UPGRADE
The European Journal for the Informatics Professional
http://www.upgrade-cepis.org

Edizione italiana a cura di ALSI e Tecnoteca
http://upgrade.tecnoteca.it

Ingegneria del software basata su princìpi empirici

di
Martin Shepperd

(Traduzione italiana a cura di Danilo De Riso (ALSI – www.alsi.it) dell’articolo
Empirically-based Software Engineering
pubblicato sul Vol. IV,No. 4, Agosto 2003
della rivista online UPGrade, a cura del CEPIS)

Riassunto italiano: Questo articolo traccia una panoramica dell'attività nel campo dell'ingegneria del Software basata su princìpi empirici. Afferma che quest’area di ricerca è importante se gli operatori devono prendere decisioni sulla base di evidenze sovrastanti le proprie opinioni soggettive. L'articolo descrive 4 settori dove i dati empirici hanno migliorato la nostra comprensione della tecnologia del software. Questi sono: l'orientazione agli oggetti, le verifiche, le specifiche formali ed i fattori di fallimento di un progetto. L'articolo conclude che l'ingegneria del software basata su princìpi empirici è destinata probabilmente a crescere in importanza ma che rimangono alcune sfide aperte, perlomeno nella valutazione dei processi e dei manufatti su larga scala, nella gestione degli aspetti umani e creativi dei processi e nel superamento dei pregiudizi che inducono a non pubblicare i risultati "negativi".

Parole chiave: inchiesta, ricerca empirica, esperimento, Ingegneria del software.

1. Introduzione

L'idea dello studio dello sviluppo e della manutenzione dei sistemi software come disciplina tipica dell’ingegneria è stata portata avanti per la prima volta negli anni ‘60 da Randall ad un seminario NATO, dando origine all’ingegneria del software. Successivamente sono stati fatti molti progressi e molte delle cose che ora diamo per scontate come parte integrante del moderno sviluppo del software sono conseguenza diretta della ricerca dell’ingegneria del software di questi ultimi 35 anni. Tra di esse citiamo le tecniche di specifica dei requisiti, i metodi per la modellazione e le modalità di ragionamento riguardo l'architettura del software, la modularizzazione e la riutilizzazione dei componenti, i metodi orientati agli oggetti ed i linguaggi di programmazione, le tecniche di verifica e di convalida, i metodi di *project management*, lo sviluppo ad impostazione incrementale ed iterativa e così via.

Tuttavia, in questa lista di cose che potrebbero essere considerate come dei successi, sono molte le idee che non hanno resistito alla prova del tempo o sono state promosse come la panacea di tutti i mali ma, con il beneficio del senno di poi, si sono rivelate essere di importanza solo marginale o locale.

Il ritmo di crescita ed il tasso di cambiamento all'interno dell'industria del software è semplicemente straordinario. Ma questo ha determinato molti problemi. C’è una grande tradizione di progetti software errati rilasciati in ritardo e fuori preventivo, associati a ben noti disastri software.

Di conseguenza la richiesta di miglioramenti supera considerevolmente l’offerta! Ne può conseguire un certo margine di credulità ed una necessità di separare il grano dal fieno.

Inoltre, non si tratta solo di determinare se la nuova tecnica di ingegneria del software sostenuta sia buona, cattiva o indifferente, ma di determinare, più precisamente, in quali circostanze la tecnica si rivelerà buona, cattiva o indifferente. Il fatto che lo sviluppatore del software X utilizzi con

successo una nuova tecnica non garantisce che lo sviluppatore Y del software avrà un'uguale riuscita. X può avere politiche differenti di reclutamento e di addestramento del personale, sviluppare prodotti differenti per requisiti di qualità differenti, avere criteri di amministrazione differenti e possedere una cultura differente da Y.
Per questo motivo, una ricerca accurata basata su princìpi empirici è essenziale per modificare il ruolo dell'ingegneria del software dall'essere una disciplina basata sull'opinione, facendola diventare una disciplina basata sulla prova.
Il resto di questo articolo tratta brevemente lo sviluppo e l'evoluzione della ingegneria del software basata su princìpi empirici. Segue una descrizione di quattro settori di attività che potrebbero essere caratterizzate come valutazioni di tecnologia. Questi sono: orientazione agli oggetti, controlli, specifica formale e fattori di insuccesso di un progetto software. L'articolo si conclude riepilogando quei settori dove si sono registrati considerevoli progressi e considerando le problematiche da risolvere in futuro.

