S-5 “Software di calcolo”

S.5.1.-Premessa

Sebbene il software vada progressivamente assumendo un ruolo sempre più importante nella pratica ingegneristica, non sembra che la conoscenza della sua struttura né delle sue regole si stia parimenti diffondendo, come invece dovrebbe. Al contrario, v’è una percolosa confidenza nell’uso dei programmi, alla quale non corrisponde una approfondita conoscenza non soltanto dei risultati teorici che sono alla base di certi algoritmi o metodi di calcolo, bensì anche di tutti i possibili accorgimenti necessari al fine di avere il pieno dominio dei risultati della elaborazione. I risultati di una elaborazione vengono ad essere trasformati in una specie di verdetto indiscutibile, che da solo sarebbe in grado di decidere del problema. Tende a scomparire il dubbio, la discrezionalità, la argomentata discussione dei risultati.

Sulla crescente importanza del software basterà dire che i progetti normalmente eseguiti (un po’ in tutti i settori della ingegneria), hanno ormai sempre un qualche elaborato software a corredo. Spesso i calcoli di progetto di fondano sui risultati di una qualche elaborazione automatica, e ciò è ormai considerato del tutto naturale. La delicata catena di decisioni che congiunge il problema in esame con i risultati di una elaborazione viene normalmente taciuta o scarsamente presa in considerazione, quasi che gli anelli di questa non possano che essere percorsi in modo deterministico, portando alla mappa dei risultati esibita, che si propone come oggettiva fotografia della realtà fisica.

La creazione di programmi per elaboratore è una disciplina molto giovane, in tumultuoso sviluppo non soltanto per quanto attiene ai contenuti dei programmi, ma più ancora per quanto riguarda le tecniche di sviluppo, le architetture con le quali questo è concepito, la teoria dei test, i linguaggi, i sistemi operativi, ecc. ecc.. Una rivoluzione permanente che, come tutte le rivoluzioni, può comportare pericolosi squilibri e- a volte – anche abbagli madornali.

Questa nota vuole cercare di dare in rapida sintesi – e pertanto senza alcuna pretesa di essere esaustivi -, alcune indicazioni generali utili per un uso più accorto dei programmi per elaboratore. In specie saranno trattati i software di calcolo, ma molte delle considerazioni fatte saranno estendibili ad altri tipi di software.

S.5.2.-Sviluppo software

A costo di innumerevoli sforzi le teorie hanno ricondotto la messe di dati sperimentali entro cornici di razionalità che consentono non solo di spiegare le esperienze, ma anche di guidarle, e, in parte, prevederle. Le teorie non rappresentano però la *spiegazione* del reale, bensì solo una sua possibile interpretazione (perfezionabile e falsificabile), affetta da approssimazioni ed errori. Ciò viene spesso dimenticato nella pratica, nell’ottimistico assunto che la teoria *descriva* la realtà fisica: invero la teoria si limita ad imitarla, e spesso è inadeguata al farlo in modo compiuto perché il problema – nonostante sia magari frequente e di importanza basilare- è troppo complesso. Questo è un errore che si commette frequentemente accettandone più o meno tacitamente le conseguenze, e spesso però dimenticando la sua esistenza. Ad esempio nessun calcestruzzo è elastico lineare, ma i

calcoli dei telai vengono spesso fatti con la teoria dell'elasticità, pretendendo però di prestar fede alla terza cifra decimale.

L'impiego dei modelli teorici al fine di pervenire a risultati ingegneristicamente fruibili, comporta nella maggior parte dei casi la trasformazione delle equazioni governanti il problema allo studio in equazioni più semplici, le quali approssimano in modo controllato la soluzione delle equazioni originarie. Si introduce così un secondo tipo di errore, invero di natura assai benigna rispetto a quella degli altri: l'*errore numerico* legato al metodo di approssimazione usato nel calcolo. Il calcolo numerico mostra in maniera rigorosa in che modo questo genere di errore possa essere dominato, quantificando limiti superiori in funzione di opportuni indici numericamente valutabili. L'errore numerico può pertanto essere stimato in modo quantitativo, e questo è un risultato di eccezionale importanza perché apre la strada all'uso concreto dei metodi di calcolo automatici.

