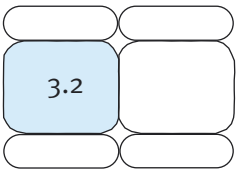
# MIGRAZIONE DI SISTEMI SOFTWARE LEGACY

**Paolo Tonella**
**Mariano Ceccato**
**Davide Marchignoli**
**Cristina Matteotti**
**Thomas Roy Dean**

Diversi produttori di software devono la loro posizione sul mercato a sistemi software sviluppati parecchie decadi fa, con tecnologie che sono diventate obsolete. Se dal punto di vista economico tali sistemi rappresentano una risorsa inestimabile, in quanto codificano una quantità enorme di regole di business e di conoscenza, dal punto di vista tecnico pongono notevoli problemi. In questo articolo riportiamo una sintesi dell'esperienza maturata nel corso della migrazione verso Java di un grosso applicativo bancario, considerando, in particolare, la strutturazione del flusso di controllo e del modello dati.




## 1. INTRODUZIONE

Molti sistemi software legacy (tipicamente in ambito bancario) sono difficili e costosi da mantenere ed evolvere a causa delle tecnologie e dei linguaggi di programmazione usati. Spesso tali sistemi derivano da software scritto diverse decadi fa, quando le tecnologie disponibili non erano ancora fortemente orientate alla strutturazione del flusso di controllo e del modello dei dati. La riscrittura di tali sistemi da zero non è di solito praticabile, in quanto si tratta di sistemi di grosse dimensioni (anche svariati milioni di linee di codice) che contengono in forma implicita tutte le regole di business relative al dominio su cui operano. Inoltre non è di solito possibile fermare la manutenzione di questi sistemi per il tempo necessario a riscrivere il software. L'alternativa che minimizza il rischio e consente un passaggio maggiormente controllato verso tecnologie più moderne ricade nella categoria del *reengineering* o della migrazione [9].

I problemi principali che si incontrano in un progetto di migrazione da un linguaggio di programmazione e una tecnologia obsoleti verso una piattaforma moderna hanno a che vedere con il grado di strutturazione delle istruzioni e dei dati. La disponibilità di istruzioni di salto incondizionato (GOTO) consente di scrivere codice estremamente non-strutturato (codice a "spaghetti"), non consentito dai linguaggi di programmazione più recenti. La possibilità di posizionare le variabili con un *overlay* arbitrario in memoria consente d'altro lato di definire un modello dati non-strutturato, in cui la definizione (scrittura) di una variabile influenza il contenuto di altre variabili che si sovrappongono ad essa parzialmente o completamente.

In questo articolo descriviamo i problemi relativi alla strutturazione del flusso di controllo e del modello dati, e gli algoritmi e le strategie che si possono adottare per affrontarli. Le tecniche descritte nell'articolo sono piuttosto generali e possono essere applicate a sistemi software scritti in linguaggi diversi e basati su tecnologie diverse, ma accomunati da un flusso di controllo e/o da un'organizzazione delle strutture dati di tipo non-strutturato. Per quanto riguarda il flusso di controllo, diversi linguaggi di programmazione ampiamente usati

in sistemi legacy fanno ricorso a istruzioni di salto non condizionato. Per esempio il Cobol e l'RPG. Per quanto riguarda le strutture dati, il problema si riscontra principalmente nei linguaggi di più basso livello (per esempio, *mainframe assembly*); in parte anche in linguaggi di più alto livello come il Cobol (istruzione REDEFINE). Contrariamente a quanto si potrebbe pensare,linguaggi di basso livello come l'*assembly* dei mainframe, sono ampiamente usati nei sistemi legacy, talvolta scritti interamente in tali linguaggi, ma più spesso contenente moduli specifici scritti in tali linguaggi. Dunque, entrambi i problemi affliggono un numero elevato di sistemi legacy e necessitano delle tecniche loro dedicate nei progetti di *re-engineering* e migrazione verso linguaggi di programmazione moderni.

In questo articolo affronteremo i due problemi menzionati qui sopra con riferimento a un caso di studio particolare. Si tratta di un'applicazione bancaria legacy di grosse dimensioni (8 milioni di linee di codice). Il linguaggio di partenza è il BAL (*Business Application Language*), un linguaggio di programmazione proprietario che richiede un ambiente di sviluppo e una piattaforma di esecuzione proprietari (B2U, *Business under Unix*). Il linguaggio target è Java su *application server* e data base relazionale. Per la migrazione di questo applicativo verso Java abbiamo sviluppato uno strumento di trasformazione automatica del codice, *bal2java*. Questo strumento, scritto nel linguaggio di manipolazione del codice TXL [4], implementa gli algoritmi di eliminazione dei GOTO, descritti nel paragrafo 2 di questo articolo, e l'algoritmo di strutturazione del modello dati, delineato nel paragrafo 3. Nonostante gli algoritmi siano descritti con riferimento al linguaggio BAL, la loro applicabilità va ben oltre tale linguaggio, estendendosi sostanzialmente ad ogni linguaggio di programmazione che ammetta flusso di controllo o modello dati non strutturati. Nel paragrafo 4 sono riportati alcuni dati sperimentali raccolti nel corso della migrazione. L'articolo si conclude con una discussione dell'esperienza maturata e degli insegnamenti che se ne possono trarre. Insegnamenti che riteniamo possano essere utili a qualunque azienda si trovi a dover affrontare un progetto di *re-engineering* simile quello descritto come caso di studio.