2. Una breve storia

Probabilmente uno dei primissimi studi pubblicati che discutevano di un sistema software da un punto di vista empirico e quantitativo è stato quello condotto da Benington [1] sulla base delle sue esperienze con il progetto SAGE, uno dei primi progetti software su grande scala. L'articolo contiene molte osservazioni che dovrebbero risultarci familiari, ma anche dati empirici dettagliati sui costi di sviluppo del software, inclusa la distribuzione dei costi per fase. Ciò ha permesso di constatare che il collaudo assorbiva una proporzione considerevole dei costi complessivi, di un ordine cinque volte maggiore rispetto alla codifica.
I due settori principali di interesse degli anni 60 e degli anni 70 erano il tracciamento e la previsione dei costi generali di progetto ed un lavoro molto accurato sulle misure di 'complessità' di codice (come ad esempio l'Halstead's Software Science e la misura di complessità ciclometrica di McCabe). I modelli di costo erano tipicamente basati sul formato, con molti modificatori (o *driver* di costo). Ciò ha costituito le fondamenta di modelli ben noti come COCOMO all'inizio degli anni '80 del secolo scorso. In perfetta antitesi, le misure di 'complessità' avevano a che fare con piccoli frammenti del codice e gran parte del lavoro si concentrava sui particolari della differente sintassi del linguaggio di programmazione e delle strutture di codifica.
In quel momento, guardando le cose col senno di poi, questo lavoro non fu percepito come particolarmente fruttuoso ma tuttavia ha permesso ai princìpi di convalida empirica di evolversi ed ha consentito l'affermarsi della proposizione che asserisce che è insufficiente per i ricercatori proporre semplicemente nuove misure o nuove idee senza supporto empirico. Inoltre ha costituito la base per molti perfezionamenti ed evoluzioni dei metodi empirici adatti alle problematiche tipiche dello studio dell'ingegneria del software.
Gli anni '80 e gli anni '90 sono stati caratterizzati dalla crescita e dalla diffusione dell'interesse nella ingegneria del software basata su princìpi empirici.
Le attività si sono diversificate molto, tanto che anche i prodotti diversi dal codice (cioè i disegni, le specifiche, i piani di collaudo, le guide utente) erano oggetto di studio empirico. Analogamente l'interesse si estese oltre i prodotti, per abbracciare i processi come la progettazione, la messa a punto (*debugging*), il collaudo, la comprensibilità e così via. Si è data inoltre maggior enfasi alla multi-disciplinarità, in particolare all'interessamento verso altri campi di specializzazione quali le statistiche, la psicologia, la scienza dell'amministrazione, l'economia e la teoria della misura.
Un importante sviluppo recente consiste nell'interesse crescente verso la meta-analisi. In pratica, come bisogna combinare i risultati dei numerosi studi empirici? Ciò può essere d'aiuto in due situazioni:
Prima, quanta più fiducia possiamo avere in un risultato se esso viene riportato da numerosi studi indipendenti e seconda, come ci si comporta nella situazione in cui non tutti i risultati sono concordi?
Il conteggio dei voti è una tecnica semplice ma non sempre la più efficace. Si stanno proponendo altre tecniche di combinazione dei risultati per raggrupparli più generalmente all'interno di un'area

di indagine (cfr., per esempio, [2]). Le sfide principali aperte consistono nell'accertarsi che i dati siano relativamente omogenei, per esempio che i risultati non siano influenzati dalle differenze culturali qualora gli studi siano intrapresi in paesi differenti, oppure laddove tali differenze esistano, siano ben conosciuti in modo da potere dividere appropriatamente la popolazione.

3. Alcune indagini di valutazione della tecnologia

In questa sezione, verranno trattati brevemente i risultati provenienti da quattro settori dell'ingegneria del software che sono stati oggetto di considerevole interesse e di dibattito nel corso degli anni.

