

Diagrammi, layout e gestione della complessità

di Andrea Baruzzo, Carlo Pescio

Perché *progettiamo* il software, anziché limitarci semplicemente a *costruirlo*? Se l'analisi serve per capire quale sistema costruire e lo sviluppo serve a realizzare tale sistema, quali obiettivi riversiamo sulla progettazione? La necessità di progettare deriva, oggi più che mai, dalla crescente complessità dei sistemi software. Sistemi spesso caratterizzati da vincoli di compatibilità rispetto a soluzioni preesistenti, o con richieste di prestazioni real-time, o influenzati da problematiche di sicurezza sempre più significative. Aggiungiamo la classica instabilità dei requisiti, il desiderio di mantenere comunque bassi i costi di sviluppo, ed otteniamo il cocktail micidiale che caratterizza tanti progetti, soprattutto nelle fasi di manutenzione ed estensione.

Progettare diventa allora importante perché costituisce un momento, nell'intero ciclo di vita di un progetto, in cui il focus principale è capire *come* costruire un sistema limitando i costi in una più ampia prospettiva di evoluzione del software. Intesa in questo modo, la progettazione diventa uno strumento per gestire la complessità del sistema. In pratica, un certo livello di complessità viene ereditato direttamente dal *problema* stesso e, come tale, non può essere facilmente abbattuto. Altri contributi derivano da aspetti tecnici legati alla *soluzione*, come le tecnologie utilizzate o la modularità del programma. Proprio nei confronti di tali aspetti il progettista può esercitare un maggiore controllo. Sulla base di questa intuizione, proviamo ad attribuire ai diagrammi un ruolo diverso da quello tipico di documentazione: il ruolo di strumento di gestione della complessità, che analizzeremo con particolare riferimento agli aspetti di layout.

Complessità essenziale e complessità accidentale

Sul finire degli anni 80, Fred Brooks (noto come "il padre di IBM OS/360") ha pubblicato un saggio di grande impatto [Brooks, 1987], ripreso in seguito da molti autori. Tra i suoi contributi, troviamo la fondamentale distinzione tra complessità essenziale e complessità accidentale. La *complessità*

essenziale nasce dal dominio ed è, quindi, da considerarsi una componente intrinseca al problema. Essa è caratterizzata dalla struttura del "mondo" che si vuole rappresentare in un modello. Dalla complessità essenziale, infatti, deriva la difficoltà nel comprendere e descrivere gli aspetti multiformi di un problema, o di comunicare ai membri del team di progetto un particolare aspetto del problema.

La *complessità accidentale*, per contro, nasce con il tentativo di definire e formalizzare il problema. Essa è causata, in fase di analisi, da vari fattori [Nguyen&Swatman, 2000], tra cui la comprensione limitata del dominio da parte degli analisti, la mancata identificazione di astrazioni chiave, la completezza ed il livello di precisione dei requisiti e, talvolta, anche dalla confusione tra problema e soluzione [Pescio, 1997]. La complessità accidentale è inizialmente il risultato di una scarsa corrispondenza tra la struttura del modello e quella del "mondo" da rappresentare, ossia della parte di dominio attinente al problema da risolvere.

Con il procedere dall'analisi al design, queste due forme di complessità sono destinate ad aumentare. In parte, la complessità essenziale cresce come conseguenza del maggiore approfondimento sugli aspetti essenziali del problema; in parte, tende a crescere con l'introduzione dei requisiti non funzionali. La complessità accidentale, invece, cresce spesso a causa della sovraingegnerizzazione di alcune parti del modello, o quando la piattaforma hardware/software (includendo sia l'hardware fisico, sia lo strato software composto da sistema operativo e linguaggio di programmazione) impone vincoli innaturali rispetto alla risoluzione del problema. In conclusione, la complessità essenziale è quindi non rimovibile così come, entro certi margini, avremo sempre una parte di complessità accidentale. È quindi fondamentale capire se e in quale modo sia possibile gestire tutta questa complessità. Una possibilità consiste nell'identificare delle "componenti ortogonali". L'esempio più noto è l'asse procedurale/strutturale, discusso nel paragrafo successivo.

Complessità procedurale e complessità strutturale

Nonostante le forme di complessità essenziale e accidentale permeino tutto il ciclo di vita del software, esistono altre due forme di complessità che vengono introdotte tipicamente durante il design e la codifica: la complessità procedurale e quella strutturale. Queste due forme di complessità, se mal gestite, portano ad un aumento di complessità accidentale. La *complessità procedurale* è quella che incontriamo quando l'unico meccanismo di partizionamento di un sistema è costituito dalla decomposizione funzionale, con una forza che porta a decomporre (riuso, facilità di comprensione) ed una che porta a tenere insieme (pigrizia, inerzia nell'apportare modifiche alla struttura del sistema, vicinanza delle funzioni alle variabili locali). Un tipico elemento rivelatore è la presenza continua di funzioni relativamente lunghe e non sempre semplici da capire¹. Dalla complessità procedurale, quindi, deriva la difficoltà di comprensione e, in parte, d'invocazione e di riuso delle funzioni all'interno di un programma. È esperienza relativamente comune che, avendo a disposizione altri meccanismi di gestione della complessità (ovvero potendone spostare un po' sul lato strutturale), una parte di essa semplicemente evapori, ovviamente perché non essenziale.

