

Progettare oggetti responsabili*

Un oggetto non è solo un “guscio sintattico”: compie azioni e manipola informazioni autonomamente. Esso è cioè dotato di responsabilità che condizionano aspetti importanti di una classe come il nome, lo stereotipo e l'interfaccia

Dr. Andrea Baruzzo

Gli oggetti eseguono operazioni, manipolano informazioni e prendono autonomamente delle decisioni. In tal senso, essi sono responsabili! Cerchiamo quindi di capire che cosa intendiamo nella progettazione del software con il concetto di responsabilità, come si individuano e perché caratterizzare una classe in termini di esse produce un design migliore. Parlando di responsabilità, il momento è opportuno anche per introdurre una tecnica di progettazione che, pur avendo molti aspetti in comune con il metodo CRC-Card [2], è stata concepita più specificamente come strumento di design che di analisi. Stiamo parlando della cosiddetta progettazione ad oggetti “responsibility-driven”, o “responsibility-driven design”. Cercheremo infine di supportare la teoria con un esempio concreto, piuttosto semplice per ragioni di spazio, ma altresì sufficiente per acquisire un po' di familiarità con i meccanismi di ragionamento basati sulle responsabilità. Vedremo in particolare come anche per una classe poco complessa progettare una buona interfaccia possa richiedere un certo sforzo di astrazione e come ragionare sulle responsabilità aiuti a raggiungere l'obiettivo finale (un design migliore).

Classi e responsabilità

Tradizionalmente le classi sono state descritte come un costrutto (tipico ma non esclusivo) della programmazione ad oggetti, grazie al quale possiamo associare dati e operazioni in un unico modulo. Il concetto di classe costituisce inoltre la base sintattica sulla quale poggiano l'ereditarietà, il polimorfismo e, in alcuni linguaggi come il C++, anche la genericità (i template). Nonostante queste caratteristiche, scrivere del codice che fa uso di classi non significa necessariamente scrivere del buon codice orientato agli oggetti. Le classi non sono del semplice “zucchero sintattico”, utile magari per evitare di passare lunghe liste di parametri ad una funzione (nella maggior parte dei linguaggi di programmazione le classi sono dotate di un puntatore implicito *this* che non serve né dichiarare, né scrivere esplicitamente per accedere ai dati interni). Un oggetto, quindi, non è un contenitore passivo di informazioni e di operazioni: è piuttosto un'unità concettuale del sistema, con un ruolo ben definito all'interno dell'architettura generale. Tale ruolo viene definito in base al nome della classe, allo stereotipo associato ad essa e alla sua interfaccia, strettamente correlata alle responsabilità che l'oggetto stesso assume rispetto al resto del sistema. Una *responsabilità* è quindi uno statement generale sull'oggetto, ossia un'affermazione, un'asserzione o una proprietà che attribuisce all'oggetto un compito specifico [8]. Questi compiti possono riguardare tre aspetti principali:

1. le *azioni* che un oggetto può eseguire (responsabilità sulle operazioni);

2. la *conoscenza* che un oggetto mantiene sui dati (responsabilità sui dati);
3. le *decisioni* più importanti che un oggetto prende e che possono influenzare altri oggetti.

I primi due aspetti (azioni e conoscenza) sono evidenti a livello sintattico. Non è possibile scrivere o anche solamente utilizzare una classe senza avere una minima consapevolezza dei concetti di dato e di operazione. Quello che, semmai, non è affatto banale garantire consiste nel preservare (durante l'evoluzione del progetto) alcune buone proprietà auspicabili per qualsiasi classe, come ad esempio la semplicità, la riusabilità, la comprensibilità, la manutenibilità.