3.1. Orientazione agli oggetti

Anche se le idee originali sottostanti la tecnologia orientata agli oggetti (*Object-Oriented Technology*, OOT), derivano dal lavoro sul linguaggio di programmazione Simula negli anni 60, solo negli anni 80 il lavoro ha acquisito popolarità ed il suo utilizzo è diventato più diffuso. Attualmente, C++ e Java sono ampiamente usati ed insegnati. Il paradigma OO (*Object-Oriented*) può essere considerato come l'ortodossìa dai tardi anni 90 in avanti. Tuttavia, abbiamo una conoscenza relativamente piccola, su base empirica, del comportamento dei sistemi implementati usando l'OOT. Dato che l'OOT continua ad essere pesantemente sostenuta, cresce l'esigenza di comprenderla meglio e prevederne il comportamento.

Una esempio di ricerca effettuata in una grande azienda europea di telecomunicazioni è stato riportato da Cartwright e da Shepperd [3] che hanno rivelato molte informazioni utili, specialmente per quanto riguarda la distribuzione degli errori all'interno del software.

L'organizzazione aveva circa 20.000 impiegati e più di 2.000 sviluppatori di software. E' stata adottata una regolamentazione per la progettazione e l'implementazione del software e l'azienda era accreditata ISO9000. Una attenzione particolare è stata dedicata alla qualità del software in particolare all'eliminazione dei difetti. Un 45% delle risorse è stato impiegato nel collaudo e nella simulazione. Il sistema studiato era parte di un sistema industriale *real-time* di telecomunicazioni molto più grande, che conteneva parecchi milioni di linee di codice (*Lines of code* - LOC), sviluppate negli ultimi dieci anni. Il suo corretto funzionamento era fondamentale per la salute finanziaria dell'organizzazione.

Il sottosistema è stato scritto in C++ ed è stato progettato usando il metodo di Shlaer-Mellor.

Dall'analisi, è stato identificato un totale di 259 difetti univoci, che sono stati localizzati nelle relative posizioni all'interno del software.

Ciò indica una densità generale di difetto di 1.94 *Kilo lines of code –1* (KLOC-1) che si paragona abbastanza favorevolmente rispetto ai livelli di errore citati da Hatton [4] di 2.9 KLOC-1. In effetti, la cifra è da prendere un po' con le pinze, dato che alcuni dei difetti registrati sono stati riscontrati dopo il collaudo integrato ma prima del rilascio. Tuttavia, oltre a fornire un certo tipo di *benchmark*, questo tipo di analisi può fornirci molti riscontri pratici. Per esempio, i dati hanno indicato un piccolo numero di classi molto soggette ad errore ed in effetti si è registrato un 22% puro sul 75% di tutti gli errori, ulteriore conferma di una regola di 20 su 80 segnalata da altre fonti, per esempio [5] e [6]. Ciò indica che non è ottimale distribuire lo sforzo di verifica del software in egual misura lungo il sistema.

Invece vi sono possibilità di grossi guadagni in rendimento se i componenti problematici possono essere identificati in anticipo poiché le attività di verifica possono essere svolte più efficacemente in maniera mirata.

Ciò conduce al secondo risultato utile di questo studio empirico. È possibile predire i componenti problematici usando le informazioni disponibili al momento della progettazione come il numero di eventi o di stati.

Terzo, sembra che i difetti si concentrino più considerevolmente intorno a particolari caratteristiche architetturali, più in particolare nelle strutture di ereditarietà (*inheritance*). Le classi che facevano parte di strutture di ereditarietà avevano una densità di errore più di tre volte più grande rispetto alle classi *singleton* (3.01 KLOC-1 difetti paragonato ai 0.90 KLOC-1). Questo è stato un risultato

particolarmente interessante - e confermato da un certo numero di esperimenti di laboratorio, come ad esempio [8] e [7] ed altre inchieste ad esempio [4] – poiché segnalava, come minimo, l'abuso di questo particolare meccanismo architetturale fornito dal paradigma orientato agli oggetti. Ciò era in contrasto col fatto che in quel momento molti sapientoni stavano promuovendo l'ereditarietà come la parte centrale dell'OOT. Di conseguenza, è stato ampiamente accettato che altri meccanismi quale la composizione di oggetti sono ugualmente importanti e che l'ereditarietà può essere un meccanismo inadeguato specialmente nelle circostanze dove un oggetto può cambiare ruolo. L'ingegneria del software empirica può dunque esser vista come complemento importante allo sviluppo, all'impiego ed all'evoluzione dei metodi di progettazione del software.

