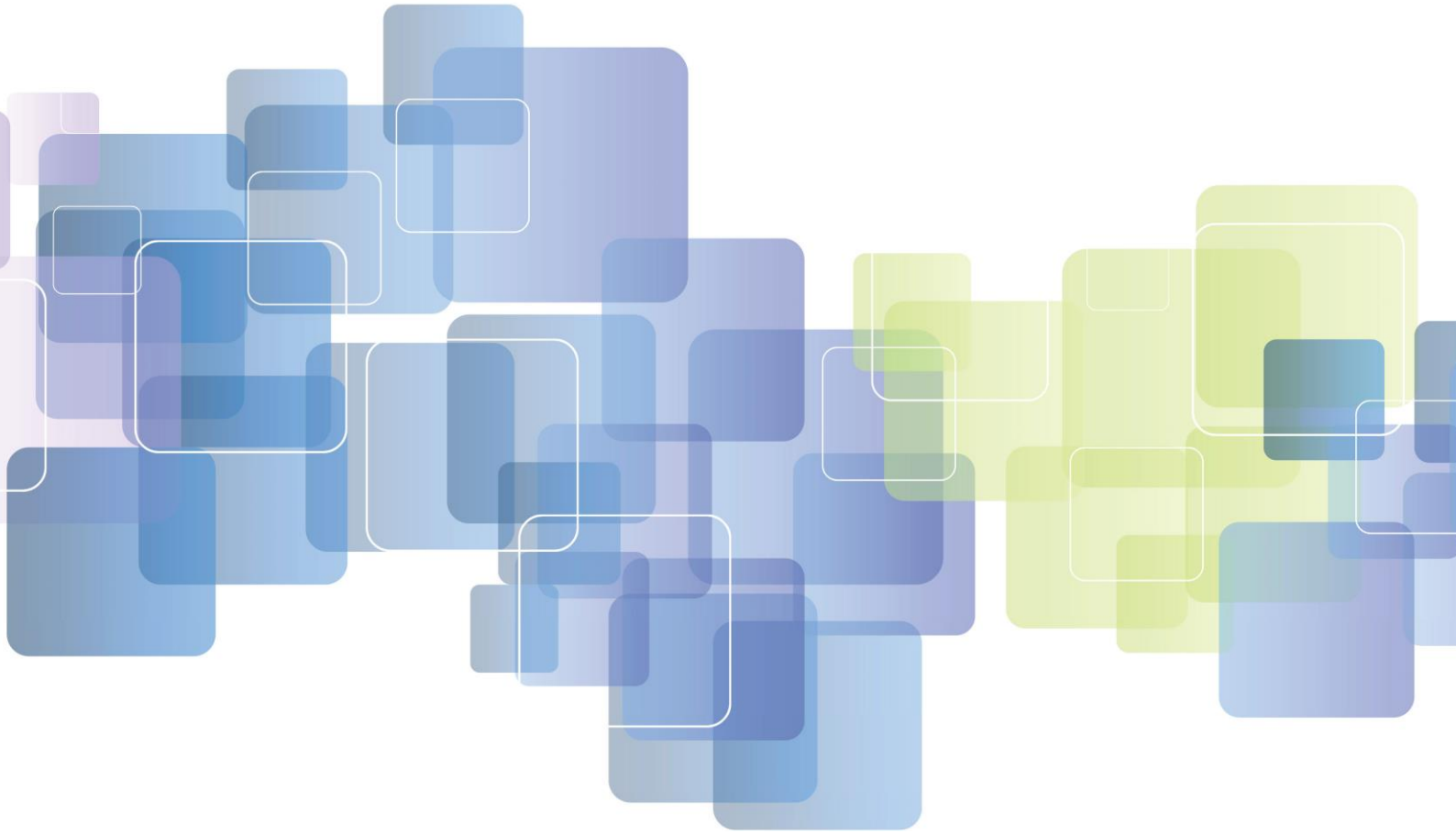
TecnetDati

Alle radici del disegno OO: i favolosi anni ' 70

Emilio C. Porcelli



Manutenibilità e Riusabilità. La posta in gioco è elevata e Meilir Page-Jones ci insegna come fare per conquistarla.

