DAGLI OGGETTI ALLE COMPETENZE: LA PROGRAMMAZIONE ORIENTATA AGLI ASPETTI

Fabrizio Rovelli
fabrizio@microservo.org

Indice

1. Introduzione 1

L'impatto della modularità sulla qualità del software 4
Paradigmi funzionali e paradigmi object-oriented 9
Oltre la decomposizione modulare 11
Verso la completa Separazione delle Competenze 14

2. Aspetti e *crosscutting concerns* 17

Classificazione e caratteristiche degli aspetti 20
Intreccio del codice a livello d'implementazione 23

3. Aspect-Oriented Programming 29

Decomposizione e composizione Aspect-Oriented 31
Subject-Oriented Programming – Hyper/J 34
Aspect Oriented Programming – AspectJ 39
Composition Filters 44
Adaptive Programming / Demeter 46

Bibliografia 49

1. Introduzione

Uno dei principi fondamentali per una buona programmazione software è quello di codificare gli aspetti importanti riguardanti il sistema che si sta sviluppando in modo chiaro e ben localizzato. E' un principio "generale" nel senso che può toccare ogni fase dello sviluppo.
Per un progettista è essenziale trattare i problemi di realizzazione di un sistema complesso in maniera il più possibile indipendente: decomporre ogni problema complesso in problemi più semplici ed aggregarne le soluzioni per ottenere la soluzione completa.
Questo approccio risulta essere uno strumento insostituibile anche per il programmatore il quale, attraverso le strutture del linguaggio scelto per implementare le soluzioni, si propone di realizzare l'effettiva separazione dei problemi.

L'Object Oriented Programming è oramai diventato il paradigma per eccellenza, avendo rimpiazzato quasi completamente quello procedurale. Uno dei più grandi vantaggi che ha introdotto è stato quello di permettere di vedere l'intero sistema come una collezione di classi, ognuna con un ben definito scopo ed una chiara responsabilità, la cui collaborazione permette di raggiungere l'obiettivo globale dell'applicazione. Strumenti come l'incapsulazione, l'ereditarietà e il polimorfismo, hanno contribuito ad un passaggio importante dall'approccio procedurale basato su procedure e funzioni, ad una migliore suddivisione basata sulle funzionalità.
Nonostante ciò, se da un lato l'object orientation promuove ed incoraggia il riuso del codice attraverso la modularità, l'esperienza sul campo ha mostrato che non sempre riesce a gestirlo così efficacemente come era stato originariamente pensato. Ci sono parti del sistema che non possono essere viste come di responsabilità di un singolo componente ma toccano o *tagliano* l'intera struttura o parte di essa. Ci sono cioè alcune funzionalità difficili se non impossibili da esprimere in modo ben localizzato usando i costrutti di modularizzazione tradizionali come le classi (di cui gli oggetti come istanze), i componenti, o le procedure. Gestione della memoria, meccanismi di sicurezza, condivisione di risorse, logging, gestione degli errori e delle eccezioni, sono solo alcune

delle caratteristiche che risentono di questo problema, ma sono anche funzionalità alla base dello sviluppo attuale del software la cui complessità cresce quanto le dimensioni.

La Separation of Concern, (*separazione delle competenze* nella nostra lingua anche se la traduzione letterale non permette di coglierne il pieno significato) rappresenta un importante principio oltre che la naturale astrazione di quanto detto. Decomporre un sistema in parti, ognuna delle quali si occupi ed incapsuli una particolare area di interesse, chiamata concern.
Il concetto di modularizzare il codice rappresenta proprio la volontà di mantenere ben distinte e separate le unità funzionali attribuendone precise competenze. Quando però una proprietà del sistema non riesce ad essere espressa attraverso un'effettiva separazione di competenza ma coinvolge più di un modulo funzionale, si genera un problema che chiameremo di *sovrapposizione delle competenze*.

L'Aspect Oriented Programming potrebbe essere il prossimo passo della costante evoluzione del paradigma Object Oriented, o forse potrebbe evolversi verso un nuovo paradigma completamente indipendente. Comunque sia, rappresenta un recente ed interessante filone di ricerca che cerca di identificare i costrutti linguistici appropriati per esprimere tali problematiche. L'AOP introduce un nuovo elemento di analisi e di programmazione chiamato *aspetto*. L'obiettivo è di fornire metodi e tecniche per decomporre i problemi in un certo numero di componenti funzionali e in un certo numero di aspetti che coinvolgono tali componenti, e quindi di comporre componenti e aspetti per ottenere l'implementazione del sistema.

L'impatto della modularità sulla qualità del software

L'obiettivo ultimo delle teorie e dei metodi che sottostanno a qualsiasi campo dell'ingegneria è quello di aiutare gli ingegneri nella produzione di prodotti di qualità. Normalmente questo è ottenuto fornendo una distinzione attenta tra prodotto e processo. Il prodotto è *cosa* è visibile ai clienti mentre il processo è il *come* questo obiettivo può essere realizzato. Il cosa e il come, tuttavia, sono i due lati della stessa medaglia. E' con il processo, infatti, che gli ingegneri possono influenzare la qualità nei loro prodotti, ridurre il time to market, e controllare il costo di produzione. Queste considerazioni generali sono valide per ogni campo dell'ingegneria non meno in quella del software. La natura stessa del software, infatti, rende la qualità del prodotto difficile da ottenere e da valutare, a meno che non si adotti un processo adeguato.

L'ingegneria del software si è evoluta negli ultimi trent'anni proponendosi di identificare tali processi di produzione oltre che principi di sviluppo, tecniche e notazioni, con lo scopo ultimo di ottenere un prodotto di qualità, sviluppato nei tempi ed entro il budget previsti.
Se da un lato ha definito principi chiave come rigore e formalismo, separazione dei concetti, anticipazione dei cambiamenti, evolvibilità e scalabilità, il grosso sforzo è stato quello di capire come raggiungere questi obiettivi. Sono stati quindi definiti metodi, metodologie e strumenti per garantire la correttezza e l'efficacia del proprio procedere.

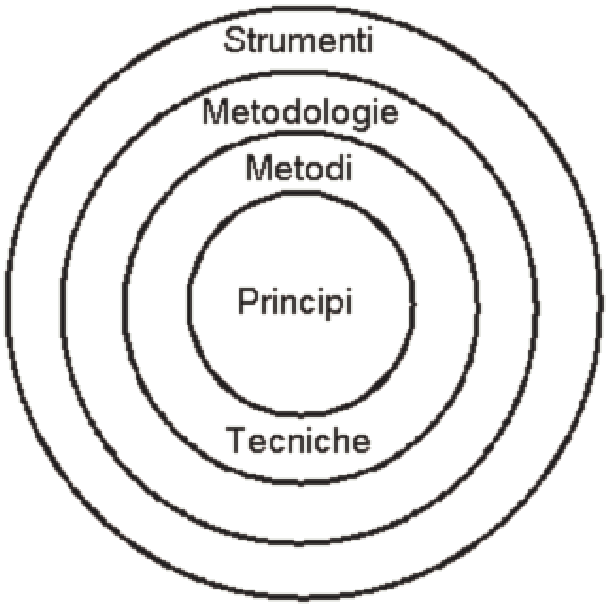

Figura 1 – Ingegneria del Software

A tutt'oggi rimangono valide le proprietà che McCall [McCall77], verso la fine degli anni 70, propose per delineare uno dei primi modelli di qualità standard, tant'è che nel 1991 ne derivò il modello ISO/IEC 9126 che venne riproposto per una seconda versione nel 2001.

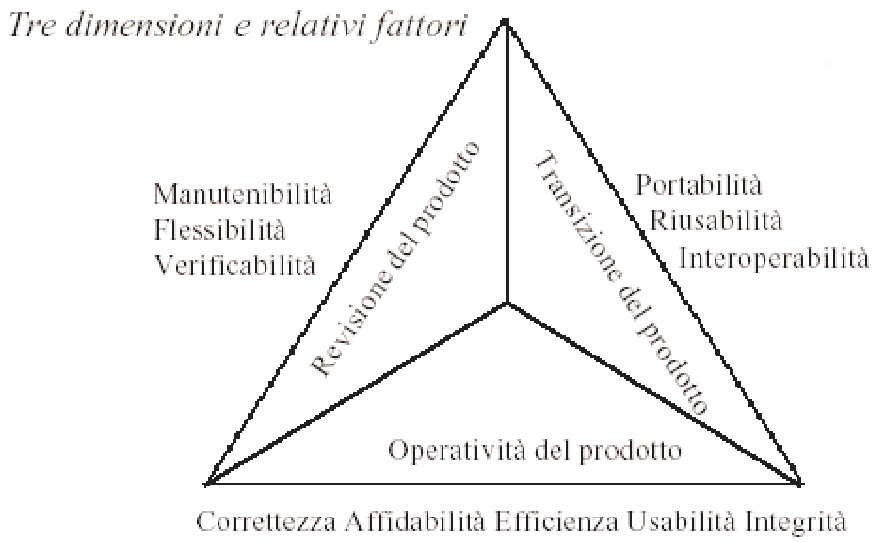

Figura 2 - Modello di qualità di McCall

Il grafico mostra chiaramente la situazione, suddividendo i fattori di qualità rispetto a tre principali dimensioni : l'operatività, la revisione e la trasposizione del prodotto.

Caratteristiche essenziali dell'*operatività* del prodotto sono :

- Correttezza: esprime in quale misura il programma rispetta i suoi requisiti.
- Affidabilità : quanto bene le funzionalità offerte rispondono ai suoi requisiti.
- Efficienza : tempi di risposta, uso della memoria ...
- Usabilità : lo sforzo per imparare ad operare con il sistema.
- Integrità : capacità di sopportare attacchi alla sicurezza del sistema.

La *revisione* del prodotto comporta :

- Manutenibilità : sforzo richiesto per localizzare e risolvere errori in un programma in esercizio.
- Flessibilità : sforzo richiesto per modificare un programma in esercizio.
- Verificabilità : sforzo richiesto per verificare il sistema, per assicurarsi che il sistema faccia effettivamente ciò per cui è stato costruito.

E per quanto riguarda la *trasposizione* del prodotto abbiamo :

- Portabilità: sforzo richiesto per trasferire il software da una configurazione ad altre configurazioni.
- Riusabilità: la misura della facilità con cui parti di software possono venire re-impiegate in altre applicazioni.
- Interoperabilità: sforzo richiesto per far cooperare/dialogare il sistema con altri sistemi.

Potremmo identificare tutte queste come *proprietà primarie* al soddisfacimento della qualità. Se quelle relative all'operatività del prodotto (esterne) risultano essere quelle di cui ne trarrà giovamento l'utente e in base le quali valuterà la qualità del prodotto, quelle relative alla revisione e alla trasposizione (interne) assumono un ruolo fondamentale, in termini progettuali, economici e manageriali, per gli sviluppatori.
Nel 2001 si stima ancora che i costi di mantenimento sono superiori al 60% dell'intero costo di produzione di cui il 20% di correzioni, il 30% di adattamenti ed il 50% di perfezionamenti. Non per nulla la produzione di software è spesso considerata come somma tra sviluppo e mantenimento.

La progettazione assume quindi un ruolo di importanza strategica nello sviluppo di sistemi, fornendo da un lato una visione ad alto livello delle componenti statiche e dinamiche fondamentali, dall'altro rappresentando il tratto d'unione tra un vasto elenco di requisiti, frutto dell'analisi, ed un mare di particolari, di minuzie, frutto del design dettagliato.
Gli studiosi della progettazione del software oggi enfatizzano la nozione di *architettura software*, perché tale nozione permette di organizzare le conoscenze progettuali, comprendere le proprietà strutturali e favorire il riuso dei componenti.

Il motto fondamentale dei progettisti di software è :

progettare per il cambiamento [Parnas72]

alla luce delle considerazioni precedenti e del fatto che, in molti casi, vengono progettate famiglie di applicazioni, non applicazioni singole.
Idealmente il progettista sceglie un'architettura software, la modella, la valuta rispetto alla specifica dei requisiti e dopo alcune iterazioni produce una specifica dettagliata di progetto che i programmatori possono implementare.

Progettare per il cambiamento ha inoltre una duplice scopo. Da un lato questa prevenzione è finalizzata ai cambiamenti che avvengono durante la manutenzione, dopo

che il prodotto è stato consegnato. Esempi possono essere i cambiamenti dei requisiti dell'utente, cambiamenti agli algoritmi (routine più efficienti), cambiamenti alla rappresentazione dei dati (si veda il problema dell'anno 2000), cambiamenti all'infrastruttura (nuovi sistemi operativi, protocolli di rete, DBMS), introduzione di nuove periferiche o modifiche del contesto sociale (cambiamenti del sistema di tassazione o del sistema monetario).
Dall'altro lato è necessario ricordare che la visione della costruzione del software paragonata ad altre discipline ingegneristiche da cui trarne idee ed approcci ai problemi, (in letteratura è spesso identificata come "impollinazione") è un'idea sbagliata.
Ne è testimonianza l'abbandono, a livello di analisi, del modello a cascata (waterfall) per un modello più flessibile come quello a spirale fino ad arrivare ai cosiddetti metodi agili come l'Extreme Programming; nello sviluppo di software, a differenza della costruzione di un progetto architetturale come un ponte, non si sa esattamente, al momento della partenza, che cosa si vuole ottenere. Un progetto che specifichi *completamente* un sistema software è perciò un'utopia e di questa "instabilità" è necessario tenerne in considerazione.

Uno degli aspetti dei moderni cicli di produzione che più influenzano la strategia di progettazione è la necessità di costruire *famiglie di applicazioni*, ovvero diversi prodotti che vengono visti come una singola applicazione che viene riutilizzata, specializzata e modificata opportunamente in contesti diversi, dando luogo a versioni differenti. Si applica un ciclo di vita speciale, esteso, in cui la fase di progetto viene effettuata su un dominio e non per un'applicazione.

Le preoccupazioni principali durante la fase di progetto sono :

- il disegno di un'*architettura software* adeguata al problema
- la *decomposizione* dell'architettura in sottostrutture
- il dettaglio di interfacce e implementazione dei moduli
- il *riuso* di schemi di progetto, di codice, di applicazioni

Anche se viene stabilita l'architettura di base, rimane problematica la sua decomposizione.
In tutti questi anni gli sforzi si sono concentrati sulla cosiddetta *decomposizione modulare* da cui ne scaturisce il comune concetto di modularità.
Al fine di stabilire cosa si intende per architettura modulare e metodo di progetto modulare, è necessario introdurre dei criteri e dotarsi di opportuni principi:

- Scomponibilità: un metodo di progetto deve favorire la decomposizione del problema in sottoproblemi risolubili separatamente.
- Componibilità : l'architettura deve favorire il riuso di elementi software già esistenti e in grado di configurarsi in base alle funzionalità richieste.
- Comprensibilità : le funzioni di un modulo devono essere comprensibili indipendentemente dal resto del sistema.
- Continuità : il metodo deve minimizzare le variazioni necessarie per far evolvere il sistema e soddisfare eventuali nuove specifiche.
- Protezione : ogni modulo deve limitare la propagazione di errori a run-time verso gli altri moduli.