3.2. Controlli

Un altro settore molto interessante è l'uso dei controlli, specialmente al momento della progettazione, come tecnica efficace per la tempestiva individuazione e rimozione dei difetti. Basilarmente ciò consiste nella lettura o controllo al banco dei manufatti come la documentazione di progetto, il codice, ecc., da parte di personale indipendente. Successivamente, in genere, viene effettuata una riunione in cui sono valutati i difetti rilevati e, se accettati come validi, sono documentati. Cfr.[9] per una descrizione dettagliata.

Tuttavia, mentre tutti sono d'accordo che i controlli sono una tecnica importante, si è sviluppato invece un acceso dibattito su come questi ultimi dovrebbero essere condotti, per esempio, qual è il numero ottimale di ispettori?, E' un contatto "faccia a faccia'' essenziale?, e così via. Questi sono ancora una volta esempi di questioni di ingegneria del software che possono essere risolte efficacemente dall'analisi empirica.

Pochi anni fa sono stati eseguiti un certo numero di esperimenti e di studi. Un buon esempio recente è quello di [10] che espone i risultati di un esperimento controllato usando 169 studenti a cui è stata chiesta la revisione di un documento di specifica di requisiti di 35 pagine riempito con 86 errori. Si è riscontrato che le squadre che hanno usato una miscela di tecniche di lettura hanno sorpassato quelle che hanno usato una singola tecnica. Un'altra scoperta di rilevante importanza è la differenza nelle prestazioni fra gli individui e quindi la necessità di identificare i lettori più inefficienti in modo da potere abbinare il personale con le capacità adatte alle giuste mansioni. Inoltre sono stati tratti modelli empirici, in termini di costo-beneficio, per determinare la formazione ottimale della squadra e lo sforzo di controllo necessario per diminuire i ritorni al margine, cioè l'ennesimo controllore od ora, in generale, rende più benefici dell'(n+1)esimo controllore od ora. L' analisi ha classificato anche i difetti per tipologia.

Si è riscontrato che la tecnica di lettura effettuata col presupposto di interessarsi prima dei difetti più importanti, risulta essere favorita rispetto a quella caratterizzata dall'obiettivo di rilevare tutti i difetti contemporaneamente.

Ovviamente, c'è da muovere una riserva relativamente all'uso di soggetti studenti ed in effetti questo è un problema ricorrente per l'ingegneria del software su base empirica.

Non è in genere possibile usare i professionisti per gli esperimenti in ragione di costo ed accesso, conseguentemente molto lavoro sperimentale pubblicato è basato sull'uso di studenti.

Alcuni autori hanno sostenuto che le differenze fra uno studente ed un soggetto professionista possono essere minori del previsto [11].

Più restrittivo è l'uso di indagini industriali a complemento degli esperimenti di laboratorio.

3.3. La specifica formale

La specifica formale è un esempio di area dell'ingegneria del software che è stata fortemente sostenuta da una parte consistente della comunità di ingegneria del software. Chiaramente l'idea di usare una notazione formale per descrivere e dimostrare le proprietà di una specifica software attrae molto, specialmente per le applicazioni che potrebbero essere considerate *safety critical*. Ma non sarebbe meglio, invece, dedicare tempo e sforzo a sviluppare n versioni, in senso analogo all'uso della ridondanza hardware, per proteggere dai guasti? Possono essere rese argomentazioni convincenti per entrambi i punti di vista. Ancora una volta gli studi empirici possono far luce su

questo dibattito, anche se bisogna notare che la specifica formale abbraccia una gamma di attività molto ampia, dalle verifiche convenzionali di ogni fase e perfezionamento all'uso di notazioni su base matematica in stile 'leggero'. Inoltre, le notazioni variano considerevolmente, dagli approcci basati su modelli che sono essenzialmente insiemistici associati al calcolo dei predicati di primo ordine, come B e VDM, ai metodi algebrici quali OBJ e LARCH, alle algebre di processo come CSS.

Un'interessante inchiesta con riscontri non totalmente positivi è stata pubblicata da [12]. Qui gli autori discutono le loro esperienze usando Z (un linguaggio formale di specifica basato su modello) per sviluppare un analizzatore (*parser*). L'esperienza non ha conseguito molto successo. La specifica è risultata povera di contenuti, con molti errori e con problemi di modularità. Sebbene la densità finale di errori di codice era abbastanza buona (1.3 errori per KLOC), molti errori erano gravi e l'affidabilità (contrariamente alla precisione) era inaccettabilmente bassa. L'uso di Z quindi è stato considerato infruttuoso dato che l'attenzione era stata focalizzata su di un fattore di qualità del software rivelatosi inadeguato, cioè la precisione, piuttosto che l'affidabilità.