## 2. STRUTTURAZIONE DEL FLUSSO DI CONTROLLO

I sistemi software legacy sono spesso scritti in linguaggi che non supportano costrutti iterativi quali: WHILE, DO-WHILE, FOR, BREAK, CONTINUE. In tali sistemi, il flusso di controllo viene definito mediante l'uso del costrutto di salto incondizionato: GOTO. I programmatori sono dunque costretti a ricorrere a codice non strutturato per ottenere la semantica dei costrutti mancanti.

Il linguaggio BAL, usato per lo sviluppo del nostro caso di studio, il sistema software Gesbank, originariamente non supportava alcun costrutto iterativo tranne il FOR. In seguito sono stati aggiunti al linguaggio costrutti iterativi quali il WHILE, DO-WHILE, BREAK e CONTINUE. Tuttavia diverse porzioni del sistema Gesbank erano già state sviluppate e i programmatori erano oramai abituati a ricorrere al costrutto GOTO in diversi contesti, avendo elaborato una serie di idiomi di programmazione basati sul GOTO. L'introduzione dei nuovi costrutti non ha cambiato radicalmente le abitudini dei programmatori, i quali spesso realizzano nuove funzionalità tramite "copia-ed-incolla" di frammenti di codice esistente. La versione attuale di Gesbank, circa 8 milioni di linee di codice, contiene 0.5 milioni di istruzioni GOTO.

In letteratura esistono diversi approcci per l'eliminazione automatica delle istruzioni GOTO dal codice (per esempio, [1, 6, 8, 10]). Tuttavia le soluzioni disponibili hanno spesso un impatto molto negativo sulla qualità del codice risultante, soprattutto in presenza di GOTO cosiddetti "irriducibili" (coppie di GOTO aventi direzioni opposte e che si intersecano). Purtroppo tali GOTO sono presenti in numero non trascurabile nel codice Gesbank. Poiché il nostro obiettivo è anche quello di evitare il deterioramento della qualità del codice, non è stato possibile applicare uno degli approcci esistenti così come disponibile. È stato necessario valutare attentamente l'impatto di ciascun approccio sulla qualità del codice risultante, in modo da scegliere il migliore o la combinazione dei migliori sulle caratteristiche specifiche del codice Gesbank. In generale, ogni specifico approccio all'eliminazione dei GOTO avrà una serie di casi in cui produce codice leggibile e simile a quello che verrebbe prodotto manualmente da un intervento di rimozione dei GOTO,

ma potrà anche avere una serie di casi "degeneri" in cui i GOTO vengono eliminati al prezzo di un deterioramento della leggibilità del codice. Per questo motivo, riteniamo che una strategia efficace consista nel valutare l'impatto sulla qualità del codice risultante di un insieme di algoritmi alternativi, per poi scegliere la combinazione che garantisce il risultato migliore. Nel nostro caso di studio, abbiamo preso in considerazione quattro approcci alternativi, descritti in maggiore dettaglio nell'articolo presentato a CSMR 2008 [3]:

❑ **Pattern-based**: in questo approccio i GOTO vengono eliminati identificando idiomi (pattern) di codifica ricorrenti basati su GOTO. Per ciascun idioma viene specificata la traduzione più naturale e diretta in codice privo di GOTO. L'elenco dei pattern trattati da questo approccio è stato prodotto manualmente, sulla base di ispezioni del codice Gesbank e di interviste fatte ai programmatori BAL. Un esempio di pattern di eliminazione dei GOTO è fornito nella figura 1 (*While_goto pattern*). In questo pattern, i programmatori ottengono l'equivalente di un costrutto iterativo di tipo WHILE tramite un primo GOTO che consente di uscire dall'iterazione quando la relativa condizione (*cond*) è vera. Quando tale condizione è falsa, viene eseguito il corpo dell'iterazione (*Stmt_seq_0*), seguito da un salto non condizionato (*GOTO L0*) che di fatto realizza l'iterazione (figura 1).

❑ **Bohm-Jacopini top level**: uno dei primi approcci all'eliminazione dei GOTO è dovuto a Bohm-Jacopini [1]. Questo algoritmo di trasformazione è in principio in grado di eliminare tutte le occorrenze di GOTO (compresi i GOTO irriducibili). Tuttavia, in presenza di salti a istruzioni innestate all'interno di altre strutture di controllo, il risultato che si ottiene da questa trasformazione è piuttosto intricato e di difficile comprensione, in quanto la struttura innestata viene completamente distrutta dall'algoritmo. Quando invece i salti si riferiscono a istruzioni non innestate (ovvero, di *top-level*), la qualità del codice risultante è accettabile. Pertanto abbiamo deciso di applicare questa trasformazione solo quando sia l'istruzione GOTO che la label corrispondente hanno livello di *nesting 0* (*top-level*).