Chi l'ha detto che per fare effetto una medicina dev'essere necessariamente amara? E che un libro da cui trarre elementi utili per il proprio lavoro dev'essere per forza noioso e concettoso? Ma buona parte dei testi di argomento informatico che conosco è noiosa e concettosa. Non tutti, per fortuna! Dei tanti capitoli del gran libro dell'Ingegneria del Software quello sul Disegno Strutturato è senza dubbio uno dei più sostanziosi, ma anche dei meno digeribili. Solo all'apparenza, però. La "Practical Guide to Structured Design" di Meilir Page-Jones pubblicato da Yourdon Press/Prentice Hall nel 1980 e poi più volte ristampato, pur senza sacrificare nulla al rigore dell'esposizione, è infatti uno dei libri cosiddetti "tecnici" più godibili e ricchi di humour che mi sia mai capitato di leggere. Larry L. Constantine, che è uno dei padri fondatori del Disegno Strutturato oltre che autore con Ed Yourdon di un classico come "Structured Design" (Yourdon Press, 1975), racconta così il suo primo incontro con Page-Jones: "Quando - era il 1989 - venni a sapere che noi due avremmo parlato alla stessa conferenza, mi diedi subito da fare per incontrare questo autore che aveva saputo cogliere così bene la sostanza del Disegno Strutturato da trasformarlo in qualcosa di più utile per chi pratica la programmazione. Lui invece, come feci presto a scoprire, guardava con una certa apprensione alla prospettiva dell'incontro, dopo tutto il suo libro aveva venduto molte più copie del mio. Almeno fino a quando non ebbi modo di chiarirgli quanto quel libro mi avesse divertito e come fossi ben contento che avesse saputo trasmettere a un così gran numero di persone il messaggio del Disegno Strutturato". Da quell'incontro è nato un sodalizio che dura tutt'ora. Meilir Page-Jones è sulla breccia da un bel po' di anni, ma come Larry L. Constantine, Ed Yourdon e tanti altri esponenti della vecchia scuola, non è rimasto ancorato alle posizioni di un tempo. Lo abbiamo infatti incontrato una sera dello scorso novembre a Roma dove ha svolto per Technology Transfer un seminario sul Disegno Object-Oriented.

CWI - Oggi la comunità informatica guarda all'incalzare degli Oggetti con occhi spesso assai diversi: chi con fiducia ed entusiasmo, chi con sufficienza o incredulità...

Meilir Page-Jones - E' vero. Anche tra gli informatici troviamo dei "Reazionari" e il loro motto è "Niente di nuovo sotto il sole". O per dirla in altro modo: "Nel campo del software non è accaduto nulla di veramente significativo a partire dagli Anni '60. In tutto questo gran parlare di Oggetti non c'è una sola cosa che non si facesse o non si sapesse già. E' cambiato solo qualche nome".

CWI - Ma ci sono anche i barricaderi, i "Rivoluzionari".

MPJ - E il loro motto è "Non c'è nulla di quanto si sapesse prima degli Anni '80 di cui oggi valga la pena di occuparsi".

CWI - E poi ci sono i "Confusionari", ma quelli è meglio lasciarli perdere. In panorama tanto variegato Lei da che parte sta?

MPJ - Nè con gli uni, nè con gli altri. Io sto nel mezzo, sto dalla parte degli Evoluzionisti. Ai Reazionari rimprovero l'ostinazione a voler giocare in difesa. In fondo posso anche capirli. Sarebbe veramente bello infatti se nel mondo del software non accadesse mai nulla di nuovo! Così, imparate quelle quattro cose, uno potrebbe vivere di rendita.

CWI - E i Rivoluzionari?