C'è anche un'altra e ben divulgata inchiesta su Z, condotta insieme da IBM Hursley e dall'università di Oxford [13]. Questo progetto ha applicato i metodi formali (soprattutto specifica limitatamente perfezionata o dimostrazione di programma) ad un aggiornamento del sistema software di elaborazione di transazioni CICS. La grandezza dell'aggiornamento per cui è stato usato Z ammontava a circa 48 KLOC che ha rappresentato circa il 18% del codice modificato (268 KLOC) ed una dimensione totale di rilascio di 768 KLOC. Il personale addetto al programma ha sostenuto che 19 errori sono stati evitati come conseguenza dell'uso di metodi formali, cioè una riduzione di più del 50%. Inoltre essi sostengono di aver ridotto i costi del 9% del preventivo totale. Ciò risulta molto impressionante e l'inchiesta è stata ampiamente citata per discutere i meriti di questo metodo di sviluppo del software, cfr. per esempio [14].

Purtroppo, un riesame più attento dei dati del CICS, di Finney e Fenton [15] rivela notevoli problemi con questa analisi. Anche se i ricercatori all'università di Oxford e dell'IBM devono essere lodati per aver intrapreso uno studio su scala industriale, ci sono molte difficoltà con i loro dati, non ultima la determinazione di cosa sarebbe accaduto se Z non fosse stato usato, in altre parole non c'è nessun controllo. Inoltre i dati degli errori sono da considerare col beneficio del dubbio poiché c'è una dimensione di tempo: quando sbaglia ed in quali circostanze? Inoltre, il personale coinvolto con la parte del sistema la cui specifica è stata sviluppata con Z era rappresentativo di tutto il personale dell'IBM? - ci sono motivi per pensare di no – E la parte del sistema la cui specifica è stata sviluppata con Z, era rappresentativa di tutto il sistema?

Questi interrogativi non sono volti a denigrare i metodi formali né gli sforzi di quanti hanno pubblicato questi risultati ma soltanto a precisare che non tutti i dati quantitativi hanno lo stesso peso ed è necessaria cautela nella loro interpretazione. Una buona trattazione delle questioni metodologiche nel condurre inchieste di ingegneria del software è data da Kitchenham et al. [16].

3.4. Fattori di insuccesso del progetto software

Come accennato nell'introduzione, malgrado i progressi considerevoli fatti dall'industria del software, ci sono ancora molti esempi di progetti che falliscono, a volte in modo straordinario e costoso.

L'ingegneria del software basata su princìpi empirici ha affrontato questo problema in due modi. In primo luogo in termini di analisi legale per determinare quali specifici e spesso locali insegnamenti potevano essere appresi. Buoni esempi di questo tipo di lavoro riguardano i problemi avuti con il sistema di spedizione delle ambulanze di Londra ed il volo 501 del razzo Ariane V dell'Agenzia Spaziale Europea. In secondo luogo vi sono le analisi più generali, usando spesso sondaggi, per identificare i fattori di insuccesso. Nell'uno o nell'altro caso, lo scopo di tale lavoro è sostituire la speculazione con il ragionamento fondato. Esaminiamo ogni metodo più approfonditamente.

Il progetto del sistema *London Ambulance Service Computer-Aided Despatch* (LASCAD) è proprio un esempio documentato di un errore di progetto software che ha provocato tragicamente un

incidente mortale dovuto ad eccessivo ritardo nei tempi di arrivo dell'ambulanza [17] [18]. Come risultato, un certo numero di insegnamenti sono stati tratti da questo errore.

Cosa interessante è che molti di essi riguardano più la gestione di progetti e meno le questioni tecniche nonostante la perdita di memoria che ha indotto il sistema a fermarsi lentamente siccome la memoria di calcolatore era stata acquisita ma poi non rilasciata. Esempi di problemi di gestione includono il processo stesso di acquisizione che ha condotto al contratto, firmato con un'azienda che era notevolmente meno costosa di tutti i relativi concorrenti. Questa offerta bassa era indicativa di un'organizzazione che non era riuscita ad inquadrare correttamente il dominio del problema a causa di inesperienza, piuttosto che di un'organizzazione più efficiente dei relativi concorrenti. Un altra categoria di problemi consisteva nella mancanza di un adeguato coinvolgimento degli utenti (il personale dell'ambulanza) che ha portato ad un'insufficiente acquisizione di requisiti e ad ostilità verso il sistema una volta introdotto. L'analisi spassionata dei progetti è spesso una fonte ricca di dati e permette agli ingegneri del software di capire molte cose.

