Questo articolo propone alcune applicazioni del colore ai diagrammi UML, consentendo a chi progetta di trasmettere maggiori informazioni in modo semplice, a basso carico cognitivo

Progettare con UML ed il colore: facciamo parlare la struttura

di Andrea Baruzzo, Carlo Pescio

C'era una volta la Documentazione. Ogni libro di Ingegneria del Software dovrebbe, probabilmente, iniziare così. In pratica, la documentazione non è mai stata il punto forte degli sviluppatori, e nulla fa pensare che possa diventarlo nel prossimo futuro. Ciononostante, trascrivere e tramandare informazioni sui progetti è fondamentale: i diagrammi tentano oggi di giocare un ruolo intermedio, da un lato descrivendo il sistema ad un livello di astrazione sufficientemente alto, e dall'altro aiutando il progettista a *pensare* al sistema che sta sviluppando. In questo modo, la sensazione di "perdere tempo" documentando *diminuisce* (sino a sparire quando si comprende che il diagramma è uno strumento di pensiero a priori, non di documentazione a posteriori), e la quantità di informazione trasmessa *aumenta*.
Un diagramma è tuttavia spesso insufficiente per trasmettere tutte le informazioni che caratterizzano un sistema complesso. È molto probabile che importanti decisioni di design si perdano in un mare di rettangolini e freccette. Quali erano i punti di estendibilità? Quali classi non possono essere modificate in manutenzione, in quanto condivise con altri sistemi? Quali classi hanno iniziativa, quali sono semplicemente chiamate all'occorrenza? Come si dipana a run-time una interazione complessa? Alcune di queste domande possono trovare risposta attraverso altri diagrammi, come il sequence diagram, che modellano gli aspetti dinamici del sistema. Purtroppo si tratta di diagrammi estremamente *fragili*, ovvero soggetti a forti variazioni anche a fronte di modifiche modeste alla controparte statica.
In questo articolo descriveremo un approccio che non rinuncia a comunicare informazioni, ma cerca di ottenere questo risultato in modo *economico*, per chi scrive il diagramma e per chi lo legge. Arrivando un passo più vicini ad un *diagramma che parla*.

L'uso del colore nella modellazione

Con l'aumento della complessità e delle dimensioni dei moderni sistemi software, diventa particolarmente importante riuscire a costruire diagrammi espressivi, soprattutto se vogliamo introdurre nei nostri modelli sufficiente informazione per ragionare su di essi, senza per questo diminuirne la leggibilità. Come linea guida generale, possiamo ritenere un diagramma sufficientemente espressivo quando, leggendolo, riusciamo a riconoscere "a colpo d'occhio" le proprietà più importanti del design che vogliamo trasmettere. Nelle precedenti puntate di OOD abbiamo visto come (ad esempio) l'elisione di dettagli implementativi sia un valido strumento per evitare di nascondere le informazioni più interessanti in mezzo ad un mucchio di aspetti secondari. Trattandosi di diagrammi, tuttavia, l'elemento più caratterizzante è ovviamente quello *grafico*: possiamo trasmettere meglio l'informazione se sfruttiamo a fondo le possibilità di tale elemento. Gran parte delle notazioni esistenti sfruttano sin dall'inizio questa possibilità. In UML, ad esempio, alcuni tipi di relazione (dipendenza, associazione, ereditarietà) sono stati ritenuti così caratterizzanti da meritare un *simbolo* distinto. Tuttavia è sempre possibile *arricchire* la notazione, in modo tale da trasmettere ulteriori informazioni. Un esempio recente in tale direzione si trova in **[3]**, dove viene mostrato con alcuni esperimenti come sia possibile trasmettere informazioni utili alla manutenzione di sistemi con semplici decorazioni (bordature in neretto, campiture in grigio) applicate ad un diagramma a blocchi SDL. Un'idea alternativa, ma sempre correlata, si trova in **[13]**, dove si utilizza la *dimensione* degli elementi grafici nei diagrammi DFD ed Entity-Relationship per fornire informazioni sulla centralità e sul contesto.
Un altro approccio per veicolare informazioni importanti senza ridurre la leggibilità e la semplicità dei diagrammi consiste nell'introduzione del colore. Il colore rappresenta infatti un

ottimo strumento per comunicare informazione aggiuntiva, mantenendo basso il carico cognitivo. Attraverso il colore siamo in grado di esprimere preziosi elementi semantici (pensiamo al ruolo delle classi in un particolare diagramma) e di esplicitare importanti assunzioni di progetto, spesso assenti dalla documentazione ed invece importantissime per comprendere meglio il design. A differenza dei più noti meccanismi di estensione in UML (gli stereotipi e le icone), il colore presenta infine un interessante vantaggio: esso continua a convogliare la stessa informazione indipendentemente da un eventuale ridimensionamento dei diagrammi.