MPJ - Sono dei pittoreschi estremisti che finiranno prima o poi col pentirsi del disdegno con cui guardano a un passato da cui avrebbero invece molto da imparare. I progressi nel campo del software, così come in qualsiasi altro ramo del sapere, non sono il prodotto di tutta una serie di rivoluzioni nichiliste, sono piuttosto il frutto di un progressivo accumularsi di conoscenze ed esperienze. E' stata e sarà Evoluzione, non Rivoluzione.

Impariamo dal passato

CWI - Poniamoci allora in un'ottica Evoluzionistica, che tra l'altro oggi sembrerebbe ben accetta anche in Alto Loco. Mettiamola già così. Se dall'essere primigenio si è giunti fino all'uomo passando se mai attraverso la scimmia - Simula è il nome del primo linguaggio genuinamente OO, ma è una semplice assonanza -, qual è stato il punto di partenza per il Disegno OO?

MPJ - Sicuramente la teoria dei Tipi di Dato Astratti (ADT) elaborata da Barbara Liskov nei primi Anni '70. Un ADT è un tipo di dato che consiste di un insieme di valori e di operazioni associate a questi valori e la cui definizione esterna, così come si presenta a chi lo usa, è indipendente dalla sua rappresentazione interna o implementazione. Le classi di oggetti non sono altro che implementazioni di ADT. Dopo gli ADT sono venuti il polimorfismo, l'ereditarietà e mille altre cose ancora. Ma per il Disegno OO gli ADT rappresentano il vero punto di partenza.

CWI - Ma non il solo. Sfogliando il suo ultimo libro che si intitola "What Every Programmer Should Know About Object-Oriented Design" (Dorset House, 1995) mi sono imbattuto in due termini, Coesione e Coupling, di cui i Rivoluzionari forse non hanno mai sentito parlare, ma che per uno vecchio del mestiere come me hanno un delizioso sapore Anni '70. Vuol dire che gli antichi princìpi del Disegno Strutturato possono essere applicati con profitto anche al Disegno delle applicazioni OO dei nostri giorni?

MPJ - E come se si applicano! Naturalmente vanno rivisitati per tener conto di meccanismi tipici dell'OO come l'incapsulazione e il principio di information hiding che, guarda caso, risalgono anche loro agli Anni '70. Allora, con l'esplosione dell'informatica nelle aziende, ci si comincio a preoccupare concretamente, oltre che di disciplinare i processi di produzione del software, anche della manutenibilità del software prodotto. Il Disegno Strutturato nasce appunto in risposta a queste preoccupazioni. Ma la manutenzione dei cosiddetti "legacy system" si è rivelata un problema ben più scottante di quanto non si pensasse vent'anni fa e ci si è resi conto troppo tardi che l'applicazione sistematica dei princìpi del Disegno Strutturato avrebbe potuto far molto per ridurlo a proporzioni più accettabili. Oggi la storia rischia di ripetersi: se trascureremo i princìpi del Disegno OO anche la manutenzione delle nuove applicazioni finirà col trasformarsi in un incubo. Senza contare che la manutenibilità non è l'unica posta in gioco. C'è anche la riusabilità, e anche qui Coesione e Coupling possono dare una mano.

CWI - Parliamo allora di Coesione. Come va interpretata in un contesto di Disegno OO?

MPJ - Nel mio libro la definisco "la misura della interrelazione tra i metodi nell'interfaccia esterna di una classe". Oppure, in parole più semplici, "il modo in cui una classe si dimostra coerente nel- l'implementare un ADT". Una classe poco coesa (ed è un guaio!) presenta un insieme di metodi che hanno ben poco a che vedere gli uni con gli altri. Mentre in una classe a coesione elevata (ed è un bene!) tutti i metodi contribuiscono all'astrazione del tipo implementato dalla classe.

CWI - E in pratica?