❑ **Erosa**: anche l'algoritmo proposto da Erosa [6] consente in principio di eliminare tutti i GOTO, compresi quelli irriducibili. L'algoritmo è stato sviluppato nel contesto di un compilatore ottimizzante e parallelizzante. Di conseguenza, la comprensibilità e la familiarità del codice risultante non erano un obiettivo primario. Il problema principale che talora deriva dall'applicazione di questo approccio consiste nel numero elevato di istruzioni condizionali (IF) necessarie a produrre il flusso di controllo desiderato. Talora queste istruzioni hanno un livello di *nesting* così elevato che il codice risultante è completamente illeggibile. Altre volte la trasformazione risulta assolutamente naturale e simile al codice che un programmatore scriverebbe manualmente.

❑ **JGoto**: l'ultima strategia si basa sulla trasformazione del byte code Java dopo la sua compilazione. Anziché trasformare il programma sottoposto a migrazione, viene arricchito il linguaggio target. Si tratta di una strategia a cui ricorrere quando tutte le altre possibilità falliscono. Consiste nel generare invocazioni fittizie (*jgoto(label)*, *jlabel(label)*) dove originariamente erano presenti GOTO e label. Dopo la compilazione, il byte-code risultante viene trasformato: ogni invocazione a *jlabel* viene sostituita da una *label*, inserita nel byte-code Java, e ogni invocazione a *jgoto* diventa un'istruzione GOTO, non disponibile in Java ma ammessa nel byte-code Java.

Le quattro strategie descritte qui sopra sono state implementate all'interno del tool di trasformazione del codice bal2java. Il risultato

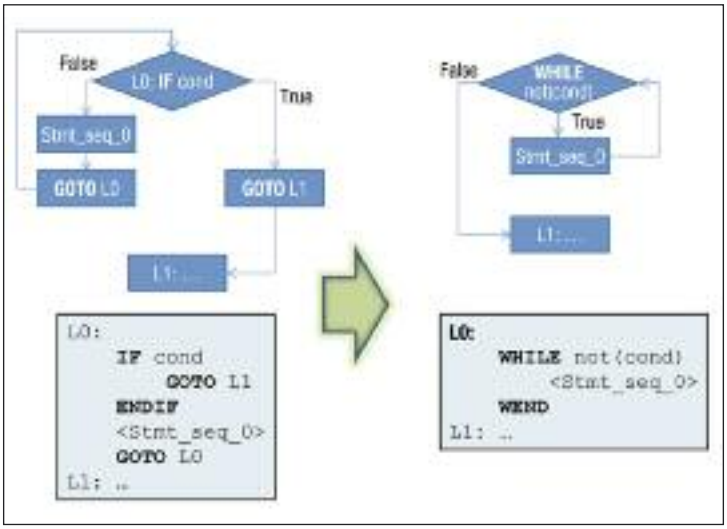



**FIGURE 1**

*While_goto pattern*

della loro applicazione al sistema Gesbank è riportato in sintesi nel paragrafo 4.

## 3. STRUTTURAZIONE DEL MODELLO DATI

Il problema di strutturazione dei dati che abbiamo affrontato nel caso di studio ha origine dalla possibilità di posizionare le variabili in maniera arbitraria in memoria. Tale possibilità e' offerta da svariati linguaggi di programmazione che espongono un modello dati di basso livello, molto vicino al posizionamento dei dati nella memoria fisica, vista come sequenza di byte. Per esempio, ciò accade per i linguaggi assembly, quali l'assembly dei mainframe. In tali linguaggi, identificare un insieme strutturato di dati, consistente di record contenitori (strutture o unioni di strutture) e dei campi contenuti, è piuttosto difficile. In letteratura, questo problema ha ottenuto scarsa attenzione, mentre invece gli autori si sono concentrati sul problema del riconoscimento di tipi complessi basati sull'uso di tipi di base (esempio, scalari)) [5] e del riconoscimento di discriminatori per le unioni di strutture [7]. Il problema di estrarre un modello dati strutturato in presenza di variabili dislocate in memoria in maniera arbitraria non appare (per quanto a nostra conoscenza) nella letteratura specialistica del settore, nonostante abbia un'importanza enorme nei progetti di migrazione e re-engineering di sistemi legacy scritti (completamente o in parte) in assembly o in linguaggi che espongono un modello piatto della memoria, vista come sequenza di byte ove posizionare le variabili. Il linguaggio di programmazione BAL, usato nel nostro caso di studio, ricade in questa categoria.

In BAL le variabili sono disposte sequenzialmente in memoria. Le variabili globali vengono disposte in uno spazio di memoria globale, mentre variabili locali e parametri di funzione vengono posizionati sulla pila dati. Il raggruppamento di variabili in record viene ottenuto in BAL collocando esplicitamente in memoria le variabili che costituiscono la stessa struttura dati, dando loro posizioni che si sovrappongono ai relativi contenitori. Il costrutto BAL principale con cui si ottiene questo risultato è il *FIELD=M*.