I colori di Peter Coad

Peter Coad è stato uno dei primi esperti ad introdurre il colore nella modellazione ad oggetti **[5]**. In quel periodo egli stava lavorando sulla definizione di un "modello universale" che potesse essere considerato come una sorta di ossatura, indipendente dal dominio applicativo, sulla quale elaborare per incrementi un qualsiasi modello più dettagliato. Diede a questo modello il nome di Domain Neutral Component (DNC). I colori nascono come tentativo di identificare a prima vista gli archetipi, ossia gli elementi costituenti il DNC. In ultimo, l'arricchimento con il colore deve, nelle intenzioni di Coad, inserire un elemento di divertimento nell'attività di progettazione. La versione finale del DNC nella visione di Coad prevede i quattro colori di seguito descritti, mirati ad evidenziare aspetti ricorrenti nei modelli di *analisi*:

Giallo – indica classi "attore", ossia elementi che partecipano ad una o più collaborazioni nel sistema mantenendo un ruolo attivo (solitamente sono anche le classi che spesso iniziano tali collaborazioni), da cui il nome di attore;
Rosa – indica classi "momento" o "tempo-evento", ossia elementi del modello ai quali viene associata informazione "temporale", come gli eventi, le transazioni, i periodi di tempo (timer);
Verde – indica classi "cosa", ossia elementi che caratterizzano oggetti del mondo reale come luoghi e persone che, seppur intelligenti, non sono centrali rispetto al particolare diagramma;
Blu – indica classi di descrizione, ossia elementi che forniscono informazioni di tipo "catalog-entry" (aggregati di valori) senza per altro fornire informazioni di tipo temporale o prendere parte a collaborazioni significative.

Analisi e design producono quasi sempre modelli differenti, proprio perché diversa è l'informazione ritenuta importante che si vuole comunicare. Affinché l'utilizzo del colore sia scalabile nei diversi tipi di modelli, dobbiamo però essere in grado di utilizzare lo stesso set di colori di riferimento, indipendentemente dalla particolare prospettiva assunta. Estendendo il significato dei colori di Coad al design **[11]**, possiamo attribuire alle classi gialle il significato di "elementi più intelligenti" del diagramma. Le classi rosa possono anche contraddistinguere messaggi asincroni, eccezioni software oppure eventi di sistema. Le classi verdi possono essere associate anche ad elementi secondari come gli elementi di infrastruttura (file e contenitori). Le classi blu, infine, sono ottime candidate per rappresentare elementi passivi, totalmente privi di intelligenza, come le strutture o classi la cui interfaccia sia composta principalmente da metodi accessori (get e set).

Una prospettiva più orientata verso il design

In realtà, passando alla fase di design, emergono alcuni aspetti importanti che non vengono convogliati attraverso la colorazione di Coad. I quattro colori definiti sopra, infatti, sottolineano aspetti di dominio, e sono quindi largamente correlati ai requisiti funzionali. Passando al design, come è stato molte volte sottolineato, si sposta l'enfasi anche sui requisiti non funzionali, come riusabilità ed estendibilità. Non è quindi sufficiente estendere i casi di applicazione dei colori di Coad; possiamo tuttavia introdurre alcuni nuovi colori, in numero limitato e sufficientemente diversi dai quattro originali, per trasmettere con immediatezza alcune informazioni molto importanti. Il set di colori che descriveremo è stato inizialmente introdotto in **[10]**, e successivamente raffinato in **[11]**. Come già accennato, l'obiettivo è comunicare, letteralmente "a colpo d'occhio", alcune importanti informazioni. Amplifichiamo quindi il potere comunicativo dei diagrammi, anche quando tale comunicazione è rivolta verso noi stessi, sia durante la progettazione (migliorando quindi le potenzialità del diagramma come

strumento di pensiero), sia in un secondo tempo, quando capiterà di rimettere mano al progetto per far fronte a nuove necessità.
In quest'ottica, è fondamentale avere un quadro immediato dei *punti di estendibilità* previsti nel design. Notiamo che questi punti coincideranno in diversi casi con le classi interfaccia, ma non saranno necessariamente coincidenti: in molti casi avremo hot-spot astratti, ed in altri ancora avremo interfacce introdotte con il solo ruolo di disaccoppiamento, senza una reale funzione di punto di estendibilità. Il colore adatto a questo concetto di design è l'arancione, che attira immediatamente l'occhio: infatti i punti di estendibilità sono tra i più importanti del nostro modello.
Un altro concetto fondamentale della programmazione (non solo ad oggetti) è il riuso. Riusare una classe significa anche rispettarne il contratto, soprattutto se proviene da terze parti (altre aziende, o altri gruppi interni all'azienda che non possono modificare le classi riusabili per adattarle al nostro progetto). In questo caso, l'obiettivo della visualizzazione è duplice: mostrare la "quantità di riuso" e chiarire immediatamente che alcune classi non potranno (ad es.) essere modificate per essere integrate nella nostra applicazione, ma dovranno essere eventualmente chiamate tramite un adapter **[6]**, o estese tramite ereditarietà. Un colore adatto per modellare questi aspetti, che pongono le classi "al di fuori" del nostro controllo, è un colore freddo, come il grigio.

