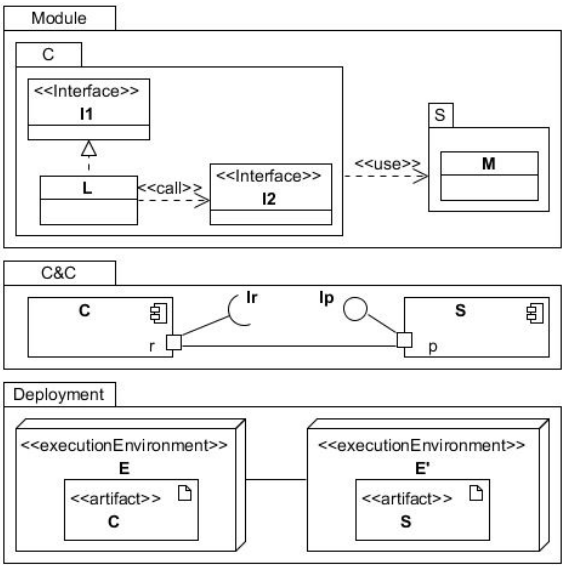

Figure 2: Architectural Views.

pressed in the C&C view and the requirements on the execution environment propagated to the other views.

The process alternates steps in which the models are enriched with containers of model elements, which are inserted in other steps. Hence, the title of the paper.

2. ARCHITECTURAL VIEW TYPES

To fix the terminology and the notation we use (all the diagrams are in UML2), we recall that, according to [3], the three main view types on SA are:

• **module views** describe the structure of the software in terms of implementation units and the relationships among them (Figure 2, top). An implementation unit may be, e.g., a class, an interface, a Java package, a level. The relationships define dependencies, like *use* and *call*, the *decomposition* of a module into submodules (illustrated with containment in the figure), and generalization/implementation relations;

In the module view, we are particularly interested in documenting the *use* relation: a module *uses* another one if the correctness of the first depends on the availability on a correct implementation of the second. This relationship makes explicit the dependencies among the modules supporting the incremental development and deployment of useful subsets of the system under development.

• **components and connectors views** (C&C) describe the architecture in terms of execution units (components), and their interactions in terms of connectors (Figure 2, middle). A component can be an object, a process, a collection of objects, a client, a server, a data store, etc. A connector represents e.g. communication paths, protocols, or access to shared memory. Components and connectors have an associated specification that defines functional requirements and qualities they have to satisfy. They are related through ports, representing component interfaces.

• **allocation views** describe the relationships between software and other structures, such as hardware or organizational chart. The most common view describes of the *deployment* of the executable artifacts on the environment where they will run (Figure 2, bottom).

3. THE DESIGN VIEWS

In this section we deal with the refinements that lead to the platform independent design views, namely introjection and design view generation, represented by the left and center dependencies in Figure 1. The next section will take care of some platform dependent characteristics.

The generation of the structure of the «design» views is made possible by the systematical application of the Distributed Proxy (DP) pattern [9], which in turn uses the *proxy* pattern [5] to decouple the communications among distributed objects from the object specific functionalities. The DP pattern addresses the problem of designing a distributed application, where complexity arises from the need to deal with the specificities of the underlying communication mechanisms, protocols and platforms. More complexity is also due to the fact that the communication mechanism can change in subsequent versions or in coexistent different configurations of the application.

3.1 Introjection: The Butterfly

In the C&C view, the DP pattern reaches its objective by introducing a *proxy* for each port and interface of the component and delegating it to deal with all the aspects of the communications through the port. In the case of remote communication the proxy (called remote proxy) represents the server in the client address space (and viceversa), and transfers data to/from the external world using the appropriate middle-ware: they take care of establishing the connections and transforming the data to/from the raw bytes that transit in the communication channels. An additional part, dubbed *Logic* as the middle layer in the Three-Tiers pattern, is introduced to take the responsibility of the functionality of the component, and it is not further decomposed by the pattern.