Nella figura 2 appare un esempio di dichiarazione di un record per una ipotetica struttura dati. Il codice (compreso tra *#ifdef* e *#endif*) comincia col dichiarare la variabile conto, di tipo stringa e di lunghezza 9 byte. L'istruzione *FIELD* che segue la dichiarazione di *conto* serve a riposizionare il puntatore alla prossima posizione disponibile in memoria per l'allocazione di variabili. Nel caso specifico, andrà a puntare all'inizio (primo byte) della variabile *conto*. Di conseguenza, le due dichiarazioni

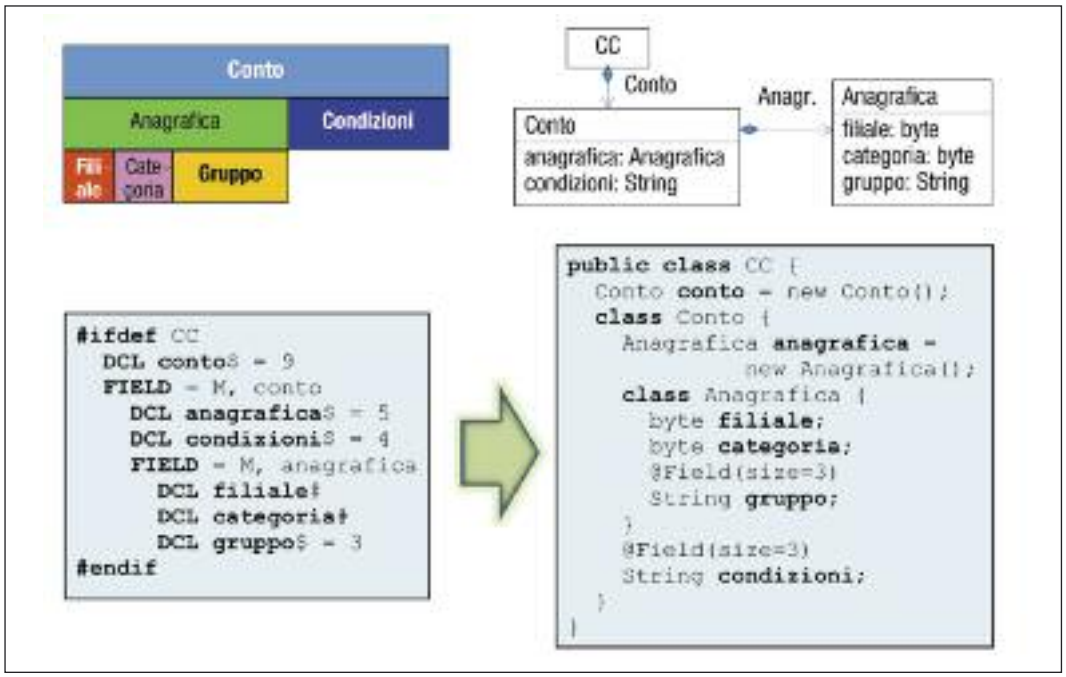



**FIGURE 2**
*Inclusione semplice con dimensionamento esatto*

successive (*anagrafica* e *condizioni*) verranno a sovrapporsi completamente a *conto*. Ciò significa che la stessa sequenza di 9 byte può essere acceduta da programma come un tutt'uno (tramite la variabile *conto*) o come due segmenti separati (*anagrafica* e *condizioni*). In modo analogo, i 5 byte che costituiscono la variabile anagrafica vengono ridefiniti (tramite *FIELD=M*) come una prima variabile *filiale* di tipo byte, seguita da *categoria* (byte) e infine da *gruppo* (stringa di lunghezza 3). L'effetto risultante dalle dichiarazioni nella figura 2 è che, ad esempio, un assegnamento alla variabile *filiale* produce il cambiamento anche del primo byte di *anagrafica* e del primo byte di *conto*. Queste tre variabili sono infatti nomi diversi dati a regioni di memoria che partono dalla stessa posizione di memoria e dunque condividono una medesima sequenza di byte.
L'uso di strutture dati posizionate direttamente sulla memoria fisica (sequenza di byte) ha diverse conseguenze negative:

1. i record non introducono alcuno scope lessicale aggiuntivo: lo spazio dei nomi è piatto e non c'è l'equivalente della notazione puntata (esempio, *conto.anagrafica.filiale*);
2. è responsabilità dei programmatori garantire il corretto dimensionamento delle variabili;
3. ci sono molti modi diversi di esprimere esattamente la stessa struttura dati.