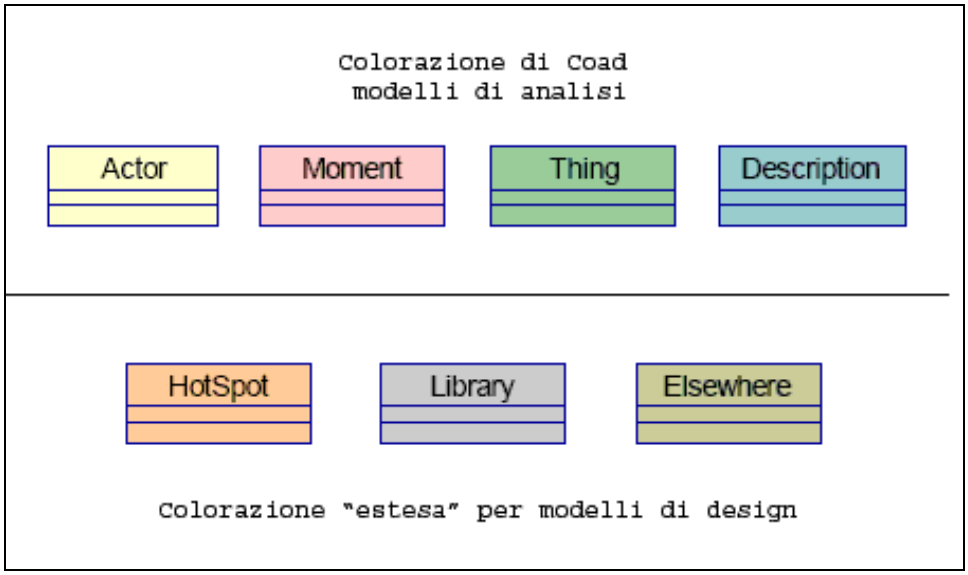

Figura 1 - Il set di colori base per i diagrammi di struttura

Infine, un ultimo colore è stato introdotto per facilitare il ragionamento multi-diagrammatico. Da diversi studi (si veda ad es. **[8]**) risulta infatti che quando un modello richiede diagrammi multipli per essere rappresentato (ad es. diversi diagrammi delle classi, uno per ogni sottosistema) è utile fornire dei "*visual clue*" per suggerire i confini del sistema, ovvero, nel nostro caso, le classi che giocano un ruolo secondario in uno specifico diagramma, ma che hanno tipicamente un ruolo diverso e più importante nel diagramma "principale" in cui sono nate. Si tratta, tutto sommato, di un caso di "riuso interno", dove una classe compare in più di un diagramma, e vogliamo fornire a chi legge una rapida indicazione che questa classe va cercata altrove (un buon tool di diagrammazione potrà poi mostrare il package di provenienza, a cui dovrebbe essere associato *un* corrispondente diagramma delle classi). Il colore associato a questo concetto è il bronzo, relativamente neutro e simile al grigio, nuovamente per non attirare l'occhio verso i contorni e per suggerire un ruolo secondario nel diagramma. In **Figura 1** riassumiamo sia la colorazione originale di Coad, sia la colorazione estesa appena discussa. Tali colorazioni compongono il set di colori base per un diagramma UML di struttura.

Facciamo parlare la struttura
Il diagramma delle classi, come è noto, si utilizza per modellare i cosiddetti aspetti statici: le classi e le loro associazioni. Non mostra però la dinamica del sistema, ovvero le interazioni che hanno luogo tra gli oggetti. UML prevede altri diagrammi (activity, sequence e collaboration

diagram) che possono colmare questa lacuna. Tra questi, il livello di massimo dettaglio si ottiene con il sequence diagram, che ha tuttavia una serie di lati negativi:
sono teoricamente necessari più sequence diagram (uno per ogni scenario rilevante che vogliamo rappresentare);
il livello di dettaglio è molto elevato, e di conseguenza anche la manutenzione è molto elevata;
è necessario cambiare diagramma per leggere i comportamenti dinamici che avvengono sulla realtà rappresentata dal modello statico;
la presenza di elevato dettaglio tende ad aumentare la complessità dei diagrammi;

In effetti, lo stesso Peter Coad che ha introdotto l'idea dei colori aveva tentato, in una sua notazione che non è sopravvissuta ad UML, di fondere aspetti statici e dinamici (si vedano ad es. i numerosi diagrammi in **[4]**). Tuttavia, al di là della scomparsa della notazione, va detto che si trattava di un approccio poco scalabile su diagrammi complessi, o dove esiste una molteplicità di scenari possibili.
Un approccio alternativo, mirato all'utilizzo da parte dei progettisti esperti, potrebbe essere quello di migliorare ciò che già facciamo: intuire, in larga misura, la dinamica a partire dalla struttura statica, usando di solito diversi "suggerimenti" come il nome delle classi e delle associazioni, gli stereotipi, il *riconoscimento* visivo di pattern nelle interazioni, e così via. Infatti, è tutto sommato normale per i progettisti discutere la dinamica sul diagramma (statico) delle classi, tipicamente "per gesticolazione", ovvero mostrando come il controllo fluisca da un oggetto ad un altro, senza dover necessariamente creare altri diagrammi. Potremmo quindi ipotizzare di utilizzare la colorazione per amplificare questa capacità degli esseri umani di lavorare anche su rappresentazioni semi-formali. Questo approccio non è stato sinora descritto in letteratura ed è quindi, per il momento, basato esclusivamente sull'esperienza derivante da alcune applicazioni in progetti reali e sul conseguente feedback ricevuto (che generalmente è stato positivo), nonché da alcune discussioni legate alla stesura di **[11]**.
La scelta di cosa rappresentare con il colore è tutt'altro che banale. Peter Coad, ad esempio, ha scelto colori ragionevolmente ricorrenti per i modelli di analisi, ma probabilmente sub-ottimi per il design, anche dopo averne esteso il significato. Pensando a colorare le interazioni, i primi pensieri si sono rivolti ad aspetti *classici*, che tutto sommato si modellano spesso attraverso stereotipi: "parametro", "creazione", "navigazione di un sotto-oggetto", ecc. Tuttavia il valore aggiunto di un simile approccio è davvero limitato. Se vogliamo ottenere un risultato di rilievo, dobbiamo mostrare qualcosa che non si vede o rappresenta con facilità. Un ottimo candidato sono i cosiddetti "*delocalized plan*" **[2]**, ovvero funzionalità non localizzate all'interno di un unico metodo (o classe), ma ottenute invece attraverso interazioni anche complesse tra oggetti anche "distanti" (a livello concettuale o di associazioni). È stato infatti più volte dimostrato come le interazioni complesse possano creare problemi di interpretazione del design (per un interessante esperimento moderno, che valuta i pro ed i contro di uno stile di controllo centralizzato piuttosto che distribuito, e confronta programmatori esperti con novizi, potete far riferimento a **[1]**).
Una volta definita l'idea di mostrare i delocalized plan, occorre pur sempre trovare un numero limitato di interazioni ricorrenti, in modo da identificare un piccolo insieme di colori significativi. Proprio quel "*ricorrenti*" può suggerire un possibile approccio: utilizzare i più famosi design pattern come fonte di ispirazione, cercando le interazioni che siano al contempo più frequenti e meno ovvie da intuire leggendo semplicemente il diagramma delle classi.