The resulting structure separates the concerns as required by any good design practice. Moreover, the proxies *internalize* the com-

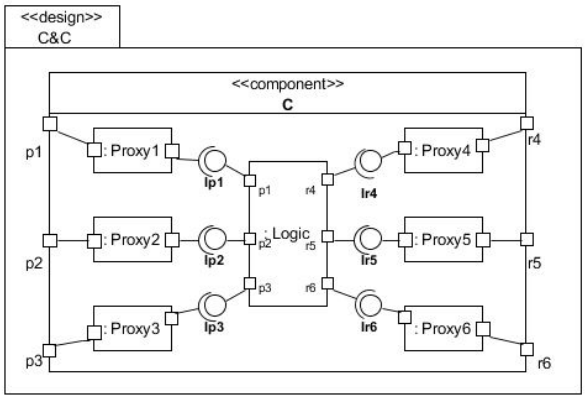

Figure 3: The butterfly.

munications, since they act, with respect to the logic, as local data sources and sinks. By taking care of all the details of the interactions with the low level communication mechanisms, they permit the design and implementation of stand alone components to be deployed independently, as required by the component based software development paradigm (CBSD).

The resulting model, after the introjection step, is the «design» C&C view: despite the introjection, we keep the delegating ports and the connectors, since the connectors are the natural place to record the platform dependent requirements on the middle-ware, as we will see in the next Section.

We refer to the systematic application of the DP pattern as the *butterfly* design pattern, because of the shape shown in Figure 3, where the "wings" take care of the communications and the "body" of the functionality. [1]

This step of design, which distributes the responsibilities among the parts (the logic and the proxies) and puts them in place in the component's butterfly structure, that is, introjects the interfaces of the components, is obtained by the algorithm shown in Table 1.

3.2 The Model and Deployment views

In order to generate the «design» Module view, we proceed in several steps: first, we create the space for the Logic and Proxies in the code. More precisely, we introduce a «component» package for each component, to place the implementation of the Logic, and a «connector» package for each connector. The latter includes two more packages, one per each role of the connector, to place the code for pairs of communicating proxies. Each of these packages is related to the package representing the component the role is attached to. The relation is a «use» one, since a component can work correctly only if there is a correct implementation of the end-point of each channel it exploits to communicate with its partners. The code that is found in the role packages, once the development is over, may be anything in between a simple delegation to a standard middle-ware and a complete implementation of (one side) of a custom communication protocol. The algorithm of this step is given in Table 2. Next, we put in place the classes related to the component functionality: we fill each «component» package with the Logic and its interfaces, reflecting the butterfly in the code. The algorithm is in Table 3. Finally, we put in place the proxies in the «connector» packages, together with the appropriate *realize* and *require* relations, as shown in Table 4.

[1]The nice symmetry of this suggestive image may often be lost, if the communications are not as symmetric as here.

```
for every component C in view C&C
  introduce in <<design>> view C&C
    a <<structuredComponent>> C with
      a part with type Logic with
        for every port P of component C
            a port P
  for every <<provided>> interface Ip at port P
                         of component C
    copy in <<structuredComponent>> C
       interface Ip and port P
    introduce in <<structuredComponent>> C
      a realize relation
                    from port P of the Logic
                    to interface Ip and
      a part P with type P_Proxy and
      a require relation from part P
                  to interface Ip
  for every <<required>> interface Ir at port P
                        of component C
    copy in <<structuredComponent>> C
     interface Ir and
    introduce in <<structuredComponent>> C
      a require relation from port P
                   to interface Ir and
      a part R with type R_Proxy with
      a realize relation from part R
                             to interface Ir
  between every part P and port P
              of <<structuredComponent>> C
     introduce a delegate connection
```

Table 1: Birth of the butterfly.

```
for every component C in view C&C
  introduce
    a <<component>> package C in view Module
for every connector between port P, P'
         of component C, C' in view C&C
  introduce in view Module
    a <<connector>> package C-P-C'-P'  with
       a <<role>> package P and
       a <<role>> package P'
    a <<use>> dependency between C and P and
    a <<use>> dependency between C' and P'
```

Table 2: Place for component and connector roles.

