

# Scoprire le collaborazioni nei sistemi orientati agli oggetti\*

L'articolo propone alcune strategie per identificare le collaborazioni tra oggetti in un sistema software object-oriented

Dr. Andrea Baruzzo

Un modello ad oggetti è completo quando sono state identificate le classi per le quali sono stati assegnati ruoli e responsabilità ed, infine, sono state definite le collaborazioni. Per gestire la complessità di un qualsiasi sistema non banale è impensabile concentrarsi esclusivamente sugli aspetti di struttura. Esistono infatti aspetti dinamici che condizionano pesantemente le dipendenze tra le classi, influenzando direttamente la complessità strutturale dell'intera architettura. Ecco perché una buona metafora per affrontare la progettazione del software consiste nel considerare un sistema come una grande comunità di oggetti indipendenti, ma comunicanti tra loro (mediante pattern ben definiti). L'interazione con altri oggetti, che chiamiamo *collaboratori*, è un aspetto essenziale perché permette ad una classe di adempiere completamente alle proprie responsabilità, distribuendo la logica di controllo (e, per certi versi, la "intelligenza" del sistema stesso) tra più componenti. Ogni collaborazione, tuttavia, ha un suo costo. Quando un oggetto chiama un metodo di un suo collaboratore, infatti, si instaura tra i due una dipendenza di chiamata che, spesso, è anche una dipendenza strutturale. Un sistema software complesso diventa quindi *economicamente gestibile* solamente se è stato ragionevolmente partizionato durante il design in sottosistemi coesi, disaccoppiati, e le cui collaborazioni aderiscono a *pattern di comunicazione dominanti* e predeterminati. Lo scopo di questo articolo è dunque quello di fornire delle strategie pratiche per identificare le collaborazioni in un sistema ad oggetti mantenendo le proprietà desiderabili appena citate (coesione, disaccoppiamento, gestione della complessità).

## Prime indicazioni per assegnare le collaborazioni

Le collaborazioni sono un ottimo test per controllare le responsabilità assegnate ad una classe [3]. Per ogni responsabilità dobbiamo chiederci sempre se ciascuna classe sia autosufficiente (ossia se sia in grado di implementare un comportamento usando esclusivamente le operazioni e i dati di cui dispone). In caso contrario, dovremmo identificare gli elementi mancanti necessari per soddisfare tale responsabilità. Questi elementi possono essere di due tipi:

- "pezzi" di informazione mancante;
- funzionalità mancanti;

Ovviamente ha sempre senso chiederci perché questi elementi non siano stati inseriti fin dall'inizio nella classe.

Potrebbe essersi semplicemente trattato di una svista, di un successivo raffinamento che completa un'implementazione parziale, oppure potremmo trovarci di fronte ad una nuova collaborazione che necessita di elementi associati ad una responsabilità attribuita in precedenza ad altri oggetti. In quest'ultimo caso, diventa importante identificare i collaboratori della classe in esame e verificare che la nuova collaborazione non implichi una dipendenza troppo costosa. La buona progettazione (non solo quella object-oriented) richiede una costante valutazione di *costi* e *benefici* derivanti da ogni decisione di design. Questo è un aspetto fondamentale da interiorizzare, altrimenti si rischia di rivestire dell'etichetta "object-oriented" un codice che, dopo qualche intervento di manutenzione, diventa molto simile allo "spaghetti-code" dei programmi scritti in C o in assembly.

Per valutare il costo di una dipendenza (di chiamata) viene utilizzata la metafora del "vicinato". Un sistema ad oggetti ben progettato dovrebbe essere *stratificato*, ossia suddiviso in layer o livelli. Ciascun layer è tipicamente organizzato in uno o più package, come illustrato in **Figura 1**. Diremo allora che due oggetti contenuti nello stesso package (ad esempio A1 e A2) sono vicini (ed, in particolare, hanno distanza 0). Due oggetti contenuti in package diversi ma facenti parte dello stesso livello (ad esempio A2 e B) hanno invece distanza 1. Più in generale, ogni qualvolta dobbiamo attraversare un layer per raggiungere un collaboratore, la distanza aumenta di una unità. In conclusione, più due oggetti sono vicini e minore risulta il costo della dipendenza. Le dipendenze meno costose sono, quindi, quelle che identificano un vicinato avente distanza non superiore a 1. Un campanello d'allarme importante è costituito dalle collaborazioni che implicano dipendenze tra classi che distano tra loro più di un livello (è il caso della dipendenza tra A2 e D, sempre con riferimento alla **Figura 1**). Mentre si stabiliscono le collaborazioni, è fondamentale pensare in termini di *costo delle dipendenze*. Solo così possiamo sperare di identificare dei pattern di collaborazione predeterminati e disciplinati. Altrimenti, il design iniziale ben presto si trasforma nell'assenza di design che contraddistingue i sistemi monolitici e centralizzati, tipici della programmazione procedurale. Sottostimare il costo di dipendenze critiche, in questi casi, si paga in alti costi di manutenzione, di debugging e di estensione poiché, nella comunità di oggetti, tutti collaborano con tutti, indiscriminatamente. Apportare una piccola modifica locale causa effetti collaterali in parti del sistema totalmente imprevedibili e spesso logicamente non correlate: un incubo così frequente che dimostra come:

- il metodo object-oriented non sia stato ancora completamente assimilato;
- l'utilizzo di tecnologie ad oggetti (linguaggi, componenti, librerie, CASE tool e altri strumenti di sviluppo) non implicano il saper pensare ad oggetti.

\* Questo documento è una versione estesa e rivista dell'articolo "Oggetti e collaborazioni", pubblicato sulla rivista *Computer Programming*, rubrica Object-Oriented Design, n°131, Gennaio 2004 – Gruppo Editoriale Infomedia. Ultimo aggiornamento: novembre 2008.

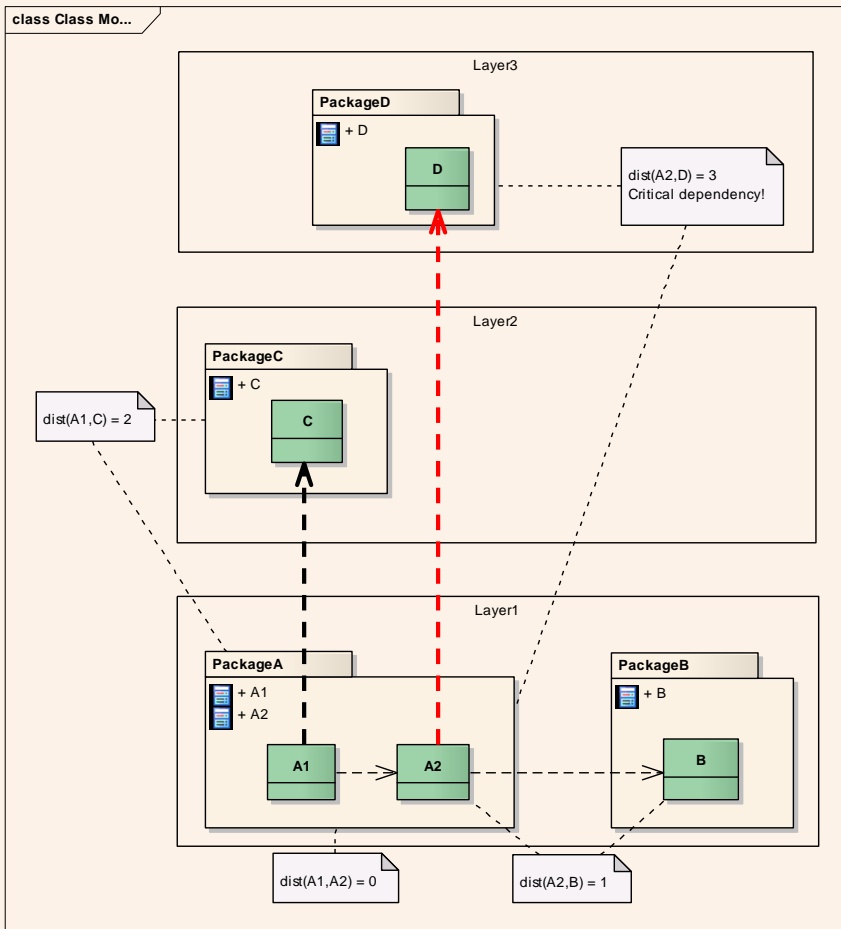


Figura 1- Metafora del vicinato per valutare il costo di una dipendenza

*“Un’architettura object-oriented è costituita da una comunità di oggetti che collaborano tra loro per fornire determinati servizi. Le collaborazioni sono quindi un ottimo test per validare le responsabilità assegnate a ciascuna classe.”*