La *complessità strutturale*, infine, emerge quando si cerca di comprendere gli aspetti di un sistema partendo da una descrizione della sua struttura interna. Nel paradigma action-oriented, favorendo la decomposizione funzionale, si favorisce anche un meccanismo di controllo centralizzato nel quale gran parte del lavoro viene svolto da un numero limitato di classi, dando origine al problema delle "god class" [Riel, 1996][Brown et al, 1998]. Il paradigma object-oriented, invece, inverte il rapporto tra procedure e struttura. Evitando di rendere dipendente la struttura dalle funzioni, si riduce il livello di accoppiamento tra dati ed algoritmi. L'eventuale propagazione della modifica al di fuori dei confini di una singola classe viene esplicitamente *documentata* (e quindi gestita, progettata) tramite le dipendenze tra le classi. Anche i sistemi strutturalmente complessi non sono esenti da problemi. Un fattore piuttosto noto è rappresentato dai

¹ Anche funzioni inizialmente semplici e con interfacce "agili" possono diventare con il tempo proceduralmente complesse e con interfacce ingombranti, ad esempio per ottenere riuso e flessibilità pensando in modo esclusivamente procedurale.

cosiddetti "delocalized plan" [Letovsky&Soloway, 1986], ovvero da operazioni concettualmente vicine realizzate in moduli fisicamente separati. Un esempio tipico di questo fenomeno è il design pattern Strategy, che definisce una famiglia di algoritmi, incapsulando ciascuna variante in una classe e riunendo poi le diverse classi in una gerarchia al fine di rendere ogni algoritmo interscambiabile [GoF, 1995]. Per chi non è ancora abituato a gestire questo tipo di complessità, la funzione "gigante" appare quasi più semplice da leggere, anche se può facilmente degenerare durante la manutenzione. Dalla complessità strutturale deriva quindi la difficoltà di comprendere gli aspetti dinamici di un sistema software e, in parte, quella di estensione. Notiamo che l'operazione di estensione di per sé è risulta spesso relativamente semplice: ciò che la rende "complicata" è la difficoltà preliminare nell'identificare il punto esatto da modificare nel codice.

Ragionare sui diagrammi

Un progettista esperto tende spesso ad *aumentare* la complessità strutturale del design per *ridurre* la complessità procedurale del codice. Non si tratta però di un semplice "spostamento" di complessità da un lato ad un altro, per una ragione tutto sommato semplice, anche se non necessariamente evidente: *complessità* non equivale necessariamente a *carico cognitivo*. In altre parole, l'essere umano è in grado di svolgere compiti anche di grande difficoltà con un carico cognitivo relativamente ridotto, quando gli *strumenti* che utilizza per svolgere tali compiti sono allineati con le sue capacità più sviluppate. Un esempio classico è la guida di un veicolo che, esaminata a livello di dettaglio, rivela una complessità notevole, ma che viene svolta senza un eccessivo carico cognitivo dalla maggior parte delle persone. Perché, dunque, la complessità strutturale può essere preferibile a quella procedurale? Perché la complessità procedurale viene gestita con criteri logico/linguistici, mentre la complessità strutturale può essere gestita a livello percettivo/visivo. In questo senso, i diagrammi ricoprono il ruolo essenziale di *strumento di pensiero* [Pescio, 2000], permettendoci di svolgere alcuni ragionamenti in modo semi-conscio, senza elevato carico cognitivo. Parafrasando Donald Norman [Norman, 1993] i diagrammi (sfruttati opportunamente) sono una delle "cose che ci fanno intelligenti", ovvero un sussidio che potenzia le nostre capacità cognitive. Non a

caso, proprio dal testo di Norman è tratto l'esempio di un gioco risolto sia a livello logico/matematico che visivo/spaziale, utilizzato in **[Pescio, 2001]** per esemplificare il concetto di diagramma come strumento *per pensare*, capace di sfruttare a pieno l'intelligenza visiva intrinseca degli esseri umani **[Robertson, 2002]**.