Coloriamo i design pattern
Esaminando i principali pattern di **[6]**, notiamo che esistono alcune interazioni minimalistiche ricorrenti, in base alle quali possiamo costruire per composizione gran parte dei pattern:
La *semplice invocazione* di metodi: è l'interazione più normale e frequente, dove una classe ne conosce un'altra per invocarne i metodi, senza utilizzo di polimorfismo o altre interazioni sofisticate. Indubbiamente, vorremo riservare a questa frequentissima interazione un colore neutrale, che non risalti, in modo da non trasformare inutilmente il diagramma in un arlecchino. Il nero si presta molto bene a questo scopo.
Il *forwarding*: un oggetto riceve una chiamata di funzione, ma non esegue alcun processing di rilievo: si limita a *girare* la chiamata ad un altro target. È una situazione comune in tutte le varie forme di wrapping, e può essere considerata una semplice specializzazione del caso

precedente. Per indicare l'interazione tutto sommato debole tra le classi, e la somiglianza con la semplice invocazione di metodi, scegliamo un colore grigio, dall'aspetto simile e neutro.
La *chiamata polimorfa* tramite una classe base. In realtà, la chiamata polimorfa assume due aspetti differenti, assolutamente *non* evidenti dal diagramma delle classi e difficili da visualizzare correttamente anche su un sequence diagram. Il primo aspetto è il cosiddetto polimorfismo esterno, o "di tipo 1" **[9]**, quando le funzioni vengono invocate polimorficamente dal chiamante esterno: la situazione tipica è quella delle chiamate tramite classe base astratta o interfaccia. Il secondo aspetto è il cosiddetto polimorfismo interno, o "di tipo 2", quando il chiamante esterno invoca una funzione (anche) non virtuale della classe base, che a sua volta invoca polimorficamente alcune funzioni implementate nelle classi derivate. Notiamo che nel primo caso la chiamata esterna "cade" direttamente sulla classe derivata (fall-through), mentre nel secondo caso abbiamo uno o più rimbalzi (da cui il nome classico di effetto yo-yo) tra la classe base e la derivata. Per visualizzare la prima interazione, prendendo proprio lo spunto dall'idea del fall-through, abbiamo scelto il blu, come in una cascata che scende dalla classe base alla derivata (peraltro nel verso opposto alla freccia UML, che non a caso modella le dipendenze statiche e non le interazioni). Per visualizzare la seconda, abbiamo scelto un colore simile ma ancora più visibile (viola), in quanto il polimorfismo interno, pur se utile e potente, è spesso associato al problema della fragile base class.
Il "*non utilizzo*": per quanto possa apparire strano a prima vista, in molte situazioni un oggetto ottiene un riferimento ad un altro e *non* lo utilizza in alcun modo, se non per restituirlo (in momenti diversi) ad altri oggetti. Esempi classici sono le factory, ma anche il memento, così come le diverse classi contenitore "pure". Abbiamo battezzato questa interazione "hot potato" in quanto la classe che ottiene il riferimento al target cerca di manipolarlo il meno possibile, e di restituirlo invece ad altri. Il colore che meglio modella questa condizione è il rosso, che ricorda immediatamente l'idea di caldo.
La *callback*: un oggetto esegue una chiamata, passando sé stesso tra i parametri, in modo tale che il chiamato possa invocarne "all'indietro" alcune funzioni. Tipicamente, in questi casi il chiamante implementa un'interfaccia e passa il riferimento a sé stesso tipizzato attraverso l'interfaccia stessa, per disaccoppiare il chiamato dalla classe concreta del chiamante. Il colore che abbiamo scelto per questa interazione è il verde.
In ultima analisi, abbiamo quindi 6 colori per le interazioni, che coprono uno spettro molto ampio. Talvolta, come già avviene per la colorazione delle classi, dovremo necessariamente scegliere una "colorazione principale" per una associazione, ad esempio nel caso in cui vengano eseguiti sia un forwarding che una chiamata con callback. In queste situazioni, è indispensabile il giudizio del progettista, che dovrà scegliere cosa comunicare nel modo più immediato a chi legge, cercando di supplire all'informazione soppressa in altro modo (tipicamente, chiarendo bene attraverso i nomi di classe, di funzione ed i ruoli delle associazioni)..

