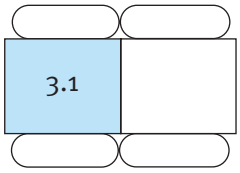
EXTREME PROGRAMMING: PROGRAMMAZIONE ESTREMA O REVISIONISMO ESTREMISTA?

Carlo Ghezzi
Mattia Monga

L'XP, nel processo di sviluppo del software, ha l'obiettivo di evitare la produzione di semilavorati diversi da quelli necessari alla realizzazione delle applicazioni. Presentata come un approccio del tutto nuovo e originale, ha in realtà le sue radici in metodi già noti e sperimentati. Secondo chi scrive, inoltre, pur introducendo utili concetti e offrendo un approccio valido in alcuni specifici contesti, l'XP è troppo spesso proposta come una soluzione universale ai problemi della moderna produzione industriale di software[1].



1. INTRODUZIONE

L'ingegnere del software è, periodicamente, bombardato dagli annunci di approcci rivoluzionari che cambieranno definitivamente il suo modo di lavorare, dandogli, finalmente, la possibilità di conseguire quegli obiettivi di produttività e qualità che ha sempre sognato. Le cose stanno, in realtà, in maniera differente, ma non è sempre facile distinguere quanto effettivamente ci sia di innovativo e originale perché la rivoluzione è, innanzitutto, terminologica e spesso si usano parole nuove per ribadire concetti già noti. In un famoso articolo del 1987 [7], F. Brooks analizzò, a fondo, quali fossero le difficoltà intrinseche e ineliminabili del software (*essential difficulties*), argomentando, in modo molto efficace e convincente, che il progresso, fino ad allora, aveva solo aggredito gli aspetti più facili (*accident*). Le conclusioni che venivano tratte erano che non esisteva una panacea capace di rendere facile lo sviluppo del software. Secondo l'efficace metafora di Brooks, non esiste il rassicurante *silver bullet* capace di eliminare "il lupo mannaro delle favole dei bambini". Purtroppo, il mito del silver bullet ha continuato ad affacciarsi sulla scena della tecnologia software, manifestandosi attraverso la sopravvalutazione o la generalizzazione di metodi, tecniche e strumenti che si dimostravano efficaci in certi contesti, ma che venivano, ingiustificatamente, proposti come soluzioni generali definitive. L'entusiasmo dei ricercatori e, ancor più spesso, la propaganda commerciale, hanno in molti casi generato false aspettative di soluzioni miracolistiche che, alla prova dei fatti, hanno subito mostrato i propri limiti. Uno dei settori in cui più frequentemente si è assistito a questo fenomeno è quello dei metodi e strumenti di supporto al processo di produzione del software. In tale ambito, per l'appunto, si colloca l'*eXtreme Programming* (XP). Il motivo dell'interesse per questo tema è evidente, data la sua rilevanza econo-

[1] Questo lavoro è stato svolto in parte nell'ambito del progetto *Sahara* cofinanziato dal MIUR (Ministero dell'Istruzione dell'Università e della Ricerca).

mica. L'obiettivo, infatti, è quello di definire la struttura del processo di sviluppo in modo da renderlo, il più possibile, efficace e controllabile. Nei metodi tradizionali, la controllabilità è ottenuta dettagliando fasi e attività attraverso cui il processo deve evolvere, definendo le regole di transizione da una fase alla successiva e documentando ognuna di esse secondo procedure standardizzate. Le persone coinvolte nello sviluppo, perciò, sono responsabili della produzione non solo del codice sorgente dell'applicazione, ma anche di molti altri semilavorati, che non sono necessariamente rivolti alla pura generazione del codice. Ogni applicazione, però, pone problemi diversi, ed è difficile e irrealistico (se non addirittura impossibile) definire un processo universalmente valido, senza cadere nei generici richiami al buon senso. Quando si cerca di adottare una qualche forma di processo senza progettarne l'adattamento alle specifiche esigenze del caso, il lavoro aggiuntivo finisce per essere percepito come burocrazia inutile, che influisce in maniera negativa sulla flessibilità e l'efficienza.
L'XP promette di mantenere la controllabilità del processo pur riducendo il lavoro di supporto e convogliando il massimo dello sforzo sulla mera produzione dell'applicazione.
Un altro problema ricorrente nella produzione del software è che spesso i risultati ottenuti da anni di lavoro finiscono per soddisfare solo le esigenze degli sviluppatori, anziché quelle dei committenti.
L'XP promette di fornire i meccanismi perché gli sviluppatori possano acquisire, *durante lo sviluppo*, la consapevolezza che ciò che stanno costruendo soddisferà pienamente, al momento della sua posa in opera, le esigenze di chi l'ha commissionato.
Quanto detto finora costituisce una premessa teorica al presente articolo che sarà suddiviso in tre parti. Nella prima (Paragrafo 2) si fornirà una breve presentazione delle principali caratteristiche dell'XP. Nella seconda (Paragrafo 3) si cercherà di collocare il metodo nel contesto dei metodi di definizione dei processi software. Con ciò si intende mostrare come l'XP, che spesso viene presentata come un approccio del tutto nuovo e originale, in realtà abbia le sue radici in metodi noti da e sperimentati da molti anni. Infine, nella terza (Paragrafo 4) si fornirà una valutazione di XP, cercando di spiegare perché, secondo chi scrive, essa pur introducendo utili concetti e ponendo reali problemi, possa essere complessivamente valutata come uno dei tanti silver bullet falliti. Alcune lezioni che essa insegna devono senz'altro essere tenute in seria considerazione dagli ingegneri del software ma, altrettanto chiaro, deve risultare il fatto che non si tratta della soluzione definitiva ai problemi dello sviluppo delle moderne applicazioni software di qualità, come tanta letteratura sull'argomento vorrebbe far credere. In altri termini, se è vero che alcune delle raccomandazioni dell'XP si inquadrano correttamente nel solco delle proposte di processi di produzione di software agili e flessibili e che in quanto tali, forniscono un bagaglio concettuale utile al progettista di oggi, nella visione radicale che realizza la sua formulazione data dai propri paladini costituisce solo un revisionismo estremistico dei metodi tradizionali, che rischia di produrre un danno anche maggiore di quello che vorrebbe eliminare.