Ovviamente, un diagramma potenzia le nostre capacità cognitive quando vi rappresentiamo ciò che è realmente importante, senza introdurre dettagli irrilevanti che ostacolano il ragionamento diagrammatico. Non si tratta, tuttavia, di una semplice azione di "sfolgimento": dobbiamo prima di tutto *evidenziare* ciò che è realmente importante, ed in seconda battuta evitare di *nascondere* in un mare di dettagli minori. Ancora più importante è evitare di suggerire, omettendo elementi essenziali, intuizioni errate, relative (ad es.) alla dinamica del sistema, od alle sue proprietà non funzionali (estendibilità, riusabilità, ecc). Da un lato, quindi, omettere dettagli poco significativi diventa essenziale per creare un diagramma facilmente comprensibile (o, tout-court, un diagramma *utile*). Dall'altro, dobbiamo esplicitare ed enfatizzare gli *elementi portanti* del sistema. È tuttavia sin troppo semplice indicare negli elementi secondari (attributi con i loro tipi, parametri con i loro tipi, funzioni private, ecc) i dettagli da scartare, e negli elementi primari (classi ed associazioni) i fattori da porre in evidenza. In realtà, un elemento può risultare *nascosto* anche se è presente nel diagramma: l'intelligenza visiva è in larga misura percettiva: se dobbiamo *studiare* il diagramma per poterlo utilizzare, abbiamo già compromesso una buona parte della sua funzione. La semplice disposizione grafica di alcuni elementi può rendere immediati (ovvero svolti a livello percettivo) alcuni semplici ragionamenti che tuttavia, in mancanza di un supporto diretto, andrebbero comunque ad aumentare il carico cognitivo del progettista.

Possiamo riformulare quanto sopra in due massime, che guideranno non solo il resto dell'articolo, ma anche una futura puntata sull'utilizzo dei "visual clues" (tra cui il ruolo dei colori nel diagramma delle classi). Un buon diagramma deve:

- Rendere immediato ciò che è importante;
- Rendere esplicito ciò che è essenziale.

Come vedremo, non sempre i diagrammi cui siamo abituati rispettano queste due importanti proprietà, ed osserveremo come questo possa portare a sorprese poco piacevoli.

Layout e complessità: tra ordine e caos

Volendo utilizzare UML principalmente come strumento di sviluppo, anziché di pura documentazione, cerchiamo ora di capire come adattare il layout di un diagramma per meglio gestire la complessità. Restrungendo l'analisi ai diagrammi delle classi, vogliamo capire quali siano gli aspetti importanti che un buon diagramma deve rendere immediati. Poiché il concetto di dipendenza permea tutto il metodo object-oriented in termini di gestione della complessità, uno di questi aspetti fondamentali è costituito dalle dipendenze circolari **[Pescio, 1999]**. Non sempre tali tipi di dipendenze sono evitabili: a volte esprimono semplicemente un aspetto essenziale del dominio. Altre volte, però, costituiscono un problema di design che va opportunamente gestito dal progettista. Il primo passo di questa gestione consiste nell'esplicitare tali dipendenze nel diagramma. L'efficacia del diagramma, tuttavia, risulta pesantemente ridotta se esse sono poi nascoste dal layout finale. Considerando il diagramma di Figura 1, riusciamo ad individuare a colpo d'occhio tutte le dipendenze circolari tra le classi?

Indubbiamente ci viene richiesto un certo sforzo, senza considerare il fatto che il diagramma originale, tratto da **[Yacoub et al, 2000]**, era arricchito da attributi, metodi e note associate alle classi non riportati in figura. Se proviamo ora a confrontare tale diagramma con quello proposto in Figura 2, possiamo osservare come, semplicemente orientando tutte le dipendenze verso l'alto, otteniamo un layout più ordinato. La linea guida "tutte le dipendenze verso l'alto", discussa ampiamente nel prossimo paragrafo, si ispira ad uno dei cardini della programmazione ad oggetti: ogni elemento deve dipendere solamente da elementi più astratti. Essa rappresenta anche una regola pratica per evidenziare rapidamente le dipendenze circolari. Se non riusciamo a disegnare tutte le dipendenze verso l'alto, allora almeno una di esse è circolare.

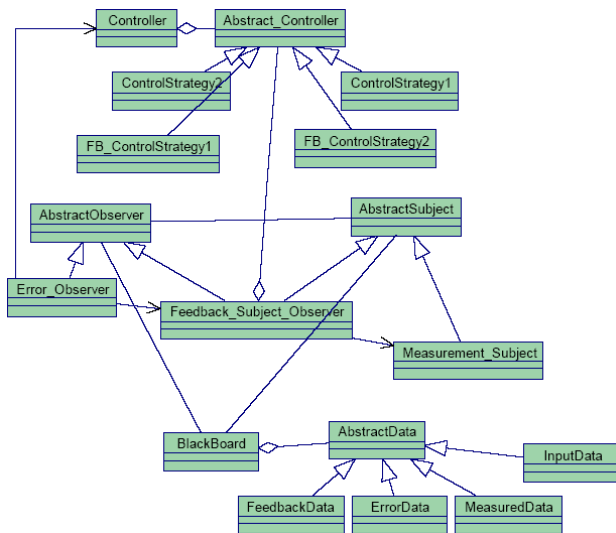


Figura 1 – Esempio di diagramma che nasconde aspetti importanti legati alle dipendenze