## Gli scenari d’uso suggeriscono le collaborazioni

Durante la fase di analisi si scrivono spesso degli scenari di utilizzo (noti anche come *use case*, o casi d’uso). Essi possono essere descritti mediante diagrammi oppure (il più delle volte in modo più efficace ed espressivo) mediante delle descrizioni testuali [4]. Questi scenari sono in parte il risultato di interviste con utenti e committenti. Essi rappresentano il sistema da un punto di vista esterno, orientato verso i servizi forniti, piuttosto che verso la struttura logica. Nonostante uno scenario non preveda l’utilizzo delle classi (che sono invece un concetto interno del sistema), è comunque evidente che esso fornisce delle indicazioni sui metodi da implementare nelle classi stesse. Dopotutto, ad un servizio fornito dal sistema deve corrispondere una o più classi che collaborano per implementare tale funzionalità. Ogni servizio è dunque una potenziale collaborazione! Rileggere la documentazione sugli scenari d’uso può essere, quindi, un ulteriore strumento di test per completare il modello ad oggetti del sistema.

## Le collaborazioni e i design pattern

I design pattern forniscono importanti indicazioni su come organizzare le collaborazioni. Un design pattern è costituito sia da una parte statica (di struttura), sia da una parte dinamica.

È a quest’ultima che facciamo riferimento quando progettiamo le collaborazioni. Anziché reinventare completamente una soluzione, risulta conveniente rileggere la descrizione dei pattern che più sono utili a risolvere il problema in esame. Ad esempio, se esistono diversi oggetti contenuti in diversi sottosistemi (package) che forniscono funzionalità utilizzate da una o più classi, può essere interessante valutare l’applicabilità del pattern *Façade* [6], fornendo un’interfaccia che concentra i servizi utili e disaccoppia la classe utilizzatrice dai singoli sottosistemi.

Prima di adottare un pattern, tuttavia, vanno valutate le sue conseguenze. In particolare, per quanto concerne le collaborazioni, dobbiamo porci le seguenti domande:

- Il pattern è compatibile con il ruolo e le responsabilità degli oggetti identificati nel nostro sistema?
- Il pattern semplifica il design? Lo rende più adattabile e flessibile o lo complica, rendendolo più difficile da capire e da mantenere (ad esempio perché aggiunge classi e dipendenze non essenziali)?
- Esistono alternative più convenienti?

Considerare l’applicabilità e le conseguenze di un pattern è indispensabile poiché essi non sono un sostituto del saper pensare. E la progettazione è un’attività prevalentemente razionale e tipicamente umana.

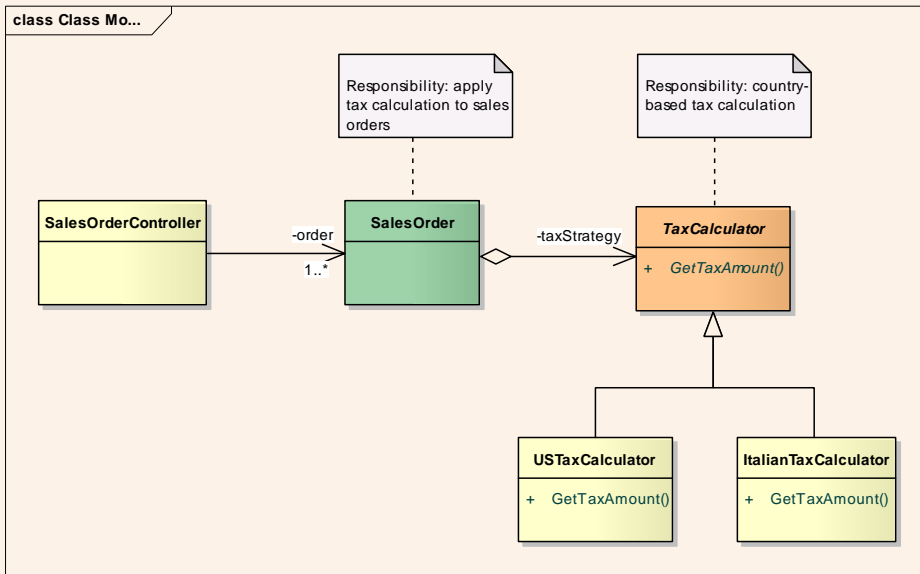


Figura 2 – Collaborazione derivata dal raffinamento di una responsabilità

*“Gli aspetti dinamici sono importanti perché condizionano pesantemente le dipendenze tra le classi, determinando uno specifico stile di collaborazione”*