MPJ - Tre esempi potranno aiutarci a capire meglio

I Esempio: le provvigioni di Mary

MPJ - Facciamo il caso di un gruppo di venditori dei quali alcuni, come Fred, sono pagati a provvigioni e altri, come Mary, a stipendio fisso. Immaginiamo allora di definire una classe VENDITORE della quale gli oggetti Fred e Mary siano due istanze. E immaginiamo anche di voler inviare alla nostra classe due messaggi con i quali chiedere che vengano calcolate le provvigioni di vendita dovute a Fred e a Mary: 1) fred.calcola-provvigione; 2) mary.calcola-provvigione; Ma di questi messaggi solo il primo ha senso, Mary infatti non viene pagata a provvigioni. E a questo punto attenti a non cadere in una certa trappola.

CWI - Quale trappola?

MPJ - Quella di pensare che a Mary spetti una provvigione di importo zero. Sarebbe una sciocchezza! Lo zero è infatti un valore aritmetico ben definito, mentre per Mary, che lavora a stipendio fisso, la provvigione è invece un qualcosa di "indefinito", di "non applicabile", e cioè un valore nullo.

CWI - Perché, allora non inserire nella classe VENDITORE un bel flag e cioè una variabile booleana che con i suoi valori ci dica, per ogni sua istanza, se viene o meno pagata a provvigioni?

MPJ - Perchè in questo modo sarebbe necessario rimaneggiare il metodo venditore::calcola-provvigione aggiungendo almeno quella IF che ci eviti di stampare sul cedolino di Mary, in corrispondenza della voce "provvigioni", una sfilza di zeri o peggio ancora qualche simbolo incomprensibile. Funzionerebbe, ma non è certamente la soluzione di Disegno migliore. Renderebbe infatti un po' più complessa la struttura del nostro metodo e di sicuro non contribuirebbe alla sua manutenibilità e riusabilità. Ma il vero problema è un altro. Non possiamo pretendere di trattare Fred e Mary come se appartenessero alla stessa classe. Sarebbe una forzatura o, per dirla in termini di Disegno OO, un caso di cattiva Coesione. Io lo chiamo di "commistione delle istanze" (mixed istances Cohesion).

CWI - E a questo punto?

MPJ - A questo punto trovato il male, trovato il rimedio: basta sottodefinire la classe VENDITORE in due sottoclassi, VENDITORE A PROVVIGIONI e VENDITORE A STIPENDIO FISSO. Così il metodo calcola-provvigione sarà membro esclusivo della sottoclasse VENDITORE A PROVVIGIONE e non ci troveremo più di fronte a una palese anomalia: un metodo che, pur facendo parte della definizione dell'ADT implementato nella classe VENDITORE, non si applica a tutte le istanze della classe stessa.

II Esempio: una classe con troppe ambizioni

CWI - La Coesione ha quindi molto a che vedere con un buon disegno delle classi.

MPJ - Proprio così e altri due esempi di cattiva Coesione ce lo possono dimostrare. Forse nessuno storcerà la bocca trovando un metodo 'arctan' tra quelli definiti per una classe (numero) REALE. Dopo tutto il metodo non fa altro che dirci, dato un numero reale 'r', l'angolo la cui tangente è appunto 'r'. Ma a questo punto è facile farsi prendere la mano. Perché, non definire per la nostra classe dei Numeri Reali anche un metodo che converta una temperatura da Celsius a Farenheit? O un importo da Dollari a Lire? Sarebbe un bell'esempio di "commistione del dominio" (mixed-domain Cohesion). Infatti gli angoli hanno ben poco a che vedere con le temperature o con le valute. In realtà un metodo 'arctan' è membro a pieno titolo di una classe ANGOLO, un metodo 'converti-temperatura' di una classe TEMPERATURA e un metodo 'converti-importo' di una classe VALUTA. In altre parole tutte queste classi appartengono a domini differenti, intendendo per dominio la loro applicabilità o riusabilità.

CWI - E la classe REALE?