Non è difficile vedere come la modularità, per sua natura, impatta profondamente sulle proprietà che McCall indicò per caratterizzare un software di qualità [McCall77] e come quindi possa essere considerata come *proprietà architetturale principale*.
Criteri come comprensibilità e scomponibilità favoriscono la manutenzione, la flessibilità e la verificabilità del sistema, come la componibilità e la continuità possono considerarsi alla base di un buon sistema riusabile e portabile.

Paradigmi funzionali e paradigmi object-oriented

Una volta stabiliti gli obiettivi da raggiungere è necessario sviluppare metodologie in grado di soddisfare i principi e le proprietà introdotte.
Fino all'inizio agli anni 80, la progettazione di un sistema software complesso era eseguita seguendo due principi base. Quello del "divide et impera", in cui il sistema veniva decomposto in componenti funzionali più semplici ed indipendenti, e quello della "centralizzazione" dove lo stato globale veniva comunque gestito in un'area condivisa, anche in caso di programmazione non sequenziale.

Le strategie di progetto erano sostanzialmente orientate alla definizione dei processi di elaborazione (progettazione funzionale) e l'enfasi era posta sulle funzioni caratteristiche del sistema, viste come trasformatrici di dati in ingresso in dati in uscita.

Il paradigma era quello strutturato, guidato da diagrammi Entità-Relazione per modellare i dati del sistema e da diagrammi Data-Flow per modellare il comportamento o il funzionamento del sistema. La decomposizione modulare era vista come decomposizione funzionale. Al modulo corrispondeva una certa funzionalità.

Il maggior svantaggio fu la scarsa consistenza tra dati e comportamento all'interno dell'intero sistema, nonché una difficile corrispondenza di concetti tra il mondo reale e implementazione.

A partire degli anni 80, l'introduzione del concetto di *abstact data type* , in cui dati e comportamento vengono strettamente accoppiati, contribuì in modo significativo a gettare le basi per le strategie data-oriented, in cui l'enfasi viene messa sulla definizione delle strutture dati, sulle operazioni che le manipolano e sulle loro proprietà.
I principi divennero: "orientamento agli oggetti", dove il sistema viene decomposto in moduli riusabili ed autonomi, e "decentralizzazione" attraverso cui lo stato globale viene distribuito anche nel caso sequenziale.

Il paradigma object oriented la fece da padrone e si impose come base per lo sviluppo di una varietà di linguaggi di programmazione, database systems e approcci di modellizzazione.
Il successo della modellizzazione object-oriented derivò sostanzialmente dal fatto che fu il primo vero traguardo al soddisfacimento delle proprietà primarie di McCall, oltre che un'ottima concretizzazione dei principi di modularità.
L'OO risultò quindi il paradigma più seguito, nonostante un'evoluzione ostacolata agli inizi degli anni 90 dal fatto che più di cinquanta approcci determinarono la cosiddetta "method war", dove ognuno cercava di imporsi come *il* metodo corretto.

Alle tecniche strutturali, basate sull'idea di costruire un sistema concentrandosi sulle funzioni, le tecniche object-oriented proposero un rovesciamento di questo rapporto costruendo il sistema a partire dalla classificazione degli oggetti da manipolare.
L'idea quindi è di non chiedersi cosa fa il sistema ma a cosa serve, di quali oggetti è composto e come si opera su di essi. Questo obiettivo permette di modellare gli aspetti della realtà nel modo più diretto e astratto possibile, mediante le nozioni di oggetto, classi e relazioni. Uno degli strumenti più interessanti che vennero introdotti fu quello delle schede CRC (Class, Responsibility, and Collaboration) [Beck89] in risposta al bisogno di documentare le relazioni di collaborazione e responsabilità tra oggetti in fase di design.

Attraverso questi elementi e con l'introduzione di concetti come *ereditarietà* e *polimorfismo* è possibile esprimere in modo chiaro e naturale i principi di una buona architettura modulare. La scomponibilità e la componibilità trovano espressione dal fatto che il sistema viene costruito in termini di moduli, sulla base del principio che ogni classe è un modulo. La natura stessa della classe, attraverso costrutti di *incapsulazione* che possano

delinearne parti private e pubbliche, fornisce uno strumento per riflettere la volontà di proteggere le parti delicate del software permettendo solo un accesso controllato e protetto a dati e funzioni. Legando gli aspetti comportamentali a quelli strutturali e modellando i concetti attraverso l'ereditarietà di ottiene un aumento della comprensibilità del sistema, la cui astrazione è garantita attraverso il polimorfismo che permette un riuso di concetti tra tipi diversi di oggetti.

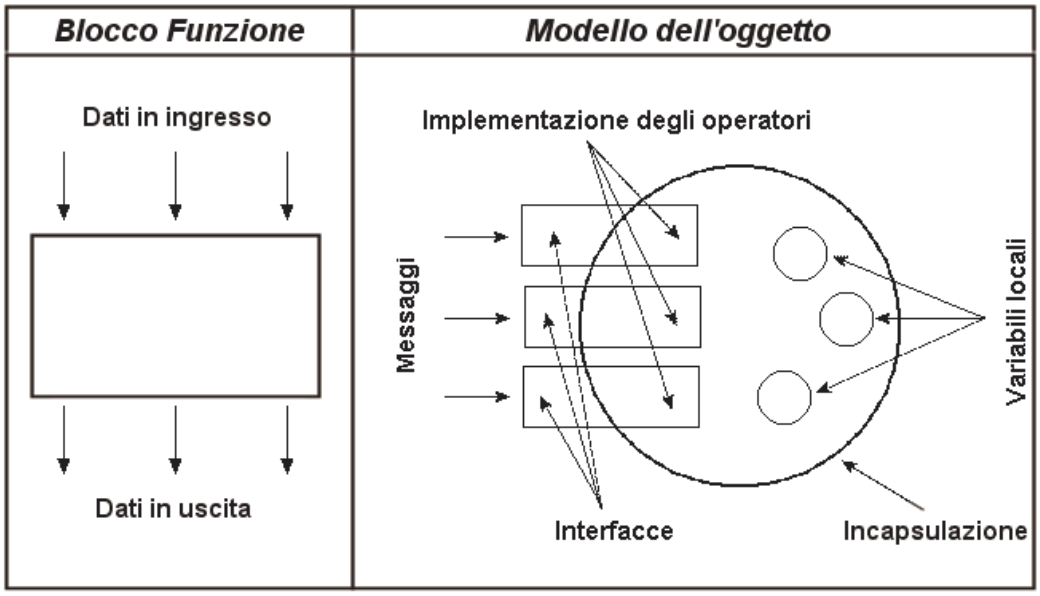

Figura 3 - Modello funzionale e Modello dell'oggetto

Oltre la decomposizione modulare

Come accennato in precedenza, l'attività fondamentale di un buon designer/architetto del software è la decomposizione del sistema al fine di soddisfare tutte le proprietà necessarie al raggiungimento di un software di qualità.

Il principio alla base delle scelte di decomposizione è quello della *Separation of Concern* (*Separazione delle Competenze*) [Dijkst76] che a tutt'oggi viene universalmente riconosciuto come uno dei principi fondamentali dell'ingegneria del software: decomporre un sistema in parti, ognuna delle quali si occupi ed incapsuli una particolare area di interesse, chiamata concern. Questo principio riconosce che non possiamo trattare più problemi alla volta ma, al contrario, dobbiamo trattare con ognuno di essi, separatamente. Afferma, tra l'altro, che gli aspetti importanti dovrebbero essere rappresentati nel

programma intenzionalmente (in modo esplicito, dichiarativo, con poca o senza aggiunta di "rumore") e dovrebbero essere ben localizzati. Questo facilita la comprensibilità, l'adattabilità, la riusabilità e molte altre buone qualità di un programma, perché intenzionalità e localizzazione ci permettono di verificare in modo più semplice come un programma implementa i nostri requisiti.

Alla luce di ciò, si può astrarre la problematica di decomposizione seguendo due principali approcci dove uno non esclude necessariamente l'altro:

- Decomposizione Modulare (Modular decomposition) : questo tipo coinvolge la decomposizione del sistema in *unità gerarchiche* come moduli, componenti, oggetti, funzioni e procedure. La parola "gerarchiche" indica che un'unità può contenere altre unità e cosi via. I contorni di tali unità sono definiti in modo tale che possano contenere le funzionalità e gli obiettivi che si propongono. Ne fanno parte la decomposizione algoritmica e quella orientata agli oggetti e lo scopo è di ottenere alta coesione e minimo accoppiamento tra le unità.
- Decomposizione ad Aspetti (Aspectual decomposition) : la principale idea è quella di organizzare i concetti (ad esempio un sistema, un dominio, una funzione o un componente) in un set di *prospettive* dove ogni prospettiva si interessa di un aspetto differente, nessuno del quale è però sufficiente a descrivere l'intero sistema. Un'importante proprietà di questo tipo di decomposizione è che ogni prospettiva produce un modello con una struttura differente, e tutti questi modelli si riferiscono sempre allo stesso concetto. Come conseguenza ci sono locazioni in un modello che fanno riferimento a locazioni in altri modelli e per questo ci si riferisce ad essi come "cross-cutting aspects" poiché attraversano e tagliano la struttura del sistema. Come vedremo in seguito, esempi di aspetti includono interazioni, strutture dati, data flow, logging, sincronizzazione, gestione degli errori, gestione della memoria e storicizzazione. La maggior parte delle attuali tecniche di modellizzazione applicano una certa forma di decomposizione ad aspetti: per esempio l'UML [BRJ99] propone differenti diagrammi per descrivere differenti aspetti dei sistemi (diagrammi delle classi, dei componenti, di distribuzione, di interazione, di stato e di attività). E' importante notare che anche se vogliamo rimanere all'interno di un aspetto, desideriamo nuovamente ridurre la complessità che percepiamo scomponendola in unità modulari.

Una rappresentazione della differenza tra aspetti e unità modulari è mostrata in Figura4. Il disegno alla sinistra mostra come le unità modulari vengano incapsulate in modo

chiaro e organizzate in gerarchie. La figura sulla destra mostra che gli aspetti tagliano (cross-cuts) un certo numero di unità modulari.

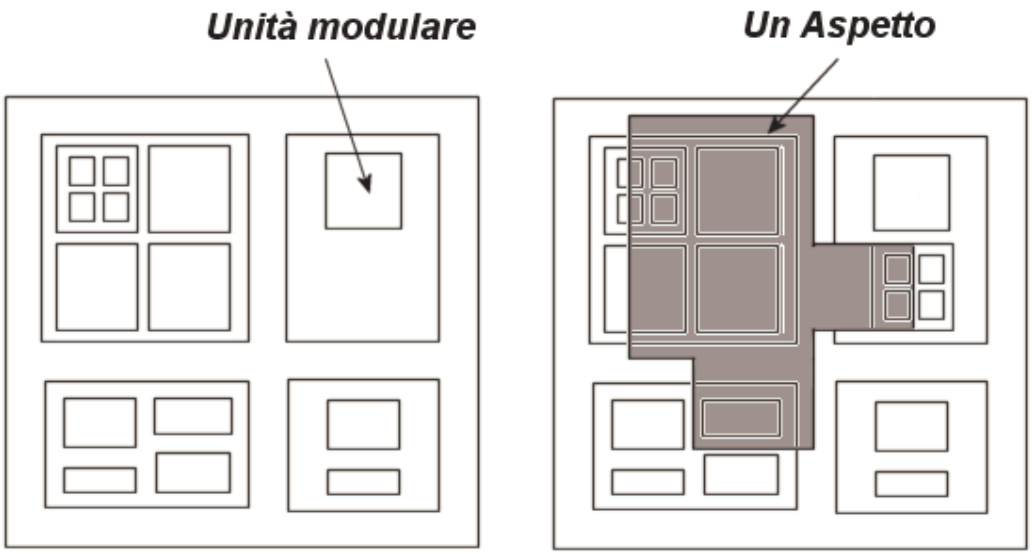

Figura 4 - Decomposizione modulare e decomposizione ad aspetti.

La seguente Figura5 mostra un'altra rappresentazione di un aspetto: il Modello A è un aspetto del Modello B perché si riferisce a diverse locazioni di B. Per esempio, B potrebbe essere una classe che implementa un certo tipo di struttura di dati astratti e A potrebbe essere una specifica di sincronizzazione riferita ai metodi di tale struttura.

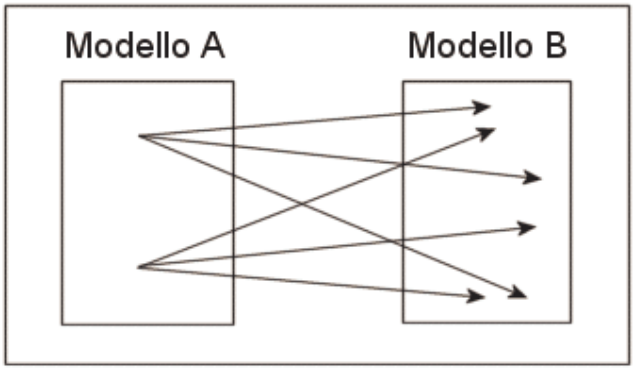

Figura 5 – Un altro esempio di aspetto

La decomposizione modulare ci permette di ottenere perfezionamenti in modo semplice aggiungendo strutture che non attraversano i contorni delle unità modulari già stabilite.

Ogni volta che dobbiamo aggiungere una struttura che oltrepassa questi contorni, stiamo applicando una decomposizione ad aspetti.
La decomposizione ad aspetti e quella modulare possono essere combinate, poiché una rappresenta il complemento dell'altra. Entrambe corrispondono alla strategia umana e naturale di modellizzazione : dipaniamo i problemi investigandoli sotto diverse prospettive e li organizziamo gerarchicamente.

La decomposizione in aspetti rappresenta le fondamenta della Programmazione Orientata agli Aspetti che verrà presentata in seguito. Come già accennato, tale decomposizione è comunque presente, in modo più o meno evidente, nello sviluppo di software, ad esempio nei diversi aspetti usati nei metodi di analisi e di design, ma la ricerca nel campo dell'AOP fornisce, alla decomposizione ad aspetti, alcune nuove prospettive. L'AOP incoraggia l'introduzione di nuovi aspetti piuttosto che l'uso di un ridotto set di aspetti generici (come nel caso dei metodi esistenti di OOA/D); enfatizza la necessità di aspetti specializzati oltre che una loro combinazione per differenti categorie di domini. L'AOP dichiara inoltre aperta la sfida di introdurre la decomposizione ad aspetti non solo nei modelli di design ma a livello di implementazione del sistema. In particolare c'è la necessità di nuovi meccanismi di composizione e nuovi costrutti di linguaggio, tutto questo, ovviamente, allo scopo di ottenere miglioramenti di qualità come la riduzione di codice "aggrovigliato", riduzione di ridondanza, migliore localizzazione del codice, migliore comprensione, manutenibilità, e riusabilità.