## Le collaborazioni e lo stile di controllo

Nell’ottimo libro *“Object Design”* [10], Rebecca Wirfs-Brock e Alan McKean forniscono una chiara ed incisiva introduzione alle collaborazioni. Gli autori propongono alcune strategie per individuare le collaborazioni. Trovo particolarmente significativo il tentativo di far ricondurre le collaborazioni alle responsabilità, ai ruoli e agli stereotipi delle classi. Questo sforzo dimostra come, nella progettazione ad oggetti, i diversi elementi del design si incastrino l’un l’altro per descrivere un modello completo del sistema. I successivi paragrafi descrivono queste strategie. Prima di passare ad esse, tuttavia, ritengo utile evidenziare un ulteriore aspetto che caratterizza le collaborazioni: il *controllo*. In questo contesto, con il termine di controllo ci si riferisce alla logica di controllo e alle decisioni che gli oggetti prendono autonomamente durante le collaborazioni. Tali decisioni influenzano fortemente il modo in cui le responsabilità sono distribuite. Sviluppare un modello di collaborazione significa individuare dei *pattern di comunicazione* che disciplinano il flusso di controllo e la sequenza di azioni tra due collaboratori. Questi pattern di comunicazione forniscono delle scelte che caratterizzano gli aspetti di controllo a diversi livelli, come le seguenti domande suggeriscono:

- Come i diversi task di un’applicazione vengono controllati e coordinati nella comunità di oggetti?
- Quali oggetti hanno la responsabilità di prendere le decisioni più cruciali nel dominio dell’applicazione?
- Come vengono gestite le eccezioni che possono verificarsi durante il normale flusso di controllo?
- Esistono degli oggetti (o dei sottosistemi) che hanno la responsabilità di fornire delle strategie di identificazione delle eccezioni ed eventuali strategie di recovery?
- Nelle collaborazioni in che misura un oggetto deve “fidarsi” dei suoi collaboratori?

Le scelte evidenziate sono tutte molto importanti per caratterizzare il concetto di controllo. Particolarmente significativo, nell’ambito di questo articolo, è l’ultimo punto sollevato. È possibile individuare due atteggiamenti estremi nell’implementare le collaborazioni tra due oggetti. Il primo di

questi atteggiamenti, che chiamiamo *stile collaborativo*, ogni oggetto che richiede un servizio ad un suo collaboratore si aspetta che quest’ultimo porti correttamente a termine il suo lavoro e che restituisca il controllo senza effetti collaterali non documentati. Tale posizione descrive un elevato livello di fiducia nei confronti degli oggetti collaboratori. Il secondo atteggiamento, che chiamiamo *stile difensivo*, è molto più guardingo poiché non presuppone che tutto funzioni sempre correttamente. Esso introduce tutta una serie di controlli e di strategie di difesa solo per permettere a due oggetti di collaborare. I controlli sono caratterizzati spesso da contratti software, ossia dall’introduzione di invarianti, precondizioni e postcondizioni nelle classi. Il lettore avrà già intuito che tale filosofia basata sui contratti è il nucleo centrale su cui è fondato il design by contract [8]. Seppure molto meno object-oriented dei contratti, esiste un’altra filosofia che, in alcuni frangenti, può risultare ancora utile: la programmazione difensiva. Entrambe queste filosofie fanno parte dello stile difensivo.

Lo stile collaborativo e lo stile difensivo rappresentano due posizioni estreme. In generale non è consigliabile adottare esclusivamente un unico tipo di stile. Nel primo caso, infatti, si producono sistemi che non sono in grado di gestire adeguatamente le situazioni eccezionali. Basti ricordare le prime versioni del sistema operativo Windows che, essendo basate su un meccanismo di controllo collaborativo, non riuscivano a proteggersi da applicazioni che assumevano il controllo e non lo cedevano mai, esaurendo tutte le risorse di sistema e costringendo l’utente al riavvio della macchina. Neanche lo stile difensivo, tuttavia, è sempre desiderabile. Nel caso della programmazione difensiva si producono sistemi complessi, piuttosto pesanti e che tendono a mascherare (se non addirittura a nascondere) gli errori [7]. Nel caso dei contratti, il costo si concretizza maggiormente nella difficoltà di analizzare, descrivere e verificare completamente la loro specifica [9]. Una strategia pragmatica consiste nell’adottare un elevato livello di fiducia per le collaborazioni tra oggetti “vicini” (situati nello stesso package o nello stesso layer), mentre è il caso di adottare un alto livello di