Chi legge la Figura 1, con un po' di fortuna può anche individuare la dipendenza circolare esistente tra *AbstractSubject*, *AbstractObserver* e *Blackboard*, ma solo osservando la Figura 2 è possibile capire immediatamente, cioè senza studiare il diagramma, che essa è anche l'unica dipendenza circolare presente. Lo scopo di questa linea guida non è comunque limitato ad evidenziare le dipendenze circolari: mettere le dipendenze verso l'alto aiuta anche a capire se stiamo sbagliando a classificare come generale qualcosa che è invece concreto [Martin, 1996], ovvero se siamo "costretti" a mettere in alto qualcosa che intuitivamente dovrebbe stare in basso (o viceversa). Se ci "permettiamo" di mettere le frecce in ogni direzione, perdiamo questa possibilità di verifica a basso carico cognitivo.

L'esempio proposto evidenzia come il layout svolga un ruolo importante nella modellazione: la riorganizzazione degli elementi può rendere un diagramma più facile da leggere, sia mentre lo si crea, sia mentre lo si rielabora o lo si trasferisce ad altri, facendolo diventare uno strumento per focalizzare l'attenzione sugli aspetti importanti del design che il progettista vuole evidenziare. In altre parole, uno strumento per comunicare e ragionare.

In generale chiameremo *stratificato* un diagramma che presenta tutte le dipendenze verso l'alto. La stratificazione di un diagramma esprime anche una condizione necessaria affinché un'astrazione, ossia un concetto del dominio o del sistema potenzialmente riusabile e potenzialmente stabile, non dipenda dai dettagli. Le dipendenze circolari riducono la riusabilità in

quanto l'unità di riuso, in questo caso, diventa l'intero ciclo e non la singola classe. Un diagramma stratificato, quindi, assicura il riuso degli elementi più astratti: non a caso, la stratificazione costituisce anche la prima regola del Systematic Object Oriented Design [Pescio, 1999]. Rappresentare *tutte* le dipendenze verso l'alto significa trasferire la regola dal dominio concettuale a quello grafico, fornendo così un importante ausilio visivo alla fase di progettazione.

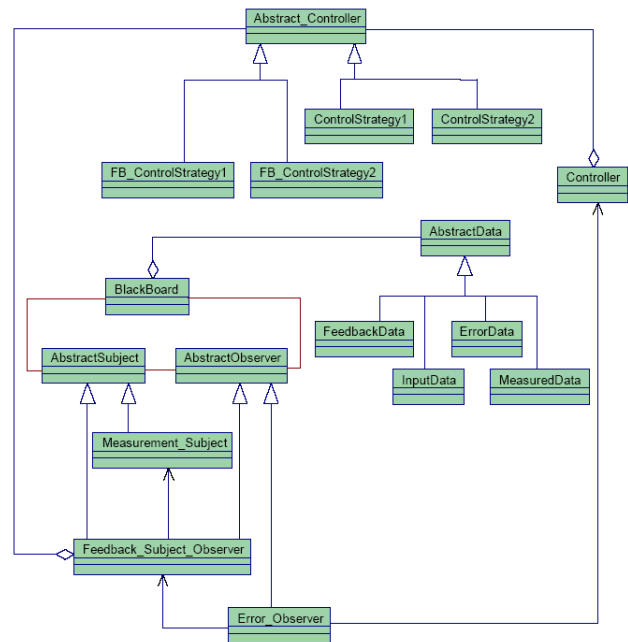


Figura 2 – Esempio di diagramma che rendere immediate le dipendenze circolari

Dopo tutta questa enfasi sulle dipendenze verso l'alto, è naturale chiedersi come si ponga la tradizione informatica nei confronti di questa linea guida. In effetti, nei diversi anni in cui si è fatto uso di diagrammi delle classi (non necessariamente UML), l'idea di "imporre" un layout con tutte le dipendenze orientate esclusivamente verso l'alto non è mai stata particolarmente caldeggiata. Il suo utilizzo nella modellazione di gerarchie di ereditarietà è invece decisamente frequente, ed in larga misura potrebbe derivare da una tradizione preesistente: storicamente, le tassonomie di classificazione (si pensi ad es. alla biologia) vengono rappresentate come alberi, la cui radice rappresenta la categoria più generale, e scendendo si trovano categorie via via più fini.