2. XP: UN TUTORIAL

Con eXtreme Programming si intende una tecnica, proposta da K. Beck, per organizzare il processo di sviluppo del software con l'esplicito obiettivo di evitare la produzione di semilavorati diversi da quelli strettamente necessari alla realizzazione dell'applicazione. Dettagliate specifiche formali, approfondite analisi e puntigliosa documentazione sono considerate attività troppo costose rispetto ai benefici apportati, in quanto limitano la flessibilità del processo che deve poter modificare i propri obiettivi in ogni momento.
La produzione di un'applicazione, secondo Beck, non è un'attività che possa essere analizzata e precisamente pianificata a priori. Invece, esattamente come quando si guida l'automobile, la condotta complessiva è il risultato di un gran numero di minimi cambiamenti di rotta che il pilota decide in base alla sua istantanea percezione di curve ed ostacoli. Il lavoro dei programmatori procede organizzando quattro attività fondamentali che si ripetono per tutto il corso del progetto:

- scrittura del codice dell'applicazione (*coding*);
- verifica delle funzionalità (*testing*);
- osservazione dell'ambiente, inteso come desideri del committente, opportunità tecnologiche, sviluppi di mercato (*listening*);
- progetto dell'applicazione (*design*).

I programmatori sono responsabili non solo della codifica dell'applicazione, ma anche della sua verifica. Anzi, particolare enfasi è posta proprio su questa attività: nessuna istruzione dovrebbe essere considerata veramente parte dell'applicazione finché non ne sia stato verificato l'effetto secondo le attese. La costruzione dell'applicazione procede in maniera iterativa, cogliendo le reazioni dei committenti e riprogettando, in maniera adeguata, le sue funzionalità e i meccanismi adottati per ottenerle.

Perché queste attività possano essere svolte efficacemente, vengono identificate alcune prassi fondamentali che aiutino i programmatori a rendere il loro lavoro il più efficiente possibile.

Pianificazione delle attività (*Planning the game*). Lo sviluppo dell'applicazione è accompagnato dalla stesura di un piano di lavoro. Il piano è definito e, continuamente aggiornato, a intervalli brevi e regolari dai responsabili del progetto, secondo le priorità aziendali e le stime dei programmatori, che partecipano, in modo attivo, alla pianificazione. Il meccanismo per attuare, efficacemente, la pianificazione è articolato come un gioco in cui sono coinvolti utenti responsabili del progetto e sviluppatori per stabilire un equilibrio dinamico fra le esigenze di tutti gli attori coinvolti. Gli utenti finali dell'applicazione presentano gli obiettivi da raggiungere descrivendo una serie di scenari (storie) che il sistema deve soddisfare. Gli sviluppatori stimano il tempo necessario per la realizzazione di ogni storia: qualora ciò non sia possibile, la storia viene suddivisa in storie più semplici. Le storie vengono ordinate da utenti e responsabili secondo la loro priorità di realizzazione, dopo che gli sviluppatori ne hanno stimata la rispettiva difficoltà. Dalla sintesi di queste valutazioni i responsabili del progetto generano la pianificazione delle attività, intesa come l'insieme di storie che dovranno essere realizzate per il prossimo rilascio e le date previste: sarà, inoltre, loro principale responsabilità misurare e controllare l'andamento delle attività rispetto alla pianificazione stessa. Questo processo viene ripetuto dopo ogni rilascio per pianificare il successivo. Una pianificazione più dettagliata viene poi decisa dagli stessi sviluppatori, i quali definiscono i "compiti" elementari necessari all'implementazione delle singole storie. Per ogni compito uno sviluppatore deve stimare i giorni necessari al suo completamento e assumersi la responsabilità della sua realizzazione. Il carico di ciascuno sviluppatore non deve superare quello concesso dalla pianificazione del rilascio, dopo essere stato pesato secondo un fattore di efficienza di gruppo che tiene conto delle necessità di cooperazione tramite riunioni, incontri con il committente ecc..

Rilasci frequenti (*Short releases*). La vita e lo sviluppo dell'applicazione sono scanditi dai rilasci di versioni del prodotto funzionanti, nel senso che realizzano qualcuna delle storie che ne descrivono gli obiettivi. Ogni rilascio rappresenta il punto conclusivo di un'iterazione di sviluppo e l'inizio di una nuova pianificazione. Per poter tener conto di cambi di prospettiva, errori di valutazione, nuovi requisiti, restrizioni di bilancio, ogni iterazione dovrebbe durare non più di qualche settimana (in genere, da due a quattro).

Metafora condivisa (*Metaphor*). Ogni progetto è guidato da una metafora condivisa da responsabili e sviluppatori. La metafora non è altro che una descrizione semplificata, ma efficace, del sistema nel suo complesso. Serve a fornire un vocabolario comune a tutte le persone coinvolte, senza scendere nei dettagli implementativi.

Progetti semplici (*Simple design*). La struttura dell'applicazione deve essere la più semplice possibile. L'architettura del sistema deve essere comprensibile da tutte le persone coinvolte nel progetto. Non devono esserci parti superflue o duplicazioni. Le parti che compongono il sistema devono essere, soltanto, quelle strettamente necessarie alle esigenze correnti. Solo quando nuove circostanze lo richiederanno, verranno progettati nuovi componenti, eventualmente riprogettando anche quelli già esistenti.

Ristrutturazione del codice (*Refactoring*). Come si è detto precedentemente, l'applicazione necessita di continue riprogettazioni per eliminare parti divenute superflue e per adattare il sistema alle nuove esigenze. Questa attività è detta *refactoring*, ristrutturazione, a intendere che, ogni volta che si intravede la possibilità di eliminare parti superflue o di semplificarne l'organizzazione, l'intera struttura del codice va adattata ai nuovi principi progettuali.

Verifica di ogni funzionalità (*Testing*). Ogni funzionalità va sottoposta a verifica, in modo che si possa acquisire una ragionevole certezza sulla sua correttezza. Ciò sia a livello di sistema (test di sistema) sia a livello del singolo metodo (test di unità). I test di sistema sono costruiti sulla base delle storie concordate con il committente che dice l'ultima parola sulla convalida del sistema. I test di unità devono poter essere rieseguiti automaticamente, con tempi dell'ordine dei minuti: allo scopo è utile avvalersi di strumenti opportuni [10]. Ogni ristrutturazione o modifica del codice deve mantenere inalterato il risultato dei test già considerati. I test vengono, generalmente, scritti prima della codifica della funzionalità. Il metodo prescrive, addirittura, che il codice da sviluppare debba soddisfare tutti i test e niente di più. Secondo Beck, la verifica è sostanzialmente una falsificazione di stampo "popperiano[2]": una teoria (il codice dell'applicazione) viene messa alla prova grazie ad una serie di osservazioni (casi di test) che essa deve spiegare (ovvero, non deve disattendere le aspettative). Ogni nuova teoria deve rimanere coerente con le osservazioni descritte precedentemente.