Abstract Factory e Command: patate bollenti e polimorfismo
Proviamo ora ad applicare la colorazione proposta a qualche design pattern. Possiamo, ad esempio, considerare Abstract Factory come la combinazione di una chiamata polimorfa con fall-through più un'interazione di tipo "hot potato". L'idea dell'hot potato, letteralmente patata bollente, nasce dalla considerazione che la classe factory di **Figura 2** crea un oggetto di tipo prodotto (*Product1* oppure *Product2*), ma poi con esso non ci fa nulla: si limita a restituirlo il prima possibile al chiamante. La responsabilità di una factory, d'altro canto, è proprio questa: isolare il resto del sistema dalla conoscenza di quale prodotto concreto creare (e dal come crearlo) **[7]**.
La seconda interazione evidente in una factory è la chiamata polimorfa con "fall-through" ("caduta libera") dalla classe base alla classe derivata. Si tratta indubbiamente di un tipo di interazione piuttosto ricorrente, anche al di fuori del pattern in esame. Più in generale, ogni qualvolta abbiamo una "caduta" polimorfa dal chiamante al vero chiamato siamo in presenza di un fall-through. Nel caso di Abstract Factory, abbiamo due situazioni di fall-through: la prima tra il *Client* e la factory concreta, disaccoppiati dall'interfaccia della factory; la seconda tra il *Client* ed il prodotto concreto, intermediato dall'interfaccia dei prodotti. In entrambi i casi, chi applica il pattern sa che il *Client*, in definitiva, eseguirà delle chiamate polimorfe: esprimendo questa conoscenza con il colore, si fornisce a chi legge il diagramma un ulteriore indizio sulla dinamica delle interazioni. La **Figura 2** illustra il pattern Abstract Factory dopo la colorazione

delle due interazioni discusse. Una nota collaterale: abbiamo inserito esplicitamente la dipendenze tra l'interfaccia delle factory (*AbstractFactory*) e quella dei prodotti (*AbstractProduct*), a differenza del testo originale nel quale tale dipendenza è stata omessa.

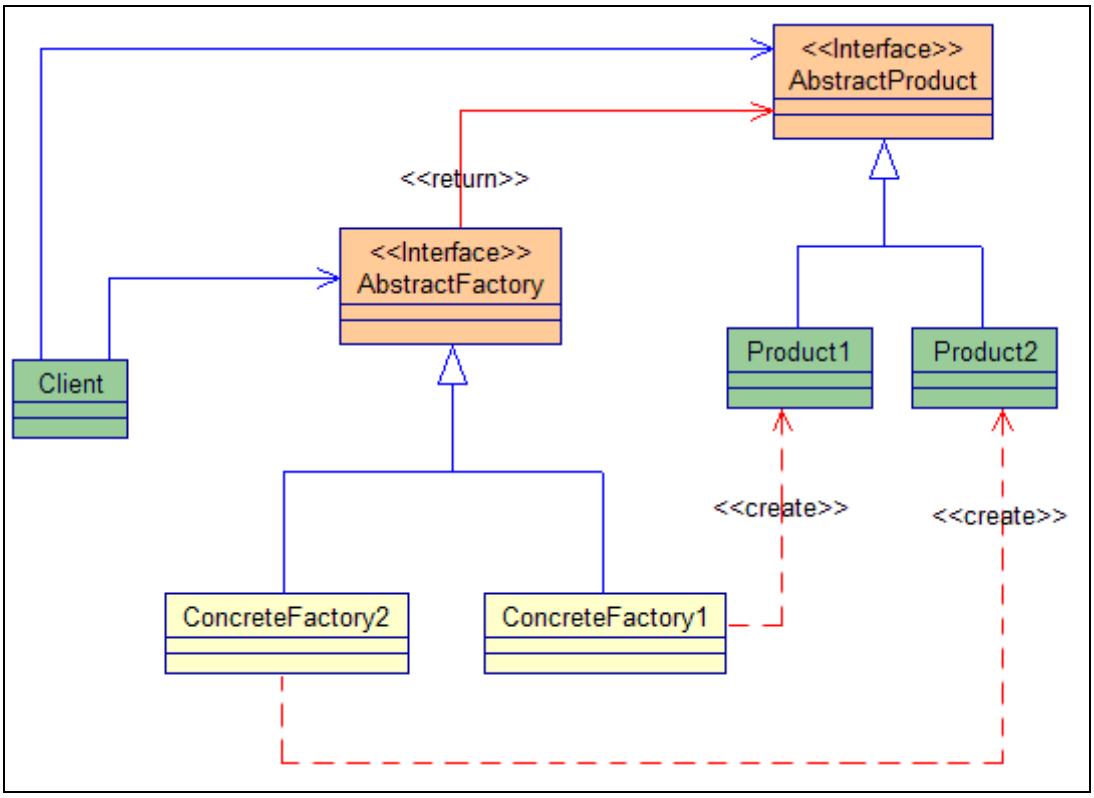

Figura 2 – Interazioni "hot potato" e chiamata fall-through nel pattern Abstract Factory

Riteniamo che il diagramma sia più espressivo e corretto se disegnato in questo modo, anche perché possiamo stereotipare tale associazione per evidenziare che una factory restituisce un prodotto. Si tratta di un esempio che dimostra anche come colore e stereotipi possono essere usati insieme in modo non ridondante.