Il terzo aspetto (le decisioni), per contro, è spesso del tutto ignorato dai programmatori, molto probabilmente a causa di un retaggio dei metodi di programmazione top-down che scompongono i sistemi software in procedure e sottoprocedure. In tali sistemi, i dati sono mantenuti separati dalle funzioni le quali possono accedere ad essi in modo indiscriminato. Questa caratteristica produce tipicamente delle architetture centralizzate, caratterizzate da pochi moduli complessi (in corrispondenza delle procedure di più alto livello) e da una moltitudine di moduli minori aventi funzionalità di puro supporto. Si tratta chiaramente di un esempio di cattiva distribuzione dell'intelligenza. Le architetture object-oriented, per contro, sono organizzate in termini di comunità di oggetti. Una caratteristica fondamentale di tali comunità consiste nella loro struttura: esse mantengono delle relazioni esplicite tra gli oggetti, i quali a loro volta cooperano in modo disciplinato, secondo dei pattern di comunicazione predeterminati, facilmente localizzabili, e ben documentati. Le decisioni importanti che un sistema deve prendere sono distribuite a diversi livelli tra gli oggetti in base al loro ruolo. Tutte queste considerazioni spostano l'attenzione dal livello sintattico, orientato al compilatore e alla macchina fisica sottostante, ad un livello più astratto, orientato al problema e all'essere umano che deve leggere, comprendere, sottoporre a debug e testing, correggere ed infine estendere il codice nel tempo. Garantire che le classi abbiano un *adeguato livello di astrazione*, una *buona integrità concettuale* e un' *interfaccia semplice* è altrettanto indispensabile dell'assicurarsi che esse non causino degli errori di compilazione.

Le responsabilità: un collante tra il problema e la soluzione

Le responsabilità forniscono alla classe un livello semantico e una certa astrazione dall'implementazione. Il programmatore mostra spesso una tendenza marcata nel progettare una classe in termini dei meccanismi di funzionamento interni. Anziché caratterizzare un oggetto mediante pochi fondamentali compiti, un errore comune consiste nell'eccedere prematuramente ad elencare tutti i possibili attributi oppure tutti i possibili metodi di una classe.

* Questo documento è una versione estesa e rivista dell'articolo “Oggetti e responsabilità”, pubblicato sulla rivista *Computer Programming*, rubrica Object-Oriented Design, n°130, Dicembre 2003 – Gruppo Editoriale Infomedia. Ultimo aggiornamento: luglio 2008.

Questa è la prospettiva sbagliata per osservare un oggetto ed attribuirne le responsabilità. Il giusto livello di investigazione consiste nel mantenere una *prospettiva concettuale* che descrivere ciò che la classe deve svolgere per risolvere un problema, astruendo dal modo in cui lo fa [3]. Secondo tale prospettiva, pensare subito alla successiva fase di codifica è sconsigliabile perché porterebbe ad una distorsione dei concetti, adattandoli al “come” saranno implementati e perdendo di vista il “pezzo di mondo” che si vuole modellare [4]. Significa inoltre prendere prematuramente un sacco di decisioni orientate alla macchina (e non al vero problema da risolvere)[†].

Viste in questa chiave, le responsabilità costituiscono una sorta di collante tra due mondi distinti: quello del problema e quello della soluzione. Eccedere nei dettagli significa perdere di vista il *problema essenziale* da risolvere. Nel loro libro “*The CRC Card Book*”, Bellin e Simone citano un antico racconto cinese che mi sembra molto appropriato per parlare dei concetti di responsabilità e di problema essenziale. Il racconto narra di quattro contadini che provengono da un campo e che, per entrare in paese, devono oltrepassare un muro di cinta dotato di un unico punto di accesso costituito da una porta chiusa da un pesante lucchetto. Gli uomini (ovviamente) non hanno con sé la chiave, né particolari attrezzi che possono essere apparentemente utili per aprire la porta. Tre di loro provarono allora a servirsi di tutto ciò che trovarono sul posto. Tirarono contro il lucchetto delle pietre, provarono a dar fuoco alla porta; infine provarono anche a scagliarsi fisicamente contro nel tentativo di abbatterla. Solo in quel momento i tre contadini videro il quarto uomo raccogliere un bastone di bambù, prendere la rincorsa e scavalcare il muro di cinta con un salto. Aprire il lucchetto o sfondare la porta costituivano solamente una sorta di camuffamento del vero problema. Il problema essenziale era invece quello di passare al di là del muro. La morale del racconto (rivisto in chiave informatica) è che *i progetti software diventano spesso eccessivamente complessi per il semplice motivo che ci si concentra troppo sui dettagli, perdendo di vista le funzionalità essenziali o la visione globale*. Pensare fin dalle prime fasi di design alle responsabilità permette di introdurre un elemento di astrazione in un processo orientato verso il dettaglio, fungendo da catalizzatore per progettare la soluzione in termini del problema giusto da risolvere. Nella parte restante dell’articolo ci serviremo di un esempio concreto per capire come si assegnano le responsabilità ad una classe.