Programmazione a coppie (*Pair programming*). La scrittura vera e propria del codice è fatta da coppie di programmatori che lavorano al medesimo terminale. Le coppie non sono fisse, ma si compongono associando le migliori competenze per la risoluzione di uno specifico problema. Il lavoro in coppia permette, scambiandosi periodicamente i ruoli, di mantenere mediamente più alto il livello d'attenzione. I locali dove si svolge il lavoro devono permettere senza difficoltà di lavorare a coppie.

Collettivizzazione del codice (*Collective ownership*). Il codice dell'applicazione può essere liberamente manipolato da qualsiasi sviluppatore. Ciò è possibile grazie al fatto che l'organizzazione è la più semplice possibile e che il codice è scritto rispettando regole condivise da tutti. La collettivizzazione è anche un meccanismo per stimolare la semplificazione delle parti più oscure del codice, che, essendo incomprensibili a tutti, fuorché agli autori, hanno un'alta probabilità di essere eliminate. Naturalmente ogni modifica non deve pregiudicare la correttezza dei test.

Integrazione continua (*Continuous integration*). Non sono previste sessioni particolari di integrazione. In effetti, dato che il codice dell'intera applicazione è sotto il controllo di tutto il gruppo di sviluppo, l'integrazione del lavoro dei singoli è continua. Deve essere costantemente possibile ottenere una versione funzionante dell'applicazione sulla quale operare le verifiche. Una piattaforma pronta per l'integrazione è sempre disponibile per i programmatori.

Rinuncia al lavoro straordinario (*40-hour week*). Lo sviluppo di applicazioni con i metodi dell'eXtreme Programming è un'attività che richiede grande concentrazione, entusiasmo e creatività. Il lavoro in condizioni di stress, con ripetute sessioni straordinarie, provoca necessariamente un deterioramento della qualità dell'impegno e deve pertanto essere evitato. Per questo motivo, la pianificazione coinvolge anche gli sviluppatori ed è aggiornata di frequente per tener conto di errori e imprevisti.

Partecipazione del committente (*Onsite customer*). Il committente deve essere coinvolto nello sviluppo perché è l'unica fondamentale fonte di convalida del sistema. Partecipa perciò alla stesura dei test di sistema e verifica, periodicamente, che il sistema realizzato corrisponda effettivamente alle proprie esigenze. Inoltre, è la principale fonte di informazione per la conoscenza sul dominio di applicazione. Secondo chi propone il meto-

[2] Il filosofo di origine austriaca Karl Popper (1902-1994) sostenne che un'ipotesi scientifica non può mai essere verificata da risultati sperimentali, che possono tuttalpiú semplicemente corroborare la nostra fiducia in essa. Viceversa, un risultato sperimentale contrario alle aspettative (falsificazione) permette di sviluppare nuova conoscenza, obbligando al rifiuto dell'ipotesi di partenza.

do, la comunicazione orale tra progettisti e committenti, unita alla definizione a priori dei test, allevia la necessità di produrre una formulazione scritta e precisa dei requisiti.

Uso di standard per la codifica (*Coding standards*). Il codice deve esplicitare le astrazioni dei programmatori ed è il loro principale strumento di comunicazione. Deve, pertanto, essere scritto in maniera uniforme e omogenea. Tutti gli sviluppatori devono essere in grado di capire e modificare ogni linea di codice scritta da altri.

3. IL CONTESTO DELLE METODOLOGIE DI PROCESSO

Dalla fine degli anni '60 in poi, con la nascita dell'ingegneria del software come disciplina, ci si è posti il problema di definire modelli del ciclo di vita del software, dalla concezione iniziale dell'idea del prodotto fino al suo sviluppo, al suo rilascio e, infine, alla sua dismissione. Tutti i prodotti industriali di cui si vogliono assicurare adeguati livelli di qualità vengono, infatti, sviluppati secondo processi sistematici ben definiti. L'esistenza di un processo di produzione ben definito viene, addirittura, considerato come un prerequisito necessario perché il prodotto industriale possa aspirare a raggiungere un livello di qualità certificabile.
Il primo e più noto modello di processo è noto come *ciclo di vita a cascata*. Il ciclo di vita a cascata, nato come esperienza all'interno di sistemi militari sviluppati fin dagli anni '50, è caratterizzato da una progressione lineare attraverso fasi (per l'appunto, *in cascata*), quasi si trattasse di una catena di montaggio attraverso cui far procedere le attività di sviluppo del software.
Nel seguito, vengono presentate le caratteristiche fondamentali di un'organizzazione a cascata del ciclo di vita.

- *La sequenzialità del processo*. Secondo i proponenti, ciò dovrebbe garantire una migliore controllabilità e prevedibilità del processo e, quindi, la capacità di tenere i costi e i tempi di sviluppo sotto controllo. I ritorni all'indietro, infatti, vengono considerati dannosi ai fini di tenere il processo sotto controllo.
- *La definizione di fasi e attività standard*. Per ciascuna fase, viene definito, con esattezza, il criterio di successo che permette di certificarne il completamento e, quindi, l'autorizzazione a procedere alla fase successiva. Per di più, spesso vengono prescritti metodi specifici che i progettisti sono tenuti a seguire per lo svolgimento delle attività della fase.
- *Gli standard documentali*. Si afferma, spesso, che un ciclo di vita a cascata è *document driven*. Con ciò si intende che i semilavorati di ciascuna fase, che costituiscono *input* per la fase successiva, sono documenti che devono seguire certi standard che l'organizzazione produttiva prescrive. Il completamento di ciascuna fase, pertanto, viene a identificarsi con l'approvazione, da parte dei responsabili del progetto, dei documenti che rispettano gli standard prescritti. Il ciclo di vita a cascata, posto ancor oggi, in molti casi, come modello generale di riferimento per lo sviluppo di software, è in realtà un modello irrealistico e, spesso, del tutto inadeguato. L'organizzazione lineare del processo non può quasi mai avverarsi in pratica ma, anzi, il risultato dello svolgimento di certe fasi comporta, spesso, la ripetizione di quanto è stato fatto in fasi precedenti. Ciò è causato, principalmente, dalle difficoltà connesse con l'acquisizione e specifica dei requisiti. È, in generale, illusorio pretendere di congelare i requisiti in una specifica esaustiva che viene prodotta all'inizio del processo, in base alla quale si procede poi al progetto dell'architettura e all'implementazione dell'applicazione. I requisiti sono spesso, inizialmente, confusi e, dunque, non sono noti in maniera precisa e, qualora lo siano, è facile che vengano fraintesi dall'ingegnere del software, che potrebbe non avere alcuna competenza circa il dominio applicativo nel quale l'applicazione dovrà operare.
Con un modello a cascata, errori o fraintendimenti nella specifica dei requisiti verrebbero scoperti molto tardi, quando l'applicazione viene rilasciata al committente o immessa sul mercato. Il risultato è dunque che, immediatamente, inizia una fase di "manutenzione", che in realtà tende solo a rimediare gli errori introdotti all'inizio e, tornando indietro, a rielaborare l'analisi e la specifica dei requisiti. È anche stato osservato che molto spesso i requisiti si chiariscono solo dopo che una qualche forma dell'applicazione viene posta nelle mani del committente e che,