MPJ - E' una classe fondamentale. Infatti non mi passerebbe mai per la testa di definire una classe ANGOLO senza aver prima definito una classe REALE. Ma questo implica che il dominio di ANGOLO è di livello più elevato del dominio di REALE. E anche che la definizione di REALE non dev'essere "ingombrata" da metodi, o altro, di pertinenza di classi di livello superiore al suo.

III Esempio: i cani di Fred

CWI - C'è poi un terzo tipo di cattiva Coesione.

MPJ - E' quello che chiamo a "commistione dei ruoli" (mixed-role Cohesion). Immaginiamo per esempio di voler sapere quanti cani possiede una PERSONA di nome Fred. E' facile, basta inviare un messaggio Fred.num-di-cani-posseduti perchè un metodo ad hoc, "num-di-cani-posseduti" per l'appunto, ci risponda dicendoci quanti sono i cani che ha Fred.

CWI - E se la risposta fosse 'zero' e cioè nessun cane?

MPJ - Niente paura, per una persona è del tutto naturale non possedere alcun cane. In altre parole non ci troviamo di fronte al problema della "commistione delle istanze" con il quale ci siamo già scontrati. E non ci troviamo nemmeno di fronte a un problema di "commistione dei domini" perchè le classi PERSONA e CANE fanno entrambe parte dello

stesso dominio.

CWI - Qual è allora il problema?

MPJ - E' molto semplice. L'appetito - si sa - vien mangiando. E se volessimo sapere non solo quanti sono i cani di Fred, ma se per caso ha anche qualche gatto, oppure quante sono le sue automobili o qual è il saldo del suo conto in banca? Attenzione! Per ciascuna di queste domande saranno necessari due metodi: il primo che ci fornisca le risposte, il secondo che provveda ad aggiornare il numero dei cani, dei gatti, delle auto e via dicendo di proprietà di Fred. E a questo punto chiediamoci cosa accadrà quando cercheremo di riusare la classe PERSONA in un nuovo contesto applicativo nel quale non ci siano cani e gatti, oppure automobili e conti bancari. La nostra classe risulterà inutilmente "ingombrata" da tutta una serie di riferimenti ad altre classi, in aperta violazione del principio di incapsulazione e a tutto danno della sua riusabilità. Tuttavia non è facile sfuggire alla tentazione di un disegno delle classi con commistione dei ruoli. Infatti pensare a un metodo come fred.num-di-cani-posseduti può sembrare la cosa più scontata di questo mondo e di conseguenza ben pochi si asterranno dal definire il "numero-di-cani-posseduti" come attributo di PERSONA. Ti diranno: nella vita di tutti i giorni, se vogliamo sapere quanti cani ha una persona, la cosa più semplice non è andarglielo a chiedere?

CWI - E invece?

MPJ - Invece è molto meglio definire tre classi: oltre a PERSONA e a CANE, una classe POSSESSORE DI CANI. Ciascuna sua istanza associa una PERSONA a un insieme di istanze - zero, una o più - della classe CANE e il nostro metodo "num-di-cani- posseduti" sarà appunto membro di questa classe. In altre parole avremo creato una cosiddetta classe "mix-in" che eredita in modo multiplo da PERSONA e CANE e da cui vengono istanziati oggetti che rappresentano persone che posseggono cani.

Buona e cattiva Coesione

CWI - A questo punto a proposito di Coesione sarà bene tirare le somme.

MPJ - E' molto semplice: un buon disegno delle classi è quello in cui la Coesione è "ideale" e cioè esente dalle tre forme di "cattiva" Coesione di cui abbiamo già parlato: 1. Coesione a commistione di istanze - alcune componenti della classe - attributi, metodi - risultano indefinite per certi oggetti istanziati dalla classe stessa. 2. Coesione a commistione di domini - la classe presenta componenti che fanno riferimento ad altre classi appartenenti a un dominio diverso dal suo. 3. Coesione a commistione di ruoli - la classe presenta riferimenti diretti ad altre classi che appartengono al suo stesso dominio. Di queste tre forme di "cattiva" Coesione le prime due si traducono inevitabilmente in problemi di disegno e manutenzione. La terza è più veniale, ma incide sulla riusabilità

CWI - E dopo la Coesione il Coupling. Se ricordo bene, Coesione e Coupling sono proprietà mutuamente esclusive.