Responsabilità sui dati

Le responsabilità sui dati descrivono le informazioni *essenziali* che un oggetto manipola. Consideriamo la seguente struttura dati che descrive l’entità geometrica punto in due dimensioni:

```
struct Point2d {
    double x;
    double y;
};
```

Trasformare questo frammento di codice C in termini di classi è un compito molto facile. Se ci limitiamo a considerare la classe come

un costrutto prettamente sintattico, ci basta riscrivere la struttura dati nel seguente modo:

```
class Point2d {
    // versione “irresponsabile” sui dati
public:
    double x;
    double y;
};
```

La classe non è solamente un costrutto sintattico del linguaggio di programmazione: dal punto di vista della progettazione rappresenta una (parziale) implementazione di un tipo di dato astratto [6]. In quanto tale, essa dovrebbe fornire le uniche operazioni in grado di alterare il valore dei suoi attributi. La classe *Point2d* non è un buon esempio di *classe responsabile dei propri dati*. Esponendo gli attributi nella propria interfaccia pubblica, essa di fatto compromette il livello di protezione fornito dall’information hiding e dall’incapsulamento. Un qualsiasi oggetto cliente può accedere direttamente allo stato interno di *Point2d*, alterandolo. Tale oggetto condivide con *Point2d* le responsabilità di gestione delle variabili *x* e *y*. Se il numero di tali oggetti è superiore a due o tre, avremo un design in cui nessun oggetto è veramente responsabile di gestire il proprio *stato interno*. Pensiamo inoltre alla possibilità che si voglia passare da una rappresentazione in coordinate cartesiane, come quella proposta nell’esempio, ad una rappresentazione del punto in coordinate polari. Se l’unica classe responsabile di conoscere il formato delle coordinate è la classe *Point2d*, allora basterà riscrivere i metodi che dipendono da tale formato, limitando notevolmente l’impatto sulle altre classi utilizzatrici. Pensate, invece, a quante altre classi il programmatore della “versione irresponsabile” di *Point2d* dovrebbe quantomeno *esaminare* per apportare eventuali correzioni. In conclusione, la progettazione ad oggetti responsibility-driven fornisce un implicito messaggio al progettista: *le responsabilità sui dati costituiscono un invito esplicito ad incapsulare* gli attributi di una classe.

```
class Point2d {
private:
    double x;
    double y;
public:
    // public interface
};
```

Responsabilità sulle operazioni

Il passo successivo consiste nel progettare l’interfaccia pubblica della classe, ossia stabilire le sue operazioni. Le *responsabilità sulle operazioni* descrivono i compiti principali ai quali la classe *assolve* e i servizi che essa fornisce alle altre classi; in altre parole ciò che essa è in grado di fare. Ancora una volta, se si pensa alla classe esclusivamente da un punto di vista implementativo, è facile finire col trovarsi un’interfaccia ingombrante e scarsamente coesa. Riprendendo l’esempio di *Point2d*, il programmatore potrebbe dotare la classe di un insieme troppo ampio di funzionalità, semplicemente perché ciascuna di esse “ha bisogno” di manipolare le coordinate di un punto.

[†] Il lettore può facilmente intravedere in questo schema un sintomo tipico della prematura ottimizzazione, spesso responsabile dell’introduzione nei progetti di sovrastruttura, complessità accidentale, e costi di manutenzione dovuti a codice poco comprensibile.

```

class Point2d {
// versione "irresponsabile" sulle operazioni
private:
// attributes
public: // public interface
// operators
Point2d operator+ (const Point2d& p);
Point2d operator- (const Point2d& p);
...
// classifiers
int Classify (const Point2d& p1, const
Point2d& p2) ;
int Classify (const Edge3d& e) ;

// operations in polar coordinates
double PolarAngle (void);
double Length (void);
...
// geometrical operations
double Distance (const Point2d& p);
double Distance (const Edge2d& e);
...
};