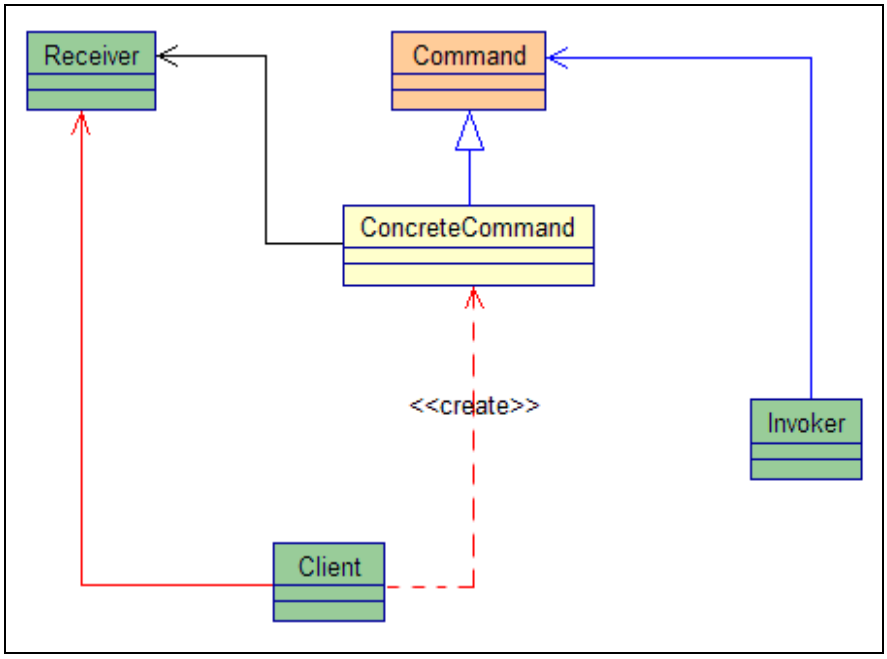

Figura 3 - Il pattern Command colorato

Nell'esempio precedente la classe prodotto che viene utilizzata dal cliente non ha alcuna conoscenza di chi la utilizza. In altre parole, essa è totalmente disaccoppiata dal resto del sistema. Una variante che introduce una nuova interazione in questo scenario è costituita dall'esecuzione di una chiamata diretta da parte della classe concreta ad un altro oggetto del sistema per svolgere dei compiti specifici. La presenza di tale chiamata, che lasciamo nera nel diagramma, può essere facilmente riconosciuta nel pattern Command illustrato in **Figura 3**. La classe cliente crea un comando concreto e ne imposta il corrispondente *Receiver*, che eseguirà in uno dei suoi metodi l'operazione associata al comando. La classe *ConcreteCommand*, che implementa il comando, effettua la chiamata diretta di tale metodo (associazione di colore nero). La classe *Invoker*, infine, effettua la chiamata polimorfa, rimanendo quindi disaccoppiata dal particolare comando concreto utilizzato.

Osserviamo un esempio di callback (Observer)

Un altro pattern interessante da colorare è Observer. Il suo intento è di mettere in relazione tra loro più oggetti in modo che la modifica dello stato di uno di questi, il *subject*, venga notificata automaticamente agli altri (gli *osservatori* concreti). Per realizzare questo meccanismo di propagazione delle modifiche viene utilizzata una callback. Il subject fornisce un'interfaccia attraverso la quale uno o più osservatori concreti possono registrarsi, passando un riferimento a sé stessi. Grazie a questo riferimento, il subject può effettuare le notifiche a tutti gli osservatori registrati, invocando la loro procedura di aggiornamento.