quindi, risulta illusorio ipotizzare che il processo di sviluppo possa procedere in modo lineare, basato su conoscenze che possono solo in parte essere note *a priori*.
In conclusione, il ciclo di vita a cascata nasce con l'obiettivo primario di ridurre il rischio che i costi e i tempi di sviluppo non siano controllabili. Nel cercare di limitare questo rischio, vengono ignorati i rischi ancora più gravi che sono dovuti alla scarsa conoscenza e all'instabilità dei requisiti, che fanno sì che la riduzione dei costi di sviluppo sia solo illusoria, e che si riversi, da un lato, in elevati costi di manutenzione e, dall'altro, in insoddisfazione dei committenti. Un modo più astratto di caratterizzare un ciclo di vita a cascata definisce il processo come una scatola nera, che riceve, in ingresso, la specifica dei requisiti dell'applicazione e che, in uscita, fornisce l'applicazione finale, ottenuta attraverso la sequenza lineare di fasi. Poiché il processo di acquisizione e specifica dei requisiti soffre dei problemi intrinseci che sono stati riassunti in precedenza, qualunque sia la scomposizione in fasi che viene scelta per articolare le attività della scatola nera, si arriverà sempre, e troppo tardi, a scoprire che occorre procedere a modifiche e ricalibrazioni, ripercorrendo in tutto o in parte il processo di sviluppo.
È anche possibile definire modelli flessibili, agili e incrementali in cui si abbandona l'idea dello sviluppo dell'applicazione intesa come un prodotto monolitico, con un processo teso a un'unica data finale di consegna. Al contrario, il processo di sviluppo viene visto come una successione di rilasci incrementali, che adatta lo sviluppo in maniera flessibile ai requisiti man mano che questi vengono esplicitati.
Viene così abbandonata l'idea che debba esistere un unico modello di riferimento da adottare come standard immutabile e universale, ma viene invece accettato che il modello di processo debba essere di volta in volta definito in base alle specifiche caratteristiche dell'applicazione.
Pur senza entrare in dettagli che esulano dallo scopo di questo lavoro (e per una trattazione dei quali si rimanda a un testo di Ghezzi et al. [8]), si ricorda che i modelli alternativi proposti di volta in volta hanno preso il nome di modelli basati su *rapid prototyping*, di modelli *user driven*, di modelli "a spirale" (per contrasto con la linearità del ciclo a cascata [1]). Esempi ben noti, e diversi tra loro, di organizzazione innovativa del ciclo di vita del software sono offerti dal cosiddetto *Unified Development Process*, sviluppato nel contesto di UML (*Unified Modeling Language*) [12], dal metodo agile e flessibile adottato da Microsoft, descritto da Cusumano e Selby [6] ed efficacemente definito come *synchronize & stabilize* e, infine, dai processi di tipo "bazaar" seguiti talvolta per lo sviluppo di software *open-source* [17]. È stato anche osservato [7] che nell'èra di Internet sempre più spesso chi sviluppa software innovativo può seguire la strategia di sviluppare versioni iniziali rendendole, in qualche modo, disponibili, anche in forma gratuita, attraverso Internet. Ciò può incoraggiare molti potenziali utenti, da un lato, a sperimentare l'uso dell'applicazione e, dall'altro, a fornire utili suggerimenti per sue modifiche che potrebbero essere, successivamente, incorporate nel prodotto.
In questo modo, si generano comunità virtuali di *early adopter* che possono essere fidelizzati al prodotto e che lo adotteranno quando questo verrà poi messo sul mercato, o che commissioneranno servizi aggiuntivi, stabilendo così un forte legame anche di natura economica con il produttore di software. La proposta dell'XP, si colloca, dunque, in un contesto generale che, da tempo, ha riconosciuto l'improponibilità sia di metodi generali e universalmente adottabili per lo sviluppo delle applicazioni, sia l'inadeguatezza, se non in casi molto specifici, di metodi rigidamente sequenziali, quali il ciclo di vita a cascata. In particolare, si colloca all'interno dei metodi agili e flessibili, che vogliono rispondere in maniera efficace sia all'instabilità dei requisiti che all'esigenza di rapide risposte al mercato.

4. UN'ANALISI CRITICA DELL'XP

In che senso la produzione di software, secondo la modalità proposta da Beck, deve essere considerata estrema? Probabilmente, l'intenzione dei proponenti era quella di suggerire l'idea che la costruzione di applicazioni è un'attività che deve essere svolta in condizioni particolarmente rischiose, in

cui i progettisti e gli implementatori, a differenza di quanto accade in altre discipline ingegneristiche, devono essere in grado di reagire, prontamente, a ogni genere di eventualità impreviste: cambiamenti dei requisiti, stravolgimenti tecnologici e dell'ambiente in cui il sistema verrà utilizzato, *turnover* dei lavoratori ecc..

Ecco, allora, che in un contesto simile il "meglio" diventa nemico del "bene" e l'unica cosa che conta è generare, prima possibile, un'applicazione utilizzabile. A parere di chi scrive, invece, il termine "estremo" caratterizza bene un approccio che non si limita a proporre una collezione di buone tecniche che l'ingegnere del software può scegliere di volta in volta, ma che fornisce una soluzione radicale in cui le singole tecniche sono integrate in un approccio estremista.

L'ipotesi, implicitamente accettata, sembra essere che l'analisi sia, in sé, un appesantimento del progetto. Il credo dei programmatori estremi è che "ad ogni giorno deve bastare la sua pena", ovvero non serve prevedere i possibili cambiamenti, perché la previsione è considerata troppo incerta per valere il suo costo. L'ingegneria del software tradizionale insegna che cambiare idea ha un costo che cresce esponenzialmente nel corso del progetto. La curva di figura 1, che si trova su molti testi classici di ingegneria del software [16], descrive, in maniera qualitativa[3], quello che è normalmente considerato l'andamento dei costi delle varianti in corso d'opera. In sostanza, motiva la ragionevolezza del vecchio adagio ingegneristico (e non solo) secondo cui "prevenire è meglio che curare". Secondo Beck, invece, al giorno d'oggi, i progressi della tecnica nella produzione del software (soprattutto la diffusione di linguaggi orientati agli oggetti e la disponibilità di strumenti automatici di verifica e di refactoring) fanno sì che l'andamento dei costi sia meglio descritto da una curva come quella di figura 2, per cui è possibile rischiare una variante tardiva per risparmiare tempo prezioso nelle prime fasi dello sviluppo. Questa curva, peraltro, non appare giustificata da reali dati di natura empirica, ma viene presentata come la conseguenza indiscutibile dei progressi tecnici e metodologici sopra ricordati, che per atto di fede assumono così il ruolo di nuovi silver bullet. Si può ammettere l'esistenza di classi di applicazioni per le quali l'approccio dell'XP può risultare vantaggioso: sistemi non critici di piccola-media dimensione, di tipo esplorativo (per esempio, nella ricerca e sviluppo) o, comunque, fortemente caratterizzati da requisiti poco definiti e instabili. Ma per tutti i sistemi di una certa complessità che vengono