```

Quest'interfaccia è molto discutibile poiché accentra troppe responsabilità nella classe punto. Il programmatore che ha scritto questo codice ha attribuito le responsabilità alla classe concentrandosi molto probabilmente al livello dei singoli attributi e delle operazioni che si possono fare su di essi. Date le coordinate di un punto, è sicuramente possibile eseguire dei confronti o delle classificazioni oppure calcolare delle distanze rispetto ad altri oggetti geometrici. Tuttavia nell'attribuire le responsabilità vanno fatte delle considerazioni anche relativamente all'integrità concettuale di un oggetto. In particolare è importante assicurare che *l'interfaccia pubblica di ogni classe descriva un livello di astrazione uniforme*. La classe proposta è discutibile perché fonde in un'unica interfaccia operazioni di più basso livello (gli operatori aritmetici o le operazioni sulle coordinate polari) con operazioni di livello chiaramente più astratto (le classificazioni o la valutazione delle distanze). Troppe responsabilità rendono un oggetto simile ad una "god class" [7]. Si tratta di una cattiva scelta di design poiché la conseguenza principale è proprio la creazione di architetture centralizzate già menzionata in apertura di articolo: poche classi, molto complesse e difficili da mantenere, che svolgono la gran parte dei compiti, lasciando i dettagli minori ad una collezione di classi piuttosto banali (tipicamente costituite da soli metodi accessori *get* e *set*).

"Le responsabilità sono affermazioni che riguardano i dati, le operazioni e le decisioni di un oggetto. Esse costituiscono un invito esplicito ad incapsulare lo stato interno di un oggetto."

Responsabilità e "intelligenza" di una classe

Ogni classe è dotata di un livello di "intelligenza" che è importante definire fin dall'inizio, nella scelta del nome e nel posizionamento della classe nell'architettura del sistema. Riprendendo l'esempio della classe punto, la prima domanda che dobbiamo porci è rivolta a capire quali siano i suoi compiti essenziali e concentrarci su questi per attribuire le responsabilità. Un buon test consiste nel provare ad escludere una responsabilità e vedere se la classe può ancora essere implementata. In questo modo abbiamo un potente filtro che ci aiuta a progettare *interfacce minimali* in grado di descrivere l'essenza di una classe. Un punto, ad esempio, deve essere responsabile di gestire le proprie coordinate, il loro formato ed, eventualmente, dei metodi di conversione per passare da un formato ad un altro. L'aggiunta di altre responsabilità va valutata con molta attenzione: dopotutto il concetto di punto è molto semplice. Più che fornire servizi e prendere decisioni complesse autonomamente, un punto costituisce uno dei "mattoni di base" con i quali costruire oggetti geometrici più complessi. Potremmo esplicitare questa caratteristica di minimalità presente nel dominio del problema attribuendo alla classe Point2d lo stereotipo di «information holder», proprio per enfatizzare il suo ruolo nel sistema (la soluzione): gestire informazioni (le coordinate in questo caso). Tali considerazioni sono fondamentali per distribuire l'intelligenza (ed in particolare la logica di controllo) tra i componenti del sistema, anziché centralizzarla in pochi oggetti. La decentralizzazione promuove un migliore controllo della complessità che produce, a sua volta, progetti più robusti e manutenibili; in altre parole, progetti migliori. Volendo infine utilizzare uno schema di colori per descrivere il design espresso dal modello UML, una classe punto sarebbe probabilmente un'ottimo candidato, in quanto ad information holder, per il colore blu, riservato tipicamente alle classi descrittive (data-oriented) e sostanzialmente passive (si veda [1]).

Perseguendo l'obiettivo di meglio distribuire l'intelligenza del sistema, l'attribuzione di un ruolo passivo alla classe punto e l'individuazione di funzionalità di alto livello come la classificazione forniscono l'opportunità di creare nuove classi dotate di una maggiore intelligenza. Riconoscere questi "salti d'astrazione" non è comunque sempre facile. Una considerazione ci può essere d'aiuto. La gestione delle coordinate e del loro formato, essendo dei dettagli interni, mantiene la classe Point2d auto-contenuta e fortemente coesa. I concetti di distanza e di relazione geometrica, invece, non rientrano direttamente nella definizione di punto. Il calcolo della distanza, ad esempio, assume la conoscenza di altre entità geometriche (linee, segmenti, oggetti 3D generici) e di un modello matematico di riferimento per il suo calcolo (la distanza euclidea, per esempio). Confinare queste responsabilità a livello della classe Point2d significa esporre funzionalità d'alto livello e *assunzioni di implementazione* nell'interfaccia sbagliata.