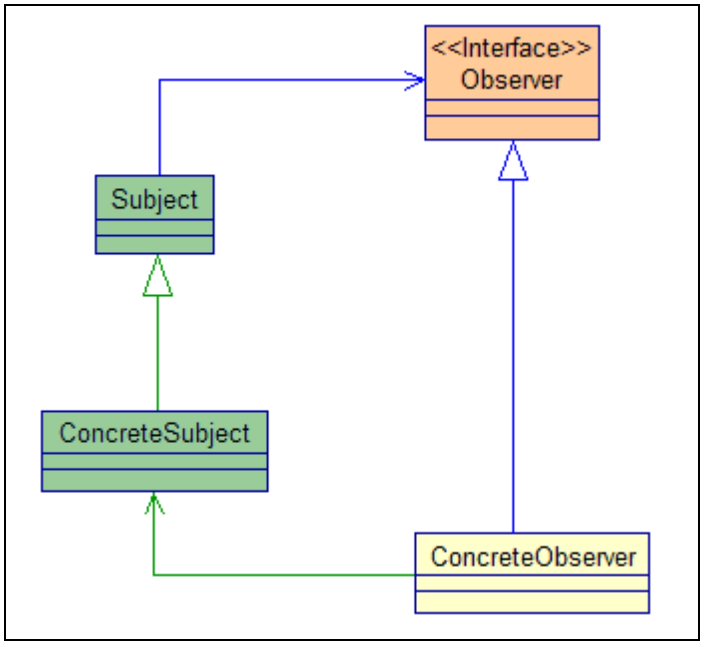

Figura 4 - La callback nel pattern Observer

La **Figura 4** illustra il pattern Observer. Colorando la dipendenza tra *ConcreteObserver* e *ConcreteSubject* in verde, evidenziamo esplicitamente la presenza della callback che altrimenti potrebbe essere scambiata per una semplice dipendenza. Attraverso la colorazione possiamo capire a colpo d'occhio che tale dipendenza è in realtà un'hook installato da *ConcreteObserver* in *ConcreteSubject* per effettuare la propagazione delle modifiche. L'interazione *Subject-Observer-ConcreteObserver*, invece, è la fall-through già vista in precedenza, per cui la coloriamo di blu. Notiamo che la colorazione verde *si estende* alla generalizzazione tra *Subject* e *ConcreteSubject*. In effetti, è in genere Subject che effettua la vera callback, partecipando così al delocalized plan dell'Observer.

Un Adapter per delegare

Il pattern Adapter è un classico esempio di utilizzo della delegazione (forwarding) nella progettazione ad oggetti. Lo scopo principale del pattern è di convertire l'interfaccia di una classe (*Adaptee*) nell'interfaccia che il cliente si aspetta (*Target*), senza modificare la classe di

partenza. La soluzione consiste nel derivare da *Target* la classe *Adapter* la quale implementa l'interfaccia richiesta dal client, delegando alla classe Adaptee la gran parte del lavoro. La dipendenza grigia tra *Adapter* e *Adaptee* in **Figura 5** evidenzia come la prima sia semplicemente una classe wrapper e che la parte più significativa dell'implementazione sta invece in *Adaptee*.

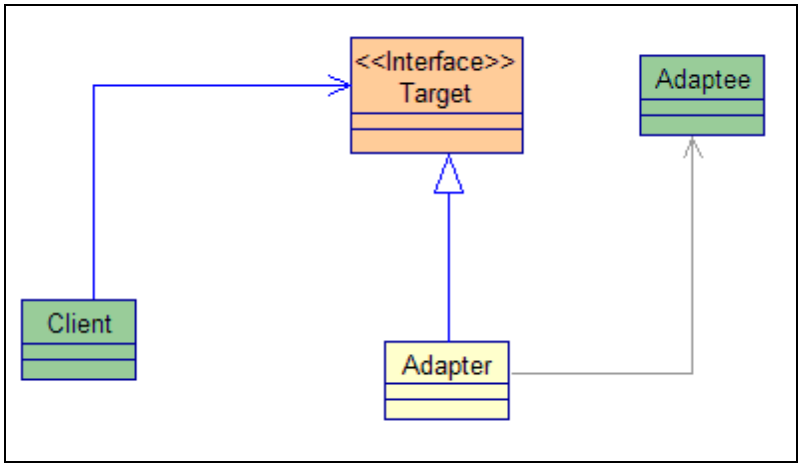

Figura 5 - La delegazione nell'Adapter

Il Template Method e l'effetto yo-yo

Nel pattern Template Method, una classe base astratta (ma non interfaccia) fornisce uno o più metodi concreti, direttamente chiamabili dal client (quindi senza fall-through immediato sulla classe derivata). All'interno di questi metodi, che costituiscono in genere dei veri e propri algoritmi astratti, la classe base chiama uno o più dei suoi metodi astratti. Tali metodi devono, naturalmente, essere implementati nelle classi derivate, che si devono pertanto preoccupare unicamente di definire alcuni singoli passi (custom) degli algoritmi astratti implementati nella classe base. Naturalmente, nulla impedisce alle classi derivate di chiamare a loro volta metodi della classe base, che a loro volta possono nuovamente invocare metodi astratti. Si ottiene, in questi casi, una dinamica nota come "effetto yo-yo", dove il controllo passa continuamente dalla classe base alla derivata.

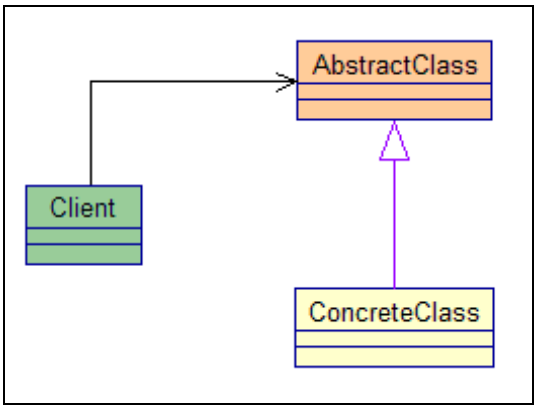

Figura 6 – L'effetto yo-yo nel pattern Template Method

Pur essendo uno dei pattern strutturalmente più semplici, la dinamica del Template Method può quindi risultare anche complessa. Inoltre, si crea spesso un forte accoppiamento tra classe base e classe derivata, che di norma deve essere scritta *conoscendo* le sequenze di chiamata da parte della classe base, e che in seguito vincola la classe base a rispettare tali sequenze anche in versioni future. Un visual clue molto esplicito, come la colorazione viola, aiuta a far emergere la necessità di approfondire i dettagli di una porzione di design che potrebbe altrimenti essere scambiata per un ben più innocente fall-through. L'importanza di evidenziare simili interazioni è legata, come accennato precedentemente, anche al problema della fragile