MPJ - Ricorda benissimo. Nel Disegno Strutturato il Coupling era una misura del numero

e della forza delle connessioni tra due procedure. La Coesione di una classe presenta molte analogie con la Coesione di una procedura, anche se con qualcosa in più: un maggiore livello di incapsulazione. Di conseguenza nel Disegno OO il Coupling diventa una misura del numero e della forza delle connessioni tra le classi. Tanto minore la Coesione, tanto maggiore il Coupling. E viceversa.

Nati assieme

CWI - Sfogliando il suo libro non mi sono imbattuto soltanto in due vecchie conoscenze come la Coesione e il Coupling. Ho trovato anche alcuni termini per me del tutto inediti, Connascence e Contranascence. Sono parole inventate?

MPJ - Connascence non è una parola inventata. Infatti la troviamo tale e quale nel Chambers' Twentieth Century Dictionary e sotto forma dell'aggettivo "connato" nientemeno che nel Webster Dictionary. Connascenza viene dal latino, significa 'essere nati assieme' e si presta anche a un gioco di parole. Suona infatti esattamente come "connaissance", che in francese vuol dire conoscenza.

CWI - Tutto questo ha qualcosa a che vedere con il Disegno di un software OO?

MPJ - Per quanto riguarda il software essere nati assieme spesso vuol dire anche condividere lo stesso destino e sapere molto l'uno dell'altro. E infatti la Connascenza di due componenti software A e B significa l'una o l'altra di queste cose: 1. possiamo postulare che qualche cambiamento di A richiederà un cambiamento (o almeno un attento controllo) di B se vogliamo preservare la correttezza del tutto; 2. oppure che un qualche cambiamento di portata più generale richiederà la modifica sia di A che di B, sempre che desideriamo preservare la correttezza del tutto.

CWI - Detta così sembrerebbe che la Connascenza abbia molti punti di contatto con il Coupling del Disegno Modulare. Non veniva definito come la probabilità che, dovendo modificare un certo modulo, si dovesse mettere le mani anche in altri moduli?

MPJ - C'è del vero in quello che dice e la stessa osservazione mi è venuta da Larry Constantine. Ma attenti a non cadere nelle posizioni dei Reazionari, quelli del "Niente di nuovo sotto il sole". La Connascenza ha sicuramente qualcosa a che vedere con il Coupling, ma si spinge molto più in là, come un semplice esempio ci potrà dimostrare. Facciamo il caso di due componenti software, A e B. La componente A contiene una dichiarativa int i: // linea A e la componente B una assegnazione i:= 7; // linea B. Tra A e B ci sono almeno due forme di Connascenza. Se, ipotesi per la verità poco probabile, A venisse modificato in char i:, anche B dovrebbe essere a sua volta modificato. E lo stesso accadrebbe su A venisse modificato in int j:.

CWI - Non mi sembra così drammatico. In tutto questo non c'è nulla che un buon text editor non sia in grado di scoprire e risolvere.

MPJ - E' vero. Il nostro era un semplice esempio di Connascenza esplicita. Ma la Connascenza può essere anche implicita ed è qui che nascono i drammi: non è sempre possibile accorgersi che una componente è connata con un'altra. Ho inventariato tutta una serie di forme di Connascenza e per buona parte di esse non c'è text editor che tenga. Oltre che statica - riguarda la codifica delle classi che uno può scrivere,

compilare e lincheditare -, la Connascenza infatti può essere anche dinamica e in questo caso riguarderà il comportamento di due o più oggetti al run-time. Per non parlare poi della Contronascenza.

Puntare sull'incapsulazione

CWI - Questo sì che è un termine inventato. A me suona come il contrario di Connascenza.