Una soluzione migliore consiste nello spostare questa funzionalità ad un livello più alto, creando ad esempio la nuova classe *ObjectToObjectDistance*, responsabile del calcolo delle distanze tra oggetti geometrici. Come il nome fa intuire, la nuova classe è caratterizzata da un'interfaccia simile ad un'API, contenente tutti gli algoritmi per il calcolo delle distanze tra le entità geometriche del motore grafico. Una simile classe può essere caratterizzata dallo stereotipo «service provider» poiché il suo ruolo è proprio quello di fornire agli altri oggetti un servizio. Il salto d'astrazione tra i due livelli può essere espresso architetturelmente identificando due package distinti: un package *Core*, che contiene

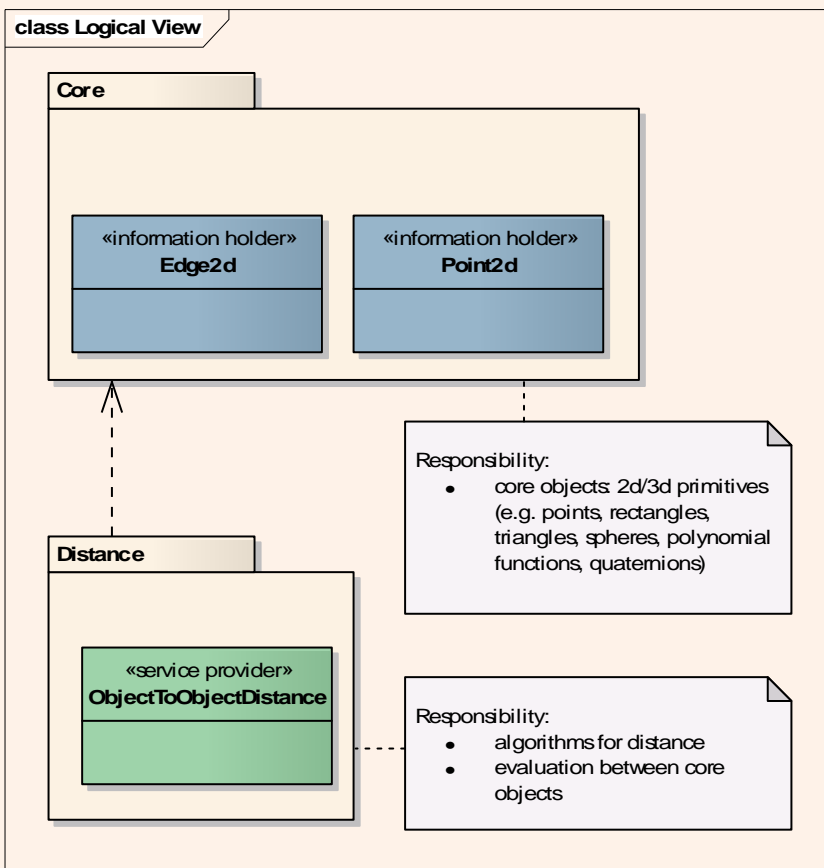


Figura 1 - distribuzione delle responsabilità tra classi di tipo information holder e classi di tipo service provider

gli oggetti grafici primitivi (tra cui anche i punti), e un package *Distance*, che dipende dal package *Core* per il calcolo delle distanze (Figura 1).

Un effetto collaterale, ma molto importante, di tutto questo ragionare sulle interfacce delle classi e sul loro livello d'astrazione consiste nel evidenziare già dalle fasi iniziali di design delle opportunità di collaborazione tra gli oggetti. Ciò risulta utile sia per introdurre nel modello ad oggetti anche degli elementi dinamici, esplicitabili mediante dei diagrammi di sequenza in UML, sia per chiarire alcune dipendenze tra le classi.