Verso la completa Separazione delle Competenze

La separazione delle competenze segue il fondamentale principio di nascondere la complessità attraverso l'astrazione. Astraendo le competenze e separandole, gestirne i relativi comportamenti diventa sostanzialmente meno complesso.
Sfortunatamente, gli aspetti rilevanti in un'applicazione sono solitamente "sovrapposti" e in mutua dipendenza tra loro; quindi, se tentiamo di rappresentare tutti questi aspetti esplicitamente e localmente, introdurremo inevitabilmente molta ridondanza. E' un problema da non sottovalutare poiché dobbiamo assicurarci che tutte le rappresentazioni ridondanti rimangano consistenti. Il modello globale del sistema rischia di diventare molto complesso vista la necessità di controllare tutte le differenti rappresentazioni. D'altra parte, se decidiamo di mantenere una rappresentazione con bassa ridondanza, alcuni aspetti risulteranno ben localizzati ma altri non lo saranno. E' un'idea simile alla rappresentazione di un segnale nel dominio del tempo e in quello delle frequenze. Nel

dominio del tempo riusciamo a vedere l'ampiezza del segnale in ogni istante di tempo ma non riusciamo a vederne le componenti in frequenza; nel dominio delle frequenza, d'altra parte, riusciamo a vedere le componenti in frequenza ma non l'ampiezza dell'intero segnale in un dato istante. Se decidiamo di tenere entrambe le rappresentazioni ne vedremo entrambe le proprietà ma introdurremo ridondanza.
Idealmente vorremmo implementare e memorizzare la soluzione di un problema attraverso una rappresentazione efficiente, ossia una con minima ridondanza, ed avere qualche meccanismo di supporto che permetta di estrarre dal modello ogni prospettiva di cui abbiamo bisogno. Stesse informazioni sotto diversi *aspetti*.

Progettare un sistema basandosi su tale principio è un compito che a tutt'oggi è più arduo di quello che si possa pensare e, con le tecnologie attuali, risulta essere una soluzione impraticabile. E' necessario quindi orientarci su soluzioni più pratiche: lo scopo delle odierne tecniche di modellizzazione è di sviluppare un modello che rispecchi i requisiti (funzionali e di qualità, come performance, throughput, availability, failure safety, ecc.) e, allo stesso tempo, bilanci i seguenti obiettivi :

- avere gli aspetti di maggior importanza il più possibile *localizzati*.
- diminuire la complessità di *composizione* di tali aspetti (ci si riferisce a questo problema come view rendering o complexity of compiler) poiché la decomposizione e la conseguente localizzazione di certi aspetti può considerevolmente aumentare la complessità delle trasformazioni necessarie alla loro ricomposizione.
- *minima ridondanza*
- abilità di soddisfare con lo stesso grado *cambiamenti* previsti e imprevisti.

Vedremo con maggior dettaglio ognuno di questi obiettivi quando discuteremo delle tecniche e dei meccanismi che sono stati proposti nel campo della programmazione orientata agli aspetti.

Prima di concludere è necessario ribadire la necessità di introdurre meccanismi per poter seguire l'approccio della separazione delle competenze in tutto lo sviluppo del sistema. A tale scopo è possibile distinguere due differenti livelli di separazione.
In un *livello concettuale* bisogna fornire una chiara definizione delle competenze, e identificarle in modo tale che ognuna di esse possa essere distinta concettualmente dalle altre. E' necessario inoltre assicurarsi che i concetti siano individualmente "primitivi" nel senso che non risultino composizione di altri concetti.

In un *livello d'implementazione*, la separazione delle competenze deve provvedere a un'adeguata organizzazione che isoli tali competenze. L'obiettivo di questo livello è di separare i blocchi di codice che implementano le differenti competenze, provvedendo a rendere minimo l'accoppiamento tra di essi.

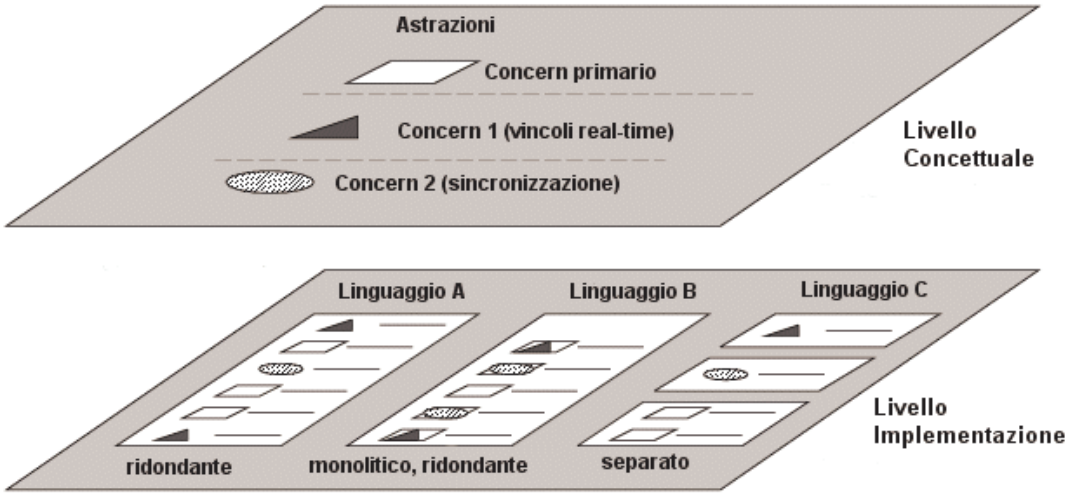

Figura 6 - Livello concettuale e livello d'implementazione

I concern identificati nel livello concettuale sono usualmente mappati nel livello d'implementazione attraverso i costrutti messi a disposizione dai linguaggi di programmazione. I concern non primari nel livello concettuale hanno però l'effetto indesiderato di essere rappresentati in costrutti monolitici (di linguaggio) che cercano di gestire i differenti concern allo stesso tempo. Nonostante l'astrazione concettuale venga riconosciuta in molte metodologie di design, pochi linguaggi di programmazione ne permettono un'effettiva implementazione. L'organizzazione del codice risulta monolitica e intrecciata come rappresentato nei Linguaggi A o B in Figura6.
Solo attraverso un'effettiva applicazione della separazione delle competenze ad entrambi i livelli se ne possono ottenere i reali benefici.

2. Aspetti e *crosscutting concerns*

Prima di introdurre le tecniche proposte dall'Aspect Oriented è necessario descrivere in modo dettagliato cosa si intende per aspetti e inquadrare in modo più preciso quelli che vengono definiti cross-cutting concern.

Lo sviluppo di un sistema software può essere visto da un utente come un dominio tecnico su cui lavorare, ma per uno sviluppatore è visto come un dominio funzionale che coinvolge, in modo naturale, diversi domini tecnici trasversali, come sistemi operativi, basi di dati, reti e così via. A loro volta tali domini possono essere considerati domini funzionali di altri sottodomini tecnici come la sicurezza, la persistenza o i protocolli.

Poiché questi domini tecnici possono essere pensati e discussi indipendentemente, sembra abbastanza naturale mantenerli separati a livello di specifica e di sviluppo, in modo tale che ai programmatori non sia necessaria la *competenza* di tutti i domini coinvolti. E' interessante notare come, con questo tipo di approccio, ogni individuo coinvolto nello sviluppo di un sistema software, possa concentrarsi con maggior dettaglio sugli aspetti di sua competenza: ogni sviluppatore può offrire la sua esperienza in merito a specifici obiettivi di sviluppo.

La situazione è quella di avere un sistema che deve soddisfare una determinata competenza primaria (o concern primario), attorno al quale è possibile identificare aspetti o concern secondari che soddisfano proprietà del concern primario o dell'intero sistema (Figura7).

La particolarità di *alcuni* di questi concern, che possono ad esempio riguardare la sincronizzazione, la persistenza, il profiling, il logging o la gestione degli errori, è quella di non essere direttamente esprimibili in modo esplicito attraverso alcun meccanismo di incapsulazione degli attuali linguaggi di programmazione OO. Come conseguenza le competenze vengono distribuite all'interno della struttura del concern primario, i cui moduli diventano, a livello di codice, un "mix" di concern differenti (Figura8).

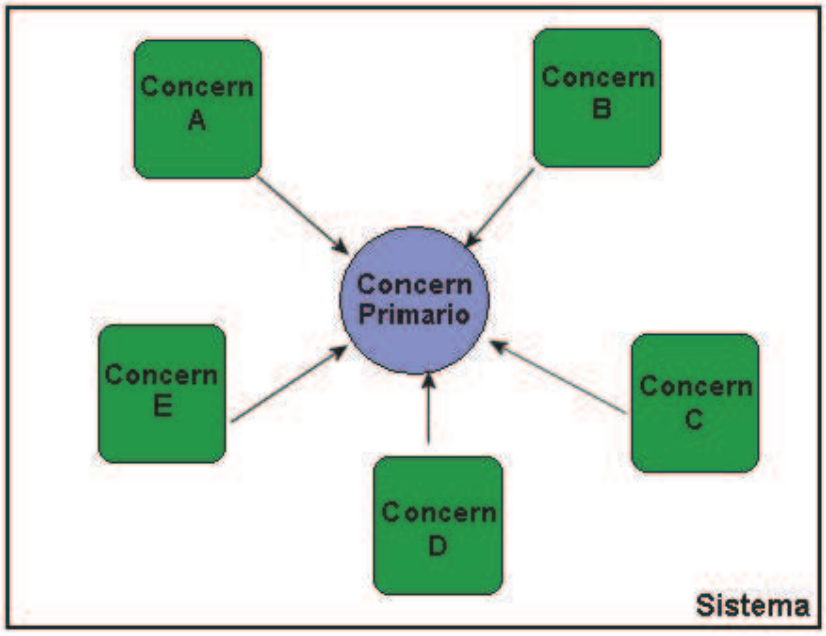

Figura 7 – Concern primari e secondari

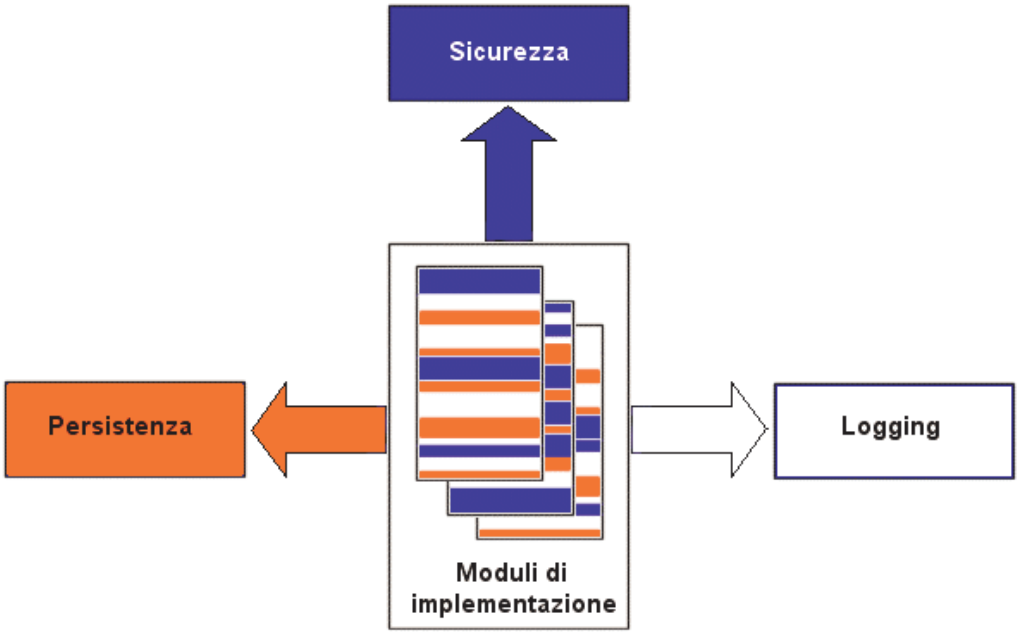

Figura 8 – Concern intrecciati nei moduli funzionali

Per realizzare la separazione delle competenze fino alla fase di programmazione, sono già stati introdotti approcci e infrastrutture software (framework).
Come già discusso, le tecniche più popolari ed efficaci per realizzare questo obiettivo sono state quelle object-oriented e quelle component-based che permettono di realizzare astrazioni ad alto livello per un dato dominio, in modo che possano essere facilmente riutilizzate ed integrate all'interno di un programma. Tuttavia queste tecniche devono affrontare alcuni difficili problemi.

Un problema noto come "*inheritance anomalies*" [Matsu+93] è emerso dalle discussioni tra ricercatori del settore sui conflitti tra sincronizzazione e riuso, e sulla necessità di una migliore separazione tra codice di sincronizzazione e funzionalità primarie in sistemi concorrenti. Più precisamente, i vincoli sulla concorrenza comportano spesso la ridefinizione di numerosi metodi nelle sottoclassi, riducendo in questo modo l'efficacia in termini di riuso dei componenti che l'ereditarietà dovrebbe favorire. Tutti i tipi di inheritance anomaly, sia relativi alla programmazione ad oggetti concorrente che relativi a sistemi real-time ad oggetti, comportano una ridefinizione forzata della maggior parte delle operazioni ereditate nelle classi derivate e questo vanifica gli sforzi spesi nel progetto di una gerarchia d'ereditarietà, non permettendo uno sviluppo incrementale.

La generalizzazione di queste problematiche ha portato alla luce la necessità di provvedere a problemi più generali che, in prima istanza, nascono dal tentativo di effettuare un'integrazione di proprietà che seguono differenti regole di composizione. Tali problemi, come già accennato, prendono in considerazione il fatto che molti concern non-funzionali di un sistema non riescono ad essere fisicamente separati rispetto ai moduli che si occupano delle caratteristiche primarie, ma devono essere distribuiti e sparsi all'interno della loro struttura perdendone ogni caratteristica di modularità.

Potremmo quindi definire un *aspetto* nei seguenti termini: a livello di design, rappresenta un crosscutting concern, quindi una competenza che taglia e attraversa la struttura del concern primario e di altri concern; a livello di implementazione, un appropriato costrutto linguistico di programmazione che permetta a questi concern di essere catturati ed espressi in unità modulari. In generale quindi un aspetto è un crosscutting concern ben modularizzato.

Classificazione e caratteristiche degli aspetti

E' importante a questo punto proporre una classificazione degli aspetti [Constan02]. Da un lato, un possibile modo è quello di classificare gli aspetti dal punto di vista del sistema: sotto quest'ottica è possibile distinguere tra aspetti funzionali e non funzionali e classificarli rispetto al loro grado di attraversamento all'interno della gerarchia delle classi o tra i livelli di un'architettura. Da un punto di vista degli aspetti stessi e delle loro reciproche interazioni, è possibile invece distinguere tra aspetti statici e dinamici, o caratterizzare alcune loro importanti proprietà come il grado di trasparenza, l'ortogonalità e il supporto al design by contract.

Aspetti funzionali e non-funzionali

Come abbiamo visto, gli aspetti sono competenze dipendenti dal dominio del problema. Come tali, possono soddisfare requisiti funzionali e non-funzionali. I requisiti funzionali gestiscono il comportamento del sistema in termini di servizi, mentre i requisiti non-funzionali forniscono proprietà e vincoli a questi servizi, interessandosi delle performance e delle semantiche del sistema. Esempi di requisiti non funzionali sono la sincronizzazione, lo scheduling, l'autenticazione, il tracing e il logging. L'interesse dell'Aspect Oriented Programming è rivolto in particolare agli aspetti non-funzionali.