MPJ - Invece è una particolare forma di Connascenza che ci dice come qualche volta l'integrità del software va preservata mantenendo le differenze e non, come nell'esempio di prima, le somiglianze tra due componenti.

CWI - E cioè?

MPJ - L'esempio migliore ce lo offre l'ereditarietà multipla. Se una classe C eredita da una classe A e da una classe B, i metodi di A e di B non possono condividere lo stesso nome. Se così fosse ci sarebbe Contronascenza dei nomi tra i metodi di A e di B.

CWI - E a questo punto ci troveremo a fare i conti con due capisaldi dell'OO, l'overloading e il polimorfismo.

MPJ - Proprio così. Due metodi possono infatti condividere lo stesso nome anche se la loro implementazione è differente. Se le nostre due classi, A e B, dispongono entrambe di un metodo polimorfico M, quale delle due versioni di M sarà ereditata dalla classe C? L'unico modo per venire a capo di conflitti come questi consiste nel ridenominare i metodi ereditati e in molti casi per farlo sarà necessario intervenire a mano. Non a caso l'ereditarietà multipla gode di fama sinistra tra molti addetti ai lavori.

CWI - Quindi un buon disegno deve tener conto anche di Connascenza e Contronascenza.

MPJ - La ricetta che suggerisco è molto semplice: usare l'incapsulazione come antidoto alla Connascenza e cioè 1. minimizzare la Connascenza complessiva, ivi compresa la Contronascenza, ripartendo il sistema in una collezione di componenti ben incapsulate; 2. minimizzare la Connascenza che travalichi le frontiere tracciate dall'incapsulazione; 3. massimizzare invece la Connascenza tra tutto cio che risiede entro queste frontiere. Le differenze tra un buon disegno e un cattivo disegno delle classi saltano all'occhio così come il diverso impegno che richiederà la loro manutenzione.

Dagli amici mi guardi Iddio!

CWI - Che nel disegno delle classi sia bene puntare sull'incapsulazione sembrerebbe scontato.

MPJ - Ma non lo è, basta pensare alle nefaste conseguenze delle funzioni "friend" del C++. CWI - Perchè nefaste? MPJ - Perché, sembrano fatte apposta per violare il principio di incapsulazione. Facciamo il caso di una funzione friend che chiameremo FF e che sia "amica" di una classe C1: pur essendo una componente che risiede al di là dei suoi

confini, FF può accedere in modo incondizionato alla porzione privata di tutti gli oggetti della classe C1. Con una Connascenza tra FF e C1 tanto elevata qualsiasi modifica a C1 richiederà che anche FF venga modificato o quanto meno controllato con molta attenzione.

CWI - Ma se FF è "amica" della sola classe C1, non potremmo considerarla come membro a pieno titolo di quella classe? A questo punto il discorso della violazione dei limiti della classe verrebbe a cadere.

MPJ - E' vero. Purtroppo, oltre che di C1, FF può essere "amica" anche di C2, di C3 e così via. Per loro sfortuna numerosi disegni di applicazioni C++ hanno esattamente questa struttura e con amici del genere si può fare tranquillamente a meno dei nemici...

CWI- Quindi le funzioni friend vanno messe al bando. Senza eccezioni?

MPJ - Una eccezione c'è e riguarda il testing strutturale delle classi. L'incapsulazione vuole che le classi si presentino come Scatole Nere, in altre parole nulla dello stato di un oggetto deve trapelare al suo esterno. Ma questo, in fase di test, può rappresentare un serio intralcio e qui le funzioni friend, trasformando le Scatole da Nere a Bianche, possono trarci d'impaccio. Ma, una volta reso questo importante servizio, andranno accantonate.

Quale notazione per l'OO?

CWI - Un'ultima curiosità. Cosa sono e come nascono i simbolismi usati nel suo libro per rappresentare classi, messaggi, metodi e via dicendo?