The generation of the Deployment view is similar. The algorithm in Table 5 makes use of the function *image*, with the following meaning. The generation of the Module view establishes a natural correspondence between the elements in the «design» C&C view and those in the «design» Module view: given an element e of the «design» C&C view, image(e) the generated element in the «design» Module view. For instance, the image of a component is the homonym «component» package, etc. In the few cases in which an element gives rise to more then one element, they can be sorted out by their type. A similar argument applies to the relation between the «design» C&C view and the Deployment view, once generated.

The Deployment view is generated by reflecting each component in an executable artifact deployed in its own execution environment, and each connector in a communication path between the environments of the connected components. Note that the artifact need not be the only one actually used to deploy the component: often it will be so, but the designer is free to use this artifact just to configure the deployment of a (complex) component made up

```
for every <<structuredComponent>> C
              in <<design>> view C&C
  introduce in <<component>> package C
                  in <<design>> view Module
    a class Logic
  for every interface I in <<structuredComponent>> C
    copy in <<component>> package C
      interface I
  for every require relation from port P
              of component C to interface Ir
    introduce in <<component>> package C
      a require relation from class Logic
                       to interface Ir
  for every provide relation from port P
              of component C to interface Ip
    introduce in <<component>> package C
      an realize relation from class Logic
                        to interface Ip
```

Table 3: Components in place.

```
for every component C in view C&C
  for every port R with <<required>> interface Ir
    let Conn be the <<connector>> package
                whose name includes CR and
        Ro be the <<role>> package in Conn
              depending from C
    introduce
      class R_Proxy in package Ro and
      an realize relation from R_Proxy
                          to interface Ir
  for every port P with <<provided>> interface Ip
    let Conn be the <<connector>> package
                whose name includes CP and
        Ro be the <<role>> package in Conn
              depending from C
    introduce
      class P_Proxy in package Ro and
      a require relation from R_Proxy
                    to interface Ir
```

Table 4: Proxies in place.

of several pieces. An explicit «manifest» dependency is also introduced between related executables and components. Note that the designer is not constrained in the allocation of the execution environments in the available/necessary machines, so that he can take in due consideration also the available/necessary physical connections where to group the required communication paths between the components.

A simple example: We consider two components *C* and *S*, with ports and provided/required interfaces as in Figure 2, middle. Ports *C.r* and *S.p* are connected via the interfaces *Ir* and *Ip*, respectively. The design views generated by the algorithms are shown in Figures 4, 5, and 6.

3.3 More on ports and roles

It may be worth to discuss the implementation of connectors in an actual middle-ware, to make concrete the nature of the involved objects. The simplest framework is likely Java RMI, where the required and provided interface across a connector must coincide, and extend `java.rmi.remote`. So, the `rmic` compiler can generate the classes of the objects that take care of the inter-component communications, namely the *Stub* for the client and the *Skeleton* for the server. These objects are precisely the one playing

```
for every component C in <<design>> view C&C
  introduce in view Deployment
    an <<executionEnvironment>> C with
      an executable artifact C with
         a <<manifest>> dependency
            to component C and
         a <<build>> dependency
            to image(C) in view Module
for every connector in view C&C
       from port P of C to port P' of C'
  introduce in view Deployment
    a communication path
       from image(C) to image(C') with
         role names P and P'
```

Table 5: Place for artifacts and artifacts in place.

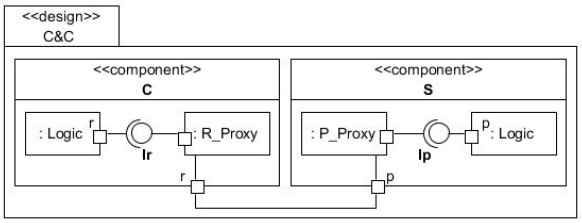

Figure 4: The introjected scenario.

the Proxy roles in the butterflies of the pair of components connected via the common interface, and are created via operations `rebind` and `lookup`, respectively.