Un altro esempio di un guasto riferito al software è stato il volo 101 del razzo Ariane V, 37 secondi dopo il lancio quando un *integer overflow* ha causato la deviazione massima dei propulsori principali e la conseguente corretta interruzione del volo. Il carico utile (*payload*) è stato stimato a circa 500 milioni di dollari. Anche questo caso è stato documentato ed un dettagliato *post mortem* è stato pubblicato dall'Agenzia Spaziale Europea [19]. In esso vi erano molte cose da imparare, ma forse le due che colpiscono di più sono che dati i rapporti complessi fra i molti subappaltatori e l'ESA, il software non è stato collaudato correttamente. In questo caso se il codice fosse stato eseguito, una volta rilasciato, con dati di volo adeguati, è garantito che il codice avrebbe sbagliato! In secondo luogo, il sistema è stato visto giustamente come *mission-critical* in modo da integrarvi la ridondanza come salvaguardia contro il guasto di un componente specifico. Quindi c'erano due sistemi inerziali paralleli di riferimento (dove il guasto si è prodotto).

Purtroppo entrambi avevano lo stesso codice in esecuzione e quindi entrambi sono venuti a mancare simultaneamente. La lezione da imparare è che le nozioni della ridondanza hardware non sono traslabili pedissequamente al software.

Altri ricercatori hanno adottato un approccio abbastanza differente, studiando i progetti software in generale piuttosto che guardando i particolari individuali di un caso specifico. In generale i ricercatori esaminano una popolazione più ampia di progetti software usando tecniche come i sondaggi. Un esempio recente interessante è il lavoro di Procaccino et al.. [20]. E' stato analizzato un campione di 21 professionisti del software di una grande organizzazione finanziaria nordamericana e sono stati usati una miscela di interviste strutturate e questionari per ottenere le informazioni sui fattori di successo e di fallimento di un progetto, con particolare rilievo verso quei fattori che potrebbero essere determinati nella fase iniziale o nel corso della vita di un progetto. Ciò ha fornito dati su un totale di 42 progetti. Un risultato notevole è la difficoltà nel determinare esattamente cosa costituisce successo od insuccesso. In questo studio, il *management* ha considerato il 78% dei progetti come riuscito, mentre gli sviluppatori hanno valutato soltanto il 49% come tale.

Inoltre sarebbe stato interessante ottenere i punti di vista degli utenti su questa materia! Ciò mostra almeno una sfida aperta nel condurre studi come questi dal momento che, in altre parole, variano le percezioni.

Ciò nonostante, emerge un certo numero di fattori dalla loro analisi, compresa l'importanza della partecipazione dell'utente, ma un gran numero di utenti spesso generava "*tanto conflitto quanto consenso*". Un altro fattore era la presenza di uno sponsor che era 'impegnato visibilmente' nel progetto. La dimensione del progetto inoltre è risultata influente, con i grandi progetti che sono più probabilmente soggetti ad errore.

Mentre questo studio si è basato soltanto su di una singola organizzazione, risultati simili sono stati segnalati da altri studi, che aumentano così la nostra fiducia nei risultati, ricordando però che abbiamo bisogno sempre di capire il contesto all'interno del quale tali risultati sono stati tratti. Lo studio di Procaccino et al. è stato effettuato in una grande organizzazione. Dovremmo essere prudenti nel generalizzarlo, per esempio, alle piccole organizzazioni che sviluppano software per videogiochi.

4. Riepilogo ed orientamenti futuri

Si è detto che l'ingegneria del software basata su princìpi empirici risulta essenziale se la disciplina e la pratica devono progredire oltre l'opinione pura. Senza convalida empirica correttamente condotta ed indipendente, è difficile giudicare se un particolare strumento o tecnica del software sia utile o no.

La situazione è resa più complessa dal fatto che molti commentatori hanno interessi personali.

Se desidero ottenere fondi di ricerca o consulenza per, ad esempio, il collaudo software allora posso dare esagerata enfasi ai problemi connessi con un collaudo non eseguito accuratamente, e così via.