Aspetti inter-class e intra-class.

In un sistema object oriented, gli aspetti possono essere caratterizzati rispetto al loro livello di crosscutting sulla gerarchia di classi del sistema. Questo approccio permette di identificare aspetti intra-class (o intra-object) e inter-class (o inter-object).

Naturalmente le due categorie non sono mutuamente esclusive, e un aspetto può attraversare diversi metodi di una classe ma allo stesso tempo può attraversare differenti oggetti. Come vedremo, l'AspectJ, il modello Composition Filters e il DemeterJ (Adaptive Programming) sono in grado di fornire una definizione di aspetto che permette di gestire entrambi i livelli di crosscutting, inter-object e intra-object.

Aspetti tra layer.

Per quanto riguarda lo sviluppo di grandi sistemi software a più livelli, un'altra distinzione è quella tra aspetti che possono esistere a livello di sistema e aspetti a livello di applicazione poiché alcuni di essi possono attraversare questi diversi layer [Lodew00].

Esempi di aspetti multi-layer sono la Quality of Service [IWQoS02], che può attraversare l'application layer , l'operating system layer e il network layer, o l'aspetto crittografica che applica differenti algoritmi in ogni livello.

Aspetti statici e dinamici.

Può essere fatta un'ulteriore classificazione considerando due particolari dinamiche :

- l'associazione a run-time tra aspetti e componenti
- i cambiamenti dinamici delle politiche di gestione degli aspetti.

Nel primo caso un aspetto A_1 può attraversare i componenti C_1 e C_2 al tempo T_1 ma potrebbe attraversare i componenti C_1, C_2 e C_3 al tempo T_2. Questo comporta che il "livello" di cross-cutting viene esteso a run-time. Nel secondo caso un aspetto A1 può tagliare i componenti C_1 e C_2 al tempo T_1 adottando una politica di gestione P_1 e, al tempo T_2, la sua politica può cambiare in P_2. In questo caso, il cambiamento di politica può richiedere un'adattabilità run-time dell'aspetto, ad esempio la sostituzione di un riferimento di un aspetto con un altro. Ovviamente possiamo avere combinazioni dei due casi.

Un aspetto introdotto a run-time e un aspetto la cui politica deve cambiare a run-time non appartengono necessariamente alla stessa categoria. L'introduzione di aspetti a run-time (vedi grado di trasparenza) prevede la necessità di sistemi aperti [Kiczales97] che possano gestirne le dinamiche di inserimento . Se un aspetto è invece progettato in modo da soddisfare un certo comportamento o fornire servizi che possono essere sensibili allo stato (di sistema o di un altro aspetto) è una preoccupazione dell'aspetto stesso e del suo comportamento.

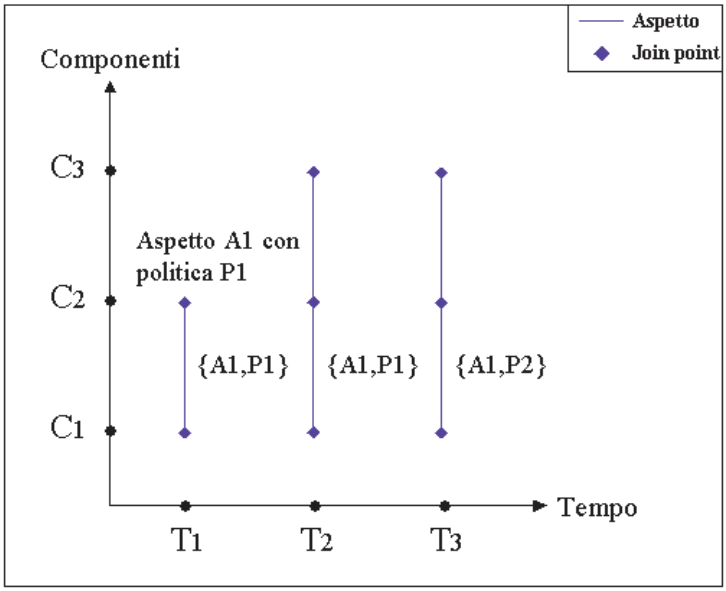

Figura 9 – Aspetti statici e dinamici

A seconda che la politica di un aspetto debba adattarsi o meno a run-time possiamo introdurre le notazioni di aspetti dinamici o statici.
Un aspetto come l'autenticazione può essere considerato statico poiché è improbabile che la sua politica di autenticazione cambi durante l'esecuzione del programma; d'altra parte una politica di scheduling in sistemi real-time potrebbe più probabilmente adattarsi a run-time. E' per questo che possiamo considerare ad esempio la schedulazione come un aspetto dinamico.

Grado di trasparenza

Può essere interessante analizzare il livello con il quale un aspetto è "invasivo" verso gli altri componenti o gli altri aspetti del sistema, cioè il grado di modifica necessaria al codice client per introdurre un aspetto o cambiarne la politica. Idealmente un sistema dovrebbe supportare adattabilità statiche e dinamiche non invasive (plug-compatibility) e il codice client non dovrebbe preoccuparsi dell'introduzione di aspetti nel sistema [Filman00]. Una tale caratteristica, alla base anche delle architetture component-based, permette di introdurre aspetti nell'applicazione senza alterare il codice esistente permettendo alti gradi di riuso e minime modifiche al sistema. Per ottenere proprietà statiche di adattabilità, devono essere forniti meccanismi a livello di linguaggio (come i joinpoint, e gli advice nell'AspectJ) mentre l'adattabilità dinamica può essere gestita da meccanismi che osservino la semantica corrente del sistema, come i contratti.

Supporto al Design by Contract

Possiamo osservare gli aspetti anche rispetto al grado con cui supportano il Design by Contract [Meyer92]. Secondo questo principio, un sistema software è visto come un set di componenti comunicanti la cui interazione è basata su precisi mutui obblighi (contratti). La possibilità di sviluppare attraverso tale principio in un contesto di AOP è stato affrontato in [Klaer+00]. Il DBC è stato introdotto nel linguaggio di programmazione Eiffel dove il contratto è profondamente legato nella definizione della classe e viene ereditato. Separare i contratti dai componenti funzionali permette di raggiungere alti livelli di riusabilità e adattabilità e nel caso di comportamenti concorrenti permette di eliminare il problema delle inheritance anomalies [Aksit94]. Attualmente, l'implementazione dell'AspectJ permette un supporto per la definizione di contratti come aspetti attraverso l'implementazione di *pre* e *post* condizioni.
L'uso dei contratti nell'AOSD permette, come accennato prima (gradi di trasparenza), di implementare un possibile meccanismo di adattabilità dinamica degli aspetti. Le definizioni dei contratti possono essere controllate a compile-time o a run-time per verificare che la composizione in atto sia valida e corretta in accordo al set di contratti forniti dallo sviluppatore. Un tale meccanismo risulta di particolare importanza in

architetture che vogliono supportare l'adattabilità a run-time. La ragione è attribuibile al fatto che l'adattabilità di un aspetto può implicare alterazioni del flusso dati e di esecuzione del programma base, generando possibili sovrapposizioni di priorità (scorretto ordine di attivazione) di un aspetto rispetto ad un altro.
Poiché un aspetto può essere creato per un determinato programma ma può potenzialmente essere usato da altri programmi, è necessario controllare se l'aspetto è valido e corretto rispetto al contesto di un dato programma base. La possibilità che un aspetto possa verificare autonomamente la sua correttezza rispetto al sistema nel quale è stato agganciato permette di ottenere codice più robusto oltre che una migliore integrazione con il sistema base. A tal fine è necessario un meccanismo di verifica dei contratti a run-time che controlli inoltre i contratti degli aspetti che dinamicamente vengono aggiunti al sistema [Findl+01].

Non-ortogonalità
E' altamente improbabile che gli aspetti siano completamente indipendenti gli uni rispetto agli altri. Molti esempi concreti nelle progettazioni reali coinvolgono aspetti che tendono ad essere interdipendenti. Tale situazione è definita come non-ortogonalità tra aspetti ed è importante perché coinvolge l'intera semantica del sistema. In stretta relazione a questo c'è la cosiddetta problematica dell'ordine di attivazione già introdotta nella discussione sul design by contract.
Consideriamo ad esempio un sistema concorrente che supporta il protocollo readers-writers. La sincronizzazione deve essere attivata e verificata prima della schedulazione e una possibile inversione di tale ordine potrebbe violare le semantiche del sistema. Se dovesse poi essere introdotto un ulteriore aspetto di autenticazione, questo dovrebbe a sua volta essere gestito prima della sincronizzazione.
Ci sono proposte concrete sia a livello di framework, attraverso moderatori di aspetti basati su pattern [Consta99], sia a livello di linguaggio, attraverso l'introduzione di meccanismi adeguati per la gestione delle attivazioni (AspectJ e Composition Filters supportano tale tecnologia).

Intreccio del codice a livello d'implementazione

Nel primo capitolo è stato evidenziato come esista una profonda difficoltà a mappare gli aspetti progettati a livello concettuale sul piano dell'effettiva implementazione.
Se gli aspetti funzionali riescono molto spesso ad essere ottimamente espressi mediante le consuete decomposizioni e metodologie OO, non si può dire lo stesso di quelli non funzionali il cui codice risulta distribuito, intrecciato e ridondante tra i moduli funzionali.

In Figura10 è riportato come esempio [Kiczal+98] il risultato grafico di un'analisi condotta sul progetto software Jakarta Tomcat dell'Apache Software Foundation. Le linee rosse rappresentano il codice che gestisce il logging dell'applicazione. A differenza di altre funzionalità come il parsing XML o l'URL pattern matching, il logging risulta estremamente non modularizzato e il codice distribuito all'interno dell'applicazione.

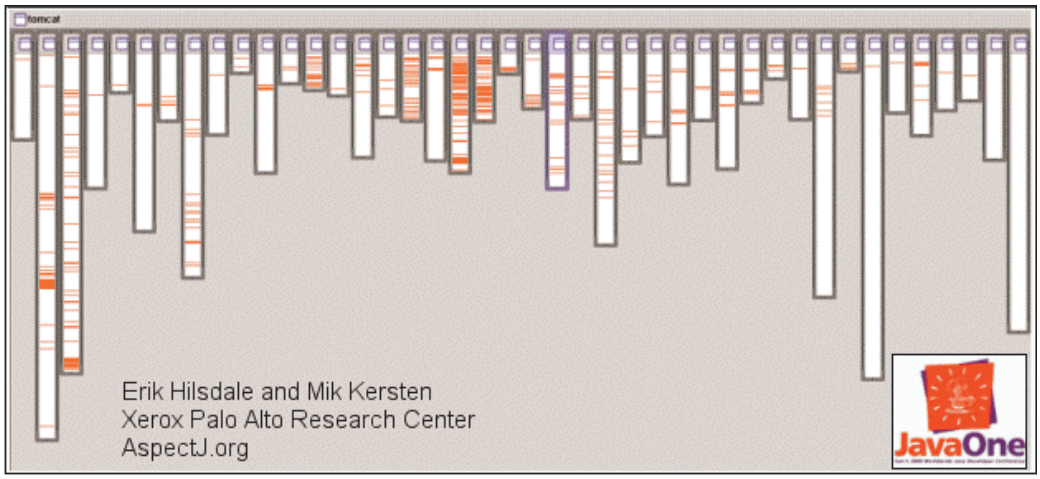

Figura 10 – Il logging risulta distribuito all'interno dell'applicazione

I differenti approcci introdotti nel campo dell'Aspect Oriented Software Development che vedremo nel prossimo capitolo, dichiarano di essere possibili soluzioni a questo problema.

Vediamo ora di descrivere brevemente alcune situazioni in cui è presente il problema dei crosscutting concern per metterne in luce le problematiche di localizzazione del codice. Se l'esposizione precedente ha permesso di classificare diversi tipi di aspetti a livello di design, la seguente trattazione mostrerà le difficoltà di operarne una buona separazione a livello d'implementazione.

Consideriamo il codice intrecciato presentato in Figura11, risultato dell'applicazione di un *Pattern Observer* [Hanen02].

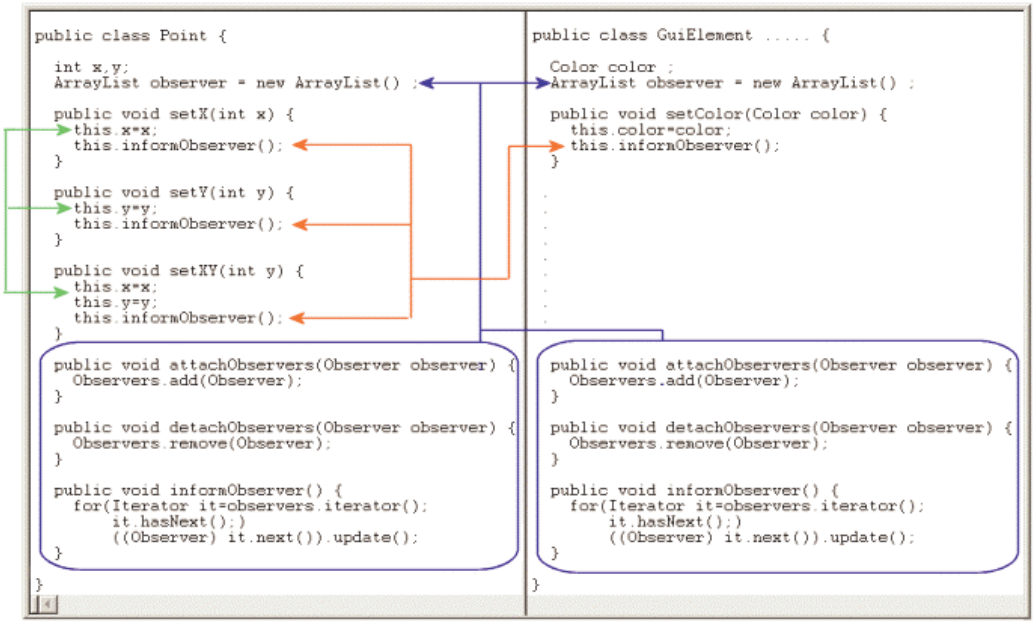

Figura 11 - Implementazione del Pattern Observer

– Entrambe le classi `Point` e `GuiElement` contengono metodi ridondanti per collegare e scollegare gli observer oltre che la stessa definizione dell'istanza `observer`. L'intenzione di queste definizioni è la stessa e per questo è possibile obiettare che entrambe le classi contengono codice ridondante e intrecciato. E' importante notare come tale codice non sia di effettiva competenza delle classi ma degli observer coinvolti. Siamo cioè in presenza di un intreccio di codice oltre che una sovrapposizione di competenze.