Figure 7 shows the behavior of two communicating components, in the context of the class diagram of Figure 8. The remote object, on the server side, has type Logic: when the component is activated, it invokes the Build object of the P role, which in turn invokes the *rebind* operation in such a way that i) the «skeleton» Proxy (a subtype of the server Proxy) object is created; ii) the remote object is registered in the Naming service, at a well known location; iii) the «stub» Proxy (a subtype of the client Proxy) code is generated and uploaded in the Naming service, ready to be downloaded in the client.

On the other side, the activation of the Client involves the Build object of the R role, which performs a lookup operation on the

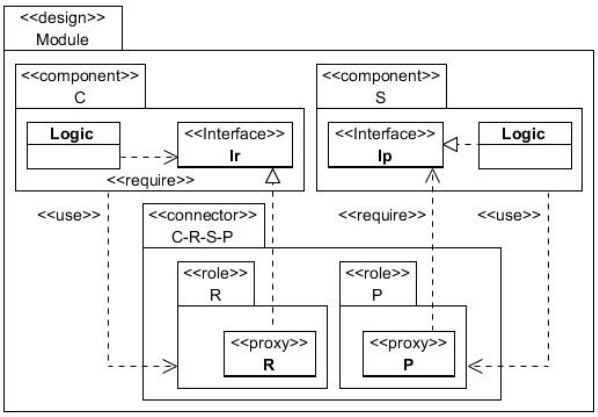

Figure 5: The design module view of the simple scenario.

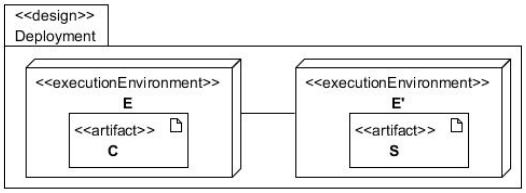

Figure 6: The design deployment view of the simple scenario.

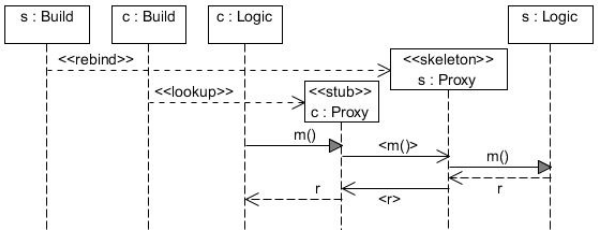

Figure 7: Proxies in RMI.

Naming service, to obtain a «stub» Proxy object, which is passed to the client Logic, so that it can access the remote component.

The Build classes in the connector roles can be viewed as concrete factories of the component ports, depending on the chosen middle-ware.

Using the butterfly pattern with RMI, the Proxy classes of the connector need no code: they simply stand for the types of the stub and skeleton objects, that are anonymous types in the framework. With another middle-ware, it may be necessary to flesh the proxy with code for themselves, may be by identifying them with classes of the framework itself.

4. QUALITY REQUIREMENTS

We now consider how to support recording some kinds of non functional requirements that arise from the design, in such a way that they can be propagated automatically to the views where they have to be obeyed. We consider run-time requirements on the distribution of the components and the execution environments and requirements that affect the development, like which programming language to use to develop a given component. In our approach, these requirements are quite naturally introduced in the «design» C&C view and propagated to the Module view (those related to development time) and to the Deployment view (those related to run time), where the pertinent engineer can take them in consideration.

UML provides a natural and flexible way to express the requirements, via its notion of *Constraint*, since no specific logic language

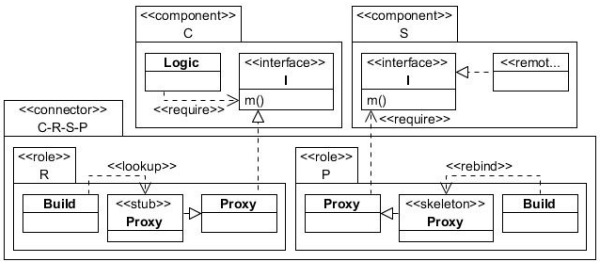

Figure 8: «design» Module view for RMI.