A guardare bene, esistono almeno due situazioni in cui la tradizione va in direzioni opposte:

- sistemi "stratificati" vengono spesso rappresentati con le dipendenze orientate verso il basso.
- le associazioni del tipo "tutto-parti" vengono spesso rappresentate con il "tutto" al centro e le parti distribuite sui quattro lati, spesso in forma simmetrica ("a ragno"), o con il tutto in alto e le parti in basso ("a medusa").

Come vedremo, in entrambi i casi la scelta "tradizionale" mostra alcuni limiti, che fanno comunque pendere la bilancia a favore dell'orientamento verso l'alto.

Dipendenze nei design stratificati

Organizzare un sistema in strati, o layer, è una pratica relativamente comune, tanto da essere stata da tempo classificata come pattern [Buschmann et al., 1996]. In pratica (pensiamo al classico stack ISO/OSI o TCP/IP) un sistema stratificato viene quasi sempre rappresentato con le dipendenze orientate "verso il basso", guidati forse dall'idea che un livello "si appoggi" sui livelli sottostanti, piuttosto che "richiedere servizi" a livelli più generali (chi vuole, può trovarvi facilmente l'eco di una visione gerarchica delle organizzazioni, che resta a ricordare le invariabili ramificazioni della legge di Conway [Conway, 1968]: la struttura [anche grafica] del software tende a rispecchiare la struttura [comunicativa] dell'azienda).

A prima vista, potremmo considerare la discordanza come un semplice problema geometrico: è sufficiente ribaltare il diagramma ed ecco che, improvvisamente, rispetterà la regola proposta (ammesso, ovviamente, che il diagramma originale utilizzasse *solo* dipendenze verso il basso). In realtà, la questione è leggermente più complessa, e si riallaccia alla massima di "rendere esplicito ciò che è essenziale". Riprendiamo infatti alcune forze "essenziali" che portano alla nascita di un sistema stratificato, facendo sempre riferimento a [Buschmann et al., 1996]:

- Interfaces should be stable, and may even be prescribed by a standards body;
- Parts of the system should be exchangeable. Components should be able to be replaced by alternative implementations [...].

D'altra parte, nel diagramma stratificato "classico" non è presente alcuna separazione tra interfaccia ed implementazione. Viene quindi reso implicito un fattore importante, direttamente correlato con le forze che il

pattern intende gestire. Sembra quindi che il classico diagramma stratificato (a sinistra in Figura 3) sia assai meglio rappresentato dalla sua controparte destra, che evidenzia un *elemento essenziale del modello*.

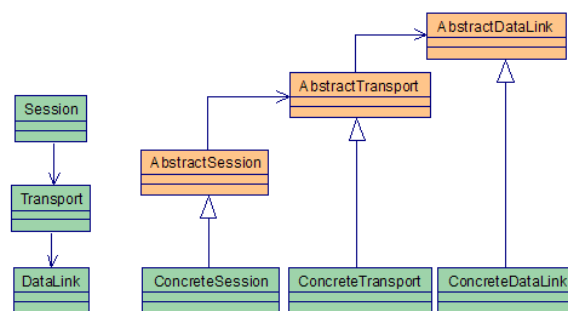


Figura 3 – Rappresentazioni alternative di un sistema stratificato

Ad onor del vero, il testo succitato considera l'ipotesi di utilizzare un *Adapter* od un *Bridge* per creare sistemi più facilmente estendibili: queste sarebbero varianti strutturalmente più complesse rispetto alla semplice interposizione di un'interfaccia, ed il conseguente aumento di complessità andrebbe motivato con un incremento di altre proprietà desiderabili, ad es. la possibilità (per l'*Adapter*) di agganciarsi a sistemi preesistenti che non implementano direttamente l'interfaccia fornita.

Occorre a questo punto chiedersi se sia giusto considerare il diagramma di sinistra una *buona rappresentazione* di un sistema stratificato, o se sia meglio usare la forma più prolissa di destra. La risposta, in fondo, possiamo trovarla sempre nello stesso testo, dove gli autori citano come esempio di sistema stratificato un programma per il gioco degli scacchi organizzato secondo questi layer: pezzi elementari del gioco; mosse di base; tattiche di medio termine; strategia complessiva². Possiamo rappresentare il sistema come in Figura 4 ed è anche possibile che ci risulti chiaro e soddisfacente.

²Notiamo peraltro che gli stessi autori elencano i layer nell'ordine dato sopra, peraltro su righe distinte, in pratica capovolgendone l'ordine rispetto al diagramma...