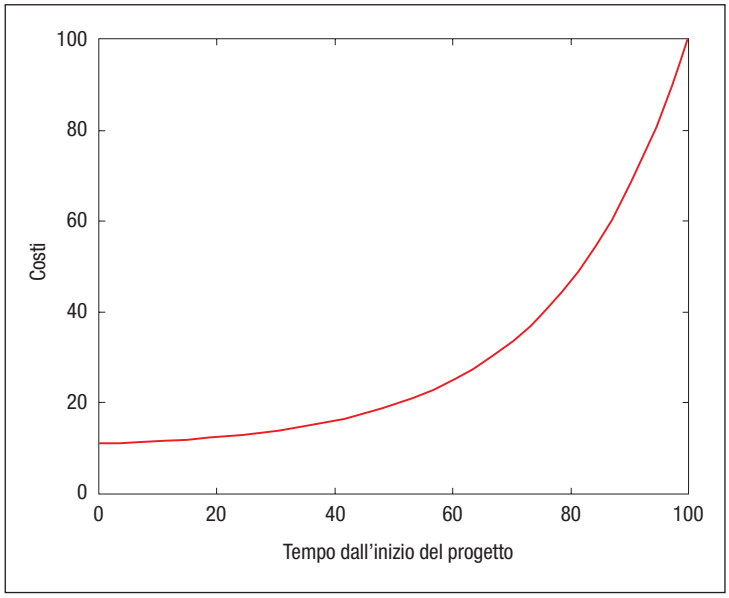


FIGURA 1
Relazione tradizionale fra costi delle modifiche e momento in cui vengono attuate

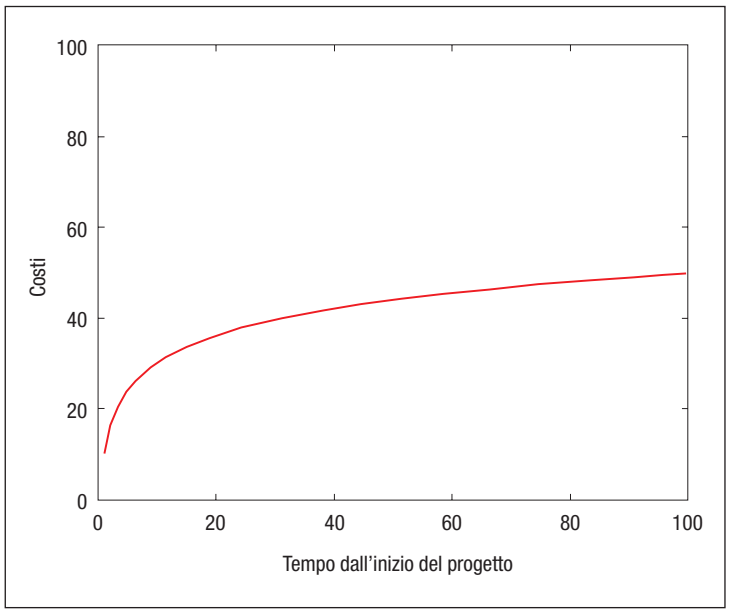


FIGURA 2
Relazione fra costi delle modifiche e momento in cui vengono attuate secondo XP

[3] Per una curva rappresentante dati reali si veda l'articolo di Bohem [2].

costruiti prevedendone una certa sopravvivenza nel tempo, l'approccio dell'XP deve essere stemperato in forme meno estreme, in particolare, attraverso un maggiore investimento nelle attività di acquisizione, analisi e specifica dei requisiti. Sarcasticamente, Beck vede in tali attività una pura perdita di tempo: una produzione di graziosi diagrammi di dubbia utilità che nessuno utilizza durante lo sviluppo dell'applicazione. Non c'è dubbio che, in molti casi, questa fase cruciale venga interpretata come il burocratico aderire a standard aziendali che prevedono la produzione di moduli cartacei e di diagrammi talvolta superflui. Forse perché la metodologia proposta risulta essere particolarmente difficile da applicare e facile ai fallimenti? Non sembra opportuno credere che Beck abbia scelto questo nome per insinuare alcunché di simile, né per sottolineare il suo approccio integralista nel rifiuto delle pratiche correnti dell'ingegneria del software. È anche vero che, in assenza di una focalizzazione mirata, esiste il rischio di concentrarsi su aspetti dell'applicazione che non hanno alcun interesse o scarsa priorità per il committente. Ma esiste un rischio anche più serio che lo sviluppo dei prodotti software degeneri in un processo senza fine del tipo *code & fix* [8]. La specifica dei test che devono essere superati da ciascuna nuova versione del prodotto è, di fatto, solo un criterio illusorio di accettazione. Per loro natura, i test definiscono solo un campione finito di possibili comportamenti del prodotto: non possono, dunque, esaurirne né la specifica né l'accettabilità. In sistemi dalle caratteristiche critiche e, a maggior ragione, in sistemi *safety critical*, inoltre, è del tutto improponibile che non si ponga l'accento sulla necessità di un'analisi a priori. La letteratura è ricchissima di esempi di malfunzionamenti che sono da ascrivere ad un'errata comprensione iniziale o a un'inefficace formulazione dei requisiti [19, 20]. Un'analisi fondata su metodi rigorosi o formali potrebbe, invece, ridurre i rischi o del tutto eliminare le cause dei malfunzionamenti. Ciò è coerente con quanto accade in settori ingegneristici dalla tradizione maggiormente consolidata, che hanno spesso reso obbligatorie alcune forme di analisi preliminari: si pensi ai calcoli del cemento armato o ad altre forme di verifica che devono essere svolte, a priori, per fare convalide progettuali che devono necessariamente precedere la realizzazione. È, quindi, opportuna l'enfasi che l'XP pone nel ricordare che costi, tempi, qualità dei risultati e generalità del prodotto non sono variabili indipendenti l'una dall'altra, ma si influenzano a vicenda: progettisti e manager tendono a volte a dimenticarlo provocando l'esplosione dei costi. Il suggerimento di Beck è, in sintesi, quello di sacrificare la generalità, rinunciando ad anticipare aleatorie e imprevedibili evoluzioni future e, quindi, rinunciando a produrre soluzioni progettuali che facilitino tale evoluzione. Così facendo, però, Beck cade in contraddizione con ciò che Parnas ha insegnato attraverso il principio di *design for change* [17] e che generazioni di ingegneri del software hanno (con maggiore o minor successo) messo in pratica nell'ultimo ventennio. Lo sforzo di individuazione delle possibili evoluzioni future del sistema non deve certo diventare un esercizio sterile e del tutto teorico che rischia di prolungare i tempi di analisi senza reali riscontri pratici. Tuttavia, una costante attenzione alle probabili future evoluzioni costituisce uno dei principi basilari su cui si fonda l'ingegneria del software. Dunque non si può che convenire sull'opportunità che i progettisti profondano molte energie e tutta la loro esperienza e sensibilità nell'identificazione di quelle parti del sistema suscettibili di maggiori modifiche, strutturando le applicazioni in modo tale che i cambiamenti più probabili siano anche i meno costosi, cosicché i loro sforzi siano ripagati nel tempo e gli investimenti ammortizzati.