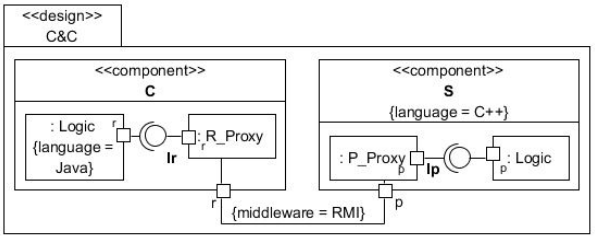

Figure 9: Requirements in the «design» C&C view

is prescribed and almost any modeling element can be decorated with constraints.

We consider only a few kind of constraints, and a simple encoding, since our purpose here is only to show how few simple and expressive rules can lead to an easy propagation of the constraints. Consider first the constraints on the implementation language: a single one can be used for a whole component, or one for the logic and some queer others for the proxies: attaching the constraint to the affected element in the model naturally conveys its scope, as shown by the constraints {language = ...} in the «design» C&C view in Figure 9: Java is to be used only for the Logic of C, and the choice for R_Proxy is delayed, while the whole C' has to be coded in C++.

The next example is related to the middle-ware to be used for a given communication between two components: the natural places where to put a constraint is the connector making it possible. For instance, in Figure 9, RMI is required for the communication between C anc C'.

Before considering how to propagate these constraints to the Module and Deployment views, we need a few remarks.

First, it is reasonable to assume that the constraints are classified with their impact, i.e., if they are relevant at development time, run time, or both. For instance, in general a requirement on the language must be taken into account during the development, but it may have an impact also on the run time environment, e.g., when a specific virtual machine is needed. Similarly, the specification of the middle-ware used for a connector has an impact at development time, since the appropriate libraries must be used, but often also at run time, namely, whenever a well known Naming service has to be available and initialized, for the connection to take place.

Besides, we assume that a propagation function prop(c) is defined for the constraints, so that they are adapted to the target view of the propagation according their classification.

Using also the image function introduced for the algorithm in Table 5, the schema of the propagation algorithm of the constraints is very simple and essentially the same for both views, but for the influence of the target view on the form of the propagated constraint:

```
for every constraint C on element E in view C&C
  affect image(E) in view V with
    prop(C) for view V
```

A simple example (cont'ed): The results of building the Deployment views and propagating the constraints from the «design» C&C view to the other two are shown in Figure 10.

4.1 Example: Security

As an example of quality requirement, we assume that the communication between ports *r* and *p* in Figure 2, middle, is encrypted,

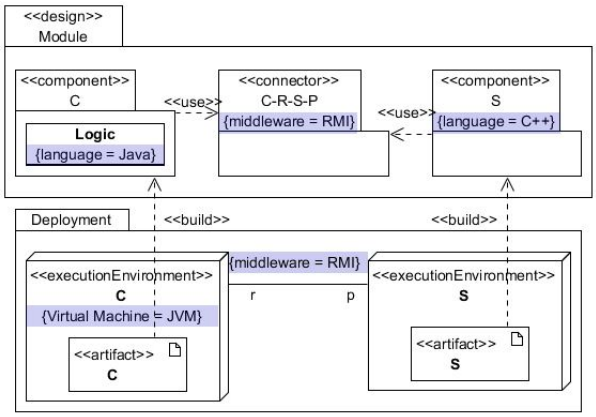

Figure 10: Propagated requirements

to secure sensible data. Security requirements may be expressed using one of the several UML *profiles* for security, e.g. [7, 8]. The minimal annotation may be a stereotyped connector, e.g. imagine to stereotype «secure» the connector between ports *r* and *p*. The architect can decide to implement this requirement using SSLRMI. In this case, the constraint on the connector is propagated as discussed in the previous section.