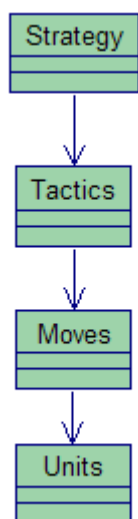


Figura 4 – Rappresentazione tradizionale dei layer nel gioco degli scacchi

Tuttavia nel modello esiste almeno un punto ampiamente discutibile, se non apertamente errato, ed il diagramma non ci ha affatto aiutato a capirlo. I layer dovrebbero essere sostituibili visto che, in assenza di interfacce esplicite, siamo *abituati a presumere* l'esistenza di interfacce implicite. Tuttavia, in un caso reale potremmo probabilmente sostituire il layer *Units* (in funzione delle responsabilità che gli avremo attribuito, non meglio specificate nel testo). Ad esempio, se il layer si occupa della visualizzazione, possiamo cambiarlo per offrire una visualizzazione alternativa.

Pensiamo ora a sostituire il layer *Moves*, che contiene mosse elementari, come l'arrocco. Qui non esiste molto spazio di manovra: se è possibile sostituirlo, il nuovo layer dovrà contenere altre mosse. Ora, pur senza scendere in ulteriori dettagli, è evidente che il layer *Tactics* difficilmente potrà funzionare correttamente se cambiamo le mosse a disposizione. Con un ragionamento analogo, difficilmente il layer *Strategy* potrà funzionare in modo indipendente dal layer *Tactics*.

Perché abbiamo questo problema? Perché la rappresentazione classica del sistema stratificato è sovrasemplificata, sino ad essere scorretta. Lascia all'immaginazione del lettore dettagli che *non possono essere intuiti correttamente* nei diversi casi. Nel caso del gioco degli scacchi, una rappresentazione (probabilmente) sensata è quella di Figura 5, che mostra una sola interfaccia, ed un accoppiamento tra classi concrete per il resto del sistema. Il layer strategico è più concreto del layer tattico, e quindi posizionato più in basso, e così via. Nel caso del layer ISO/OSI, abbiamo invece una più forte presenza di

interfacce standard, che permettono di sostituire le effettive implementazioni. Non possiamo *ridurre* entrambi i diagrammi alla stessa rappresentazione senza *perdere* alcuni elementi essenziali del design.

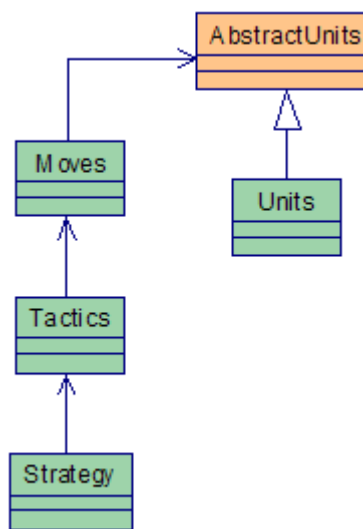


Figura 5– Rappresentazione più corretta dei layer nel gioco degli scacchi

Tornando alle questioni di layout, è ovvio come queste siano del tutto strumentali: il problema sussiste al di là del layout adottato per il diagramma finale. Ricordiamo però che il valore di una norma di stile sta proprio nel condurre in modo più naturale verso un progetto migliore³. Rendere esplicite le interfacce tra i layer rende esplicite alcune proprietà essenziali del sistema, senza lasciarle all'immaginazione (con i rischi di cui sopra). Modellare le dipendenze verso l'alto in un sistema stratificato ci aiuta a capire se ogni strato si appoggia realmente ad uno strato più generale, astratto o concreto che sia.

Dipendenze Tutto-Parti

Le dipendenze tutto-parti vengono spesso rappresentate con il tutto al centro e le parti disposte sui quattro lati ("a ragno"), oppure con il tutto in alto e le parti in basso ("a medusa"). La rappresentazione "a ragno", riscoperta individualmente da molti progettisti, ha un suo carattere di comodità che emerge in particolari contesti. Tra i suoi

³ In altre parole, una norma di stile dovrebbe incoraggiare altre pratiche che migliorano la qualità generale del prodotto. Non dovrebbe, per contro, favorire pratiche che portano ad un peggioramento qualitativo del prodotto. Per una analogia a livello di norma di codifica, si veda la discussione della raccomandazione 55 in [Pescio,1995], soprattutto nella parte finale.

lati interessanti troviamo:

- un utilizzo parsimonioso dello spazio, comodo quando le parti sono molte;
- un senso estetico di simmetria, più evidente quando le parti sono molte;
- una centralità del tutto, che fornisce un visual clue non disprezzabile a chi legge.

Esaminando con più cura le argomentazioni di cui sopra, tuttavia, emerge un primo elemento che tende a controbilanciare l'opportunità di scegliere il layout a ragno: questa si manifesta, infatti, quando le parti sono molte. Un diagramma del genere, tuttavia, sottende un approccio alla progettazione tramite controllo centralizzato: numerose parti, tipicamente dotate di scarsa intelligenza ed autonomia, aggregate da un tutto super-intelligente, che riassume in modo tipicamente non riusabile l'intera logica. In breve, la *god class* già citata. Vista da questa angolazione, una pratica di layout che semplifica la rappresentazione di sistemi di bassa qualità non va incoraggiata, allo stesso modo in cui non incoraggeremmo un stile di codifica che rende più "accettabile" il codice che viola le più elementari nozioni di leggibilità.

