REVERSE ENGINEERING DI SISTEMI SOFTWARE LIMITI E POTENZIALITÀ

Article

2 authors:

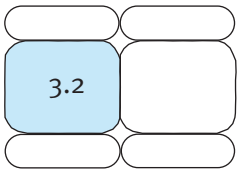
Filippo Ricca
Università degli Studi di Genova
162 PUBLICATIONS **3,063** CITATIONS
SEE PROFILE

Paolo Tonella
Fondazione Bruno Kessler
264 PUBLICATIONS **6,262** CITATIONS
SEE PROFILE

REVERSE ENGINEERING DI SISTEMI SOFTWARE
LIMITI E POTENZIALITÀ

Filippo Ricca
Paolo Tonella

Il reverse engineering è un processo di analisi mirato all'identificazione delle componenti e relazioni di un sistema software. Da alcuni esperti è considerato un potente mezzo, mentre da altri un processo non realizzabile in pratica. Chi ha ragione? In questo articolo vengono sottolineati i limiti e le potenzialità del reverse engineering. Prima di concludere con due storie di successo a livello industriale, l'articolo propone anche una possibile classificazione dei tool usati nel processo di reverse engineering.



1. INTRODUZIONE

Col passare degli anni, modifica dopo modifica, i sistemi software invecchiano e la loro struttura tende inevitabilmente a degenerare. Spesso i manager e gli ingegneri del software sono costretti a dover gestire un sistema molto grande, vitale per l'organizzazione, costato molto e al tempo stesso difficile da comprendere e mantenere. A peggiorare le cose è il fatto che solitamente la documentazione non esiste e se esiste non è aggiornata. Questo punto di arrivo, che ha come conseguenza quella di rendere molto costosi e difficili tutti gli interventi di manutenzione, costringe spesso i manager a una scelta difficile: riscrivere il sistema da zero oppure ristrutturarlo completamente.
Le tecniche di *reverse engineering* possono essere utili nel caso in cui viene scelta questa seconda soluzione. Il *reverse engineering* [2], in italiano ingegneria inversa, è un processo di analisi di un oggetto o dispositivo (un componente elettronico, un aeroplano, un motore, un software ecc.) che ha come obbiettivo unicamente la comprensione. Solitamente il passo successivo del *reverse engineering* è quello di costruire un nuovo oggetto/dispositivo avente caratteristiche e funzionalità simili a quello analizzato.
Il *reverse engineering* è stato usato spesso dalle forze armate al fine di copiare la tecnologia bellica dei nemici [7]. In ambito militare, celebri esempi di *reverse engineering* sono il missile *Scud Mod A*, il bombardiere russo *Tupolev Tu-4* e il personal computer *Agatha*. La Corea del nord in tempi recenti si è dotata di missili a lunga gittata denominati Scud Mod A. Questi missili sono stati costruiti seguendo un processo di ingegneria inversa a partire dai missili sovietici Scud BS. Durante la seconda guerra mondiali alcuni bombardieri americani B-29 diretti in Giappone furono costretti ad atterrare in Unione Sovietica per problemi tecnici. In quel tempo i Sovietici non erano dotati di aerei così avanzati e sfruttando questo caso fortuito decisero di copiare la tecnologia dei B-29; nacquero così i Tupolev Tu-4. Il personal computer Agatha, vera punta di diamante dei sovietici per diversi anni, è stato creato a partire dal computer americano Apple II.

In ambito informatico lo scopo del *reverse engineering* è abbastanza diverso rispetto a quello militare. Solitamente l'ingegneria inversa è utilizzata dai progettisti per aumentare il grado di conoscenza di un sistema software quando questo deve essere sottoposto ad una operazione di modifica. Nei casi reali non è raro che i progettisti siano costretti ad eseguire operazioni di manutenzione del codice senza avere una conoscenza approfondita dello stesso. Questa eventualità si ha tutte le volte che sul software agiscono progettisti o ingegneri che non hanno partecipato alle fasi di sviluppo – caso assai frequente visto che alcuni sistemi hanno oramai più di 30 anni – oppure quando occorre modificare il proprio codice dopo che è passato un po' di tempo.
Le tecniche e i *tool* di *reverse engineering* forniscono i mezzi per generare un'adeguata documentazione del codice; in particolare sono in grado di produrre documenti mai esistiti o recuperare quelli che si sono persi con il passare degli anni. Il risultato di questo processo è un insieme di rappresentazioni alternative del sistema, spesso grafiche, che aiutano il progettista nella fase di manutenzione ed evoluzione del codice.

2. MANUTENZIONE DEL SOFTWARE

Il ciclo di vita di un sistema software inizia con l'analisi del dominio e dei requisiti, continua con la fase di sviluppo (progettazione e codifica) e termina con la dismissione – momento in cui viene stabilito che il sistema software non è più utile per l'organizzazione. Dopo il rilascio, cioè quando il sistema diventa fruibile da parte degli utenti, il software entra in una fase molto delicata che solitamente dura molto, la manutenzione.
Per manutenzione si intende il complesso di operazioni necessarie a:
- conservare il sistema software funzionante ed efficiente con il passare degli anni;
- mantenere le funzionalità del sistema continuamente aggiornate.

Dopo avere condotto diversi studi empirici sull'evoluzione/manutenzione del software, Lehman and Belady [17] hanno concluso che ci sono delle "leggi" che valgono per tutti i sistemi software.

Leggi sull'evoluzione del software
- Un programma che è utilizzato in un ambiente reale deve necessariamente essere modificato o diviene progressivamente meno utile in quell'ambiente.
- Man mano che un software evolve, la sua struttura diviene sempre più complessa. Risorse extra devono essere impiegate per preservare e semplificare la struttura.
- Le funzionalità offerte dal sistema devono essere continuamente aggiornate ed estese per mantenere gli utenti soddisfatti.
- La qualità di un sistema tende a peggiorare modifica dopo modifica. Risorse extra devono essere impiegate al fine di contenere questo degrado irreversibile della struttura.

Nell'riquadro sono riportati i loro risultati più significativi.
La manutenzione è una fase molto critica e ancora oggi, nonostante il miglioramento delle tecniche di sviluppo e dei processi di produzione, rappresenta per tutte le aziende una spesa molto cospicua. Sommerville [23] la stima in un valore compreso tra il 50% e il 75% della spesa totale di produzione del software. Mantenere ed evolvere un sistema software esistente è difficile perché bisogna tenere presenti diversi aspetti: il veloce "turnover" degli sviluppatori, la continua crescita dei sistemi software in termini di grandezza e complessità e l'evoluzione della tecnologia.
I costi sono molto alti perché la manutenzione è composta di una serie di attività estremamente complesse:
- ***comprensione del codice***: consiste in un'analisi approfondita del software (o solo di una parte). Prima di effettuare una modifica occorre, infatti, capire come funziona il codice e dove apportarla. Questa è sicuramente l'attività più complessa tra tutte; è estremamente complicato comprendere il proprio codice e peggio ancora capire il software scritto da altri;
- ***analisi di impatto***: serve a capire quali funzionalità vanno aggiornate in modo da renderle consistenti con la modifica da implementare;
- ***modifica del codice***: consiste nella fase di codifica del cambiamento;
- ***validazione***: consiste nel verificare che la modifica non abbia introdotto nuovi errori.