Il dato aneddotico può risultare di una certa utilità, però la difficoltà può risiedere nel capire correttamente il contesto in modo da permettere una generalizzazione ragionevole dell'esempio ad altre situazioni.

L'investigazione empirica quindi dovrebbe fungere da custode delle nuove idee all'interno dell'ingegneria del software. Mentre molti progressi sono stati realizzati nel corso degli anni, restano naturalmente molte sfide aperte.

In primo luogo, c'è la scalabilità. Molti dei problemi dell'ingegneria del software sono associati con la scala delle mansioni intraprese e dei manufatti. Purtroppo tutto questo non può essere riprodotto sempre adeguatamente in laboratorio. Tuttavia, l'accesso ai siti industriali è limitato spesso da considerazioni di costo o di riservatezza. Una collaborazione migliore fra i ricercatori e l'industria sarà di valore inestimabile.

In secondo luogo, ci stiamo occupando di esseri umani e di un'attività fondamentalmente creativa o di progetto. Ciò introduce un quantitativo enorme di variabilità nei processi di ingegneria del software, che senza un'estesa riproduzione in laboratorio, può fornire una visione distorta su ciò che può essere un effetto di piccole proporzioni se estrapolato dal fenomeno di interesse.

In terzo luogo, dobbiamo affrontare più decisamente il preconcetto della pubblicazione che favorisce i risultati 'positivi' discriminando quelli 'negativi'.

Questo può avere effetto sulla meta-analisi, se non sono disponibili tutti i 'risultati negativi' nel pubblico dominio. Può anche esercitare pressione sui ricercatori allo scopo di pescare risultati a strascico o subcoscientemente gonfiare le prove delle asserzioni rese.

Nonostante queste sfide, il campo riveste un importanza crescente ed ha un futuro forte nel sostegno all'industria del software nel prendere decisioni redditizie.

5. Fonti

La seguente sezione fornisce alcuni punti di partenza per i lettori interessati ad esaminare più approfonditamente gli aspetti dell'ingegneria del software basata su princìpi empirici.

Ci sono un certo numero di buoni libri, tuttavia, i più ben conosciuti sono probabilmente "*Software Metrics: A Rigorous and Practical Approach*" di Norman Fenton e di Shari Pfleeger [21] ed un lavoro recente di Claes Wohlin ed altri. "*Experimentation in Software Engineering: An Introduction*" [22] che si occupa di molte delle questioni pratiche e metodologiche dello svolgimento di questo tipo di ricerca.

Un certo numero di riviste scientifiche inoltre pubblicano articoli in merito. *Empirical Software Engineerin*g: una rivista scientifica internazionale è dedicata a questo tema, tuttavia, molte riviste contengono materiale utile, comprese: *IEEE Transactions on Software Engineerin*g, il *Journal of Systems & Software* ed *Information & Software Technolog*y. In aggiunta riviste come *IEEE Software* contengono spesso articoli più corti, rivolti più al professionista.

Ci sono inoltre un certo numero di congressi annuali dedicati all'ingegneria del software basata su princìpi empirici. I più importanti sono i seminari *IEEE International Metrics Symposium* e l' *Empirical Assessment and Evaluation in Software Engineering (EASE).*

Inoltre abbiamo molte buone risorse sul web. La *International Software Engineering Research Network (ISERN)* è una rete internazionale di molti dei gruppi di ricerca principali che attivamente si occupano di ricerca di ingegneria del software basata su princìpi empirici. L' Home Page è

<http://www.iese.fhg.de/isern/>, che contiene informazioni su vari esperimenti in corso, rapporti tecnici che possono essere scaricati, ecc.

Altri gruppi di ricerca che gestiscono siti Web includono:
Experimental Software Engineering Group at the University of Maryland (USA):
<http://www.cs.umd.edu/projects/SoftEng/tame/>.
Empirical Software Engineering Research Group at Bournemouth University (UK):
<http://dec.bournemouth.ac.uk/ESERG/>.

Riferimenti

[1] H. D. Benington Production of large computer programs, Symp.on Advanced Computer Programs for Digital Computers, Wash-ington, D.C., Office of Naval Research, 1956.

[2] W. Hayes. Research synthesis in Software Engineering: a case for meta-analysis, 6th IEEE International Softw. Metrics Symp., Boca Raton, Fl: IEEE Computer Society, 1999.