base class che si manifesta proprio in presenza di questi rimbalzi tra classe base e classe derivata. La colorazione di **Figura 6** può, in questo caso, essere l'occasione per esplicitare attentamente il contratto della classe base. L'aspetto più complesso nella progettazione di buone gerarchie, infatti, risiede soprattutto nel riuscire a garantire non solo la correttezza iniziale, ma anche la robustezza della gerarchia durante l'evoluzione del codice nelle successive versioni **[12]**.

Altri aspetti interessanti

Questo articolo esamina solo alcuni aspetti, con particolare riferimento alle dinamiche tipiche che possiamo riscontrare nei pattern di design più conosciuti. Esistono tuttavia altri elementi interessanti da analizzare e modellare per rendere i diagrammi più espressivi e comunicativi. Pensiamo ad esempio alla concorrenza e al problema della rientranza (più thread che "entrano" contemporaneamente nello stesso oggetto). Pensiamo ancora all'elisione di elementi "ovvi" per ridurre ulteriormente il carico cognitivo, senza perdere molta informazione, come nel caso di alcune interfacce nelle callback. Questi ed altri temi verranno affrontati in un prossimo articolo.

Conclusione

UML nasce come linguaggio di Modellazione, ed un modello deve esprimere le caratteristiche essenziali del progetto. Un *buon* modello deve esprimere queste caratteristiche in modo *economico*, ovvero, massimizzando il risultato comunicativo: ad uno sforzo *ragionevole* di chi crea il modello deve corrispondere uno sforzo ragionevole di chi lo legge. Molti modelli, purtroppo, non vengono creati secondo questo criterio. Quando sono pensati per la macchina (generatori di codice), contengono dettagli irrilevanti che appesantiscono il carico cognitivo degli umani. Ma anche quando non rivelano a colpo d'occhio le intenzioni dell'autore, lasciando a chi legge il difficile compito di immaginare il comportamento del sistema, i diagrammi diventano poco economici per il lettore, e quindi vengono tralasciati. Allo stesso modo, se per comunicare il comportamento dobbiamo stendere dettagliati sequence diagram, si ha uno sbilanciamento economico sul versante di chi crea il diagramma, che nuovamente tende ad abbandonare.

Le idee che abbiamo presentato in questo articolo tentano di riportare un equilibrio tra gli sforzi, consentendo a chi progetta di trasmettere maggiori informazioni in modo semplice, a basso carico cognitivo, e *giocoso* quanto basta per inserirsi al meglio nella pratica della progettazione, che trae grande beneficio da momenti di produttività riflessiva ma tutto sommato rilassata. Provate a metterle in pratica e, se avete dubbi o considerazioni, contattateci via email: il tema è innovativo, ed ogni feedback (soprattutto quelli derivanti dalla pratica in progetti reali) è più che benvenuto.

Bibliografia

[1] Arisholm, Erig; Sjøberg, Dag I.K. – "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software", IEEE Transactions on Software Engineering, Vol. 30 No. 8, August 2004.

[2] Baruzzo, Andrea; Carlo, Pescio – “Diagrammi, Layout e Gestione della Complessità”, Edizioni Infomedia, Computer Programming No. 136, Giugno 2004

[3] Bratthall, Lars; Wohlin, Claes – “Is it Possible to Decorate Graphical Software Design and Architecture Models with Qualitative Information? – An Experiment”, IEEE Transactions on Software Engineering, Vol. 28 No. 12, December 2002.

[4] Coad, Peter – "Object Models: Strategies, Patterns, and Applications", Prentice-Hall, 1995.

[5] Coad, Peter – “Show Your Colors”, in The Coad Letter No. 44, September, 1997

[6] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M. – “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995

[7] Ibid, pag 87

[8] Jinwoo, Kim; Hahn, Jungpil – "Reasoning with Multiple Diagrams: Focusing on the Cognitive Integration Process", Nineteenth Annual Conference of the Cognitive Science Society, Stanford University, 1997.

[9] Pescio, Carlo – “Il problema della "fragile base class" in C++”, Computer Programming No. 41, Novembre 1995.

[10] Pescio, Carlo – “Architettura di Sistemi Record-Oriented, Parte 2”, Computer Programming No. 79, Aprile 1999.

[11] Pescio, Carlo – “UML Manuale di Stile”, draft disponibile gratuitamente, www.eptacom.net/umlstile

[12] Szyperski, Clemens – “Component Software: Beyond Object-Oriented Programming”, cap. 7, pag. 102, Addison Wesley, 1998

[13] Turuken, Ozgur; Schuff, David; Sharda, Ramesh; Ow, Terence T. – ”Supporting System Analysis and Design Through Fisheye Views”, Communications of ACM, Vol. 47 No. 9, September 2004.

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.