La manutenzione, oltre ai costi diretti, implica anche un insieme di costi indiretti. Il più evidente in ambito industriale è lo sbilanciamento nell'allocazione delle risorse verso la fase di manutenzione. Quello che accade in realtà è che i programmatori sono impegnati, quasi sempre, a mantenere codice già rilasciato dedicando poco tempo allo sviluppo di nuovi progetti. Il risultato è che le risorse effettive impegnate per lo sviluppo di nuovi progetti sono spesso insufficienti.

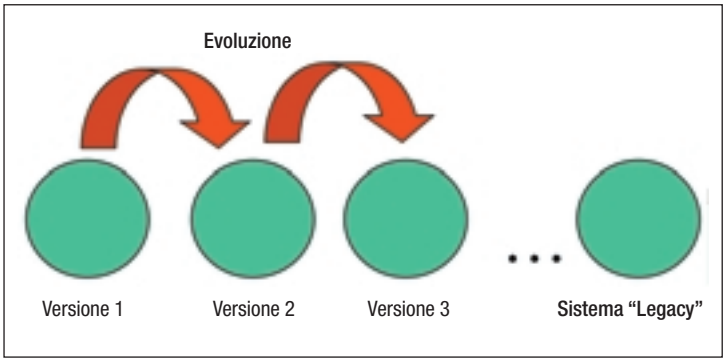


FIGURA 1
Le modifiche ripetute degradano la struttura interna del software. Spesso il risultato è un sistema legacy

Un altro aspetto che rende questa fase molto delicata è il fatto che una manutenzione non strutturata e caotica porta al rapido degrado del sistema, facendo diventare le successive modifiche sempre più complesse. Un sistema software raggiunge lo stato di *legacy* (Figura 1) quando subisce nel tempo diverse modifiche e la fase di manutenzione diventa sempre più complessa e costosa.

3. SISTEMI "LEGACY"

Un sistema *legacy* [1] è un software che è stato mantenuto in vita per molti anni e che ha le seguenti caratteristiche:

- ❑ è vitale per un'organizzazione ed è pesantemente utilizzato. Per esempio deve essere operativo 24 h su 24 e 7 giorni su 7;
- ❑ su di esso l'organizzazione ha investito molto, sia in termini economici che di tempo;
- ❑ non può essere dismesso facilmente in quanto è l'unico *repository* di conoscenza. Le procedure dell'azienda (le cosiddette *business rules*) non sono registrate in nessun altro posto e sono contenute solo nel codice;
- ❑ solitamente è un codice enorme costituito da milioni di linee;
- ❑ è scritto in un linguaggio di vecchia generazione (per esempio: Cobol, Assembler, PL/1, RPG ecc.) ed è progettato secondo vecchie concezioni (per esempio: programmazione non strutturata, salti incondizionati ecc.). Inoltre spesso è eseguito su piattaforme obsolete;
- ❑ solitamente utilizza un database obsoleto sempre che non faccia uso di un *file system* (file ad accesso sequenziale, accesso diretto ecc.);
- ❑ solitamente è un'applicazione monolitica. Presentazione, logica applicativa ed accesso ai dati sono fusi assieme in un unico blocco (le componenti non sono separate né logicamente né fisicamente);
- ❑ è difficile da comprendere, mantenere ed espandere in quanto solitamente la documentazione non esiste e se esiste non è aggiornata con le modifiche che sono state, nel corso degli anni, apportate al software (codice e documentazione non sono allineati).

Molti software che furono sviluppati negli anni '70 e '80, in linguaggi come Cobol, Assembler, PL/1, RPG e che ancora oggi sono utilizzati in diversi ambiti, possono essere considerati a pieno titolo dei sistemi *legacy*. Di fatto i *legacy* sono ancora molto comuni [18] e sono tuttora ampiamente utilizzati. Questo accade soprattutto nella Pubblica amministrazione, in quanto la loro dismissione non può avvenire facilmente perché non giustificabile economicamente. Sebbene l'espressione sistema *legacy* è spesso associata a sistemi scritti in linguaggi di programmazione obsoleti non è raro trovare sistemi recenti che presentano problemi simili [3]. È importante sottolineare che le tecniche di *reverse engineering*, originariamente pensate per i sistemi *legacy*, si sono rivelate estremamente utili anche in supporto alla comprensione di sistemi software scritti secondo approcci moderni (per esempio, codice orientato agli oggetti o applicazioni Web).

4. APPROCCI POSSIBILI PER TRATTARE I SISTEMI LEGACY

Il momento più critico per un'organizzazione si ha quando il sistema software che è stato utilizzato con successo per anni è diventato, con il passare del tempo, inefficiente o troppo costoso da mantenere. Una soluzione è quella di rimpiazzare il vecchio sistema con uno moderno più semplice da mantenere e più efficiente. Questa soluzione è raramente scelta dai manager perché presenta elevati rischi per l'organizzazione e al tempo stesso spesso ne raddoppia i costi (l'organizzazione dovrà investire nel nuovo sistema e contemporaneamente mantenere funzionante quello vecchio). Costruire da zero un nuovo sistema che riproduce ed estende le funzionalità di uno vecchio è

un'operazione complessa, soggetta a molti errori e che solitamente prende molto più tempo di quello preventivato. Molti progetti statunitensi di questo tipo iniziati nel biennio 1992/93 si rivelarono un autentico disastro; nel 1995 quasi l'80% erano falliti, nel senso che erano stati ridimensionati, posposti o addirittura cancellati [18]. Altra complicazione è il fatto che non è possibile avere la certezza che il nuovo sistema, una volta sviluppato, riprodurrà fedelmente parte delle funzionalità di quello vecchio.
Il ***legacy dilemma*** consiste proprio in questo: *è più conveniente ristrutturare completamente il vecchio sistema e continuare ad usarlo oppure accantonarlo e riscriverne uno nuovo che riproduce ed estende le funzionalità del vecchio*?
Di fatto tra i due estremi, detti sostituzione a taglio netto (o riscrittura) e migrazione totale (o ristrutturazione totale), esistono anche soluzioni intermedie. In generale, esistono diversi approcci possibili per trattare i sistemi *legacy* [18]:

- ***manutenzione del sistema legacy nella sua forma originale***: spesso questa soluzione non è praticabile in quanto troppo costosa;
- ***sostituzione a taglio netto o riscrittura***: consiste nell'accantonare il *legacy* e procedere con una riscrittura da zero del nuovo sistema. Questa è sicuramente la soluzione che presenta più rischi;
- ***wrapping***: si tratta di aggiungere al *legacy system* un nuovo livello software che nasconde l'implementazione effettiva delle funzionalità e presenta il vecchio codice sotto una nuova forma. Il *wrapping* può essere realizzato a livello di interfaccia utente o a livello di sistema. Nel primo caso la vecchia applicazione appare all'utente completamente trasformata, anche se di fatto non lo è, e con un interfaccia grafica moderna. Nel secondo caso il nuovo livello software presenta il sistema *legacy* attraverso un'interfaccia software ben definita, tipicamente ad oggetti, che permette la connessione del vecchio software con i nuovi moduli che costituiscono l'ambiente distribuito (Figura 2). In questo modo la vecchia applicazione appare sotto forma di uno o più oggetti, del tutto simili a tutti gli altri, ed in grado di operare nello stesso ambiente, accettare messaggi dagli altri moduli e rispondere alle richieste [18];
- ***migrazione parziale***: solo una parte del sistema, quella che crea più problemi o quella più obsoleta, è trasformata. Solitamente è mantenuta in vita la parte del *legacy* che implementa le *business rules*, mentre sono migrate le interfacce utente e i dati. La differenza tra migrazione parziale e *wrapping* è nella prospettiva di evoluzione del sistema. Nel primo caso sono trasformate alcune componenti software per facilitare gli interventi futuri di manutenzione sul sistema. Nel secondo caso l'implementazione effettiva delle funzionalità è nascosta da un nuovo strato software nella speranza di non dover più intervenire sul sistema *legacy*;
- ***migrazione totale***: la migrazione totale consiste nel trasformare completamente il sistema *legacy* in uno nuovo con l'intenzione di migliorare la manutenibilità. Questo processo può presentare grossi rischi e spesso richiede molti anni; pertanto la migrazione totale deve avvenire gradualmente e incrementalmente, in modo da mantenere sempre operativo il sistema.

Tipicamente le paure e le incertezze legate agli approcci più estremi fanno preferire le soluzioni intermedie. *Wrapping* e migrazione parziale raramente danno luogo alla soluzione migliore, ma hanno il pregio di non esporre l'organizzazione a grossi rischi e soprattutto di costare meno.

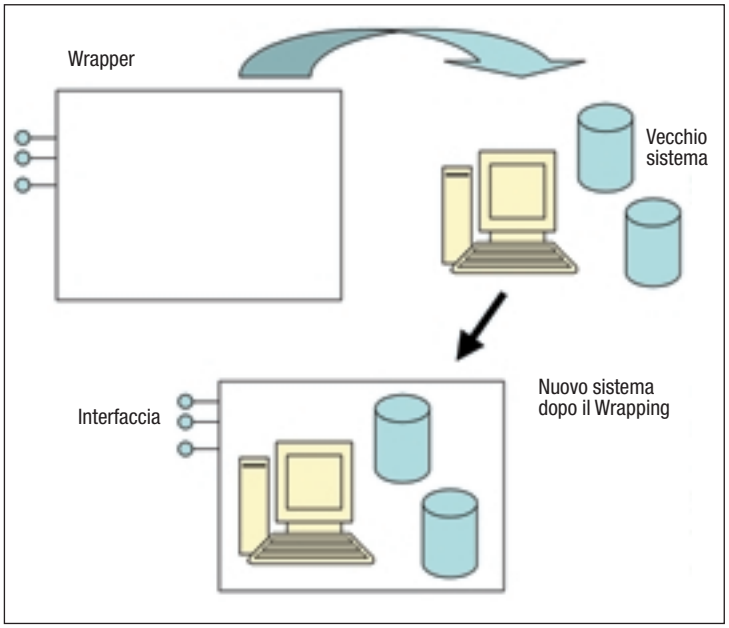


FIGURA 2
Operazione di wrapping a livello di sistema [18]

Per questo motivo le soluzioni intermedie sono scelte più frequentemente.

5. REVERSE ENGINEERING

Per molti anni l'ingegneria del software si è concentrata soprattutto sullo sviluppo di nuove applicazioni trascurando in parte la fase di manutenzione. Ultimamente però, con il crescere dell'importanza del software e soprattutto con la proliferazione dei sistemi *legacy*, questa fase è stata notevolmente rivalutata sia nel mondo della ricerca che in quello dell'industria.
Come già accennato, uno dei momenti più critici della manutenzione è rappresentato dalla comprensione del codice, fase nella quale il programmatore cerca di capire la struttura interna del software (o parte del software) e il suo funzionamento prima di effettuare le dovute modifiche.
È proprio nella fase di comprensione che risultano particolarmente utili le tecniche di *reverse engineering*. Queste tecniche forniscono i mezzi per recuperare le informazioni perse con il passare degli anni o che non sono mai esistite. L'obbiettivo è quello di costruire progressivamente dei modelli mentali del sistema in modo tale da migliorare la comprensione dello stesso. Mentre questa operazione non è complessa per piccoli sistemi software, dove lettura ed ispezioni del codice sono spesso sufficienti, diventa estremamente problematica nel caso di sistemi *legacy* a causa della loro grandezza e complessità.
Dato un sistema legacy ci sono diversi approcci per cercare di ricostruire le funzionalità del software e la loro interazione con i dati:

- ***lettura della documentazione esistente e del codice***: è difficile usare questo approccio quando la documentazione è datata, incompleta o inesistente. Inoltre la lettura del codice diventa improponibile quando le linee di codice sono nell'ordine del milione (si dice che questa tecnica non scala con il crescere del sistema);
- ***intervista agli utenti e agli sviluppatori***: questo è un buon metodo ma è applicabile solo raramente. È infatti difficile trovare gli sviluppatori originali che hanno partecipato allo sviluppo del codice;
- ***utilizzo di tools*** e delle tecniche proprie del *reverse engineering* per generare rappresentazioni ad alto livello del codice: vedremo nel paragrafo 7 una lista di *tools* di *reverse engineering*.

In letteratura esistono due definizioni diverse di *reverse engineering*, una "forte" e una cosiddetta "debole" [22]:

- ***definizione forte*** di *reverse engineering*: è un processo che a partire dal codice *legacy* permette di estrarre le specifiche formali del sistema. Le informazioni di design del sistema vengono derivate dal codice come passo intermedio (Figura 3) e le specifiche formali estratte possono essere usate per creare una nuova implementazione del sistema. In questa definizione ci sono tre assunzioni implicite:
  - il processo è automatico;
  - le specifiche sono ad un buon livello di astrazione in modo tale che il sistema possa essere re-implementato in un altro linguaggio o secondo concezioni diverse;
  - il tempo e lo sforzo per derivare le specifiche è inferiore rispetto a quello richiesto per costruire il sistema da zero;
- ***definizione debole*** di *reverse engineering*: è un processo automatico o semi-automatico (ovvero assistito dall'utente) che a partire dal codice *legacy* permette di derivare una base di conoscenza del sistema. La base di conoscenza è costituita da rappresentazioni alternative del codice *legacy*, spesso ad un livello più astratto e grafico, che mettono in risalto alcune proprietà e caratteristiche del sistema stesso.

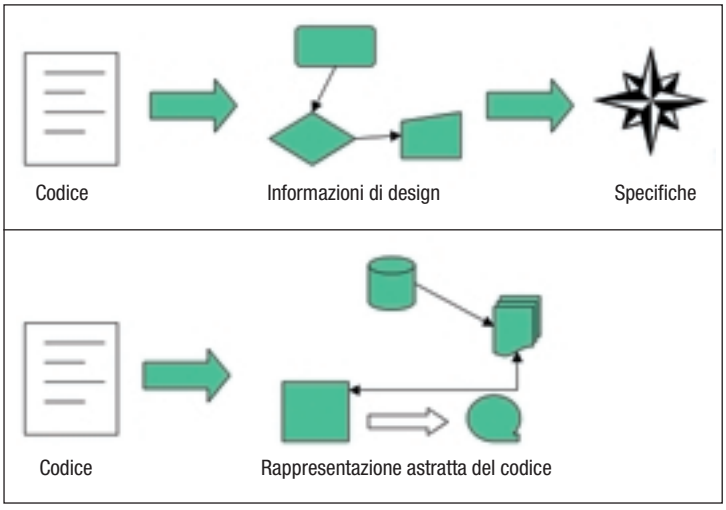


FIGURA 3
Definizione forte (in alto) e debole (in basso) di reverse engineering

Questa seconda definizione è molto meno ambiziosa rispetto alla precedente. Nella definizione debole non viene richiesto come risultato un insieme di specifiche formali del sistema ma solo una rappresentazione astratta del codice. Altra differenza fondamentale è che il processo non deve essere completamente automatico bensì assistito dall'utente.