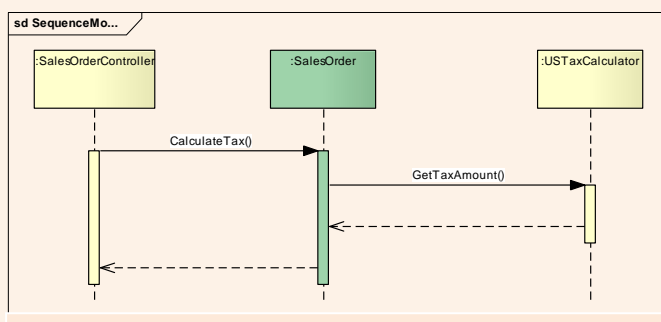
autonomia (e quindi di protezione) per le collaborazioni tra oggetti "lontani" (collocati in layer diversi).

### Le responsabilità implicano collaborazioni

Dal punto di vista delle collaborazioni, le responsabilità sui dati, sulle operazioni e sulle decisioni "intelligenti" possono essere riformulate nelle seguenti azioni:

- Conoscere;
- Fare (o eseguire);
- Decidere.

Molto spesso queste tre azioni interagiscono tra loro in modi anche complessi. Salvo casi molto banali, un oggetto che deve compiere un'azione ha spesso anche la responsabilità (o la necessità) di conoscere un'informazione. Quando l'informazione è complessa, ossia non è descrivibile mediante un semplice tipo primitivo come un intero o una stringa, l'oggetto dipende da un gruppo di oggetti esterni che, nel loro insieme, rappresentano tale informazione. È quindi necessario che gli oggetti in questione comunichino tra loro. L'oggetto da cui parte l'azione diventa un cliente degli oggetti che contengono i dati necessari per completare l'azione stessa. Essi, a loro volta, devono fornire i metodi necessari affinché la comunicazione con l'oggetto cliente possa realizzarsi. Facciamo un esempio per capire meglio. Consideriamo un programma di vendita on-line di un certo tipo di prodotti. Immaginiamo che l'azienda produttrice venda i suoi prodotti sia in Italia, sia negli Stati Uniti. Affinché gli importi siano calcolati correttamente, è necessario che il sistema software gestisca le informazioni relative alle specifiche tasse in vigore nei due Paesi. Come illustrato in **Figura 2**, la classe *SalesOrder* ha la responsabilità di applicare la corretta tassa per un qualsiasi ordine in base al Paese verso cui esso viene emesso. Questa responsabilità implica, di conseguenza, la conoscenza sia del sistema di tassazione italiano, sia di quello americano: due responsabilità della gerarchia di classi *TaxCalculator*. *SalesOrder*, attraverso il metodo *GetTaxAmount()*, diventa così un collaboratore di *USTaxCalculator* e di *ItalianTaxCalculator*.



**Figura 2 - Collaborazione descritta mediante un diagramma di sequenza**

Tale collaborazione può essere descritta mediante il semplice diagramma di sequenza di **Figura 3** nel quale *SalesOrderController* invoca il servizio di calcolo delle tasse fornito dalla classe *SalesOrder*. Può essere interessante notare come la classe *SalesOrderController* non è a conoscenza del fatto che la responsabilità di conoscere la corretta tassa è fornita alla classe *SalesOrder* dalla gerarchia *TaxCalculator*. Tale gerarchia è incapsulata all'interno di *SalesOrder*. Questa soluzione rappresenta l'applicazione del pattern *Strategy* [6].

*“Sviluppare un modello dinamico significa individuare dei pattern di comunicazione tra gli oggetti che disciplinano il flusso di controllo, equilibrando la complessità del sistema.”*

### I ruoli e gli stereotipi suggeriscono le collaborazioni

Il ruolo di un oggetto può suggerire un certo tipo di collaborazione con gli altri oggetti del sistema. Tale ruolo viene ben espresso da un opportuno stereotipo associato alla classe. Alcuni stereotipi standard suggeriti dalla progettazione basata sulle responsabilità sono i seguenti:

- «information holder»;
- «structurer»;
- «service provider»;
- «controller»;
- «coordinator»;
- «interfacer».