Esistono almeno altre due argomentazioni a favore dell'adozione del layout "verso l'alto" anche per le dipendenze tutto-parti, che si applicano sia al layout a ragno, sia a quello a medusa. La prima, su cui ci soffermeremo appena, riguarda l'unificazione diagrammatica tra due concetti di modellazione affini, ovvero il tutto-parti e l'ereditarietà privata. In diversi contesti, la scelta tra le due alternative va valutata con cura, in quanto entrambe possono modellare lo stesso concetto (con una maggiore potenza nel caso di ereditarietà privata, ammesso che sia presente nel linguaggio target e che non causi problemi collaterali, ad es. introducendo un fork-join nel grafo di ereditarietà). Da questo punto di vista, se adottiamo il layout "verso l'alto" per la modellazione dell'ereditarietà (pubblica o privata che sia), e se consideriamo la quasi equivalenza dell'ereditarietà privata e del tutto-parti, sembra naturale adottare lo stesso layout anche per il secondo caso, ponendo quindi il tutto in basso rispetto alle parti aggregate.

La seconda argomentazione è leggermente più complessa. Consideriamo il diagramma di Figura 6, che nel lato sinistro mostra una automobile ed una delle sue parti (il motore),

come spesso viene rappresentata, ovvero con il "tutto" in alto. Alcuni considerano del tutto corretta questa rappresentazione, anche alla luce delle considerazioni esposte sinora, portando la seguente linea di ragionamento: possiamo sostituire un motore a benzina con uno diesel o con uno ad idrogeno, quindi la classe *Car* è più generale e la classe *Engine* è più specializzata.

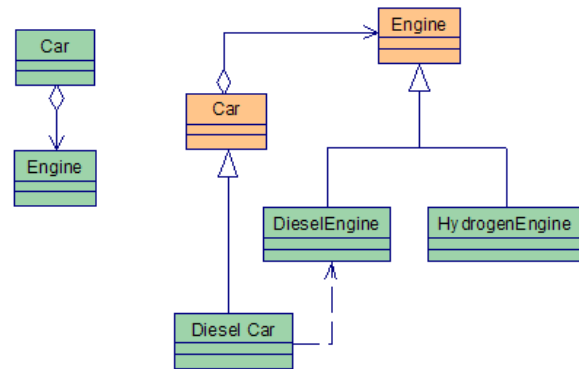


Figura 6– Rappresentazioni alternative del tutto-parti "automobile"

Purtroppo questo ragionamento non è corretto, sia nel mondo reale, sia in quello più astratto della modellazione. Nel mondo reale, i parametri del motore influenzano anche il progetto di altre parti dell'automobile, e di conseguenza il motore non è sostituibile con tanta facilità: di fatto, l'automobile (come insieme) è quasi sempre più specializzata del motore che integra. Nel mondo della modellazione ad oggetti, invece, il diagramma di sinistra non dice esplicitamente che il motore è sostituibile: abbiamo pur sempre un accoppiamento tra classi concrete. Può darsi che l'autore avesse in mente questo concetto ma che, per mantenere il diagramma semplice, non l'abbia esplicitato. In questo caso, riportato in forma esplicita, un modello che mostra la dipendenza auto-motore in modo più preciso è quello di destra. Ora, nell'ottica di una possibile semplificazione del diagramma, potremmo decidere di omettere alcuni "dettagli" e mostrare solo le classi *Car* ed *Engine*. Anche in questo caso, sarebbe quanto mai indicato mettere in pratica una "conservazione della forma" nel passaggio da implicito ad esplicito: se la dipendenza *Car-Engine* di sinistra riassume, in modo implicito (ed in questo caso un po' discutibile), l'espansione della controparte destra, ne dovrebbe quanto meno preservare l'orientamento: in altre parole, il passaggio semplificativo da esplicito ad implicito non deve alterare il messaggio trasmesso attraverso la forma. Notiamo peraltro, come argomentazione conclusiva, che il motore, così come le ruote, e così come ogni elemento

costitutivo di un "tutto", è per sua natura più generale del "tutto", in quanto abitualmente riusabile in altri contesti. Di conseguenza, volendo posizionare ciò che è generale in alto, le parti devono per necessità salire verso l'estremo superiore del diagramma.