Tali caratteristiche del modello dati di partenza rendono difficile la ricostruzione di un modello dati strutturato, quale quello imposto da Java. Per esempio, la dimensione della variabile *conto*, che contiene l'intero record, potrebbe essere erroneamente dichiarata pari a 8 o 10 byte. Le istruzioni FIELD=M che definiscono i campi dei sotto-record potrebbero apparire in ordine diverso e con variazioni, pur risultando nella stessa dislocazione in memoria delle variabili.
Nella parte destra della figura 2 mostriamo la traduzione in Java della struttura dati BAL di sinistra. Le dichiarazioni di variabili innestate (tramite *FIELD=M*) diventano classi contenute, come illustrato dalla relazione di composizione (*part-of*), rappresentata nel diagramma delle classi che nella figura 2 compare sopra alla traduzione in Java. Le stringhe BAL usate come contenitori di record diventano oggetti Java. Per esempio, le stringhe *conto* e *anagrafica* diventano due oggetti (*conto* e *anagrafica*) dichiarati come attributi delle classi *CC* e *Conto* rispettivamente. La traduzione nella figura 2 si basa sull'ipotesi che i record siano acceduti attraverso i loro campi (le foglie dell'albero di composizione delle classi generate) o come un tutt'uno attraverso un contenitore. Quando un intero record viene letto o scritto, il codice Java generato ricorre a primitive di serializzazione (*readFrom, writeTo*) degli oggetti generati.
Non sempre la disposizione delle variabili in memoria rispetta la struttura illustrata nella figura 2, in cui ogni ridefinizione si limita a spezzare un contenitore più grande in sotto-campi più piccoli. Può accadere che su uno stesso contenitore siano definite più viste indipendenti. Quando ciò accade, il contenitore definisce una *union* di più viste (*varianti*) alternative. In Java non è disponibile alcun supporto nativo per le union, pertanto nella traduzione è necessario ricorrere a un insieme di interfacce (per le varianti) e a una classe che realizza la union implementando tutte le interfacce. Ciò consente di accedere ai dati contenuti nella classe attraverso una qualsiasi delle viste disponibili. La consistenza tra queste viste è garantita da codice generato automaticamente, in grado di cambiare la variante attiva, quando viene acceduta una variante diversa da quella correntemente attiva, e in grado di copiare i dati dalla variante corrente su quella che deve essere attivata, se necessario.
Oltre alle union, le strutture dati di partenza possono presentare altri problemi, relativi a:

1. **inversioni**: contenitore e campi contenuti (esempio, *conto*, {*anagrafica, condizioni*} nella figura 2) possono essere dichiarati in ordine inverso (prima le parti e poi il contenitore);
2. **contenitore mancante**: il contenitore potrebbe essere assente;
3. **dimensionamento scorretto**: le dimensioni dei sotto-campi potrebbero non corrispondere a quelle dei rispettivi contenitori.

La soluzione al problema 1 consiste nel riconoscere l'inversione e nel ripristinare automaticamente l'ordine corretto delle dichiarazioni. Per il problema 2 si ricorre all'introduzione automatica di un contenitore fittizio. Il problema 3 può essere affrontato riconoscendo una serie di pattern e applicando un'opportuna euristica di strutturazione dei dati BAL per ciascun caso [3]. Per esempio, l'ultimo campo conte-

nuto in un record sfora la dimensione del record, ma è seguito immediatamente dalla dichiarazione di un altro record. In tale caso è ragionevole ipotizzare che il campo appartenga interamente al record precedente.

## 4. CASO DI STUDIO

Il sistema Gesbank sottoposto a migrazione da BAL a Java è un applicativo bancario che supporta tutte le funzionalità necessarie all'operatività di una banca: gestione dei conti correnti, gestione dei prodotti finanziari, operazioni allo sportello, comunicazioni alla banca centrale e alle altre authority, comunicazioni interbancarie, generazione di statistiche e di reportistica. L'interfaccia utente è a caratteri e l'architettura complessiva è client-server. L'ambiente di esecuzione è una piattaforma proprietaria detta B2U.
L'applicativo è di dimensioni notevoli (circa 8 milioni di linee di codice). Il linguaggio BAL consente direttive di *pre-processing*, che comportano un fattore di espansione del codice pari a circa 1,63. I dati persistenti risiedono all'interno di tabelle ISAM, per un totale di 1.339 file ISAM e 5.893 tabelle ISAM. I dati persistenti sono descritti in una tabella ISAM particolare, detta *dizionario*. Poiché il modello dati usato nel dizionario è lo stesso supportato dal linguaggio BAL (basato su FIELD=M), la generazione delle classi Java che modellano i dati persistenti utilizza gli stessi algoritmi definiti per le strutture dati BAL (vedi paragrafo precedente).
Tutte le strategie di eliminazione dei GOTO discusse nel paragrafo 2 sono state applicate a tutto il codice Gesbank. Il codice risultante è stato analizzato in modo da identificare la strategia o la combinazione di strategie maggiormente appropriate. I risultati sono discussi in dettaglio nell'articolo presentato a CSMR 2008 [3]. Nella tabella 1 mostriamo una sintesi di tali risultati. Per le quattro strategie considerate (per Erosa distinguiamo l'applicazione *forward* dall'applicazione *backward* dell'algoritmo) possiamo confrontare la percentuale di GOTO rimossi, il massimo livello di *nesting* presente nel codice trasformato ed il tempo di computazione richiesto dalla trasformazione automatica del codice. La strategia JGoto non elimina alcun GOTO dal codice sorgente, in quanto trasforma il byte-code risultante dalla compilazione del codice Java. Il relativo *overhead* di compilazione è trascurabile (pertanto non compare alcun tempo di trasformazione per questa strategia).
Dai dati della tabella 1 possiamo concludere che le strategie Pattern-based e Bohm-Jacopini top-level sono estremamente efficaci, in quanto consentono di rimuovere una percentuale significativa di GOTO (ovviamente, non tutti), con un impatto minimo sulla qualità del codice risultante (il livello di *nesting* aumenta al più di 1 o 2 per le due strategie). Al contrario, Erosa è sì in grado di rimuovere tutti i GOTO, ma produce un incremento inaccettabile del livello di *nesting*. Per questo motivo, si è deciso di scartare questo approccio e di ricorrere a JGoto per i casi residui in cui Pattern-based e Bohm-Jacopini top-level non sono in grado di eliminare i GOTO.
La tabella 2 consente di valutare l'efficacia delle euristiche di strutturazione dei dati BAL presentate nella sezione precedente (ulteriori dettagli sono forniti in [3]). Il caso 1, in cui si ha un dimensionamento esatto (come nella figura 2) dei contenitori, copre la maggior parte delle istanze di strutture dati presenti nel codice utente e nel dizionario. Le euristiche relative ai casi 2 e 3 consentono di aumentare significativa-