Sebbene ridotto ad equazioni più semplici, il problema da risolvere non può nella maggior parte dei casi essere risolto "a mano". Sorge dunque la necessità di scrivere programmi di calcolo che effettuino tutte le operazioni necessarie in modo automatico, ripetendo milioni di volte le operazioni elementari nelle quali il problema originario è stato decomposto dal metodo numerico. La complessità è però tale che nella maggior parte dei casi non basta scrivere programmi di poche decine o centinaia di istruzioni. E' assolutamente normale che i programmi abbiano decine di migliaia di istruzioni, e taluni anche centinaia di migliaia o alcuni milioni di istruzioni.

Architetture così complesse di istruzioni interdipendenti possono, in linea di principio, essere costruite in modo "esatto". Tuttavia – nonostante tutto - è ancora molto improbabile che ciò avvenga: a tacer d'altro è esperienza comune l'aver constatato come programmi appartenenti a tipologie anche molto differenti, e realizzati con vasto impiego di mezzi, siano soggetti a malfunzionamenti e persino a risultati di dubbia affidabilità.

Viene così a comparire un nuovo tipo di errore, di natura estremamente subdola perché scarsamente quantificabile: l'errore di programmazione, o *software bug*. La disciplina che si occupa di indagare sull'esistenza di bug, e delle procedure atte a limitarne l'esistenza, è una disciplina giovanissima, di estremo interesse ed importanza: in questi anni siamo in un' era pioneristica a questo riguardo.

Agli albori della programmazione, ma talvolta ancora oggi, gli enormi insiemi di istruzioni che fanno i programmi venivano organizzati in routine o funzioni che eseguivano compiti specifici, utili ad un certo scopo del programmatore (*approccio procedurale*). La divisione in routine era ampiamente arbitraria, nel senso che il programmatore era completamente libero di raggruppare le istruzioni in funzioni secondo il proprio personale tornaconto o convincimento. La sovrapposizione, nel tempo, di strati diversi di arbitrarietà, tendeva a rendere difficile il mantenimento dei programmi, se non a volte impossibile: si danno casi di programmi letteralmente buttati nel cestino perché totalmente impossibili da mantenere. E' classico l'esempio della famigerata serie di "goto" che obbligando a saltare qua e là nei sorgenti poteva rendere assolutamente impossibile capire ove ci si trovasse dentro un sorgente, e perché.

In seguito si è trovato un modo logicamente più pulito di raggruppare le istruzioni in modo da consentirne un'analisi partizionata ed efficiente. Si tratta della programmazione "a oggetti" che le più recenti linee di sviluppo considerano imprescindibile per la efficiente gestione di un programma complesso. Nella programmazione "a oggetti" (OOP, object oriented programming), il programmatore si sforza di definire oggetti astratti che nel corso della elaborazione interagiscono tra loro, portando a compimento lo scopo del programma. Nella programmazione ad oggetti continuano ad esistere le funzioni, ma queste vengono ad essere innervate negli oggetti, così da rispettare il criterio logico che prevede che ogni oggetto possa fare certe cose di sua pertinenza, ma non altre che non gli competono. Ad esempio, nella programmazione di un simulatore di volo potrà probabilmente esserci un oggetto "timone di coda". Il *timone di coda* potrà modificare la propria posizione, ma non potrà certo dire ai motori alcunchè. L'oggetto *timone di coda* avrà propri controlli e propri dati (per esempio un angolo di incidenza), accessibili solo mediante protocolli

rigidamente assegnati all'oggetto stesso: si parla in questo caso di incapsulamento dei dati. La rigidità delle procedure necessarie a modificare un dato consente di universalizzarne il controllo, e così – ad esempio - non potrà mai essere possibile assegnare un angolo maggiore di 35° al timone in questione. Grazie a queste tecniche lo sviluppo è in qualche modo costretto a seguire linee di rigore logico che – oltre a facilitare il mantenimento – rendono di per sé meno probabili gli errori. Lo sforzo di astrazione e di nitore logico che oggi occorre compiere per realizzare un buon programma è però totalmente invisibile all'utilizzatore, che vede solo i risultati ma non sa come ci si è giunti. Del resto, persone come Bjarne Stroustrup, che ha inventato il linguaggio C++ ([3]), e che nella sua opera ha creato un intero insieme di nuove regole formali di straordinaria astrazione e nitore (pur con le immancabili critiche degli sviluppi successivi, come java), sono del tutto sconosciute all'immensa platea degli utilizzatori.