6. SUCCESSO O FALLIMENTO?

Rispondere alla domanda se il *reverse engineering* sia stato e sia tuttora un successo è estremamente difficile. La risposta chiaramente dipende dal risultato che ci aspettiamo dal *reverse engineering* e quindi dalla definizione che scegliamo (forte o debole).
Se scegliamo la definizione forte allora dovremmo concludere che il *reverse engineering* è stato ed è un fallimento. Lo stato dell'arte è molto lontano dagli obbiettivi prefissati da questa definizione. I *tool* commerciali ed accademici non sono in grado di recuperare automaticamente dal codice le specifiche formali. Inoltre anche limitandosi alle informazioni di design non esistono *tool* in grado di recuperare dal codice delle rappresentazioni astratte usabili in modo completamente automatico. Le rappresentazioni astratte che si ottengono dai *tool* di *reverse engineering* devono poi, per essere utilizzabili, venire raffinate manualmente. Inoltre i *tool* devono essere "guidati" nel processo di recupero delle informazioni: l'intervento umano è indispensabile per avere dei risultati accettabili.
Per quanto concerne la terza assunzione della definizione forte, quella relativa ai tempi di sviluppo, risulta molto difficile verificarla e dipende da sistema a sistema. La definizione forte presenta inoltre un altro grosso problema, relativo alla seguente domanda: *è possibile estrarre specifiche formali dal codice in modo completamente automatico*? In generale non è infatti decidibile l'equivalenza tra due rappresentazioni formali alternative di un sistema, quali codice e specifiche.
Se invece viene scelta la definizione debole allora la risposta alla domanda iniziale è completamente diversa: il *reverse engineering* è stato ed è un successo. Esistono, infatti, in commercio e nel mondo accademico innumerevoli *tool* che sono in grado di recuperare in modo semi-automatico dal codice, rappresentazioni astratte (grafiche e testuali) molto utili nella fase di comprensione del software.
Al momento le ricerche si stanno concentrando principalmente su tecniche e *tool* di estrazione di informazioni di design a diversi livelli di astrazione.