| Strategy | GOTO rimossi | Max nesting | Tempo |
|---|---|---|---|
| Pattern-based | 21,4% | 29 (+1) | 1 h 2 m |
| Bohm-Jacopini top-level | 79,3% | 30 (+2) | 22 m |
| Erosa (backward) | 100,0% | 162 (+134) | 117 h 38 m |
| Erosa (forward) | 100,0% | 207 (+179) | 144 h 22 m |
| JGoto | 0% | 28 (+0) | - |

**TABELLA 1**
*Risultati prodotti dalle quattro strategie di eliminazione dei GOTO*

| Casi | Codice utente | Dizionario |
|---|---|---|
| Caso 1: dimensionamento esatto | 425.988 | 8.279 |
| Caso 2: sottodimensionamento | 27.100 | 65 |
| Caso 3: sovradimensionamento | 13.850 | 4 |
| Caso 4: errore | 5.286 | 45 |

**TABELLA 2**
*Numero di occorrenze dei vari casi di strutture dati BAL*

| | Codice utente | Dizionario |
|---|---|---|
| Classi | 510.108 | 12.402 |
| Interfacce | 148.621 | 753 |
| Union | 29.394 | 263 |

**TABELLA 3**

*Codice Java generato per le strutture dati BAL*

mente la quota di strutture BAL tradotte automaticamente in classi Java. I casi di errore residui sono in numero sufficientemente basso da consentire un intervento manuale. Ai programmatori BAL è stato chiesto di fissare i problemi all'origine delle incompatibilità di dimensione che hanno prodotto questi errori. Spesso l'intervento necessario è stato banalmente un ridimensionamento del contenitore. In alcuni casi è stato necessario introdurre un contenitore mancante o un campo FILLER, avente lo scopo di completare la ridefinizione di una variante di una union o di un sotto-record.
Nella tabella 3 viene mostrato il risultato della traduzione in Java delle strutture dati BAL. La maggior parte delle strutture dati di partenza può essere tradotta tramite classi Java "standard". In un numero limitato di casi è stato necessario simulare le union in Java, tramite l'implementazione di interfacce, associate alle diverse varianti della union. Nella tabella 3 riportiamo il numero di union generate, nonché il numero di interfacce generate per le varianti delle union.

## 5. CONCLUSIONI

I progetti di migrazione, specialmente per sistemi software di grosse dimensioni, sono intrinsecamente ad alto rischio e diverse sono le cause che possono portare al loro fallimento. Dall'esperienza di migrazione riportata in questo articolo si possono ottenere le seguenti regole empiriche:

**1.** La valutazione dei risultati prodotti dalla migrazione del codice coinvolge più dimensioni, spesso in conflitto l'una con l'altra. Per esempio, nell'eliminazione dei GOTO abbiamo riscontrato una dicotomia tra la capacità di trattare in modo automatico tutti i possibili casi di GOTO e la manutenibilità del codice trasformato.

**2.** Dato un sistema software complesso, come Gesbank, non è possibile risolvere tutte le istanze di un problema con una singola soluzione. La varietà dei casi presenti impone la valutazione e la combinazione di una molteplicità di alternative.

**3.** Soluzioni con proprietà teoriche interessanti (esempio, capacità di trattare GOTO irriducibili) possono rivelarsi in pratica inusabili, a causa delle caratteristiche specifiche del codice sottoposto a migrazione.

**4.** È importante identificare sempre soluzioni di *backup*, (esempio, JGoto) a cui ricorrere quando si incontrano ostacoli insormontabili nel corso del progetto di migrazione, nonostante queste possano introdurre del codice non del tutto soddisfacente. Rappresentano infatti una tappa importante verso la soluzione "ideale".

**5.** I progetti di migrazione hanno obiettivi di breve e obiettivi di lungo periodo. Nel breve termine non è possibile fissare obiettivi troppo elevati in quanto irrealistici e spesso bisogna accettare dei compromessi. Tuttavia, rimane importante avere chiara la strategia di lungo termine, in modo da innescare un processo di miglioramento continuo, che continua anche quando la migrazione è terminata.

## Bibliografia


[1] Bohm C., Jacopini G: Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, Vol. 9, n. 5, 1966, p. 366-371.