Invero, la stessa creazione di programmi viene spesso nella vulgata corrente equiparata alla scrittura di immensi elenchi telefonici: il lavoro del programmatore sarebbe unicamente quello di prevedere l'esecuzione di istruzioni volte al raggiungimento di un ben determinato obbiettivo, il quale sarebbe raggiunto per mezzo della realizzazione di un certo numero di passi obbligati. Un lavoro grigio ed assai scarso di spunti innovativi. Non è così.

In realtà, lo sviluppo di software avente determinati compiti è possibile in mille modi diversi, non solo per quanto attiene all'aspetto visibile del prodotto (ovvero la sua interfaccia grafica), ma anche e soprattutto per quanto riguarda le strategie di programmazione, il rigore logico con il quale è stato concepito, il livello di astrazione della sua struttura formale, la cura nel prevedere i casi particolari, la generalità minore o maggiore con la quale è stato sviluppato. Inoltre (se ne parlerà meglio più avanti) non è affatto detto che sia immediata la programmabilità del problema: spesso chi sviluppa software deve fare l'indispensabile lavoro di razionalizzazione, omogeinizzazione e formalizzazione che trasforma un ammasso incoerente in un ordinato svolgersi di eventi.

In pratica chi realizza un software deve ideare un modello che includa il problema da esaminare e che sia capace di trattarlo con la necessaria generalità. Il modello non è affatto scontato e spesso è tutt'altro che semplice inventarlo, sia per la vastità del problema, sia per la intrinseca caoticità che spesso un problema reale presenta. Se il modello è sbagliato o incompleto, se non è capace di contenere tutti i casi che si presenteranno in realtà, il software è destinato ad avere seri problemi, che potranno magari evidenziarsi anche a distanza di mesi dall'effettivo rilascio.

S.5.3.-Validazione, test e certificazione del software

La creazione di vaste architetture di moduli interconnessi, spesso realizzati da produttori diversi, porta come inevitabile conseguenza alla creazione di programmi affetti da errori. Sebbene molto lavoro sia stato fatto nella direzione di una maggiore affidabilità dei prodotti software (si citano qui ad esempio tra i tantissimi testi sull'argomento [1] e [2], il titolo del quale, "fuori dalla fossa di catrame", è già indicativo), ad oggi è ancora sostanzialmente vero che data un'applicazione complessa esistono in essa un certo numero di bug sfuggiti ai controlli preliminari, ed agli stessi controlli degli utilizzatori. Basterebbe tener conto di quanto avviene ad ogni rilascio di un nuovo sistema operativo, e fare mente locale a tutti gli innumerevoli piccoli o grandi problemi che tutti quanti noi abbiamo sperimentato negli ultimi anni con le più diverse applicazioni, per convincersi del fatto che il problema dell'esistenza di bug riguarda in pratica tutti: il fatto è che la dimensione dei software tende a crescere al crescere del fatturato delle case madre, per cui la quantità di soldi investiti in test e controllo qualità per migliaio di istruzioni potrebbe non essere una funzione monotonamente crescente con il crescere del fatturato. D'altro canto è ormai banale dire quanto

drastico sia stato lo sviluppo non solo del numero delle applicazioni (cosa già di per sé significativa), bensì soprattutto delle tecniche di programmazione, dei linguaggi, degli standard negli ultimi dieci anni. L'innovazione continua, se da un lato ha indiscutibilmente portato alla creazione di tecniche e ambienti di lavoro di via via maggiore affidabilità, dall'altro lato ha richiesto cambiamenti pressochè continui, che hanno sicuramente favorito la creazione di errori. Infatti i programmatori hanno dovuto più di una volta rivoluzionare le loro conoscenze, imparando tecniche nuove, ed inoltre immensi giacimenti software di ore-uomo, trovatisi improvvisamente esclusi, si sono dovuti riscrivere o riadattare.