Malgrado la proposta dell'XP, intesa come metodo generale unitario sia, da chi scrive, ritenuta poco convincente, si deve constatare che essa ha recentemente riscosso una certa popolarità.

Le ragioni sono duplici: da un lato, ragioni effimere che derivano dalla moda e dalla novità dell'approccio, dall'altra ragioni più profonde che derivano dall'aver riproposto in una forma nuova principi e metodi consolidati. Le ragioni del primo tipo sembrano dovute principalmente a due fattori collaterali, più che al loro intrinseco valore:

- l'uso della metafora e dello slogan teorizzato come mezzo per comunicare e condividere gli

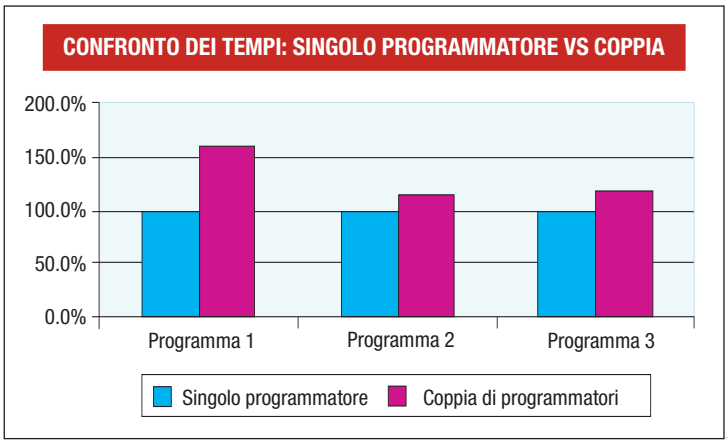
obiettivi; come insegna la realtà quotidiana, però, metafore e slogan, pur essendo formidabili metodi di aggregazione di massa, sono spesso delle grossolane semplificazioni che lasciano spazio a pericolose ambiguità, e trascurano aspetti fondamentali dei problemi;

- l'uso di semplici ricette empiriche per gestire la complessità dello sviluppo; se è vero che ogni sforzo deve essere fatto per dominarne l'intrinseca complessità, Brooks ha insegnato che non esistono facili scorciatoie. Lo sviluppo di software è una complessa attività di progettazione che si basa *in primis* sulle capacità individuali delle persone e sulla loro attitudine a cooperare nel lavoro di gruppo. Non esistono ricette generali prefissate che si possono dare a supporto di questa attività, ma solo una serie di principi, metodi, tecniche e strumenti che gli ingegneri del software, di volta in volta, devono essere in grado di aggregare in un processo atto a sviluppare lo specifico progetto sul quale sono impegnati. Purtroppo, invece, periodicamente, il fascino illusorio della semplificazione estrema dato dai ricettari standard riappare come un silver bullet nello scenario dell'ingegneria del software.

Si esaminano ora, invece, le ragioni positive della diffusione dell'XP. Innanzitutto, si osserva che, malgrado si sia cercato in passato di introdurre metodi sistematici di sviluppo, la produzione artigianale di software (code & fix) è un approccio ancora, estremamente, diffuso. Le metafore e le ricette (magari sostenute dall'uso di strumenti semplici e opportuni) proposte dall'XP possono essere un modo per insinuare alcune idee dell'ingegneria del software in ambienti altrimenti restii. Un po' come è successo recentemente con javadoc, lo strumento che Sun distribuisce con Java e che permette di ottenere documentazione direttamente dai commenti del codice: la *literate programming* non è, di per sé, un'idea rivoluzionaria, né una tecnica nata con Java, tuttavia, si deve riconoscere l'indubbia efficacia che ha avuto la diffusione di un simile strumento sulla qualità della documentazione del software prodotto in ambito accademico o di ricerca e sviluppo. Così, per esempio, l'organizzazione dello sviluppo basata sulle "storie" riflette la necessità ben nota di coinvolgere il committente nella validazione del sistema, fin dalle prime fasi della sua costruzione, e l'opportunità dell'uso di elementi tangibili per la pianificazione e il tracciamento dei progressi del lavoro. La collaborazione del committente allo sviluppo, quando è praticabile, è senz'altro auspicabile, anche perché riduce la conflittualità contrattuale. Molto spesso, però, quello che si vorrebbe davvero è il coinvolgimento degli utenti finali che potrebbero avere obiettivi diversi da quelli identificati dal committente e, soprattutto, diversi fra loro. Non convince, invece, l'idea che la mera partecipazione del committente all'interno del progetto riduca la necessità di un'attenta documentazione di specifica. Se si può convenire che tale specifica possa non essere necessaria sul momento, la sua necessità si giustifica quando occorre revisionare e far evolvere l'implementazione. In che modo è possibile risalire dal codice alle sue motivazioni? Come si può risalire dall'implementazione ai principi ispiratori delle scelte di progetto?