Responsabilità e incapsulamento

Abbiamo visto, quindi, che una classe responsabile di gestire i propri dati costituisce per il progettista un chiaro invito all'incapsulamento, ossia a rendere privati gli attributi. Ancora una volta, però, è necessario andare oltre l'aspetto formale, sintattico, per comprendere appieno questo principio di programmazione.

Una pratica molto comune (e non solo tra gli sviluppatori provenienti da linguaggi di programmazione strutturati come il C) consiste nel rendere privati i dati, ma anche nel dotare sistematicamente l'interfaccia della classe di metodi accessori per ciascuno di questi dati. Cerchiamo di ragionare sull'esempio della classe *Point2d* per capire se si tratta di una buona scelta di design. Riportiamo un breve frammento della classe con la dichiarazione delle tre coordinate e i relativi metodi *get* e *set*.

```
class Point2d {
private:
    double x;
    double y;
public:
    ...
    double GetX(){return x;}
    double GetY(){return y;}
};
```

In generale, la presenza di metodi *get* e *set* indebolisce l'incapsulamento della classe perché espone direttamente gli attributi interni di una classe ai suoi utilizzatori. Di solito si tratta quindi di una pessima scelta di design poiché l'effetto finale non è molto diverso dal dichiarare gli stessi attributi pubblicamente. La presenza di molti metodi accessori implica la violazione del principio secondo cui *i dati e le operazioni sui di essi dovrebbero essere associati in un unico modulo: la classe*. In particolare va considerata sempre con molto sospetto la scelta di esporre attraverso i metodi accessori interi oggetti. Esistono tuttavia quattro importanti eccezioni che costituiscono ragionevoli violazioni di questa linea guida, di seguito elencate:

1. i metodi accessori definiscono il ruolo di "service provider" della classe (in questo caso essi formano un livello d'astrazione uniforme rispetto agli altri metodi presenti nell'interfaccia);

"Ragionare sulle responsabilità aiuta a progettare interfacce essenziali. Troppe responsabilità rendono un oggetto simile ad una "god class", chiaro retaggio di un design centralizzato il cui livello di accoppiamento complessivo è molto alto."

2. i dati forniti dai metodi accessori non provengono da più classi (in rispetto della legge di Demeter [5]);
3. la classe che contiene i metodi accessori non possiede invarianti sugli attributi che espone;
4. i metodi accessori costituiscono l'interfaccia di classi utilizzate esclusivamente per la comunicazione tra l'interfaccia utente (GUI) e il modello ad oggetti sottostante (ad esempio i JavaBean).

La classe *Point2d* rientra sicuramente nei primi tre casi, per cui dotarla di metodi accessori è da ritenersi ancora una ragionevole scelta di design. Possiamo subito renderci conto, infatti, che il compito essenziale della classe come fornitore di servizi consiste proprio nel fornire le coordinate di un punto. I suoi metodi accessori, inoltre, non espongono altri oggetti: forniscono (o impostano) esclusivamente un valore, ossia un tipo di dato primitivo. Particolarmente interessante, infine, il terzo punto: *Point2d* non richiede invarianti di classe (si presuppone che le coordinate possano assumere un qualsiasi valore compreso nell'intervallo di definizione del tipo *double*). Se almeno una di queste tre proprietà venisse meno, allora inserire i metodi accessori nell'interfaccia di *Point2d* potrebbe essere piuttosto discutibile.

Il messaggio implicito in questo esempio è che i metodi accessori vanno utilizzati solo se strettamente indispensabile: non si tratta assolutamente di una scelta "automatica" che si compie per pigrizia o per rendere l'interfaccia più funzionale. Il costo dei metodi accessori va analizzato ovviamente in chiave evolutiva poiché essi creano accoppiamento tra gli utilizzatori di una classe e i suoi dettagli implementativi interni.