7. TOOL DI REVERSE ENGINEERING: COMMERCIALI ED ACCADEMICI

In commercio e nel mondo accademico esistono diversi programmi che sono classificati come *tool* di *reverse engineering*. Questa sezione propone una possibile classificazione, necessariamente non esaustiva, di questi tools. Una lista più completa può essere trovata su Internet all'indirizzo [8].
Tra i *tool* più utilizzati per scopi di comprensione del codice troviamo:
1. ***pretty printer***: **f**ormattano il codice in una forma più leggibile. Questi programmi si rivelano particolarmente utili tutte le volte che il codice è stato scritto con convenzioni di formattazione obsolete o particolarmente difficili da capire. DMS [10], un *tool* commerciale di trasformazione automatica del codice che dispone tra le varie funzionalità anche quella di ri-formattare il codice, è stato utilizzato per analizzare il codice C vincitore di un edizione del IOCCC, una competizione internazionale che premia il codice C più "offuscato" (Figura 4);
2. ***visualizzatori di codice*** (*code viewer*): producono viste alternative del software. Alcune sono testuali altre grafiche. A questa categoria appartengono i generatori di diagrammi di flusso (per esempio, il *tool* commerciale *Code Visual to Flowchart* V3.5 [11]) e i navigatori di codice (per esempio, il *tool Source-Navigator* [12]), che permettono di "navigare" con facilità all'interno del codice grazie all'ausilio di particolari viste ad alto livello (per esempio, la vista gerarchica delle classi nei sistemi ad oggetti). Un *tool freeware* che rientra in questa categoria è il *CodeCrawler* (riquadro a p. 59) [16]. Questo *tool* implementa e visualizza il concetto di viste polimetriche, ovvero viste del software arricchite con informazioni semantiche (me-

```
#include                                        <math.h>
#include                                      <sys/time.h>
#include                                      <X11/Xlib.h>
#include                                     <X11/keysym.h>
                                              double L,o,P
                                             ,_=dt,T,Z,D=1,d,
                                             s[999],E,h=8,I,
                                             J,K,w[999],M,m,O
                                             ,n[999],j=33e-3,i=
                                             1E3,r,t,u,v,W,S=
                                             74.5,l=221,X=7.26,
                                             a,B,A=32.2,c,F,H;
                                              int N,q,C,y,p,U;
                                             Window z;char f[52]
                                             ;GC k;main(){Display*e=
XOpenDisplay(0);z=RootWindow(e,0);for(XSetForeground(e,k=XCreateGC(e,z,0,0),BlackPixel(e,0))
;scanf("%lf%lf%lf",y+n,w+y,y+s)+1;y++);XSelectInput(e,z=XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0)),KeyPressMask);for(XMapWindow(e,z);;T=sin(O)){struct timeval G={0,dt*1e6}
;K=cos(j);N=1e4;M+=H*_;Z=D*K;F+=_*P;r=E*K;W=cos(O);m=K*W;H=K*T;O+=D*_*F/K+d/K*E*_;B=
sin(j);a=B*T*D-E*W;XClearWindow(e,z);t=T*E+D*B*W;j+=d*_*D-_*F*E;P=W*E*B-T*D;for(o+=(l=D*W+E
*T*B,E*d/K*B+v+B/K*F*D)*_;p<y;){T=p[s]+i;E=c-p[w];D=n[p]-L;K=D*m-B*T-H*E;if(p[n]+w[p]+p[s
]==0|K<fabs(W=T*r-l*E+D*P)|fabs(D=t*D+Z*T-a*E)>K)N=1e4;else{q=W/K*4E2+2e2;C=2E2+4e2/K
*D;N-1E4&&XDrawLine(e,z,k,N,U,q,C);N=q;U=C;}++p;}L+=_*(X*t+P*M+m*l);T=X*X+l*l+M*M;
XDrawString(e,z,k,20,380,f,17);D=v/l*15;i+=(B*l-M*r-X*Z)*_;for(;XPending(e);u*=CS!=N){
XEvent z;XNextEvent(e,&z);
++*((N=XLookupKeysym
(&z.xkey,0))-IT?
N-LT?UP-N?&E:&
J:&u:&h):--*(
DN-N?N-DT?N==
RT?&u:&W:&h:&J
);}m=15*F/l;
c+=(l=M/l,l*H
+l*M+a*X)*_;H
=A*r+v*X-F*l+(
E=.1+X*4.9/l,t
=T*m/32-l*T/24
)/S;K=F*M+(
h*1e4/l-(T+
E*5*T*E)/3e2
)/S-X*d-B*A;
a=2.63/l*d;
X+=(d*l-T/S
*(.19*E+a
*.64+J/1e3
)-M*v+A*
Z)*_;l+=
K*_;W=d;
sprintf(f,
"%5d %3d"
"%7d",p=l
/1.7,(C=9E3+
O*57.3)%0550,(int)i);d+=T*(.45-14/l*
X-a*130-J*.14)*_/125e2+F*_*v;P=(T*(47
*l-m*52+E*94*D-t*.38+u*.21*E)/1e2+W*
179*v)/2312;select(p=0,0,0,0,&G);v-=(
W*F-T*(.63*m-l*.086+m*E*19-D*25-.11*u
)/107e2)*_;D=cos(o);E=sin(o);}}
```

```
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>
double L, o, P, _ = dt, T, Z, D = 1, d, s[999], E, h = 8, I, J, K, w[999],
       M, m, O, n[999], j = 3.3e-2, i = 1e3, r, t, u, v, W, S = 7.45e1,
       l = 221, X = 7.26, a, B, A = 3.22e1, c, F, H;
int N, q, C, y, p, U;
Window z;
char f[52];
GC k;
main()
{
  Display * e = XOpenDisplay(0);
  z = RootWindow(e, 0);
  for (XSetForeground(e, k = XCreateGC(e, z, 0, 0), BlackPixel(e, 0));
  scanf("%lf%lf%lf", y + n, w + y, y + s) + 1; y++);
  XSelectInput(e, z = XCreateSimpleWindow(e, z, 0, 0, 400, 400,
               0, 0, WhitePixel(e, 0)), KeyPressMask);
  for (XMapWindow(e, z);; T = sin(O))
    {
      struct timeval G = { 0, dt * 1e6 };
      K = cos(j);
      N = 1e4;
      M += H * _;
      Z = D * K;
      F += _ * P;
      r = E * K;
      W = cos(O);
      m = K * W;
      H = K * T;
      O += D * _ * F / K + d / K * E * _;
      B = sin(j);
      a = B * T * D - E * W;
      XClearWindow(e, z);
      t = T * E + D * B * W;
      j += d * _ * D - _ * F * E;
      P = W * E * B - T * D;
      for (o += (l = D * W + E * T * B, E * d / K * B + v + B / K * F * D) * _; p < y;)
        {
          T = p[s] + i;
          E = c - p[w];
          D = n[p] - L;
          K = D * m - B * T - H * E;
          if (p[n] + w[p] + p[s] == 0 | K < fabs(W = T * r - l * E + D * P) | fabs(D = t * D + Z * T - a * E) > K)
            N = 1e4;
          else
            {
              q = W / K * 4e2 + 2e2;
              C = 2e2 + 4e2 / K * D;
              N - 1e4 && XDrawLine(e, z, k, N, U, q, C);
              N = q;
              U = C;
            }
          ++p;
        }
      L += _ * (X * t + P * M + m * l);
      T = X * X + l * l + M * M;
      XDrawString(e, z, k, 20, 380, f, 17);
      D = v / l * 15;
      i += (B * l - M * r - X * Z) * _;
      for (; XPending(e); u *= CS != N)
        {
          XEvent z;
          XNextEvent(e, & z);
          ++ * ((N = XLookupKeysym(& z.xkey, 0)) - IT ? N - LT ? UP - N ? & E : & J : & u : & h);
          -- * (DN - N ? N - DT ? N == RT ? & u : & W : & h : & J);
        }
      m = 15 * F / l;
      c += (l = M / l, l * H + l * M + a * X) * _;
      H = A * r + v * X - F * l + (E = 1e-1 + X * 4.9 / l, t = T * m / 32 - l * T / 24) / S;
      K = F * M + (h * 1e4 / l - (T + E * 5 * T * E) / 3e2) / S - X * d - B * A;
      a = 2.63 / l * d;
      X += (d * l - T / S * (1.9e-1 * E + a * 6.4e-1 + J / 1e3) - M * v + A * Z) * _;
      l += K * _;
      W = d;
      sprintf(f, "%5d %3d"
              "%7d", p = l / 1.7, (C = 9e3 + O * 5.73e1) % 0550, (int) i);
      d += T * (4.5e-1 - 14 / l * X - a * 130 - J * 1.4e-1) * _ / 1.25e4 + F * _ * v;
      P = (T * (47 * l - m * 52 + E * 94 * D - t * 3.8e-1 + u * 2.1e-1 * E) / 1e2 + W * 179 * v) / 2312;
      select(p = 0, 0, 0, 0, & G);
      v -= (W * F - T * (6.3e-1 * m - l * 8.6e-2 + m * E * 19 - D * 25 - 1.1e-1 * u) / 1.07e4) * _;
      D = cos(o);
      E = sin(o);
    }
}
```

FIGURA 4

Codice C vincitore di un'edizione del IOCCC prima e dopo il pretty printing (solo una porzione)

Due storie di successo in ambito industriale

Tra i tanti *tool* di *reverse engineering* quelli che forse hanno avuto un maggior impatto a livello industriale sono stati *Codecrawler* [16] e *Rigi* [9].

Il *Codecrawler* è un *tool open-source* di visualizzazione del codice che permette diversi tipi di viste grafiche. Questo tool, scritto completamente in Smalltalk, è stato usato con successo in diversi progetti industriali di *reverse engineering*. La tabella seguente (presa da [3]) fornisce una lista di sistemi (industriali e non) che sono stati analizzati usando questo *tool*. Per questioni di riservatezza industriale gli autori non hanno potuto fornire descrizioni dettagliate dei sistemi analizzati (per questo motivo nella tabella i sistemi industriali sono indicati in modo generico con S1, S2, S3 ecc.).

UNA SELEZIONE DI SISTEMI ANALIZZATI CON CODECRAWLER

Sistema	Linguaggio	Linee di codice	Classi
S1 (Network Switch)	C++	1.200.000	2300
S2 (Network Switch)	C++/Java	140.000	400
S3 (Multimedia)	Smalltalk	600.000	2500
S4 (Payroll)	Cobol	40.000	-
Squeak (Multimedia Env.)	Smalltalk	300.000	1800
JBoss	Java	300.000	4900
S5 (Logistics)	C++	120.000	300

Come si può vedere dalla tabella il *tool* è stato utilizzato per analizzare sistemi scritti in linguaggi differenti (ad oggetti ed imperativi) e di dimensioni molto diverse tra loro. Gli autori sostengono che il *tool* non ha problemi di scalabilità e che il tempo medio impiegato per l'analisi dei sistemi inseriti nella tabella è stato per ognuno meno di una settimana di lavoro. Il risultato tipico prodotto per ogni sistema analizzato è stato un *report* contenente le viste polimetriche più significative e una lista di possibili problemi individuati (classi/moduli troppo estesi, metodi troppo lunghi, attributi o metodi non usati, *dead code* ecc.). Per i sistemi di dimensione media sono stati proposti anche interventi di *restructuring* e *re-engineering*. In un sistema, è stato proposto, per ridurre la complessità eccessiva della gerarchia della classi, l'utilizzo del design pattern "*template method*" [4]. I progettisti di questi sistemi, spesso inizialmente scettici, hanno giudicato l'informazione estratta dal *CodeCrawler* come estremamente rilevante [3] utilizzandola, in certi casi, per scopi di documentazione o come base per futuri lavori di ristrutturazione. La conclusione finale dei progettisti del *CodeCrawler* [3] è che le viste polimetriche non sostituiscono completamente l'ispezione e la lettura del codice ma aiutano nella fase di comprensione del codice.