Questi diversi ruoli sono caratterizzati da un altrettanto diverso tipo di collaborazioni. Nel seguito proviamo ad esaminarli brevemente uno ad uno.

«information holder»

Gli oggetti «information holder», ad esempio, presentano collaborazioni molto limitate il cui unico obiettivo è quello di acquisire le informazioni di cui essi sono responsabili. Non prendono decisioni rilevanti e non coordinano importanti attività.

«structurer»

Gli oggetti «structurer», invece, hanno la responsabilità di organizzare e strutturare le informazioni in gruppi di oggetti, mantenendo tra loro relazioni complesse. Alcuni oggetti devono essere gestiti, altri devono essere resi accessibili quando servono, altri ancora devono essere organizzati in pool di memoria. In generale un oggetto «structurer» prevede interazioni con i suoi collaboratori per ottenere ed organizzare i dati. Esso deve gestire questo tipo di connessioni e può dover mantenere delle proprietà di integrità concettuale tra un insieme di informazioni correlate.

«service provider»

Lo stereotipo «service provider» si distingue a sua volta dai precedenti in quanto caratterizza oggetti che forniscono funzionalità molto specifiche, altamente specializzate e tipiche del dominio dell'applicazione. Il tipo di collaborazioni che li contraddistingue è legato molto di più ad aspetti algoritmici o computazionali che ai dati e al modo in cui essi sono organizzati.

«controller»

Il quarto tipo di ruolo è quello degli oggetti «controller». Un controller prende delle decisioni e definisce il flusso di controllo dell'applicazione. Le collaborazioni tipiche di un controller possono essere di due tipi:

- acquisire le informazioni di cui si ha bisogno per prendere delle decisioni;
- invocare altri oggetti nella sequenza corretta di operazioni che sono necessarie per completare una particolare azione.

«coordinator»

Gli oggetti «coordinator», invece, sono maggiormente focalizzati nel mantenere connessioni con altri oggetti con l'obiettivo di passare loro informazioni oppure di reagire ad eventi, *delegando* ai collaboratori il compito di soddisfare le specifiche richieste. Essi, a differenza dei controller, non prendono *tutte* le decisioni "intelligenti", ma si basano sulla delegazione, favorendo uno stile di controllo maggiormente decentralizzato.

«interfacer»

L'ultimo stereotipo proposto rappresenta gli oggetti «interfacer». Questi oggetti rappresentano dei ponti che collegano parti disgiunte e indipendenti del sistema software. Sempre con riferimento alla **Figura 1**, qualora una collaborazione dovesse esistere tra oggetti collocati in layer diversi, sarebbe opportuno che essa avvenga per mezzo di interfacce (un esempio di oggetti «interfacer»), piuttosto che attraverso classi concrete.

Ragionare sull'assegnazione di questi ruoli alle classi del sistema può costituire una buona traccia di partenza per individuare potenziali collaborazioni.

### Ancora a proposito delle CRC-Card

In alcune puntate della rubrica Object-Oriented Design<sup>†</sup> abbiamo discusso delle strategie per identificare le classi, le responsabilità e le collaborazioni [1][2]. Alla luce di quegli articoli, possiamo ora capire come le CRC card siano un ottimo strumento di analisi, utile anche in una fase iniziale di progettazione. Esse, infatti, fanno convergere *in un unico posto* elementi fondamentali del modo di pensare object-oriented: il nome degli oggetti, lo stereotipo (e quindi il suo ruolo nell'architettura), le responsabilità sui dati e sulle operazioni, le collaborazioni (e quindi le dipendenze con altri oggetti). Personalmente non sono un "fanatico" :-)) di una metodologia o di un processo in particolare. La "buona" progettazione ad oggetti dipende non dipende dall'utilizzo di una singola tecnica. Tuttavia progettare bene non può prescindere dalla conoscenza e dall'applicazione ragionata di buoni principi di progettazione. Se in alcune situazioni risulta difficile (o innaturale) utilizzare il metodo CRC-Card, non c'è ragione di adottarlo ciecamente. Perdiamo pure il metodo, ma non perdiamo i principi generali che lo guidano (e che guidano la progettazione ad oggetti): progettare buone classi, assegnare buone responsabilità, implementare buone collaborazioni.