Responsabilità e information hiding

L'esempio della classe *Point2d* è stato utile per capire come il ragionare sulle responsabilità di una classe permette di produrre un design migliore. Abbiamo visto come la scelta accurata del nome, dello stereotipo e del livello d'astrazione dell'interfaccia di una classe siano elementi fondamentali che guidano il processo di attribuzione delle responsabilità. Questo ci permette di tenere sotto controllo il problema delle interfacce grasse e delle god class, individuando un'interfaccia minimale ed essenziale. Abbiamo visto, inoltre, come le responsabilità sui dati non vadano implementate indebolendo l'incapsulamento della classe. Concludiamo con un ultimo aspetto da chiarire riguardante l'occultamento delle informazioni. Che cosa succede se cambiassimo il formato di rappresentazione delle coordinate di un punto? Combinata all'utilizzo di metodi accessori, questa scelta non comporta forse un inevitabile violazione del principio di occultamento dei dati (information hiding)? Possiamo considerare queste domande come un'opportunità di verificare se esporre lo stato interno attraverso dei metodi accessori non mina la robustezza della classe. Per rispondere dobbiamo tenere presente una considerazione importante. *Una responsabilità sui dati (unita ad un corrispondente metodo accessorio) non implicano necessariamente la presenza di un attributo di classe.* La classe *Point2d*, infatti, potrebbe essere modificata in qualunque momento, sostituendo le coordinate cartesiane in coordinate polari, pur mantenendo i metodi accessori *getX* e *getY*.

```
class Point2d {
// una possibile versione responsibility-
// driven
private:
    double theta;
    double length;
public:
    double GetPolarAngle(){return theta;}
    double GetLength(){return length;}
    double getX(){return
    GetLength()*cos(GetPolarAngle());}
    double getY(){return
    GetLength()*sin(GetPolarAngle());}
    ...
};
```

In questo caso i metodi *getX* e *getY* calcolano "al volo" le coordinate nel formato cartesiano, senza rompere il codice degli oggetti clienti. Dal punto di vista di tali oggetti, infatti, la versione appena scritta della classe *Point2d* fornisce un'interfaccia di servizio che non svela quale sia il formato interno scelto dal programmatore per rappresentare le coordinate del punto. Senza accedere all'implementazione dei metodi, l'unica cosa che si conosce sono i suoi servizi pubblici, ossia il "cosa" fa la classe (fornire le coordinate sia in formato cartesiano, sia in quello polare), non il "come" lo fa, che rimane un dettaglio implementativo! Questo è un'importante principio di progettazione responsibility-driven. *Implementare una responsabilità sui dati non significa limitarsi a restituire l'attributo: significa capire l'informazione essenziale da fornire agli altri oggetti del sistema e costruirci attorno un servizio pubblico.* Se questa informazione è poi un valore calcolato al volo sulla base di altri oggetti, oppure se è un attributo da restituire, si tratta una decisione implementativa che non deve riguardare gli oggetti clienti.

Conclusione

Le classi secondo la prospettiva object-oriented rappresentano astrazioni sui concetti del dominio del problema. Esse dovrebbero esporre un'interfaccia pubblica essenziale e nascondere la loro complessità all'interno di metodi e attributi. Progettare le classi sulla base delle loro responsabilità permette di raggiungere più facilmente questi obiettivi. Per scrivere delle classi robuste e manutenibili, in particolare, è necessario dotare gli oggetti di responsabilità che non indeboliscano la loro integrità concettuale e che, possibilmente, distribuiscono l'intelligenza del sistema, anziché centralizzarla.

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.

Bibliografia

- [1] Baruzzo, Andrea; Pescio, Carlo. – *“Progettare con UML e il colore: facciamo parlare la struttura”*, Computer Programming n°145, Aprile, Gruppo Editoriale Infomedia, 2005
- [2] Bellin, David; Simone, Susan S. – *“The CRC Card Book”*, Addison Wesley, 1997
- [3] Booch, Grady – *“Object-Oriented Analysis and Design with Applications 2/E”*, Addison Wesley, 1997
- [4] Damiani, Ernesto; Madravio, Mauro – *“UML Pratico con Elementi di Ingegneria del Software”*, Addison Wesley, 2003
- [5] Lieberherr, Karl J. – *“Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns”*, PWS Publishing, 1995
- [6] Meyer, Bertrand – *“Object-Oriented Software Construction 2/E”*, Prentice Hall, 1997
- [7] Riel, Arthur J. – *“Object-Oriented Design Heuristics”*, Addison Wesley, 1996
- [8] Wirfs-Brock, Rebecca; McKean, Alan – *“Object Design – Roles, Responsibilities, and Collaborations”*, Addison Wesley, 2003