  L'object-oriented propone una possibile soluzione a questo problema poiché le due classi possono avere la stessa superclasse contenente le informazioni relative all'implementazione all'observer. Una tecnica come il refactoring con *estrazione di superclassi* [Fowle+99] sembra quindi adatta ad ottenere i risultati voluti, ma ci sono tuttavia diverse ragioni per non affrontare il problema attraverso l'ereditarietà. Nella programmazione orientata agli oggetti, l'ereditarietà esprime una relazione di sottotipi (subtype) più che un meccanismo di riuso. Questo tipo di relazione non è quella desiderata perché considera ed impone concettualmente una radice comune di entrambe le classi. D'altra parte estrarre una superclasse in un linguaggio di programmazione che supporta solo l'ereditarietà singola non è fattibile, per cui si rende necessaria un'organizzazione ad eredità multipla con tutte le problematiche che può comportare [Meyer97].

– Lo statement `this.informObserver()` si presenta tre volte in altrettanti metodi della classe `Point` ed una volta nella classe `GuiElement`. Lo scopo dello statement è quello di informare tutti gli observer in ascolto di ogni cambiamento di stato dell'oggetto. In particolare, lo stato di un'istanza di `Point` è descritto attraverso le coordinate x ed y mentre lo stato di `GuiElement` è descritto attraverso `color`. Questi statement intrecciano il codice dei metodi della classe introducendo ridondanza. Per questo tipo di codice intrecciato a *granularità più fine* non c'è una soluzione object-oriented poiché non si tratta di metodi intrecciati nell'oggetto ma di statement dell'observer all'interno dei metodi di `Point`.

– Gli assegnamenti `this.x = x` e `this.y = y` esistono nel metodo `setXY` oltre che nei metodi `setX` e `setY` della classe `Point`. Da un certo punto di vista questo tipo di codice intrecciato sembra essere più complesso dei precedenti. L'observer dovrebbe essere informato esattamente quando viene cambiato lo stato del punto; il metodo `setXY` non può per questo invocare `setX` e `setY` perché l'observer verrebbe informato due volte a causa del `this.informObserver` necessario in tutti e tre i metodi. In questa situazione, la necessità di inserire statements di competenza dell'observer, indebolisce la modularità dei metodi stessi all'interno della classe e il codice che risulta ridondante è quello della classe stessa oltre che dell'observer.

Un altro esempio spesso discusso di competenze intrecciate è quello relativo al codice d'implementazione del pattern Singleton [GoF95], presentato in Figura12.

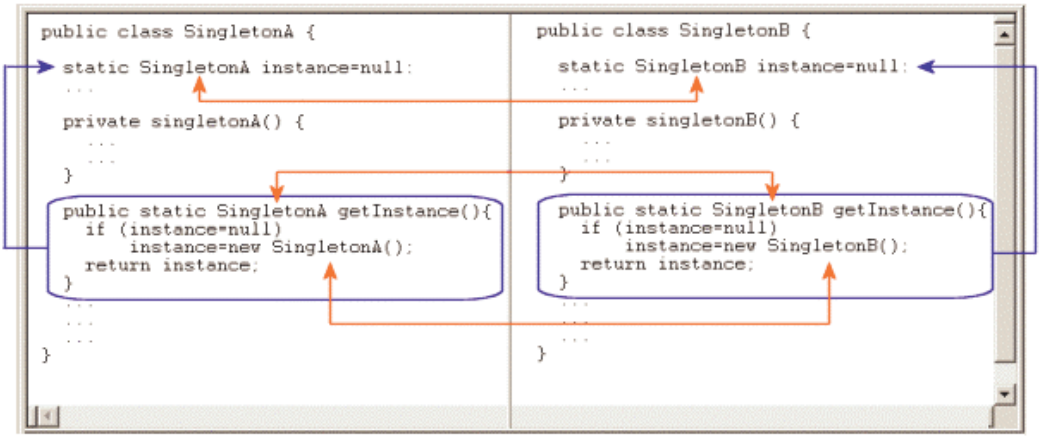

Figura 12 - Implementazione del Pattern Singleton

Da una parte il corpo di entrambi i metodi `getInstance` è pressoché lo stesso, per cui si potrebbe considerare il corpo dei metodi come codice intrecciato. D'altra parte le dichiarazioni di `getInstance` differiscono solo nei tipi che restituiscono. E' quindi più appropriato considerare l'intero metodo come esempio di codice intrecciato. Linguaggi di programmazione come Java che non permettono di dichiarare né metodi covarianti né tipi generici, non forniscono alcuna soluzione appropriata a questo tipo di codice intrecciato.
Anche in questo caso abbiamo un aspetto, la "singola istanziazione", che non riesce ad essere separato e localmente incapsulato, ma viene distribuito all'interno dei moduli funzionali di *ogni* singleton, inserendo ridondanza (si osservi anche come la definizione di `instance` differisca solo per il tipo di dato e come quindi risulti ridondante nelle due classi). Un migliore meccanismo di decomposizione dovrebbe permettere la creazione di un modulo (l'aspetto "singleton") in grado di verificare la singola istanza di tutte le classi coinvolte.

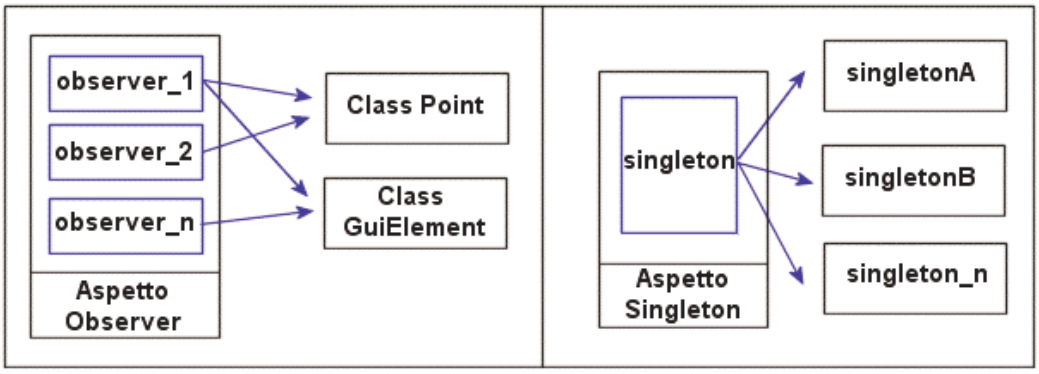

Figura 13 – Aspetto Observer e Aspetto Singleton

Gli esempi presentati permettono di identificare differenti tipi di codice intrecciato; alcuni di questi possono essere ricondotti all'esposizione fatta all'inizio del capitolo, quando è stata presentata una classificazione, più ad alto livello, degli aspetti. Se nella prima classificazione l'obiettivo era quello di esporre le diverse modalità con cui un concern poteva presentarsi a livello di progettazione e design del sistema software, ora l'attenzione è rivolta ai differenti tipi di codice intrecciato (tangled-code) che si possono presentare in fase di implementazione.

Crosscutting code dipendenti e indipendenti dall'unità modulare.
Questa caratteristica descrive se il codice dipende o meno dal modulo funzionale a cui è intrecciato o dalla "posizione" dove il codice viene inserito.

Questo tipo di tangled-code dipende, in pratica, dal tipo di unità che attraversa. Per esempio gli statement `this.informObservers()`, che possiamo trovare in entrambe le classi del pattern observer, sono esempi di codice intrecciato *dipendente dall'oggetto*, perché dipende dalla variabile `this` e presuppone che l'oggetto riferito attraverso `this` contenga un metodo `informObserver()`. Esistono anche tangled-code *dipendenti dal metodo*, che dipendono cioè da informazioni relative al metodo che intrecciano, come parametri o variabili locali. E' da notare che, anche se il codice è dipendente dall'oggetto, questo non significa che "agisca" unicamente sullo stesso oggetto: `this` in `GuiElement` si riferisce a differenti oggetti rispetto a `this` in `Point`.
Una visione più ad alto livello, considerando cioè il tangled-code come parte di un aspetto "observer", ci porta a quelli che sono stati definiti aspetti inter-class ed intra-class.

Crosscutting code costanti e variabili

I metodi per collegare e scollegare gli observer (`attachObserver(…)` e `detachObservers(…)`) sono esattamente gli stessi nelle due classi coinvolte cioè il loro codice non presenta "variazioni" e per questo possono essere definiti *costanti*. Anche gli statement `this.informObservers()` non cambiano (anche se `this`, come già detto, è riferito a differenti oggetti) e possono considerarsi un altro esempio di crosscutting code costante. Al contrario, l'implementazione del pattern Singleton contiene tangled-code *variabile* : il tipo di ritorno del metodo `getInstance` cambia a seconda del contesto in cui è inserito. Anche il corpo di questi metodi è variabile perché nel `SingletonA` viene creata una nuova istanza al `SingletonA` e nel `SingletonB` una nuova istanza al `SingletonB`.
Quando si parla di *crosscutting costante* si intende quindi che la modalità con cui certi codici tagliano una certa struttura è completamente descritta senza alcuna alterazione. In altre parole si intende che la strategia del codice non necessita di modifiche e può essere usata così com'è in nuovi e diversi contesti. Un *crosscutting variabile*, pur richiedendo tecniche di composizioni più complesse, ha sicuramente il vantaggio di ottenere maggior adattabilità, poiché basa la sua strategia su parametri che dipendono da dove viene inserito (ad esempio le classi che taglia) e permette una maggior integrazione in contesti diversi.
Anche in questo caso l'AspectJ, come altri approcci che vedremo, ne permette un'efficace gestione di entrambi.

3. Aspect-Oriented Programming

L'obiettivo dell'Aspect Oriented Programming è quello di fornire metodi e tecniche per decomporre i problemi in un certo numero di componenti funzionali e un certo numero di aspetti, e comporre componenti ed aspetti per ottenere l'implementazione del sistema.
Una progettazione basata sui concetti della separation of concern necessita, come abbiamo visto, di meccanismi che a livello di implementazione permettano di ottenere questa separazione.
In generale l'approccio AOP opera come mostrato in Figura14. Un linguaggio AOP fornisce costrutti linguistici per separare sintatticamente componenti ed aspetti e permette di integrare, tramite un meccanismo di unione detto *weaver*, aspetti e codice funzionale per ottenere l'implementazione del sistema.

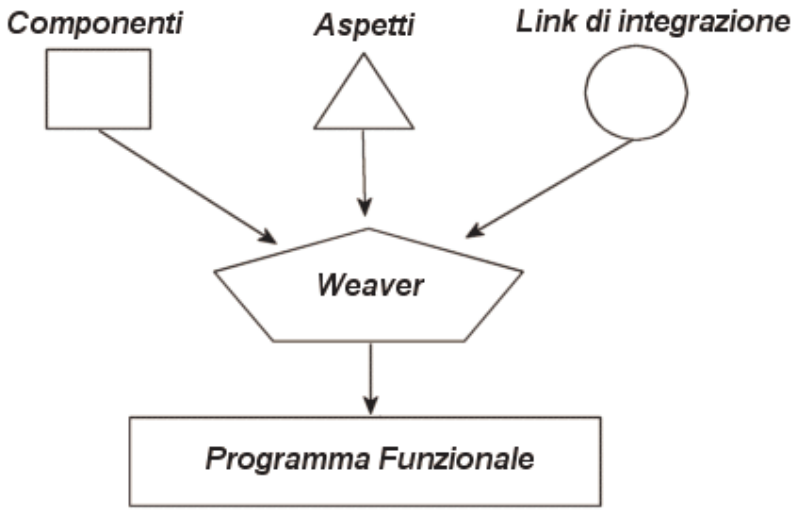

Figura 14 - L'idea base dell'aspect oriented programming

```
1  package stack;
2
3  interface Constants{
4      final int StackSize = 10;
5  }
6
7
8  public class Stack implements Constants{
9
10   synchronized                                    // synch
11   public void push(Object o){
12     while(top == StackSize-1) try{                // synch
13          wait();                                  // synch
14       } catch (InterruptedException e){           // synch
15          e.printStackTrace();                     // synch
16       }
17       elements[++top] = o;
18       if (top == 0) notifyAll();                  // synch
19   }
20
21   synchronized                                    // synch
22   public Object pop(){
23     Object ris;
24     while(top == -1) try{                         // synch
25          wait();                                  // synch
26     } catch (InterruptedException e){             // synch
27          e.printStackTrace();                     // synch
28     }
29     ris = elements[top--];
30     if (top == StackSize-2) notifyAll();          // synch
31     return ris;
32   }
33
34     private int top = -1;
35     private Object[] elements = new Object[StackSize];
36 }
```

Figura 15 -Implementazione intrecciata dello Stack

Dobbiamo risolvere principalmente due questioni :

- *Quali sono gli aspetti importanti da separare.* Quello che stiamo cercando sono set di concern riusabili da utilizzare nella decomposizione di problemi; alcuni di questi saranno maggiormente "application specific", ad esempio gli aspetti di configurazione per servizi di rete, mentre altri saranno più generali, come la sincronizzazione o il controllo di flusso. Ovviamente, differenti domini richiederanno differenti set di concern. Alcuni di questi, sia specifici che generali, risulteranno essere *aspetti* secondo la definizione data nel capitolo precedente e sarà comunque importante, come già discusso, operarne la scelta bilanciando tra complessità, ridondanza e localizzazione.
- *Quali meccanismi di composizione possono essere usati.* Se un aspetto non può essere separato in modo chiaro usando i consueti approcci come le chiamate a metodi e procedure (a cui ci si riferisce spesso come *generalized procedures*, caratterizzate dal fatto che sono chiamate dal codice client), abbiamo bisogno di altri meccanismi di composizione. Da questi meccanismi si vuole ottenere un vasto spettro di

binding, oltre che un buon grado di adattabilità non invasiva, ad esempio attraverso composizioni o trasformazioni automatiche piuttosto che cambiamenti manuali.

Quando necessario, per permettere una miglior esposizione dei dettagli implementativi, useremo un semplice esempio di sincronizzazione di una struttura dati. Il codice presentato in Figura15 è composto da statement che implementano le funzionalità dello stack e statement (indicati con `//sync`) che implementano la sincronizzazione dei metodi. Il concern primario e l'aspetto di sincronizzazione risultano fortemente intrecciati.

Decomposizione e composizione Aspect-Oriented

Analizzeremo quatto principali approcci che si propongono di soddisfare l'incapsulazione di varie proprietà di sistema - inclusi gli aspetti che attraversano i moduli delle unità funzionali - e forniscono concreti modelli di composizione. Queste tecniche estendono il modello di programmazione Object Oriented ma è importante osservare come alcune di queste possano essere applicate anche in modelli procedurali [Highl+99]. Chiamate a funzioni, parametrizzazioni statiche e dinamiche ed ereditarietà sono tutti esempi di importanti meccanismi di composizione supportati dai linguaggi convenzionali. Nonostante ciò, come già discusso in precedenza, non tutti gli aspetti rilevanti possono essere adeguatamente composti usando tali meccanismi e nasce la necessità di introdurne dei nuovi. Gli *aspect* introdotti dall'Aspect Oriented Programming, i *subject* (e relative regole) proposte nella Subject Composition, i filtraggi nel modello d'oggetto della Composition Filters e le strategie di attraversamento del Demeter sono alcuni esempi che discuteremo.