Conclusioni

Abbiamo speso molte parole per descrivere e giustificare una norma di layout estremamente semplice. Questo tipo di giustificazione e di analisi accurata è tuttavia necessario, soprattutto in un settore dove abbondano le "regole" proposte più sulla base dell'istinto personale che di argomentazioni ponderate. Per chi preferisce la pratica alla teoria, ci limitiamo ad un semplice suggerimento: provate ad adottare lo stile di layout descritto. Un minimo di adattamento iniziale sarà probabilmente necessario, e forse vi troverete ad usare un'area di disegno più estesa. Al di là di questo, se trovate che il design si appesantisca graficamente, che alcune classi *rivelino* troppe dipendenze, che una classe assuma una collocazione "innaturale", prendetela come un'occasione di riflessione. Il vostro diagramma vi sta parlando, vi sta suggerendo qualcosa. Invece di zittirlo alterando il layout, verificate che non esista un più profondo problema di modellazione. Donald Schön ha battezzato questi momenti come "conversazione riflessiva con i propri materiali": si tratta di una visione molto interessante del rapporto tra uomini e strumenti, di cui parleremo in una futura occasione.

Bibliografia

[Buschmann et al., 1996] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael - "Pattern-Oriented Software Architecture, Volume 1", Wiley&Sons, 1996

[Brooks, 1987] Brooks, Frederick P. - "No Silver Bullet: Essence and Accidents of Software Engineering", Computer, 1987

[Brown et al, 1998] Brown, William H.; Malveau, Raphael C.; McCormick III, Hays W.; Mowbray, Thomas J. - "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis", Wiley and Sons, 1998

[Conway,1968] Conway, M. E. - "How do committees invent?", Datamation, Vol. 14 No. 4, April 1968.

[GOF, 1995] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M. - "Design Patterns", Addison Wesley, 1995

[Letovsky&Soloway, 1986] Letovsky, S.; Soloway, E. - "Delocalized plans and program comprehension", IEEE Software, Vol. 3, No. 3, May 1986.

[Martin, 1996] Martin, Robert - "The Dependency Inversion Principle", C++ Report, 1996

[Nguyen&Swatman, 2000] Nguyen, L.; Swatman, P. - "Essential and incidental complexity in requirements models", 4th International Conference on Requirements Engineering Proceedings (ICRE00), 2000

[Norman, 1993] Norman, Donald - "Things that Make us Smart", Addison-Wesley, 1993. Edizione italiana: "Le cose che ci fanno intelligenti", Feltrinelli, 1995.

[Pescio, 1995] Pescio, Carlo - "C++ Manuale di Stile", Edizioni Infomedia, 1995.

[Pescio, 1997] Pescio, Carlo - "When Past Solutions Cause Future Problems", IEEE Software, September/October 1997

[Pescio, 1999] Pescio, Carlo - "Systematic Object-Oriented Design - Parte 1", Computer Programming No. 81, Giugno 1999.

[Pescio, 2000] Pescio, Carlo - "UML Manuale di Stile", draft disponibile gratuitamente, www.eptacom.net/umlstile.

[Pescio, 2001] Pescio, Carlo – “Modelli che parlano: realizzare diagrammi espressivi”, Convegno su UML organizzato dal team di Programmazione.it, Milano 24 settembre 2001, disponibile presso il sito <http://www.eptacom.net>

[Robertson, 2002] Ian Robertson, “Opening the Mind's Eye”, St. Martin's Press, 2002. Edizione italiana: “Intelligenza Visiva”, RCS Libri.

[Yacoub et al, 2000] Yacoub, Sherif M.; Ammar, H. H. – “Toward Pattern-Oriented Frameworks”, Journal of Object-Oriented Programming, Vol. 12 No 8, January 2000

Biografia

Andrea Baruzzo è laureato in Scienze dell'Informazione presso l'Università degli Studi di Udine. È membro del gruppo infoFACTORY presso il Laboratorio di Intelligenza Artificiale ed Applicazioni Avanzate per la Rete Internet. Si occupa di ricerca, formazione e consulenza sia in ambito accademico, sia in ambito aziendale. Le sue principali aree d'interesse sono l'analisi, la progettazione e lo sviluppo di sistemi ad oggetti (OOA/OOD/OOP), la qualità del software e le tecniche di machine learning. È inoltre membro di IEEE Computer Society.

Carlo Pescio svolge dal 1991 attività di analisi e progettazione di sistemi, consulenza e formazione, cercando di coniugare gli sviluppi più promettenti della ricerca con le necessità pragmatiche dell'industria.

Tra i suoi interessi più recenti ricadono lo studio delle dinamiche sociali nei team di sviluppo, gli approcci innovativi alla formulazione dei requisiti ed il ragionamento diagrammatico, ma raramente resiste a lungo lontano dal codice.

È autore di oltre 90 pubblicazioni, apparse sui più importanti periodici internazionali, dedicate principalmente a C++, Object Oriented Design e Software Engineering.

Laureato in Scienze dell'Informazione, è membro di IEEE Computer Society, ACM, e dell'IEEE Technical Council on Software Engineering.