[2] Ceccato Mariano, Roy Dean Thomas, Tonella Paolo, Marchignoli Davide: *Data model reverse engineering in migrating a legacy system to Java*. In: Proc. of the 15-th Working Conference on Reverse Engineering. IEEE Computer Society, Antwerp, Belgium, October 2008.

[3] Ceccato Mariano, Tonella Paolo, Matteotti Cristina: *Goto elimination strategies in the migration of legacy code to Java*. In: Kontogiannis K., Tjortjis C., Winter A., editors, In: Proc. of the 12-th European Conference on Software Maintenance and Reengineering, p. 53-62. IEEE Computer Society, Athens, Greece, April 2008.

[4] Cordy Jim: The TXL source transformation language. *Science of Computer Programming*, Vol. 61, n. 3, 2006, p. 190-210.

[5] van Deursen Arie, Moonen Leon: *Exploring legacy systems using types*. In: Proc. of the 7-th Working Conference on Reverse Engineering, 2000, p. 32-41.

[6] Erosa A.M., Hendren L.J.: *Taming control flow: a structured approach to eliminating goto statements*. In: Proc. of the Int. Conf. on Computer Languages, 1994, p. 229-240.

[7] Komondoor R., Ramalingam G.: *Recovering data models via guarded dependences*. In: Proc. of the 14-th Working Conference on Reverse Engineering, 2007, p. 110-119.

[8] Ramshaw L.: Eliminating goto's while preserving program structure. *Journal of the ACM*, Vol. 35, n. 4, 1988, p. 893-920.

[9] Ricca Filippo, Tonella Paolo: Reverse Engineering di sistemi software. Limiti e potenzialità. *Mondo Digitale*, n. 3, 2006, p. 52-62.

[10] Zhang F., D'Hollander E.H.: Using hammock graphs to structure programs. *IEEE Trans. on Software Engineering*, Vol. 30, n. 4, 2004, p. 231-245.

## Migrare, riscrivere o incapsulare? Strategie aziendali di gestione del rischio

Le aziende che operano nel campo delle tecnologie dell'informazione in generale, e dello sviluppo del software in particolare, si trovano costantemente sottoposte ad una forte pressione che spinge verso l'innovazione e l'adozione di nuove tecnologie. Aziende di successo possono vedere notevolmente ridotta la loro quota di mercato o possono addirittura essere estromesse dal mercato dai concorrenti a causa di un mancato adeguamento dei loro prodotti alle novità tecnologiche.
Nel settore bancario, a causa della criticità dei dati manipolati, l'innovazione si accompagna necessariamente ad altri requisiti importanti quali l'affidabilità, la robustezza, la scalabilità e la maturità delle tecnologie usate. Per questo motivo la spinta all'innovazione è spesso mitigata da una tendenza a mantenere soluzioni tecnologicamente superate ma affidabili (per esempio, Cobol su mainframe). D'altra parte gli utenti finali dei sistemi bancari stanno diventando sempre più competenti circa le tecnologie informatiche che usano e di conseguenza chiedono funzionalità e prestazioni in linea con lo stato dell'arte (interfacce grafiche, accesso via Web services, interoperabilità sui dati ecc.), senza ovviamente rinunciare all'affidabilità. Le aziende che operano nel settore del software bancario (come IBT) si trovano pertanto costantemente di fronte al dilemma se innovare, assumendosi il rischio legato all'adozione di nuove tecnologie, o mantenere il sistema nella sua forma attuale, dando però così un margine di sorpasso tecnologico ai concorrenti. Talvolta il dilemma si risolve in favore dell'innovazione, soprattutto se ci sono motivi per ritenere che la perdita di quota di mercato dovuta a prodotti concorrenti maggiormente innovativi sia una possibilità concreta. La valutazione strategica di IBT circa le tecnologie in uso (piattaforma proprietaria B2U, linguaggio proprietario BAL ed ambiente di compilazione ed esecuzione proprietario) è stata esattamente di questo tipo: innovare per rimanere competitivi sul mercato. L'innovazione si accompagna però a enormi rischi. Sono documentati diversi progetti di passaggio a nuove tecnologie che hanno avuto esiti fallimentari e talora disastrosi [9]. La scelta di una strategia di innovazione adeguata è pertanto fondamentale e di solito si riduce a tre alternative principali:

- **Incapsulamento (wrapping)**: è l'alternativa a rischio minore, ma è anche quella che dà meno vantaggi nel processo di sviluppo. Il software esistente viene mantenuto e i programmatori continuano a lavorare con ambiente, strumenti e linguaggi usuali. Per rendere però disponibili all'utente finale funzionalità più avanzate, il software viene incapsulato in modo da esportare procedure e dati necessari ai moduli esterni (magari sviluppati con tecnologie allo stato dell'arte), in grado di fornire le funzionalità richieste dall'utente finale.
- **Riscrittura**: è l'alternativa a maggiore rischio. I problemi principali legati a questa alternativa sono:
  1. a documentazione delle cosiddette "business rules", spesso carente o del tutto assente;
  2. la disponibilità di risorse aziendali adeguate a un progetto di riscrittura.