L'altro *tool* che vogliamo menzionare è *Rigi* [9], un *tool* altamente personalizzabile che permette di ricavare alcune viste e *report* (decomposizione del sistema basata sui tipi, struttura a file, architettura del sistema, analisi delle librerie condivise ecc.) dal codice in modo interattivo o non interattivo. Gli autori sostengono che per sistemi molto grandi è necessario lasciare all'utente la possibilità di manipolare le viste in modo non-interattivo, ovvero facendo uso di *script*. *Rigi* supporta un linguaggio di *scripting* basato su Tcl [21] ed un esteso insieme di comandi con i quali è possibile produrre astrazioni personalizzate del sistema e *report*. Il *tool* può analizzare diversi linguaggi di programmazione. Anche Rigi è stato applicato con successo a diversi sistemi commerciali. Uno tra questi è presentato in [25], dove un sistema industriale di buone dimensioni (700.000 linee di codice) scritto in C e C++ è stato completamente analizzato con *Rigi* con l'intenzione di ottenere alcune informazioni sulla sua struttura (viste grafiche e *report* testuali) e potenziali anomalie. Anche in questo caso gli autori, per questioni di riservatezza industriale, non hanno potuto rivelare il dominio dell'applicazione e il nome del sistema. Durante la fase di *reverse engineering* gli autori hanno sfruttato la possibilità offerta da *Rigi* di personalizzare le varie analisi mediante la scrittura di *script*. La prima decomposizione ottenuta è stata quella a livello architetturale. Lo scopo di questa vista è quello di suddividere il sistema in sottosistemi, capire le funzionalità offerte da ogni modulo e soprattutto determinare il livello di accoppiamento tra le varie sottoparti. Da questa analisi gli autori hanno ricavato alcune informazioni importanti: il sistema è composto di sette sottosistemi principali molto connessi (cioè l'accoppiamento è alto) di cui cinque contenenti un interfaccia grafica (GUI). Gli sviluppatori originali hanno giudicato questa vista molto utile trovando nel loro sistema relazioni inaspettate tra i vari moduli. Un'altra vista grafica molto apprezzata è stata quella gerarchica a livello di directory, sottodirectory e file sorgenti. Tra i vari *report* utilizzati nella fase di comprensione del codice quelli più utilizzati sono stati: la gerarchia delle superclassi (relazione di ereditarietà tra classi), le variabili globali (relazione tra funzioni e variabili globali), i sottosistemi (relazione tra sottosistemi e artefatti contenuti) e le dipendenze tra i vari artefatti. Gli autori in [25] hanno concluso il loro lavoro di analisi con alcune considerazioni. Dato che il numero di routine e variabili globali è alto comparato con quello delle classi, il sistema sembra non aderire completamente al paradigma *Object Oriented*, ovvero il sistema è scritto in C++ ma segue uno stile imperativo (questa affermazione è confermata anche da altre analisi). Visto che sono presenti 412.315 linee di commento su 700.000 linee globali di codice, gli autori hanno concluso che il sistema è ben documentato.

triche del software). Nella figura 5 viene mostrata la vista "complessità del sistema" prodotta dal *CodeCrawler* usando come input un sistema medio scritto in Smalltalk. Le metriche usate sono il numero di attributi (larghezza dei nodi), il numero di metodi (lunghezza dei nodi) e il numero di linee di codice (colore) per ogni classe del sistema. Dalla figura 5 è possibile capire che il sistema presenta alcune classi con un numero

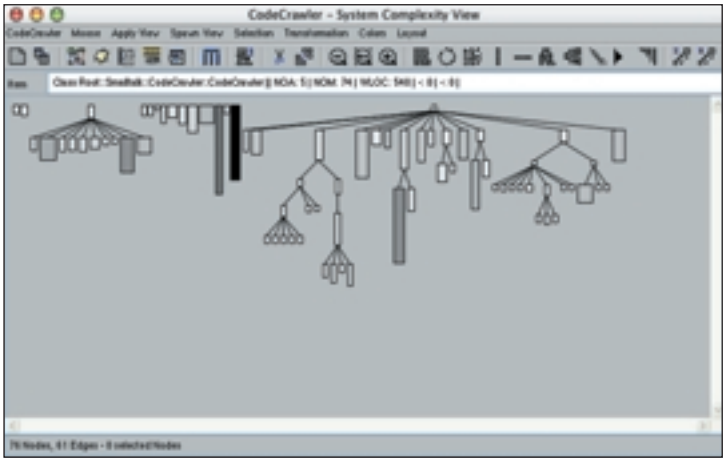


FIGURA 5

Vista "complessità del sistema" prodotta da CodeCrawler

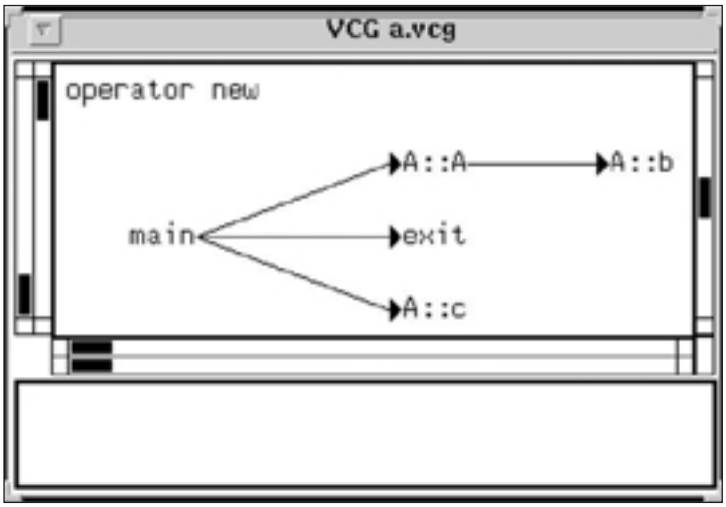


FIGURA 6

Grafo delle chiamate. Il main chiama il costruttore della classe 'A', 'exit' e il metodo 'c' della classe 'A'. A sua volta il costruttore della classe A chiama il metodo b

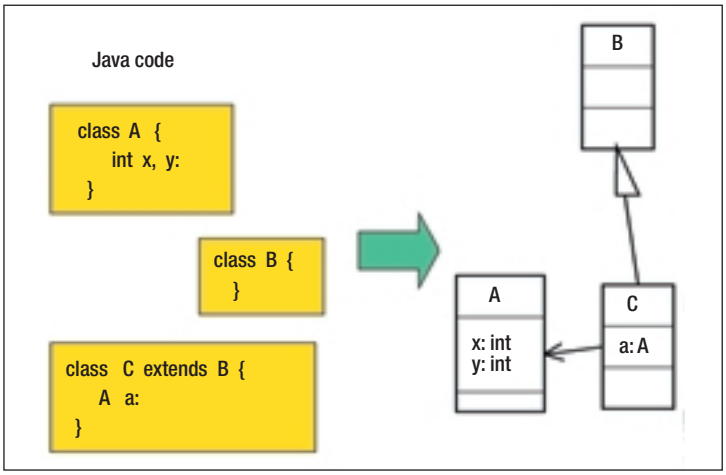


FIGURA 7

Estrazione del diagramma delle classi UML da codice Java

eccessivo di metodi e molte linee di codice e che quindi andrebbero ristrutturate;