Idealmente vorremmo ottenere un meccanismo di composizione di aspetti che permetta:

- minimo accoppiamento tra aspetti e componenti
- modalità e tempi di binding differenti tra aspetti e componenti
- adattabilità non invasiva degli aspetti verso eventuale il codice esistente

Minimo accoppiamento.

Quello che si richiede è minimo accoppiamento tra aspetti (e tra aspetti e componenti) piuttosto che la completa separazione, perché la completa separazione è possibile solo in alcuni specifici casi.

La ragione di ciò è da ricercarsi nel grado di ortogonalità degli aspetti e nel fatto che questi descrivono differenti prospettive sugli stessi singoli modelli. Cerchiamo di chiarire questa osservazione attraverso un esempio.

Il disegno presentato in Figura16-A mostra tre prospettive ortogonali di un oggetto tridimensionale; in questo esempio le prospettive sono un trapezio e un triangolo sui piani laterali e un cerchio sul piano di fondo. L'oggetto è quindi scomponibile in prospettive. Pensiamo invece ora di usare i tre piani come "generatori", disegnando alcune figure bidimensionali su di essi e cercando di interpretarli come prospettive di un oggetto tridimensionale da costruire. Anche se queste risultano sono ortogonali (in termini di angoli), le figure sui piani non possono essere scelte indipendentemente. La Figura16-B mostra un set di figure bidimensionali che risultano inconsistenti per un modello, con le quali cioè non è possibile costruire l'oggetto 3-D considerandole come sue prospettive ortogonali, mentre in Figura16-C è rappresentato un set consistente di figure bidimensionali.

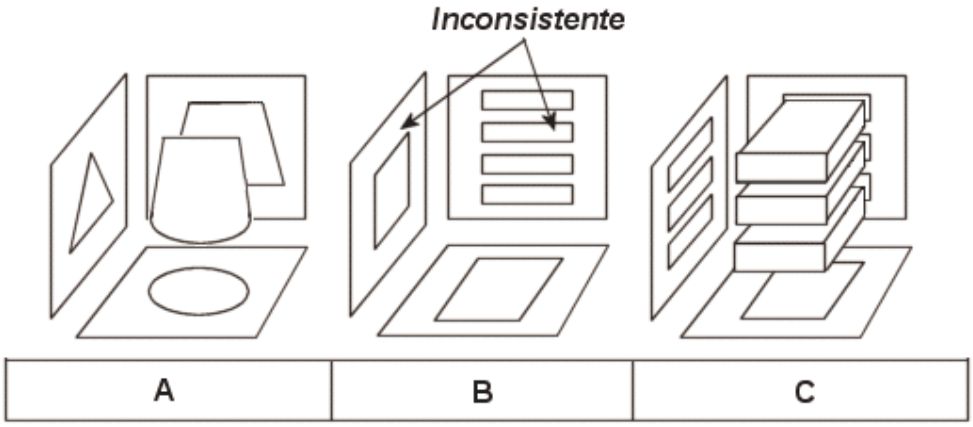

Figura 16 - Esempio di prospettive ortogonali consistenti e inconsistenti

Il minimo accoppiamento è il principio su cui si basa l'introduzione di alcuni elementi comuni in tutti gli approcci orientati agli aspetti: i cosiddetti *join point*.
In linea di principio, i join point sono i punti in cui gli aspetti "influiscono" sui componenti; vengono utilizzati in fase di decomposizione per mappare (attraverso riferimenti) le corrispondenze aspetto-oggetto e dal meccanismo di integrazione, il weaver, per fondere gli uni agli altri. Si può dire che la natura stessa dei join point definisce i diversi tipi di approccio all'AOP.

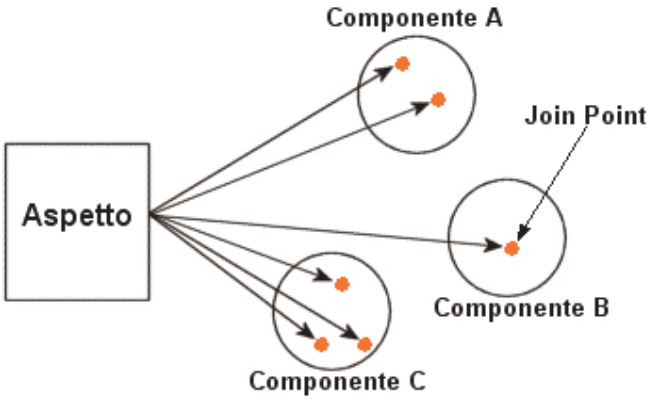

Figura 17 - Join Point

Binding.

E' importante che un meccanismo di composizione supporti una tecnologia adeguata di binding temporale intendendo con ciò la possibilità di "legare" aspetti e componenti prima o durante il runtime. Oltre a questo, l'obiettivo è quello di supportare sia binding statici che dinamici. Queste caratteristiche sono profondamente legate alla tecnologia che viene utilizzata del weaver e del compilatore e incidono sul grado di adattabilità degli aspetti visto nel capitolo precedente.

Attraverso un binding statico gli aspetti possono essere legati ai componenti e non possono essere legati nuovamente; un esempio è il binding statico ed il metodo di inlining nel C++. Con un binding dinamico un aspetto può essere automaticamente integrato prima dell'uso e tolto dopo l'uso. Un binding di questo tipo può essere utile in quei casi in cui si ha un frequente cambiamento di aspetti. L'usuale implementazione, come quella in C++ o Java, fa uso delle virtual function table.

Esistono, in effetti, tipi di binding più sofisticati, come quelli parametrizzabili o quelli sviluppati nella tecnologia HotSpot della Sun Microsystems, grazie ai quali è possibile ottenere un meccanismo di composizione più generale con supporto dinamico e statico allo stesso tempo.

Adattabilità non invasiva.

Con questo termine si intende l'abilità di adattare un componente o un aspetto senza modifiche manuali. Idealmente si vorrebbe esprimere ogni cambiamento come un'operazione additiva, utilizzando qualche operatore di composizione per aggiungere i cambiamenti al codice esistente. Il codice client non dovrebbe essere fisicamente modificato ma le espressioni di composizione stesse dovrebbero regolarne il cambiamento di semantica rispetto al codice originale.

Si potrebbe pensare che l'ereditarietà stessa è un operatore di composizione: possiamo infatti definire una nuova classe "per differenze". Sfortunatamente l'ereditarietà è non

invasiva solo rispetto alla superclasse ed il codice client deve essere cambiato in quelle situazioni in cui si vuole creare un oggetto della nuova classe derivata. Quello che si cerca è un meccanismo non invasivo rispetto sia ai componenti che al codice client, caratteristica che comunque non è possibile ottenere in tutti i casi.
Nei prossimi paragrafi vedremo esempi concreti di meccanismi di composizione che supportano questo tipo di adattabilità non invasiva, come quello ottenuto dalle regole di composizione e combinazione nella Subject Oriented Programming o quello attraverso joinpoint e advice proposto nell'AspectJ. Un'ultima osservazione riguarda il fatto che il basso accoppiamento è strettamente legato alle caratteristiche di adattabilità. Se infatti l'accoppiamento tra gli aspetti è minimo, risultano minimi anche i cambiamenti richiesti.

Subject-Oriented Programming – Hyper/J

E' stata proposta da Harrison e Ossher della IBM Thomas J. Watson Research Center come estensione del paradigma object-oriented per risolvere il problema di gestire diverse prospettive (*subjective perspectives*) applicate agli oggetti da modellare [Clarke+99]. SOP rientra in quella che viene definita Multi-Dimensional Separation of Concerns [Osshe+00], l'approccio IBM all'aspect-oriented programming.
Consideriamo ad esempio un oggetto che rappresenta un `libro`. Per il dipartimento marketing di una casa editoriale sarà importante che includa attributi come `area_di_appartenenza` o `breve_estratto`, mentre il dipartimento di produzione sarà interessato ad altri attributi come il `tipo_di_carta` o la `rilegatura`. La gestione in diversi contesti di un oggetto non è l'unica ragione dell'introduzione di prospettive: l'approccio cerca di risolvere il problema, discusso nel primo capitolo, di integrare diversi sistemi, sviluppati con un alto grado di indipendenza, ad esempio due applicazioni sviluppate per due differenti dipartimenti della casa editoriale. Oltre a questo, l'obiettivo è rendere possibile l'integrazione di nuove estensioni in un sistema esistente in modo non invasivo.

Ogni prospettiva lavora su un cosiddetto *subject*. Un subject è una collezione di classi o frammenti di classe (ad esempio i mixin[Flatt+98]) legati da una relazione di ereditarietà o altre relazioni, come quella d'aggregazione, d'associazione e così via. Quindi un subject è semplicemente un modello d'oggetto parziale o completo.

I subject possono essere composti attraverso particolari regole di composizione.

- *Regole di Corrispondenza* : permettono di specificare la corrispondenza tra classi, metodi ed attributi appartenenti a *differenti* subject che devono essere composti.
- *Regole di Combinazione* : permettono di specificare come le classi, i metodi e gli attributi dei subject sui quali sono state fissate regole di corrispondenza, contribuiscono al risultato finale.
- *Regole di Corrispondenza-Combinazione* : permettono di specificare, allo stesso tempo, corrispondenze e combinazioni.

Una prospettiva è quindi un concern del sistema e i subject contribuiscono alla rappresentazione del concern attraverso le composizioni e i filtraggi ottenuti dalle regole. Ogni subject costituisce una nuova dimensione del processo di design [Osshe+99], e tali dimensioni possono essere, in linea di principio, ortogonali l'un l'altra.

Le *regole di corrispondenza* permettono di specificare le corrispondenze, se ce ne sono, tra classi, metodi ed attributi di oggetti che appartengono a diversi subject. Per esempio potremmo usare una regola per esprimere che il libro nell'applicazione del dipartimento di marketing è lo stesso libro dell'applicazione per il dipartimento di produzione, anche se le corrispondenti classi hanno differenti nomi. Le regole possono creare corrispondenze anche tra metodi ed attributi di queste classi. Per esempio, potremmo dichiarare che l'attributo `titolo` nella prima classe è lo stesso attributo `titolo_libro` nella seconda. E' inoltre possibile usare regole di corrispondenza "automatiche" che permettano di stabilire le corrispondenze delle due classi per tutti i membri che hanno gli stessi nomi e adottare una tecnica di override per alcuni di essi, definendone le specifiche regole di corrispondenza.
Dopo aver stabilito le corrispondenze tra le due classi (libro), possiamo usare le *regole di combinazione* per specificare come queste corrispondenze devono essere combinate. La classe risultante includerà sia i metodi e gli attributi indipendenti, sia la combinazione dei metodi e attributi "mappati" in accordo alla regola di combinazione. Per esempio, un metodo di una classe potrebbe sovrapporsi al corrispondente metodo dell'altra classe, oppure entrambi i metodi potrebbero essere eseguiti in uno specifico ordine.
Quando vengono composti due o più subject sviluppati in modo indipendente è necessario solitamente sviluppare un subject addizionale, chiamato *glue subject*, che faccia da collante tra i due e includa il codice necessario per combinarli.
Le *regole di corrispondenza-combinazione* sono regole di convenienza e forniscono un metodo veloce per specificare, allo stesso tempo, corrispondenze e combinazioni.

La IBM ha sviluppato un prototipo di supporto ai subject (Hyper/J) che può essere integrato nei tools quali VisualAge for C++, VisualAge for Java e Smalltalk. Un'utile

caratteristica nel prototipo per Java è la possibilità di specificare le corrispondenze e le combinazioni in modo visuale attraverso un'interfaccia grafica.
Utilizzando l'approccio SOP possiamo separare in modo efficace la sincronizzazione e il codice funzionale presentato in Figura15 perché entrambi possono essere incapsulati in due separati subject. Gli elementi in gioco sono principalmente quattro: l'implementazione puramente funzionale dello stack (Figura18), l'aspetto di sincronizzazione (Figura20), un modulo di mappatura dei concern (Figura19) e le regole di collaborazione e combinazione (Figura21).

```
package stack;

interface Constants{
    final int StackSize = 10;
}


public class Stack implements Constants{

  public void push(Object o){
      elements[++top] = o;
  }

  public Object pop(){
    return elements[top--];
  }

    private int top = -1;
    private Object[] elements = new Object[StackSize];
}
```

Figura 18 - Implementazione funzionale della classe Stack

```
package stack : Feature.Kernel
class stack.Stack : Feauture.Kernel
class stack.SynchStack : Feature.Synch

operation pop : Feature.Kernel
operation push : Feature.Kernel

field top : Feature.Kernel
field elements : Feature.Kernel

operation stack.SynchStack.innerPop : Feature.Synch
operation stack.SynchStack.innetPush : Feature.Synch
```

Figura 19 - Mappatura dei concern

```
package stack;

public class SyncStack implements Constants{

  public void push(Object o){
    synchronized(lock){
       while(top == StackSize-1) try{
          lock.wait();
          } catch (InterruptedException e){
            e.printStackTrace();
          }
       innerPush(o);
       if (top == 0) lock.notifyAll();
    }
  }

  public Object pop(){
    synchronized(lock){
       Object ris;
       while(top == -1) try{
          lock.wait();
          } catch (InterruptedException e){
            e.printStackTrace();
          }
       ris = innerPop();
       if (top == StackSize-2) lock.notifyAll();
       return ris;
    }
  }


  private void innerPush(Object o){}
  private Object innerPop() {return null;}
  private Object lock = new Object();

  private int top = -1;
  private Object[] elements = new Object[10];
}
```

Figura 20 - Un esempio di implementazione del concern sincronizzazione in SOP

```
hypermodule HSyncStack

      hyperslice:
          Feature.Kernel
          Feature.Synch

      relationship:
          overrideByName;

          equate operation
              Feature.Synch.innerPush,
              Feature.Kernel.push

          equate operation
              Feature.Synch.innerPop,
              Feature.Kernel.pop

end hypermodule
```

Figura 21 - Un esempio di hyper-module

La mappatura dei concern permette di dichiarare in modo esplicito l'esatta corrispondenza tra unità funzionali e concern di appartenenza: ad esempio, ogni `operation pop` appartiene alla dimensione `Feature` e precisamente al concern `Kernel` che implementa la sola funzionalità dello stack. `Feature` non è semplicemente un concern ma una dimensione. Una delle caratteristiche di questo approccio, che rientra in quella che viene definita Multi-Dimensional Separation of Concern, è infatti quella di gestire più "livelli" di concern, chiamati *dimensioni*. Una dimensione ha la caratteristica di poter contenere più concern e potremmo pensarlo come un ulteriore meccanismo di aggregazione. Ad esempio potremmo aggiungere una dichiarazione nella mappatura dei concern del tipo :

```
operation stack.Stack.main : Development.Test
```

per indicare che `main` appartiene alla dimensione `Development` riguardo al problema di testare la classe.

Il subject che incapsula l'aspetto di sincronizzazione (Figuara20) assomiglia a tutti gli effetti ad una normale classe. E' necessario però notare che i metodi `innerPop()` e `innerPush()` sono vuoti e la loro implementazione viene fornita attraverso la composizione.