In generale il software bug ha una natura estremamente subdola perché è completamente privo di attinenza con il contesto, non segue alcuna regola fisica né numerica, spesso non si ripresenta allo stesso modo ma genera problemi random. Il software bug su taluni sistemi può risultare completamente inavvertito, mentre su altri può portare a crash brutali. In assenza della manifestazione conclamata del bug, essenzialmente legata alla sua ripetibilità, può essere assai difficile trovare la causa del problema. Di fronte ad un comportamento anomalo l'utilizzatore dovrebbe prendere con cura nota dei dettagli del problema e cercare di fornire il massimo numero di informazioni dettagliate al produttore. A volte il bug non è in realtà un errore, ma semplicemente la manifestazione di un uso improprio o non previsto di una applicazione: tipicamente è il caso di programmi scritti per un ambiente e fatti eseguire in simulatori di quell'ambiente. Il software bug può manifestarsi dopo aver fatto cambiamenti nel sistema che nulla hanno a che vedere (apparentemente) con l'applicazione che genera il bug. Caso tipico i programmi che cominciano a funzionare scorrettamente se si cambiano le impostazioni sull'ora o sulla tastiera, o sul formato dei numeri: programmi magari provati con impostazioni americane che non funzionano con altre impostazioni. A chi scrive è capitato di scoprire che un noto programma sbagliava brutalmente, ma all'altro capo dell'oceano atlantico il bug non veniva riscontrato, sullo stesso modello di calcolo e con la stessa versione del programma. Nemmeno una reinstallazione risolveva il problema. Fu necessario disinstallare la applicazione eppoi reinstallare la stessa applicazione perché il problema sparisse. Situazioni come questa lasciano interdetti. Il bug si annidava evidentemente in un componente che non veniva installato dalla routine di installazione, in quanto già presente sul computer. Ma il componente installato conteneva qualche problema, che poi originava il bug.

Non c'è alcuna correlazione tra la gravità di un bug e la difficoltà che il produttore può incontrare nel risolverlo. A volte piccoli problemini o migliorie funzionali richiedono una riscrittura, mentre errori catastrofici si possono correggere il mezz'ora di analisi. Per questo motivo il Produttore dovrebbe fornire versioni emendate in tempi ragionevoli, e non in mesi come purtroppo oggi avviene in molti casi. A meno che il software non stia incontrando pesanti revisioni sembra legittimo chiedere l'invio di patch entro pochi giorni o al massimo due o tre settimane dalla segnalazione.

Volendo dare una classificazione della gravità dei bug dal punto di vista dell'utilizzatore, ed avendo in mente in specie i programmi di calcolo, si può introdurre questa codifica:

1. lievissimo: il bug si limita a generare piccoli problemi estetici;
2. lieve: il bug comporta macchinosità nelle operazioni di input o incompletezza della documentazione in output (per esempio numeri troncati o incompleti);
3. marcato: il bug genera errori in casi molto particolari con plateali effetti sui risultati, immediatamente riscontrabili anche da un utente non particolarmente attento (ad esempio numeri 1.#INF, sporcamenti di dati, diagrammi fuori da ogni logica, numeri assurdi, e così via);

4. grave: il bug genera problemi sistematici su operazioni rare in situazioni molto particolari e non dà chiari segni della sua esistenza (ad esempio un numero che deve essere 18000. viene erroneamente calcolato 3700);
5. gravissimo: il bug genera numeri errati in casi frequenti ed importanti, gli effetti non sono immediatamente riscontrabili, neanche dall'utente più atttento;

In una scala da 1 a 10 i bug sono così classificati in [1]:

1. lieve (problemi estetici, come un disallineamento di output);
2. moderato (output confusi o ridondanti, il bug modifica le performace di sistema);
3. irritante (nomi troncati, invio di conti per 0€, operazioni macchinose e artificiose);
4. disturbante (rifiuto di eseguire operazioni legittime);
5. serio (perdita della traccia di transazioni avvenute, perdita di attendibilità);
6. molto serio (errori di attribuzione nelle transazioni);
7. estremo (problemi frequenti e arbitrari, non sporadici e su casi particolari);
8. intollerabile (distruzione di database, per di più non immediatamente riconosciuta);
9. catastrofico (la decisione di spegnere è tolta da noi perché il sistema va in crash);
10. infestante (problemi diffusi ad altri sistemi ed altri computer).

Alcuni ricercatori hanno cercato di quantificare il numero di bug mediamente presenti nei sorgenti, e le stime sono estremamente interessanti: sempre in [1] si dà la misura statistica di 2,4 bug ogni mille istruzioni. Si comprende come un numero assai elevato di bug sia atteso in applicazioni di decine o centinaia di migliaia di istruzioni.

Data per scontata l'esistenza di errori sorge immediatamente il problema di validare i software, in specie quelli di calcolo.