3. ***generatori di diagrammi***: analizzano il codice sorgente e producono in output dei diagrammi che rappresentano alcune proprietà del sistema. Tra i diagrammi generati troviamo il "data flow", il grafo delle chiamate (Figura 6), il grafo delle dipendenze tra file e il grafo delle relazioni di ereditarietà tra le varie classi di un sistema ad oggetti. Un *tool* commerciale che rientra in questa categoria e produce in output diversi tipi di diagrammi, tra i quali il grafo delle chiamate e il grafo delle dipendenze tra file, è *Imagix 4D* [13];

4. ***tool di analisi***: recuperano dal codice sorgente un insieme di metriche utili ai fini della comprensione. Alcune metriche sono semplici, come le linee di codice o il numero di metodi/funzioni per file, altre sono più complesse come la coesione e l'accoppiamento tra moduli;

5. ***tool di recupero del design***: recuperano le relazioni interne tra le varie componenti software (per esempio, relazioni tra classi nei sistemi ad oggetti; Figura 7) e producono in output un diagramma che rappresenta, in un qualche formalismo, il design del sistema analizzato. Esempi di *tool* che ricadono in questa categoria sono *Rational Rose* [14] ed *EclipseUML* [15]. Questi *tool* estraggono diagrammi UML da codice Java;

6. ***tool di generazione ed analisi delle tracce di esecuzione***: l'uso di informazione dinamica, cioè informazione raccolta durante l'esecuzione del software, è stata utilizzata spesso nel contesto del *reverse engineering* ad integrazione delle tecniche statiche (cioè quelle tecniche che si basano solo sull'analisi del codice). Le tecniche statiche sono conservative, nel senso che ogni possibile comportamento del sistema sotto analisi è rappresentato nei risultati, ma meno potenti di quelle dinamiche (problema dell'indecidibilità di alcune asserzioni a livello statico). Le tecniche di analisi dinamica hanno tra i vantaggi la garanzia dell'esistenza di almeno un caso in cui il comportamento del sistema è quello descritto dall'analisi. Tra gli svantaggi troviamo la parzialità delle analisi (non tutti i comportamenti del sistema sono rappresentati nei risultati) e il limite in termini di scalabilità ed interpretazione dei risultati;

7. ***tool di pattern matching***: ricercano, all'interno del codice, i cosiddetti pattern, ovvero soluzioni software consolidate per un problema ricorrente. I pattern possono essere a diversi livelli di astrazione: a livello di design (design pattern [5]), a livello di architettura (per esempio, l'architettura *client-server*) e a livello concettuale. Il riconoscimento in modo automatico di alcuni *pattern* a livello sintattico o meglio ancora di alcuni concetti a livello semantico [20] semplifica la fase di comprensione del codice.

8. LIMITI E POTENZIALITÀ

Bisogna essere realistici a riguardo dei risultati che si possono ottenere da un'operazione di *reverse engineering*. Sia in ambito industriale che in quello accademico è risaputo che il *reverse engineering* è un compito laborioso, difficile e dai costi molto elevati. Inoltre il *reverse engineering* non è mai fine a se stesso; dopo la fase di *reverse engineering* spesso seguono fasi altrettanto difficili e laboriose come il *re-structuring* o il *re-engineering* (vedere il riquadro).
Il limite più grande di questo approccio è che una parte sostanziale di lavoro deve essere svolta manualmente. I *tool* sono utili, ma vanno "guidati" nel processo di recupero delle informazioni, ed inoltre i risultati ottenuti non sono quasi mai immediatamente usabili; per essere utili devono essere raffinati manualmente.
I risultati prodotti in modo automatico devono essere, da un lato, integrati, infatti, non tutte le informazioni utili possono essere ricavate dal codice o dalla sua esecuzione (*informazione mancante*) e, dall'altro, filtrati (*sovrabbondanza di informazione*). Capita spesso che la quantità di informazioni prodotte da un *tool* di *reverse engineering* sia ingestibile.
Per esempio, nel caso dei *tool* che estraggono diagrammi UML spesso l'informazione mancante è molta:

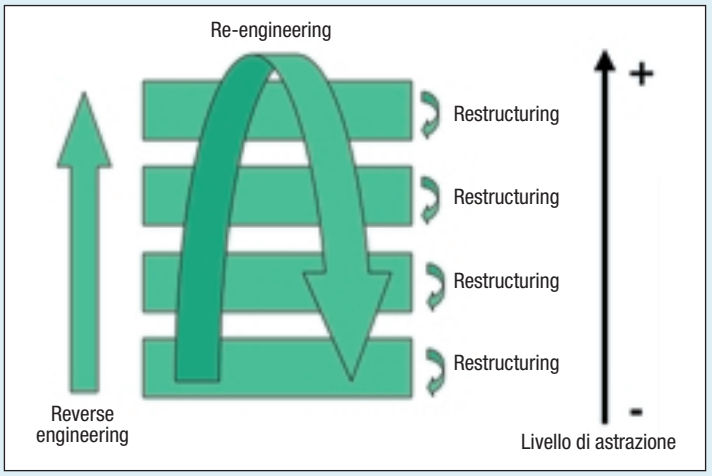
- una volta implementate, aggregazione e associazione sono indistinguibili, per cui il *tool* non può scegliere tra le due (solitamente quando non è possibile decidere il *tool* fa la scelta più conservativa, ovvero opta per le associazioni);
- la molteplicità non può essere recuperata dal codice (staticamente è indecidibile determinare il numero di entità coinvolte in una data relazione);

Forward engineering, restructuring e re-engineering

Sono termini strettamente connessi al *reverse engineering* [2].

Forward engineering è il tradizionale processo di sviluppo del software. Inizia con l'analisi dei requisiti e termina con l'implementazione del sistema. In un certo senso il *reverse engineering* può essere considerato l'operazione inversa. Si analizza il codice sorgente con l'intenzione di derivare una rappresentazione più astratta del sistema di partenza.

Restructuring è una trasformazione di un programma da una rappresentazione ad un'altra che preserva il comportamento esterno del sistema, ovvero le funzionalità. I *refactoring* proposti da Fowler [4] ricadono in questa categoria. È importante sottolineare che la trasformazione avviene allo stesso livello di astrazione, ovvero tra codice e codice oppure tra design e design. Un esempio di *restructuring* a livello di codice è la conversione di un programma che fa uso di salti incondizionati (codice a "spaghetti") in uno strutturato senza i comandi di "goto". Un esempio di *restructuring* a livello di design è invece la sostituzione di una struttura dati con un'altra (per esempio: sostituire il *file system* con un DBMS).