  Nei sistemi software legacy, spesso il codice sorgente è anche la documentazione più fedele delle regole di business implementate nel sistema. La riscrittura deve pertanto partire da un lungo e complesso processo di acquisizione di conoscenza a partire dal codice esistente prima che si possa progettare il nuovo sistema. I rischi maggiori sono relativi alla difficoltà di ottenere un insieme di specifiche completo e accurato. Le risorse necessarie a tale attività vengono inoltre sottratte al normale processo di sviluppo del software, che continua sul sistema legacy mentre viene prodotto il sistema nuovo. Di conseguenza tali risorse sono spesso inadeguate e sovraccaricate di altri compiti. È dunque difficile pianificare i tempi e i costi della riscrittura, che deve comprendere anche una fase di test tale da garantire la stessa affidabilità che possedeva il sistema in via di sostituzione. Ciò può richiedere un tempo almeno tanto lungo quanto la riscrittura stessa.
- **Migrazione**: è un'alternativa a rischio intermedio, anche se all'interno di questa alternativa troviamo uno spettro di opzioni che si avvicinano da un lato all'incapsulamento e dall'altro alla riscrittura. Il vantaggio prin-

*segue*

cipale della migrazione è che in principio può avvalersi di strumenti automatici di analisi e trasformazione del codice. Ciò consente di limitare il coinvolgimento di risorse umane sia nella fase di transizione che in quella di testing, anche se un processo totalmente automatico di solito non è praticabile. A un estremo della migrazione troviamo la generazione di codice in un linguaggio moderno (per esempio, Java da Cobol) senza alcuna pretesa di leggibilità e manutenibilità del codice. L'assunzione è che (come nell'incapsulamento) si continuerà a lavorare con il vecchio codice, ma il codice generato consentirà di agganciare in esecuzione ambienti e tecnologie moderni. All'altro estremo, si punta sull'analisi automatica (reverse engineering) per facilitare l'estrazione delle business rules, ma si lascia ai programmatori la maggior parte dell'attività di ri-codifica.
La scelta strategica di IBT è stata quella di migrare, cercando di minimizzare l'attività manuale. Si è fatto pesantemente ricorso a strumenti di trasformazione automatica del codice e di analisi del codice in supporto agli inevitabili interventi manuali. Un'altra scelta critica riguarda il rilascio all'utente finale. La strategia IBT è basata su un ambiente di esecuzione misto, che consente il rilascio incrementale dei programmi migrati.

Paolo Tonella è a capo dell'unità di ricerca SE (Software Engineering) presso la Fondazione Bruno Kessler–IRST di Trento. Si è laureato e ha ottenuto il dottorato di ricerca presso l'Università degli Studi di Padova. È autore del libro "Reverse Engineering of Object Oriented Code", Springer, 2005, oltre che di svariati articoli pubblicati su rivista e presentati a conferenza, su temi che spaziano dal reverse engineering e re-engineering, al testing del software, alla programmazione ad aspetti. Nel 2007 è stato elencato tra i primi 50 ricercatori nell'area del Software Engineering in un articolo pubblicato su *Communications of the ACM* (Vol. 50, n. 6, June 2007, p. 81-85).
E-mail: tonella@fbk.eu

Mariano Ceccato è ricercatore presso la Fondazione Bruno Kessler–IRST di Trento e insegna "Laboratorio di Analisi e Testing del Software" presso l'Università degli Studi di Trento. Ha ottenuto la laurea in Ingegneria Informatica dall'Università di Padova nel 2003 e il dottorato di ricerca in Informatica dall'Università di Trento nel 2006. È autore di numerosi articoli pubblicati su riviste e presentati a conferenze internazionali. I suoi interessi di ricerca comprendono l'analisi automatica e la trasformazione del codice sorgente, la sicurezza, il testing, e l'ingegneria del software empirica.
E-mail: ceccato@fbk.eu

Davide Marchignoli è responsabile del progetto di migrazione Gesbank presso IBT. Ha conseguito nel 2002 un dottorato di ricerca in informatica presso l'Università di Pisa occupandosi principalmente di teoria dei tipi. Da allora si occupa principalmente di progettazione di sistemi software.
E-mail: davide.marchignoli@ibttn.it

Cristina Matteotti ha conseguito nel 2007 la laurea triennale in Informatica presso l'Università degli Studi di Trento. Da allora lavora come software engineer presso la Fondazione Bruno Kessler–IRST di Trento, ove realizza tool per la trasformazione automatica del codice.
E-mail: matteotti@fbk.eu

Thomas Roy Dean è professore associato presso la Queen's University di Kingston, Canada. In passato ha condotto attività di ricerca su sistemi di controllo del traffico aereo e sulla formalizzazione dei linguaggi. Per oltre cinque anni ha lavorato presso la Legasys Corp. come ricercatore senior, occupandosi di trasformazioni avanzate del codice e di tecniche di evoluzione in un contesto industriale. I suoi interessi attuali includono la trasformazione del software, l'evoluzione dei siti Web e la sicurezza delle applicazioni di rete.
E-mail: tom.dean@queensu.ca