Il problema della validazione del software è un problema formidabilmente complesso e di straordinaria attualità. Se sottoponiamo un software complicato ad un test cosa possiamo concludere in merito al software provato? La verità rigorosa è che possiamo solo concludere che in quelle condizioni (anche relativamente alla macchina ed agli altri processi contemporaneamente eseguiti), con quei dati di partenza, e seguendo quella certa successione di operazioni, il software funziona correttamente. Ciò ovviamente non prova che il software, in altre condizioni, con altri dati, o seguendo un diverso percorso, o su un diverso elaboratore non possa dare luogo ad errori o malfunzionamenti.

In software complessi esistono migliaia e migliaia di bivii, o trivii, quadrivii, ecc., per cui è davvero impossibile – ad oggi - sperimentare tutti i percorsi possibili. L'esecuzione di un test (*esperimento software*) prova solo quanto avviene con certi dati di ingresso ed in un dato momento-macchina, mentre l'esperimento fisico (ad esempio quello di Galilei dalla Torre di Pisa) è assai più fruttuoso, perché il risultato è garantito dalla intima coerenza delle leggi fisiche, che non sono state "programmate", ma, ancorchè ignote, esistono in quanto tali. All'interno di un software, per le cause più varie ivi incluso il dolo, si possono annidare istruzioni che hanno effetti totalmente bizzarri, privi di alcuna logica che non sia quella dell'errore o della distrazione. Accade così che magari con dati diversi, o facendo una diversa serie di operazioni che dovrebbero portare allo stesso punto, o magari a causa della presenza contemporanea di altri processi incorrelati a quello in fase di test, o per l'installazione sul computer di altri programmi che modificano componenti di sistema, l'applicazione dia luogo ad errori o malfunzionamenti. Nessuna legge fisica sta dietro la costruzione di un programma software, ma solo il più o meno ordinato e controllato insieme di migliaia o milioni di istruzioni, e magari decine di componenti di varie case produttrici.

Poiché al momento non si dispone di uno strumento capace di provare sistematicamente tutti i percorsi possibili di un programma, la creazione di test significativi è una attività difficile ed onerosa. La difficoltà consiste nel fatto che occorre predisporre delle prove realmente significative, delle vere e proprie trappole che evidenzino il bug, senza sapere a priori dove questo possa annidarsi. L'onerosità è dovuta al fatto che si deve trovare un modo indipendente dal programma per verificare i risultati.

Anche il singolo test pone rilevanti problemi di principio. Consideriamo infatti i risultati di un test eseguito su un complesso programma di calcolo. Se il problema allo studio non è un problema standard, descritto in letteratura, e se i risultati si presentano all'occhio esperto in maniera credibile (non vi sono cioè palesi irregolarità o plateali violazioni delle leggi fisiche) non esiste altro modo per provare la bontà dei risultati che la prova analogica, data la impossibilità di eseguire a mano tutte le operazioni eseguite dal programma. Se infatti è possibile che i risultati ottenuti dal programma A siano sbagliati, è fortemente improbabile che il programma B, che fornisce risultati praticamente identici, sia sbagliato anch'esso *nello stesso modo*. Occorre dunque rilevare che è solo da una analogia tra due o più sistemi potenzialmente errati che traiamo la (probabilistica) sicurezza della mancanza di errore. Diventa dunque di fondamentale importanza, nella validazione di software di calcolo, poter disporre di più programmi da eseguire in parallelo sullo stesso problema. Naturalmente nel caso dei programmi di calcolo è anche possibile provare i programmi sottoponendoli ad una trafila di prove che diano, per motivi teorici, successioni di dati aventi al loro interno delle regolarità. Per esempio, un programma di elasticità lineare dovrà dare spostamenti doppi con forze doppie, e se questo non succede il programma è errato. Le reazioni vincolari dovranno essere in equilibrio con i carichi applicati. Le azioni interne dovranno soddisfare le equazioni indefinite di equilibrio, e così via. Vi sono casi però nei quali non c'è modo di utilizzare queste regole, e casi in cui programmi da usare in parallelo non esistono o perché non ancora sviluppati, o perché non disponibili, o perché il problema può essere trattato anche in modi diversi (cfr. quanto si dirà a proposito delle norme). In questi casi la validazione deriva da test fatti "a mano" e dall'uso intelligente del programma da parte degli utilizzatori.