Relazione tra i vari termini

Re-engineering è un operazione composta da due attività (si veda la Figura): analisi/comprensione del sistema (*reverse engineering*) e ricostruzione dello stesso in una nuova forma (*forward engineering*). Il processo di *re-engineering* può includere, a differenza del *restructuring*, delle modifiche rispetto ai requisiti del sistema originale.
Un'operazione di *re-engineering* può essere eseguita per diverse ragioni. La migrazione di sistemi *legacy*, il riuso o la sicurezza del codice, la migrazione verso linguaggi più evoluti (per esempio: da C a Java) oppure la migrazione di sistemi sul Web.

• non è possibile ricavare i nomi delle relazioni, i ruoli, i vincoli e le proprietà;
• quando vengono utilizzati i contenitori debolmente tipati (per esempio, in Java 1.4 i *Vector* o le *LinkedList*) le classi effettive degli oggetti contenuti non possono essere recuperate staticamente.

L'altro difetto dei *tools* di estrazione del design è la sovrabbondanza di informazione. Spesso i diagrammi recuperati sono troppo grandi e quindi poco usabili. In un sistema ad oggetti medio, nell'ordine delle 20 mila linee di codice, è abbastanza comune avere 50-100 classi. In casi tipici come questo, il diagramma delle classi estratto dal codice è difficile da leggere per un umano le cui abilità cognitive sono attorno alle 10 entità come limite approssimativo.

Esistono due possibili soluzioni per cercare di risolvere questo problema. Il filtraggio, operazione guidata dall'utente che esclude dal diagramma generato l'informazione irrilevante, e le viste multiple [24]. Con le viste multiple l'utente suddivide il sistema in viste e, durante la fase di *reverse engineering*, decide quali elementi (per esempio in un sistema ad oggetti: classi, metodi, campi ecc.) appartengono a quale vista.

Le tecniche di *reverse engineering*, originariamente pensate per i sistemi *legacy*, si sono rivelate potenzialmente utili anche in supporto alla comprensione di sistemi software scritti secondo approcci più moderni. In particolare le tecniche di *reverse engineering* sono potenzialmente applicabili ai sistemi *open source* [19] che solitamente sono rilasciati senza o con poca documentazione. Un'altra potenzialità è nell'ambito dei processi agili (per esempio, *Extreme Programming* [6]). Nei processi agili si riconosce la centralità del codice; nel ciclo di sviluppo non sono previste esplicitamente né la fase di analisi né la fase di design. Questo fa sì che non vi sia necessariamente della documentazione ad accompagnare il codice sorgente.

9. CONCLUSIONI

I sistemi *legacy* non sono dei dinosauri in via di estinzione, bensì dei sistemi molto difficili da trattare con i quali dobbiamo convivere. Il *reverse engineering* di questi sistemi è un compito laborioso, difficile e molto costoso. I *tool* possono aiutare in questo processo anche se però bisogna essere realistici sui risultati che si possono ottenere. Se consideriamo come valida la definizione forte di *reverse*, ovvero processo di recupero delle specifiche in modo automatico, allora dovremmo concludere che il *reverse engineering* ha fallito lo scopo. Lo stato dell'arte è molto lontano da questi risultati e non si sa nemmeno se in futuro questi potranno essere raggiunti. Se invece indeboliamo la definizione e ci accontentiamo di recuperare viste astratte in modo semi-automatico, ovvero consideriamo il *reverse engineering* come un processo di estrazione cooperativa (*tool* automatici e programmatori), allora potremmo concludere che l'ingegneria inversa è stata ed è un successo.

Bibliografia

[1] Bennett K.: *Legacy systems: coping with stress*. IEEE Software, Vol. 12, Issue 1, Jan. 1995, p. 19-23.

[2] Chikofsky E.J., Cross J.H.: *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, IEEE Computer Society, January 1990, p. 13-17.

[3] Ducasse S., Girba T., Lanza M., Demeyer S.: Moose - A Collaborative and Extensible Reengineering Environment. *Capitolo del libro Tools for Software Maintenance and Reengineering, RCOST /Software Technology Series*. Franco Angeli, 2005, p. 55-71.

[4] Fowler M.: Refactoring: *Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[5] Gamma E., Helm R., Johnson R., Vissides J.: *Design patterns: elements of reusable object-oriented software*. Addison Wesley, October 1994.

[6] Ghezzi C., Monga M.: Extreme programming: programmazione estrema o revisionismo estremista?. *Mondo Digitale*, n. 4, dicembre 2002.

[7] http://en.wikipedia.org/wiki/Reverse_engineering

[8] http://smallwiki.unibe.ch/codecrawler/anon-exhaustivelistofsoftwarevisualizationtools/

[9] http://www.rigi.csc.uvic.ca.

[10] http://www.semdesigns.com/

[11] http://www.fatesoft.com/s2f/

[12] http://sourcenav.sourceforge.net/

[13] http://www.imagix.com/products/screens.html

[14] http://www-306.ibm.com/software/awdtools/developer/rosexde/

[15] http://www.omondo.com/

[16] Lanza M.: *CodeCrawler - Lessons Learned in Building a Software Visualization Tool*. CSMR 2003, 7-th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, p. 409-418.

[17] Lehman M.M., Belady L.: *Program evolution – Processes of software change*. London, Academic Press, 1985.

[18] Massari A., Mecella M.: Il trattamento dei legacy system. *Capitolo 2 del libro Sistemi informativi*, Vol. 5. Franco Angeli, 2001.

[19] Meo A.R.: Software libero e Open Source. *Mondo Digitale*, n. 2, giugno 2002.

[20] Gold N.: *Hypothesis-Based Concept Assignment to Support Software Maintenance*. ICSM 2001, 17-th IEEE International Conference on Software Maintenance, IEEE Computer Society, p. 545-454.

[21] Ousterhout J.K.: *An introduction to Tcl and Tk*. Addison-Wesley, 1994.

[22] Quilici A.: *Reverse engineering of legacy systems: a path toward success*. ICSE 1995, 17-th International Conference on Software engineering, Seattle, Washington, United States, p. 333-336.

[23] Sommerville I.: *Software engineering*. Addison Wesley, 2000.

[24] Spinellis D.: *Drawing UML Diagrams with UMLGraph*. Disponibile all'indirizzo: http://www.spinellis.gr/sw/umlgraph/.

[25] Wong K., Tilley S.R., Müller H.A., Storey M.-A.D.: *Structural redocumentation: A case study*. IEEE Software, January 1995, p. 46-54.

FILIPPO RICCA è un ricercatore presso l'ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, di Trento. Si è laureato e ha ottenuto il titolo di dottore di ricerca presso l'Università di Genova (Dipartimento di Informatica a Scienze dell'Informazione). Dal 1999 fa parte del gruppo di ricerca in Software Engineering dell'ITC-irst. I suoi interessi di ricerca attuali includono il reverse engineering, gli studi empirici, le applicazioni Web, il testing e le trasformazioni automatiche di codice.
E-mail: ricca@itc.it

PAOLO TONELLA è un ricercatore senior presso l'ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, di Trento. Si è laureato e ha ottenuto il titolo di dottore di ricerca presso l'Università degli Studi di Padova. Dal 1994 fa parte del gruppo di ricerca in Software Engineering dell'ITC-irst. È autore del libro "Reverse Engineering of Object Oriented Code", pubblicato da Springer nel 2005. I suoi interessi di ricerca attuali includono il reverse engineering, la programmazione orientata agli aspetti, gli studi empirici, le applicazioni Web ed il testing.
E-mail: tonella@itc.it