A questo punto abbiamo il subject con il codice funzionale, il subject con l'aspetto di sincronizzazione e l'esatta mappatura unità-concern, quindi manca solo la definizione delle regole di composizione e combinazione viste in precedenza.
Le regole vengono raccolte in un hyper-module, che chiameremo HSyncStack (Figura21). In questo modulo sono indicati i concern, le dimensioni e le regole che si intendono usare. In particolare sarà necessario stabilire che durante la composizione, il codice dell'unità `Feature.Sync.innerPush` dovrà essere sovrapposto dal codice di `Feature.Kernel.push` e il codice di `Futures.Sync.innerPop` da quello di `Feautur.Kernel.pop`. Il modulo che viene generato da questa fusione è una regolare classe java che potrà essere integrata nell'applicazione.

La potenza di questo meccanismo sta anche nell'abilità di specificare come deve essere compiuta la composizione tra le unità. Osserviamo come nell'hyper-module siano state definite le modalità di applicazione delle regole di composizione: sotto la direttiva `relationships`, lo statement `overrideByName` indica una corrispondenza "per nome" dei subject coinvolti mentre gli statement `equate operation` identificano gli

eventuali override. Queste sono solo alcune delle regole di composizione che si possono adottare nell'Hyper/J e la gamma messa a disposizione permette di ottenere diversi livelli di granularità.

Aspect Oriented Programming – AspectJ

La strategia maggiormente conosciuta è probabilmente quella implementata dall'AspectJ, un'estensione per il linguaggio Java progettata alla Xerox Palo Alto Research Center dal team di Gregor Kiczales [Kiczale01].

I linguaggi AOP hanno tre elementi critici : un modello dei join point, un meccanismo per identificarli ed un meccanismo per realizzarne l'implementazione associata.
L'AspectJ è basato su un ridotto ma potente set di costrutti:

- I *join point* sono punti ben definiti nel flusso d'esecuzione dell'applicazione.
- I *pointcut* permettono di rappresentare collezioni di join point oltre a particolari valori che questi possono assumere.
- Gli *advice* sono particolari costrutti, tipo i metodi tradizionali, usati per definire i comportamenti dei join point.
- Gli *aspect* sono unità di implementazione modulare dei crosscutting concern, composti da pointcut, advice e le tradizionali dichiarazioni Java.

Il *modello join point* fornisce uno strumento di riferimento con cui è possibile definire la struttura dei crosscutting concern. Nel modello dell'AspectJ i join point possono essere considerati come nodi di un grafo di chiamate (object call graph) , gestito a runtime. I nodi includono i punti in cui un oggetto del sistema riceve una chiamata ad un metodo o punti in cui viene fatto riferimento ad un attributo; gli archi rappresentano il flusso di relazioni tra i nodi. Il controllo passa attraverso i join point due volte: una prima volta quando viene attraversato, e una seconda volta dopo che il comportamento associato al determinato join point è stato eseguito.

L'AspectJ fornisce numerosi tipi di join point dinamici che permettono di ottenere diversi livelli di granularità (Tabella1).

Tabella 1

Tipo di join point	Punti nell'esecuzione del programma in cui….
methods calls constructor calls	viene chiamato un metodo o il costruttore di una classe. I join point call sono nell'oggetto chiamante o in nessun oggetto se la chiamata è da un metodo statico.
methods calls reception constructor calls reception	un oggetto riceve una chiamata ad un metodo o al costruttore. I join point reception agiscono all'interno del codice chiamato, dopo l'istante in cui il controllo è stato trasferito all'oggetto chiamato ma prima di qualsiasi chiamata a metodi o costruttori.
methods executions constructor executions	un certo metodo o costruttore è stato invocato.
field gets	un campo di un oggetto, di una classe o interfaccia viene letto.
field set	un campo di un oggetto o di una classe viene settato.
exception handler executions	un'exception handler viene invocato.

Un *pointcut designator* è un set di join point che opzionalmente espone alcuni valori nel contesto di esecuzione dei join point che contiene. Rappresenta a tutti gli effetti il meccanismo con cui i pointcut possono essere definiti ed espressi. L'AspectJ fornisce un certo numero di pointcut designator primitivi e permette di comporli per ottenerne dei nuovi. Come esempio, consideriamo diagramma UML in Figura22, che rappresenta un esempio di un semplice editor di figure.
Il pointcut designator può essere pensato come strumento di "matching" a runtime di un cero insieme di join point individuati all'interno dell'applicazione. Per esempio :

```
call (void Point.setX(int)) || call(void Point.setY(int))
```

identifica ogni chiamata al metodo `setX` o `setY` definiti in `Point` e, sinteticamente, questo codice consiste in due pointcut designator *`call`* composti attraverso un operatore "*`or`*". La sintassi di `call` (come di tutti gli altri tipi primitivi di pointcut designator) è basata sulla stessa sintassi dei metodi Java :

```
call (retun_type object_type.method_name(arg_type, …))
```

Il programmatore può comporre diversi tipi di pointcut designator (che possono far riferimento a join point di diverse classi) ed etichettarli attraverso un nome. In altre parole è possibile gestire il crosscutting tra le classi.

Ad esempio, il seguente codice definisce un pointcut chiamato `move` che gestisce ogni chiamata a metodi che possono contribuire al movimento dell'elemento Figure:

```
poincut move () :
      call(void FigureEllement.moveBy(int, int)) ||
      call(void Point.setX(int))                 ||
      call(void Point.setY(int))                 ||
      call(void Line.setP1(Point))               ||
      call(void Line.setP2(Point)) ;
```

Questo pointcut designator è basato su un'esplicita enumerazione dei join point e viene chiamato *name-based* poiché strettamente vincolato al nome degli elementi che contiene.

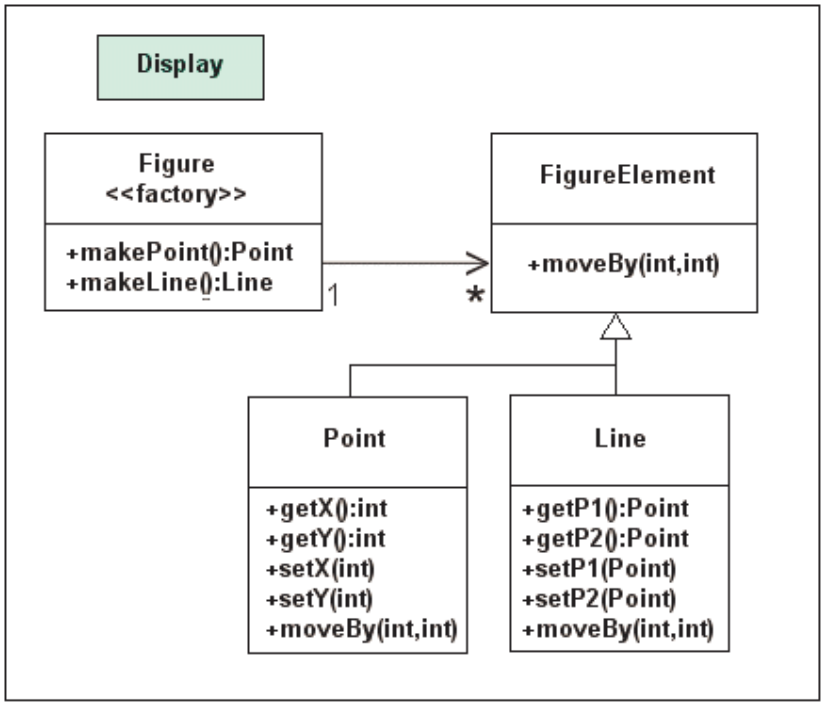

Figura 22 - UML per il semplice editor di figure

L'AspectJ permette anche di definire in modo semplice quelli che vengono definiti crosscutting *property-based*, con i quali è possibile esprimere i pointcut in termini di proprietà dei metodi coinvolti più che attraverso il loro nome. Questa caratteristica è ottenuta usando speciali caratteri (wild cards) in all'interno della segnature del metodo.

Ad esempio il codice :

```
call (* Point.*(…))
```

gestisce le chiamate a qualsiasi metodo definito nella classe `Point` (quindi `getX(), getY(), setX(int), setY(int)`); ancora, il pointcut :

```
call (* Point.get*())
```

fa riferimento a qualunque metodo definito in `Point` il cui identificatore inizia con `get` e non accetta parametri.

Gli *advice* sono il meccanismo con cui è possibile dichiarare il codice che deve essere eseguito all'occorrenza di certi join point. In pratica, dopo che il programmatore ha definito i punti in cui nell'applicazione deve essere applicato un particolare concern, quest'ultimo può essere definito ed implementato attraverso l'advice. L'AspectJ ne supporta diversi tipi tra cui *before*, *after* e *around*. L'advice before viene invocato nel momento in cui è raggiunto un join point o, in altre parole, appena prima che il metodo identificato dal join point venga eseguito. L'after advice, al contrario, viene invocato nel momento in cui il controllo ritorna attraverso il join point, ossia appena dopo che il metodo è stato eseguito ma prima che il controllo venga trasferito al chiamante. Gli advice around vengono eseguiti dopo il before e prima dell'after. Qualunque sia l'advice, il suo corpo può contenere qualsiasi codice che possa essere inserito in un consueto metodo Java.
L'esempio che segue rappresenta un semplice advice che stampa un messaggio se un elemento Figura è stato mosso:

```
after() : moves() {
 <codice di stampa del messaggio>
}
```

Un *aspect* è l'unità modulare che l'AspectJ fornisce per esprimere i crosscutting concern.
Gli aspetti sono definiti attraverso un set di dichiarazioni in un modo del tutto analogo alle classi in Java e possono includere dichiarazioni di pointcut, advice oltre ad ogni altro tipo di dichiarazione permesso dal linguaggio. Attraverso gli aspect possiamo implementare in modo del tutto modulare gli aspetti del sistema.

Ad esempio, se volessimo costruire l'aspetto di monitoraggio per il movimento delle figure potremmo implementare un *aspect* di questo tipo :

```
aspect MoveTracking {
      pointcut moves():
            call(void FigureElement.slide(int, int))  ||
            call(void Line.setP1(Point))              ||
            call(void Line.setP2(Point))              ||
            call(void Point.setX(int))                ||
            call(void Point.setY(int));
      before() : moves() {
            <codice di gestione>  }
      after(): moves() {
            <codice di gestione>  }
 }
```

Il meccanismo di composizione dell'AspectJ può essere utilizzato in modo efficace per la separazione del codice funzionale dall'aspetto di sincronizzazione presentati in precedenza. Lo stack verrà implementato come in Figura18, cioè attraverso una classe che ne gestisce la sola competenza funzionale mentre l'aspetto di sincronizzazione come nel listato di Figura23.

```
aspect SyncStack {

   pointcut lockpush():
       reception(public void push(Object o));
   pointcut lockpop():
       reception(public Object pop());

   advice  void MyStack.push(Object o) {
       before() : lockpush() {
          while(stack.top==stack.StackSize-1) try{
            wait();
          } catch (InterruptedException e){
          e.printStackTrace();
          }
       }
       after() : lockpush() {
           if (stack.top == 0) notifyAll();
       }
   }
   advice  void MyStack.pop() {
       before() : lockpop() {
           while(stack.top==-1) try{
             wait();
           } catch (InterruptedException e){
           e.printStackTrace();
           }
       }
       after() : lockpop() {
          if (stack.top==stack.StackSize-2) notifyAll();
       }
   }
}
```

Figura 23 - Aspetto di sincronizzazione

Composition Filters

L'approccio Composition Filters [Bergma94] nasce dagli studi riguardo ai problemi di inheritance anomalies nei sistemi concorrenti object-oriented, problemi che abbiamo già accennato nel capitolo precedente.
Si basa su un meccanismo a *filtri di messaggio* (message filters) attraverso i quali devono passare i messaggi scambiati tra gli oggetti. L'oggetto nel modello CF consiste in un *kernel* d'oggetto e in un *layer d'interfaccia*, come mostrato in Figura24. Il kernel può essere pensato come un nomale modello d'oggetto dei convenzionali linguaggi di programmazione object oriented, come Java o C++. Il layer d'interfaccia contiene un numero arbitrario di filtri di input ed output. I messaggi in arrivo passano attraverso i filtri di input e i messaggi in uscita attraverso quelli di output.

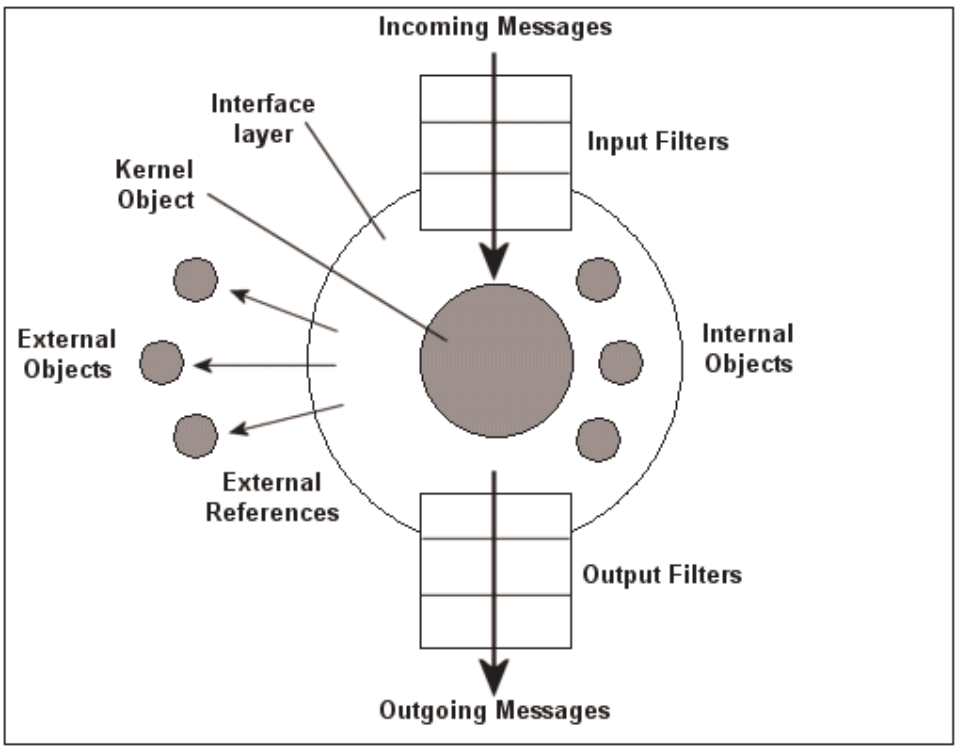

Figura 24 - Elementi di un oggetto nel modello CF

I filtri possono *modificare* il messaggio, ad esempio cambiandone il nome o modificare l'oggetto di destinazione. Più precisamente i filtri possono essere usati per redirigere i messaggi (eventualmente modificati) verso due tipi di oggetti: verso *oggetti interni*, incapsulati nel layer d'interfaccia, o verso *oggetti esterni*, riferiti dal layer d'interfaccia. I filtri possono inoltre controllare, scartare o bufferizzare messaggi, quindi l'azione è dettata unicamente dal tipo di filtro usato.