### Oltre le CRC-Card: UML

Il metodo CRC-Card è soprattutto un metodo di analisi. Tuttavia il lavoro iniziato con esso (o con qualsiasi altra tecnica di analisi che tenga in considerazione l'identificazione di classi, responsabilità e collaborazioni) può essere facilmente sfruttabile anche durante la fase di progettazione, in particolare se utilizziamo come *strumento di comunicazione* il linguaggio UML. UML è un linguaggio di modellazione che permette di rappresentare attraverso una serie di diagrammi una o più *viste* di un sistema software. Una vista costituisce una particolare prospettiva secondo la quale si osserva e si descrive il sistema (da un punto di vista esterno, user-oriented, oppure da un punto di vista interno, come accade per le viste strutturali che descrivono la struttura logica del sistema) [5]. Non tutti i diagrammi UML sono importanti allo stesso modo nelle varie fasi di un progetto. Durante il design risultano particolarmente utili il concetto di package e i diagrammi

di classe per descrivere classi e responsabilità, mentre per rappresentare le collaborazioni sono più indicati i diagrammi di interazione, come ad esempio i diagrammi di sequenza. La **Figura 2** e la **Figura 3** sono due esempi di diagrammi che potrebbero essere ottenuti durante una tipica sessione di brainstorming applicando il metodo CRC-Card. Il fatto di poter recuperare i risultati dell'attività di analisi durante il design non deve sorprendere: UML è stato pensato fin dall'inizio secondo la filosofia object-oriented proprio per ridurre la distanza tra analisi e design, fornendo un linguaggio (di modellazione, ma soprattutto di comunicazione) comune per analisti, progettisti e sviluppatori. Per questa ragione, l'utilizzo di diagrammi UML può essere affiancato all'impiego di altri metodi e anzi, se usato opportunamente, può servire a rendere il metodo stesso più incisivo ed efficace. Ancora una volta, l'invito è quello di utilizzare lo strumento che ritenete più espressivo per descrivere idee preliminari, concetti, parti di architettura, scenari d'utilizzo oppure componenti del sistema. Chiamatelo se volete "agile modeling". Ad ogni modo, sentitevi soprattutto liberi di sperimentare. Ogni strumento che ci aiuta a *pensare* e a *comunicare meglio* dà un valore aggiunto che viene ripagato nelle fasi successive del ciclo di vita del progetto software.

### Bibliografia

- [1] Baruzzo, A. – "Identificare le classi di un sistema object-oriented: il metodo delle CRC-Card", Computer Programming n°129, Novembre 2003 - Gruppo Editoriale Infomedia
- [2] Baruzzo, A. – "Oggetti e responsabilità", Computer Programming n°130, Dicembre 2003 - Gruppo Editoriale Infomedia
- [3] Bellin, David; Simone, Susan S. – "The CRC Card Book", Addison Wesley, 1997
- [4] Cockburn, Alistair – "Writing Effective Use Cases", Addison Wesley, 2001
- [5] Evitts, Paul – "A UML Pattern Language", Macmillan Technical Publishing, 2000
- [6] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M. – "Design Patterns", Addison Wesley, 1995
- [7] Maguire, Steve – "Writing Solid Code", Microsoft Press, 1993
- [8] Meyer, Bertrand – "Object-Oriented Software Construction 2/E", Prentice Hall, 1997
- [9] Szyperski, Clements – "Component Software: Beyond Object-Oriented Programming", Addison Wesley, 1998
- [10] Wirfs-Brock, Rebecca; McKean, Alan – "Object Design – Roles, Responsibilities, and Collaborations", Addison Wesley, 2003

Dr. Andrea Baruzzo

abaruzzo@computer.org

È laureato in Scienze dell'Informazione presso l'Università degli Studi di Udine.

Si occupa di ricerca, formazione e consulenza sia in ambito accademico, sia in ambito industriale. Le sue principali aree di interesse sono l'analisi, la progettazione e lo sviluppo di sistemi software ad oggetti (OOA/OOD/OOP), la qualità del software e le tecniche machine learning. È inoltre membro di IEEE Computer Society.

<sup>†</sup> La rubrica OOD viene regolarmente pubblicata dal 2003 sulla rivista Computer Programming, Gruppo Editoriale Infomedia