The balance of the design constraints may lead to a different solution, which exploits the fact that the Butterfly allow inserting an *intermediary* between a Proxy and the Logic. For instance, this can be an *adapter* [5] used to conform different interfaces: the interface of a "proxy-from-the-shelf" is likely different from that of the logic. However, an intermediary can fulfill other non-functional requirements, as securing communication by symmetric encryption. In this case the intermediary will encrypt/decrypt the message, and let the proxies to deal only with pure communication. To do that, the designer may apply the "symmetric encryption" pattern [6], with the intermediary acting as the *sender* or *receiver* of the pattern, according to its role in the component.

To accommodate this enriched structure, two packages must be created in the Module view, to place the code for the encrypter and the decrypter, respectively. Like it happens for connectors, a «use» dependency is added, from the «component» package to the new ones, and encrypter and decrypter are placed in a common container package, which can group all the shared auxiliary code needed to implement the security features.

5. DISCUSSION AND RELATED WORK

One of the advantages of the model-based development process is that a change in a part (e.g., a view) of the design can be propagated to the related parts, as advocated, e.g. in [1]. We can imagine to extend our work to accommodate, in addition to design generation, also propagation of subsequent changes. An approach for change propagation among different Architecture Description Languages has been proposed in [4], with a focus on the C&C view, and state machines.

The Module view, as generated by our refinements, reflects the structure of C&C view in the packages structure. This makes it easy to keep trace in each piece of code of the component/connector/part it is implementing, thus bridging the gap between architectural concepts and code, along the lines discussed in [2].

Like any pattern, the Butterfly pattern does not create or invent a solution, it organizes, unifies, and documents some good practices. The proposed solution decouples distributed object communication from object specific functionalities. For the communication aspects, it generalizes the *proxy* [5] and the distributed proxy [9] patterns: not only the remote communication concern is taken into account, but also other aspects, like security, can be accommodated. Moreover, in the process we propose, three views are considered and related which each other.

We presented a systematic design derivation process, which starts from an architectural C&C model and creates a platform independent design: the components are given an internal structure in the «design» C&C view, and the Module and Deployment views are created. The architect then specifies non functional constraints on the «design» C&C view, and the process propagates them to the other views. Actually, it is seldom the case that development starts from just a set of requirements: usually there are also constraints, depending on the nature of the problem, the development environment and the general context. For instance, the run-time support may be partially determined, e.g., because the physical structure of the domain entails some degree of distribution, or the implementation choices must follow the organization of the software factory with respect to the use of languages, libraries, etc. The process can easily be extended, to deal also with non functional requirements identified at the beginning of the project, not only those identified during design.

Acknowledgements. The work was partly supported by the Italian MIUR PRIN project "Security Horizons".

6. REFERENCES

[1] B. Brown. Model-based system engineering: Revolution or evolution? In *IBM Rational White Papers*, 2011.

[2] M. Broy and R. Reussner. Architectural concepts in programming languages. *IEEE Computer*, 43(10):88–91, 2010.

[3] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond, 2nd Edition*. Pearson Education, 2010.

[4] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. A model-driven approach to automate the propagation of changes among architecture description languages. *Software and System Modeling*, 11(1):29–53, 2012.

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

[6] K. Hashizume and E.B. Fernandez. Symmetric Encryption and XML Encryption Patterns. In *PLoP'09: Proceedings of the 16th Conference on Pattern Languages of Programs*. ACM, 2009.

[7] J. Jürjens. *Secure Systems Development with UML*. Springer–Verlag, 2005.

[8] T. Lodderstedt, D.A. Basin, and J. Doser. Secureuml: A uml-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441, London, 2002. Springer-Verlag.

[9] A. Rito Silva, F. Assis Rosa, T. Gonçalves, and M. Antunes. Distributed proxy: A design pattern for the incremental development of distributed applications. In *2nd Int. Workshop on Engineering Distributed Objects (2000)*, volume 1999 of *LNCS*, pages 165–181. Springer, 2001.