[3] M. Cartwright and M. J. Shepperd. An Empirical Investigation of an Object-Oriented Software System. IEEE Trans. on Softw. Eng., 26(8), pp 786–796, 2000.

[4] L. Hatton. Does OO Sync with How We Think, IEEE Software, 15(3), pp 48–54, 1998.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design, IEEE Trans. on Softw. Eng., 20(6), pp 476–493,1994.

[6] B. Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity, Object-Oriented Series, Prentice Hall: Englewood Cliffs, New Jersey, 1996.

[7] J. Daly et al. Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software, Empirical Software Engineering, 1(2), pp 109–132, 1996.

[8] B. Unger and L. Prechelt. The impact of inheritance depth on maintenance tasks – Detailed description and evaluation of two experimental replications, Technical Report, Dept. of Comp. Sci., Karlsruhe University, Germany, 1998.

[9] O. Laitenberger and J.-M. DeBaud. An encompassing life cycle centric survey of software inspection, J. of Systems & Software 50(1), pp 5–31, 2000.

[10] S. Biffl and M. Halling. Investigating the defect detection effectiveness, and cost benefit of nominal inspection teams, IEEE Transactions on Software Engineering, 29(5), pp 385–397, 2003.

[11] M. Höst et al. Using students as subjects – a comparative study of students and professionals in lead-time impact assessment, Empirical Software Engineering 5, pp 201–214, 2000.

[12] M. Neil et al. Lessons learned from using Z to specify a software tool, IEEE Transactions on Software Engineering 24(1) pp 15–23, 1998.

[13]
I. Houston and S. King. CICS Project Report – Experiences and Results from the Use of Z in IBM, in Lect. Notes Comput. Sci. Vol. 551, Prehn, S. and Toetenel, W.J., Editors, Springer-Verlag: New York. pp 588–596, 1991.

[14] J. P. Bowen and M.G. Hinchey. 7 More Myths of Formal Methods, IEEE Software 12(4), pp 34–41, 1995.

[15] K. Finney and N.E. Fenton. Evaluating the effectiveness of Z: The claims made about CICS and where we go from here, J. Of Systems & Software 35(3), pp 209–216, 1996.

[16] B. Kitchenham et al. Case-Studies for Method and Tool Evaluation, IEEE Software 12(4), pp 52–62, 1995.

[17] M. Hougham. London Ambulance Service computer aided despatch system, Intl. J. of Proj. Mngt. 14(2) pp 103–110, 1996.

[18] P. Beynon-Davies. Human error and information systems failure: the case of the London ambulance service computer-aided despatch system project, Interacting with Computers 11(6): pp 699–720, 1999.

[19] J. Lions. Report of the Inquiry Board for Ariane V Flight 501 Failure, Joint Communication ESA-CNES, Paris, France, 1996. (Disponibile anche su http://www.esa.int/export/esaLA/Pr_33_1996_p_EN.html).

[20] J. D. Procaccino et al. Case study: factors for early prediction of software development success, Information & Software Technol-ogy 44(1), pp 53–62, 2002.

[21] N. E. Fenton and S. L. Pfleeger. Software Metrics: A Rigorous and Practical Approach, London: International Thompson Publishing, 1997.

[22] C. Wohlin et al. Experimentation in Software Engineering: An Introduction, Norwell, MA: Kluwer Academic, 2000.

L'autore

Il Dott. **Martin Shepperd** è professore di ingegneria del software all'università di Bournemouth nel Regno Unito. I suoi interessi di ricerca includono l'ingegneria del software basata su princìpi empirici e l'uso dei metodi di apprendimento automatizzati per sviluppare sistemi di previsione. Ha pubblicato più di 80 articoli e tre libri. È co-Caporedattore del *Journal of Information & Software Technology* e Redattore Associato di *IEEE Transactions on Software Engineering*
<mshepper@bmth.ac.uk>.

Danilo De Riso si è laureato in Scienze dell'Informazione nel 1992 con tesi di laurea in informatica musicale ed ha conseguito nel 1986 il diploma di pianoforte principale al Conservatorio di Salerno. Si è occupato, come freelance, di traduzioni di testi di Informatica per la Jackson libri. Attualmente lavora come consulente analista programmatore, musicista e padre del bellissimo Mario, al quale questo lavoro è dedicato.