Sono presenti un certo numero di filtri predefiniti, per esempio i delegation filters (per delegare i messaggi), i wait filters (per bufferizzare), gli error filters (per le throwing exceptions) ed altri tipi di filtro possono essere creati ed aggiunti. La modifica di un messaggio da parte di un filtro può dipendere dal tipo di messaggio oltre che da alcune condizioni di stato (*state conditions*) definite attorno al kernel dell'oggetto.
I filtri di messaggio sono una potente tecnica che permette di implementare in modo ben localizzato vincoli di sincronizzazione [Bergma94], vincoli real-time [Aksit+94], transazioni atomiche [Aksit+92], controllo d'errori attraverso precondizioni [Bergma94] e molti altri importanti aspetti.

La caratteristica di redirecting dei messaggi può inoltre essere usata per implementare in modo efficiente delegazione ed *ereditarietà dinamica*. La delegazione consiste nel redirigere alcuni messaggi ricevuti dall'oggetto delegante verso un altro oggetto, il delegato, mantenendone però il riferimento. In questo caso è necessario assicurarsi che, quando l'oggetto delegato fa uso della keyword *self*, si riferisca all'oggetto delegante e non a se stesso. In questo modo i metodi degli oggetti delegati possono essere scritti come se fossero metodi del delegante per cui gli oggetti delegati possono considerarsi a tutti gli effetti come un'estensione, in modo simile alla relazione tra una classe e la sua superclasse. Nel modello Composition Filters, delegare significa redirigere i messaggi verso oggetti esterni, mentre ereditare significa redirigere i messaggi verso gli oggetti interni .

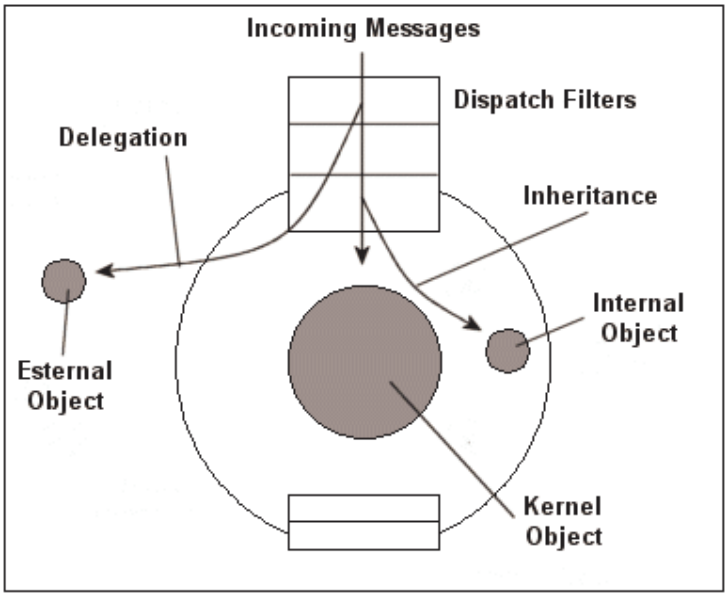

Figura 25 - Delegation e Inheritance nel modello CF

Un filtro può inoltre delegare un certo messaggio a *differenti* oggetti interni basandosi su alcune condizioni di stato. Questo significa che la superclasse di un oggetto può cambiare a seconda del suo stato, permettendo quella che viene definita ereditarietà dinamica.

Adaptive Programming / Demeter

Una generica operazione in un programma object-oriented coinvolge spesso differenti classi che collaborano. Ci sono normalmente due modi per implementare queste operazioni: mettere l'intera operazione in un metodo di una delle classi oppure dividerla e distribuirla all'interno di metodi di ognuna delle classi coinvolte.
Lo svantaggio di quest'ultimo approccio è che all'interno di ognuno dei metodi è necessario gestire troppe informazioni relative alla *struttura delle classi* (come le relazioni di tipo is-a e has-a). Una diretta conseguenza è la difficoltà di adattamento dei cambiamenti nella struttura delle classi e inoltre, la distribuzione delle operazioni su diverse classi, rende complicato l'adattamento nel caso in cui l'operazione stessa cambi.

Per risolvere questo conflitto, il progetto Demeter ha introdotto la nozione di *adaptive method*, anche conosciuto come propagation pattern [Lieber96]. Un metodo adattivo non solo incapsula il comportamento di una operazione, ma opera un'astrazione sopra la struttura delle classi coinvolte. Per ottenere questo, il comportamento è espresso attraverso una descrizione ad alto livello che indica :

- come raggiungere i "partecipanti" dell'operazione, cioè viene definita una strategia di attraversamento (*traversal strategy*)
- cosa fare quando i partecipanti sono stati raggiunti (*adaptive visitors*)

Il vantaggio più grosso è che, grazie alle direttive ad alto livello, in entrambi i casi viene indicato solo un minimo set di classi che partecipano all'operazione e le informazioni riguardo le connessioni tra queste classi viene completamente astratto. Questo meccanismo fa uso delle tecniche di reflection di Java in modo che la struttura delle classi possano essere accedute a runtime.

Il Demeter/Java, in particolare la libreria DJ, è un package Java che supporta questo stile di programmazione, fornendo anche alcuni strumenti per interpretare le strategie di attraversamento e gli adaptive visitor.

La Figura26 mostra un semplice metodo adattivo scritto in Java usando la libreria DJ. Lo scopo del codice è quello di sommare i valori di tutti gli oggetti Salary legati, attraverso una relazione has-a, all'oggetto Company.

```
import edu.neu.css.demeter.dj.ClassGraph;
import edu.neu.css.demeter.dj.Visitor;

class Company {
   static ClassGraph cg = new ClassGraph(); //class structure

   Double sumSalaries() {
      String s = "from Company to Salary";  //traversal strategy
      Visitor v = new Visitor() {
           private double sum;
           public void start() { sum = 0.0; };
           public void before (Salary host) {
                      Sum+=host.getValue(); }
           public Object getReturnValue() {
                      Return new Double (sum); }
      };
      return (Double) cg.traverse (this, s, v)
   }
   // rest of Company definition
 }
```

Figura 26 - Un semplice adaptive methods

Il funzionamento è il seguente:
la variabile statica `cg` è un oggetto `ClassGraph` che rappresenta la struttura delle classi del programma. Il `ClassGraph` è una semplice forma di diagramma delle classi UML che descrive le relazioni has-a e is-a tra le classi. La struttura della classe è gestita in `ClassGraph` attraverso la tecnica di reflection. Questa struttura è usata dal metodo traverse che ne interpreta la strategia di attraversamento. Il metodo inizia da un dato oggetto (in questo caso `this` ma potrebbe iniziare da qualsiasi altro oggetto) e attraversa uno specifico percorso ("`from Company to Salary`") eseguendo ogni metodo `visitor` applicabile lungo la strada.
In questo caso attraverso `start` viene inizializzata `sum`, poi, ad ogni passaggio per gli oggetti `Salary` viene effettuata la somma e, prima di finire, viene costruito il valore di ritorno.

La separazione della strategia di attraversamento (che specifica dove andare), l'adaptive visitor (cosa fare) e il class graph (il contesto in cui valutare) permettno a questo

meccanismo di essere riusato senza cambiamenti in un set di programmi. I dettagli, come il numero di classi tra Company e Salary, non sono importanti e risultano nascosti grazie al meccanismo di attraversamento: potrebbe infatti trattarsi di una applicazione con una grossa organizzazione di strutture (divisioni, dipartimenti, work group ecc..) con tante classi, oppure un'organizzazione con una piccola struttura.

Il termine Adaptive Programming fu introdotto alcuni anni fa da Lieberher [Lieber96] per consentire la gestione di un certo insieme di aspetti riguardanti la struttura delle classi. Successivamente fu esteso alla gestione di aspetti di sincronizzazione e remote invocation [Lieber98]. In accordo ad una definizione più recente, l'A.P. è un caso "particolare" dell'Aspect Oriented Programming dove alcuni blocchi (come i componenti e gli oggetti) sono espressi in termini di grafi ed altri (i crosscutting concern) fanno riferimento al grafo attraverso una strategia di attraversamento. Si può dire che l'A.P. è un sott'insieme dell'AOP in termini di grafi e strategia di attraversamento .

Bibliografia

[Aksit+92] M. Aksit, L. Bergmans, and S.Vural. “An Object-Oriented Language-Database Integration Model: The Composition-Filters Approch”. In Proceedings of the ECOOP’92 Conference, LNCS 615, Springer-Verag, Berlin and Heidelberg, Germany, 1992.

[Aksit+94] M. Aksit, J. Bosch, W. van der Sterren, L. Bergmans. Real-time specification inheritance anomalies and real-time filter; In Proc. ECOOP’ 94, M. Tokoro and R. Pareschi Eds, LNCS 821, Springer-Verlag.

[Beck89] Kent Beck, "A Laboratory For Teaching Object-Oriented Thinking", Apple Computer, Inc., Ward Cunningham, Wyatt Software Services, Inc., From the OOPSLA'89 Conference Proceedings October 1-6, 1989, and the special issue of SIGPLAN Notices Volume 24, Number 10, October 1989.

[Bergma94] L.M.J. Bergmanas. “Composing Cuncurrent Objects – Applying Composition Filters for the Development and Reuse of Councurrent Object-Oriented Programs. Ph.D.Thesis, Department of Computer Science, University of Twente, The Netherlands, 1994.

[Booch+99] G. Booch, J.Rumbaugh, and I. Jacobson.The “Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, 1999

[Clarke+99] S. Clarke, W. Harrison, H. Ossher, P. Tarr. "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code." Proc. OOPSLA'99.

[Consta99] Constantinos A. Constantinides, Atef Bader, Tzilla Elrad. "A framework to address a two dimensional composition of concerns." OOPSLA99 First Workshop on MultiDimensional Separation of Concerns in Object-Oriented Systems. October, 1999.

[Czarn+00] Czarnecki, K., and Eisenecker, U.W. “Generative Programming: Methods, Tools, and Applications”. Boston: Addison Wesley, 2000.

[Constan02] Constantinos A. Constantinides, Therapon Skotiniotis. "Reasoning about a classification of crosscutting concerns in object-oriented systems", Second Workshop on Aspect-Oriented Software Development, Bonn (Germany), February 21-22, 2002.

[Dijkst70] E. Dijkstra. Structured Programming. In the Software Engineering Techniques, J.Buxton and B. Randell, (Eds.), NATO Scientific Affair Division, Brussels, Belgium, 1979, pp. 84-87

[Filman00] Filman R., Friedman D. P., "Aspect Oriented Programming in quantification and obliviousness", Proceedings of OOPSLA 2000 Workshop on AdvancedSeparation Of Concerns in Object-Oriented Systems. Minneapolis, Minnesota. USA. October 16, 2000.

[Findl+01] Findler R. B., Felleisen M, "Contracts Soundness for Object-Oriented Languages", Proceedings of Object-Oriented Programming, Systems, languages, and Applications (OOPSLA 2001)

[Flatt+98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. "Classes and mixins". In Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'98), pages 171-183, San Diego, CA, January 1998. ACM.

[Fowle+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, don Roberts, "Refactoring : Improving the Design of Existing Code", Ed. Addison-Wesley Object Technology Series, 1999.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns: Element of Reusable Object-Oriented Software", Addison-Wesley, 1995

[Hanen02] Stefan Hanenberg and Rainer Unland ,"A Proposal For Classifying Tangled Code", Institute for Computer Science University, 2002

[Highl+99] T.J. Highley, Michael Lack, and Perry Myers. "Aspect-oriented programming: a critical analysis of a new programming paradigm", 1999.

[IWQoS02] International WorkShop on Quality of Sevice (IWQoS 2002), Miami Beach, May 15-17, 2002. http://www.cs.virginia.edu/~iwqos/

[Kiczale01] Kiczales G, et al., "An overview of AspectJ", In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Springer, 2001.

[Kiczales97] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, Gail Murphy. "Open Implementation Design Guidelines", in Proceedings of the 19th International Conference on Software Engineering (ICSE), Boston, USA. ACM Press. May 1997.

[Kiczal+98] Bill Griswold, Erik Hilsdale, Jim Hugunin, Wes Isberg, Gregor Kiczales, Mik Kersten , "Aspect oriented programming with AspectJ", Copyright Xerox Corporation, 1998-2001.

[Klaer+00] Klaeren G., Pulvermuller E., Rashid A., and Speck A., "Aspect Composition applying the Design by Contract Principle", Proceedings of 2nd International Symposium of Generative and Component-based Software Engineering (GCSE 2000), Springer, LNCS 2177, pp. 57-69.

[Liebe+88] K.J. Lieberherr, I. Holland, and A. J. Riel. "Object-orieted programming: an objective sense of style". In Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88), ACM SIGPLAN Notices, vol.23,no.11,1988,pp. 323-334

[Lieber96] Lieberher K.J., "Adaptive Object-Oriented Software: The Demeter Methods with Propagation Patterns", PWS Publishing Company, Boston, 1996.

[Lieber98] Lieberher K.J., "Connection between Demeter/Adaptive Programming and Aspect Oriented Programming".

[Lodew00] Lodewijk M.J. Bergmans, Mehmet Aksit , "Aspects Crosscutting In Layered Middleware Systems", TRESE Group – Cernter for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands, 2000.

[Matsu+93] Matsuoka S., and Yonezawa A., "Analysis of Inheritance Anomaly in Object-Oriented Cuncurrent Programming Language", ed. Agha G., Wegner P., and Yonezawa A., Research Directions in Cuncurrent Object-Oriented Programming, pp. 107-150, The MIT Press, 1993

[Mccall77] McCall J. A., Richards P. G., and Walters G. F. "Factors in Software Quality", Vols.I,II, and III (NTIS AD/A-049 014/015/055), Springfield: NTIS, 1977.

[Meyer92] Mayer B., "Applying Design by Contract", IEEE Computer , pp. 40-52, October 1992.

[Meyer97] Bertrand Meyer, "Object-Oriented Software Construction, Second Edition", Prentice Hall Professional Technical Reference, 1997.

[Osshe+00] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach." In Proceedings of the Symposium on Software Architectures and Component Technology, Kluwer, 2000.

[Osshe+99] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns using Hyperspaces." IBM Research Report 21452, April, 1999.

[Parnas72] Parnas. D. L. "On the criteria to be used in decomposing systems into modules", In Communications of the ACM, Vol. 15. No. 12. December 1972, pp. 1053-1058.

[Takas+92] Kazunori Takashio and Mario Tokoro. DROL: An Object-Oriented Programming Language for Distributing Real_time System. In Andreas Peapcke, editor, Object-Oriented Programming System, Language and Applications Conference (OOPSLA), pagine 276-294, Vancouver, Canada, Ottobre 1992, ACM Press.

[Tarr+99] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proceedings of the International Conference on Software Engineering (ICSE'99), 1999.

Riferimenti Web

Aspect Oriented Programming AspectJ	http://www.aspectj.org/
Subject Oriented Programming Hyper/J	http://www.research.ibm.com/sop/
Adaptive Programming Demeter	http://www.ccs.neu.edu/research/demeter/
Composition Filters	http://trese.cs.utwente.nl/composition_filters/
Aspect Oriented Software Development	http://www.aosd.net/