La tecnica del pair programming appare utile in molti casi. La programmazione è, infatti, tradizionalmente considerata un'attività solitaria praticata da persone introverse e scostanti anche se geniali. Tant'è che viene da chiedersi quanto questa mitologia del *real programmer* [18] abbia influito sulla cronica scarsità di presenza femminile fra gli sviluppatori. L'XP propone la programmazione a coppie che può, in effetti, essere un modo per migliorare la comunicazione di tecniche e obiettivi all'interno del gruppo di lavoro rendendolo più omogeneo e per introdurre ripetute ispezioni del software volte a migliorarne la qualità. Le figure 3, 4 e 5 mostrano i ri-

FIGURA 3
Confronto fra il tempo di realizzazione con e senza pair programming [4]



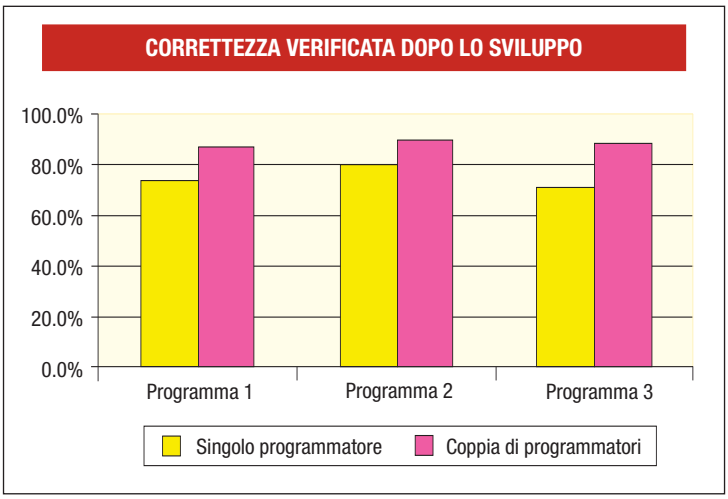


FIGURA 4

Confronto fra la correttezza con e senza pair programming [4]

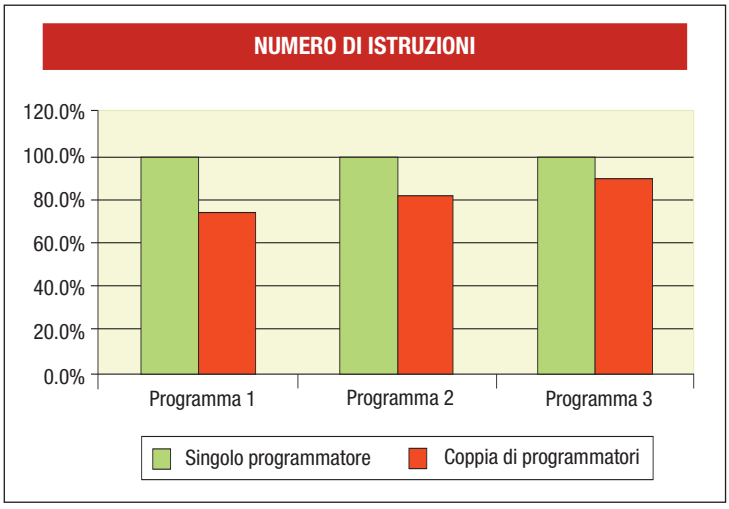


FIGURA 5

Confronto fra la compattezza del programma con e senza pair programming [4]

sultati di un esperimento [4] condotto per valutare gli effetti dell'introduzione della programmazione a coppie. La correttezza (Figura 4) e la compattezza (Figura 5) del codice aumentano, significativamente, a prezzo di una modesta diminuzione della produttività (Figura 3). Secondo J. Nosek [14], il tempo risparmiato assegnando a due programmatori il lavoro che potrebbe fare uno solo è in media il 29%. Altri studi [11] mostrano che la programmazione a coppie permette di far crescere velocemente ed efficacemente le capacità dei novizi e migliora la qualità del lavoro perché le persone coinvolte beneficiano dei rapporti interpersonali.

L'uso delle tecniche di pair programming comporta, dunque, un aumento del costo del personale, ma permette di essere più prontamente sul mercato e nel caso di progetti in cui il *time to market* è un fattore critico, può effettivamente portare vantaggi economici [13]. Inoltre, accoppiando programmatori di diversa esperienza, essa risulta essere un utile strumento didattico, anche se abbastanza costoso per tutte le parti in causa.

Un'altra tecnica che ha raggiunto una certa diffusione è la scrittura di test di unità prima della codifica dell'unità stessa. La popolarità di tale tecnica la si deve soprattutto alla disponibilità di numerosi e pratici strumenti per la loro esecuzione automatica (il più noto dei quali è Junit [10]). Nella sua essenza essa può essere considerata una forma di programmazione per contratto, in cui i contratti non sono descritti in maniera dichiarativa, ma procedurale. Pur vantaggioso, l'approccio procedurale, comporta principalmente due problemi:

- una specifica dichiarativa (sostanzialmente *intensiva*) è normalmente più sintetica e descrittiva di una procedurale (*estensiva*); per di più, una specifica procedurale, data attraverso casi di test in numero finito, risulta necessariamente parziale;
- alcune proprietà sono difficili da descrivere in maniera procedurale; si pensi ad un programma Java che faccia uso di *thread*: per scrivere dei test che mettano in evidenza delle corse critiche bisognerebbe poter manipolare la macchina virtuale Java.

La tecnica di anticipare la scrittura dei test è, talvolta, ritenuta efficace per facilitare la stessa scrittura del codice dell'applicazione. Uno studio recente [12] mostra che, in generale, ciò non accade, ma coerentemente con l'interpretazione della programmazione per contratto, la presenza dei test facilita il riuso del codice in contesti nuovi o diversi. L'integrazione continua garantisce che esista, in ogni momento, un prototipo o una versione dell'applicazione funzionante. Questa è, in generale, una condizione assai desiderabile, come del resto da tempo mettono in evidenza molti autori[4], perché per-

[4] Il processo *synchronize & stabilize* adottato da Microsoft [6] si basa fondamentalmente su questo principio.

mette di convalidare continuamente ciò che è stato costruito. Non va dimenticato però, che affidando la convalida dei requisiti sempre e solo all'esecuzione dell'applicazione, la possibilità di esplorare soluzioni alternative è fortemente ridotta. L'integrazione continua e la proprietà condivisa di tutto il codice, poi, sembrano tecniche del tutto inapplicabili quando il numero di programmatori sale e il numero dei conflitti diventa ingestibile. Considerazioni simili valgono per le ristrutturazioni del codice. Al momento gli strumenti per operare queste ristrutturazioni in maniera semi-automatica sono ancora piuttosto rozzi. Pertanto, attualmente, sembra possibile ristrutturare in maniera consistente solo porzioni di codice di dimensioni relativamente modeste.