MPJ - Anche questa è una lunga storia. Comincia infatti nel 1990 quando, con Larry Constantine e Steve Weiss, ho abbozzato una prima notazione per il Disegno OO. L'abbiamo chiamata UON, e cioè Uniform Object Notation perchè la nostra idea era quella di trovare un modo per descrivere in modo "uniforme" sistemi sviluppati con tecniche strutturate, con tecniche OO o con un misto delle due. E infatti il simbolo adottato per rappresentare le classi è una metafora che rispecchia la duplice natura degli oggetti: è in parte curvo e in parte rettilineo come voleva il Disegno Strutturato che con un cerchio simbolizzava i dati e con un rettangolo le procedure.

CWI - E poi?

MPJ - Poi nel 1994 è toccato a Brian Henderson- Sellers e a Julian Edwards rimaneggiare UON adottandolo come notazione per il loro metodo MOSES. E per finire, lavorando in parte con gli autori di MOSES e in parte da solo, ho esteso ulteriormente UON e l'ho ribattezzato Object-Oriented Design Notation. A differenza di UON, OODN dispone di tutto cio che occorre per specificare il Disegno di un'applicazione OO, dai diagrammi delle classi a quelli di transizione di stato e per la navigazione delle window.

CWI - Ma come la mettiamo con l'Unified Modeling Language (UML), frutto della convergenza dei metodi di Booch, Jacobson e Rumbaugh? E' stato infatti presentato all'approvazione dell'OMG (Object Management Group) il 16 gennaio 1997 e tutto lascia prevedere che diventerà uno standard de jure oltre che di fatto...

MPJ - Per quanto riguarda le notazioni usate nel mio libro c'era poco da scegliere. Per tutta una serie di incomprensioni con il mio editore la sua gestazione è stata lunga, anzi lunghissima. La prima stesura risale infatti al 1992 e allora le uniche alternative a UON erano le notazioni di Booch e di Rumbaugh, per me del tutto insufficienti. Io naturalmente resto affezionato a UON e a OODN, ma adesso che è venuto UML forse mi adattero a usarlo. Sempre che mi permetta di esprimere nel disegno di una applicazione OO tutti i significati che sono necessari.

Al programmatore non far sapere...

CWI - Il suo libro si rivolge ai Programmatori. Ma qui in Italia, almeno ufficialmente, di Programmatori non resta quasi più traccia. Se vuole restare a galla, uno deve fregiarsi almeno della qualifica di analista-programmatore.

MPJ - Capita anche da noi e io la chiamo "inflazione dei titoli". Ma anche se non si fanno più chiamare Programmatori ma "Software Engineer" - e qualche volta non hanno la più pallida idea di cosa sia l'Ingegneria del Software -, continuano a scrivere codice.

CWI - E allora noi continueremo a chiamarli Programmatori. Tornando sempre al titolo del suo libro, i Programmatori devono sapere di Coesione, di Coupling e di Connascenza. C'è invece qualcosa che non devono sapere?

MPJ - Forse che la loro stirpe è destinata a scomparire. Può darsi che in un futuro più o meno lontano ci troveremo ad accompagnare i nostri figli a visitare un Museo delle Arti e Mestieri, così come ce ne sono nelle nostre città. E lì, accanto a manichini e attrezzi di lavoro che raffigurano professioni del passato come quella del maniscalco o dello stagnino, avremo modo di ammirare fedelmente riprodotti i Programmatori del buon tempo antico intenti a pigiare i tasti delle loro workstation.

CWI - Dice sul serio?

MPJ - Assolutamente no! Le buone pratiche del Disegno OO porteranno senza dubbio a un aumento della riusabilità e così in futuro ci sarà bisogno di un minor numero di programmatori e di un maggior numero di bravi analisti del business. Ma non penso proprio che la stirpe dei programmatori sia destinata all'estinzione.

Tecnet Dati s.r.l.
C.so Svizzera 185 –
10149 - Torino (TO), Italia
Tel.: +39 011 7718090 Fax.: +39 011 7718092
P.I. 05793500017 C.F. 09205650154
www.tecnetdati.com