5. CONCLUSIONI

L'eXtreme Programming è nata, negli ultimi anni, come approccio radicalmente nuovo al processo di sviluppo del software. In questo articolo, dopo averne illustrato i tratti salienti, si è cercato di collocare XP all'interno del quadro complessivo dei metodi di supporto al processo. È stato inoltre evidenziato come XP abbia le proprie radici all'interno della classe di metodi agili, flessibili ed incrementali che da molti anni sono stati proposti, e largamente applicati, in alternativa ai processi a cascata che si dimostravano invece inadeguati nelle situazioni, per altro molto frequenti, caratterizzate da incertezza e variabilità nei requisiti o dalla necessità di risposta in tempi rapidi alle esigenze di un mercato in rapida e continua evoluzione.
Il giudizio che viene espresso è che, se le motivazioni alla base dell'approccio risultano largamente condivisibili, assai criticabile appare il tentativo di congelare la risposta in un metodo di processo che, a sua volta, si presenta come predefinito e universale. Mentre alcuni singoli suggerimenti e tecniche dell'XP pare che possano costituire utili strumenti nel bagaglio degli attrezzi di cui può disporre l'ingegnere del software, si è, invece, fondamentalmente critici riguardo alla loro aggregazione in un metodo unitario che venga presentato come la soluzione di tutti i problemi del processo software. Questa idea, come si è detto, è fallita in passato ed è destinata a fallire in quanto non tiene conto delle specificità che ogni singolo progetto e ogni singola organizzazione caratterizzano [5].
Da ultimo, si vuole, invece, evidenziare quello che, secondo il parere degli autori del presente articolo, è il contributo più importante dell'XP al dibattito scientifico all'interno dell'ingegneria del software. In sintesi, XP ricorda con grande determinazione che il fine ultimo dell'ingegneria del software è produrre programmi: il codice, spesso considerato semplicemente come il risultato finale di un lungo ed elaborato processo, è il vero obiettivo e su di esso (che lo si voglia o no) finiscono con il concentrarsi gli sviluppatori.
Si deve riconoscere che, troppo spesso, l'ingegneria del software ha prodotto metodi inutilmente elaborati e onerosi e che, troppo spesso, gli standard aziendali si sono dimostrati eccessivamente meticolosi e burocratici. In molti casi, gli sviluppatori non sono riusciti a vedere vantaggi sensibili nella loro adozione, ma solo un intralcio nel procedere verso l'implementazione. Gli strumenti di *Computer Aided Software Engineering* (CASE) - ovvero, un altro silver bullet del passato - sono in larga misura falliti proprio perché nel codice prodotto mancava l'evidenza della sua relazione con le fasi di progetto e, quindi, alla fine, non portavano a un codice migliore, più facile da far evolvere, ma semplicemente sviluppato più velocemente.
La lezione da trarre per il futuro è, dunque, che i metodi e gli strumenti di supporto al processo dovranno consentire una rapida transizione al codice e dovranno fornire visibili benefici in termini di verificabilità e modificabilità del codice. La documentazione di analisi e di progetto non solo dovrà automaticamente essere legata al codice ed evolvere con esso, ma dovrà giocare un ruolo attivo, sia nel generare automaticamente parte dell'applicazione che nel generare automaticamente strumenti per la sua convalida.
Sembra opportuno, a questo punto, concludere sottolineando come le affermazioni di efficacia fatte dai paladini dell'XP, siano

quasi sempre argomentate su basi puramente ideologiche e perciò prive di supporto sperimentale.
Gli studi recenti apparsi sull'argomento [4, 12, 13] corroborano solo in parte le argomentazioni dell'XP. Molto lavoro di ricerca è, dunque, ancora necessario per valutare gli effetti delle singole tecniche proposte, in modo che l'ingegnere del software possa veramente considerarle nuove frecce a disposizione del suo arco.

Bibliografia

[1] Boehm BW: A spiral model of software development and enhancement. *IEEE Computer*, Vol. 21, n. 5, 1988.

[2] Bohem BW: Software engineering. *Transactions on Computers*, Dec.1976.

[3] Brooks F: No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, Vol. 20, n. 4, Apr. 1987.

[4] Cockburn A, Williams L: *The costs and benefits of pair programming.* In Extreme Programming Examined. Addison Wesley, 2001.

[5] Cugola G, Ghezzi C: *Software processes: a retrospective and a path to the future*. Software Process 14 Improvement and Practice 4, 3, Sept. 1998.

[6] Cusumano MA, Selby RW: *Microsoft Secrets*. The Free Press, New York, NY, 1995.

[7] Cusumano MA, Yoffie DB: Software development on Internet time. *IEEE Computer*, Vol. 32, n. 10, Oct. 1999.

[8] Ghezzi C, Jazayeri M, Mandrioli D: *Fundamentals of Software Engineering, second ed*. Prentice Hall PTR, Apr. 2002.

[9] Jacobson I, Rumbaugh J, Booch G: T*he Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[10] Junit: http://www.junit.org.

[11] Lippert M, Roock S, Wolf H, Zullighoven H: JWAM and XP: *Using XP for framework development.* In Extreme Programming Examined. Addison Wesley, 2001.

[12] Muller M M, Hagner O: *Experiment about test-first programming*. In Conference on Empirical Assessment In Software Engineering EASE '02 (Keele, Apr. 2002).

[13] Muller MM, Padberg F: *Extreme programming from an engineering economics viewpoint*. In Fourth International Workshop on Economics-Driven Software Engineering Research (EDSER), Orlando, Florida, May 2002.

[14] Nosek J: The case for collaborative programming. *Communications of the ACM*, Vol. 41, n. 3, Mar. 1998, p. 105–108.

[15] Parnas DL: *Designing software for ease of extension and contraction*. In Proceedings of the Third International Conference on Software Engineering, 10-12 May 1978,

[16] Pressman RS: *Principi di Ingegneria del software*. Second ed. Mc Graw Hill, 1997.

[17] Raymond, ES: The Cathedral and the Bazaar. [Online.] Available: http://www.ccil.org/~esr/writings/cathedral-paper.html, Nov. 1997.

[18] Real programmers:http://www.cirr.com/~barkley/jokes/realprog.html.

[19] Neumann PG: *Computer Related Risks*. ACM Press 1995.

[20] Jackson M: *Software Requirements and Specification: a lexicon of practice, principles, and prejudices*. Addison-Wesley, 1995.

CARLO GHEZZI è professore ordinario di Ingegneria del Software presso il Politecnico di Milano e responsabile scientifico dell'area di ricerca sull'ingegneria del software presso il CEFRIEL. Autore di numerosi articoli scientifici e libri, è editor in chief della rivista "ACM Transaction on Software Engineering and Methodology".
e-mail: ghezzi@elet.polimi.it

MATTIA MONGA ha conseguito il dottorato di ricerca in Ingegneria Informatica e Automatica presso il Politecnico di Milano nel 2001. I suoi interessi di ricerca riguardano l'ingegneria del software in ambito Internet e i linguaggi di programmazione orientati agli aspetti.
e-mail: monga@elet.polimi.it