Dato il fatto che i test non possono essere esaustivi, la validazione del software in senso assoluto è impossibile, e ciò dovrebbe essere ben chiaro agli utilizzatori. Al momento l'unico criterio attendibile in merito alla validazione del software è quello probabilistico: la validazione serve a ridurre la probabilità di incontrare bug, non a garantire della loro assenza. L'ordinata esecuzione di test riduce la probabilità di avere problemi, ma non esime dal considerare attentamente i risultati di una elaborazione. Ad ulteriore riprova della difficoltà del problema, si tenga presente che le modifiche al programma successive ad una campagna di test, di fatto rendono quella campagna obsoleta: dovrà essere ripetuta. Ma si è dato il caso di programmi soggetti a complicate procedure di controllo qualità che sono risultati affetti da gravi errori. In uno di questi casi l'errore fu trovato sottoponendo a test una nuova applicazione che dava risultati diversi da quella già esistente: la prova analogica non era stata passata con successo, e la analitica ricerca della causa aveva evidenziato problemi nella applicazione presa come giudice. La morale è che nessun protocollo di qualità può, al momento, costituire una garanzia assoluta, pertanto nessuna "certificazione" in senso assoluto è a parere di chi scrive al momento possibile, almeno se si vogliono chiamare le cose con il loro nome.

Ne consegue che la mentalità di chi usa un programma deve essere quella di chi – magari insospettito da un risultato non in linea con le attese- predispone trappole probabilistiche volte ad evidenziare possibili errori. Nel condurre questo indispensabile lavoro i criteri da seguire sono sempre quelli: risultati identici su sistemi diversi sono con alta probabilità corretti; risultati che non soddisfano le regole teoriche dalle quali i programmi partono non sono corretti; risultati che

appaiono strani a chi ha vasta esperienza di problemi simili devono essere sottoposti ad attenti controlli; risultati ai quali si affidano parti critiche di un progetto devono sempre essere controllati.

Le considerazioni fatte portano a ritenere che almeno nell'ambito dei programmi di calcolo dovrebbe esistere uno stretto legame tra il produttore e l'utilizzatore del software. Il produttore dovrebbe fare programmi aperti all'interfacciamento con altri programmi (al fine di rendere più semplici i controlli in parallelo), dovrebbe eseguire molti test, dovrebbe informare i clienti dei bug trovati, dovrebbe produrre patch in tempi adeguati. Gli utenti dovrebbero disporre di vari programmi da usare in parallelo, e dovrebbero sempre valutare con molta attenzione i risultati ottenuti, considerando questi come una possibile rappresentazione della realtà da sottoporre ad esame, e non come una "fotografia". In questa prospettiva gli utilizzatori e gli sviluppatori diventano parte di una unica comunità professionale, nella quale vi è una sostanziale collaborazione al fine di poter disporre di strumenti via via più affidabili.

S.5.4.-Criteri per la scelta di un software

Dato un certo problema, sono spesso disponibili vari software che hanno per oggetto il problema allo studio. Si pone quindi la necessità di stabilire dei criteri utili alla scelta di un prodotto piuttosto che un altro. Volendo dare un sintetico elenco dei principali criteri possibili, occorrerà tenere in conto il fatto che non tutte le informazioni sono sempre effettivamente disponibili, e quindi alcuni dei criteri oggettivamente utili sono di fatto quasi impraticabili. Va anche detto che nessuno di questi criteri singolarmente preso può essere sufficiente a motivare la scelta finale.

Un primo gruppo di criteri o informazioni si riferisce ai contenuti, che devono essere ben chiari:

?? *Esatta classe dei modelli o problemi affrontabili, diffidando delle affermazioni generiche*. Non esistono programmi che fanno "tutto", e occorre vigilare sulle affermazioni promozionali o pubblicitarie, che a volte sono reticenti od enfatiche. Ogni programma ha un certo ambito di applicabilità che dovrebbe essere chiaramente indicato anche evidenziando eventuali limitazioni (che non sono menomazioni ma sanissime de-limitazioni). Il produttore dovrebbe indicare le aree di impiego di particolare fruibilità del programma.
?? *Albero completo dei comandi*. Sebbene sia una informazione solo in parte significativa, questa informazione può dare una idea dei contenuti di un programma.
?? *Presenza di interfacciamenti standard o aggiuntivi.* La presenza di interfacciamenti verso altri programmi (tipicamente i solutori), anche magari concorrenti, è sempre un fatto positivo, perché aumenta le possibilità di controllo da parte dell'utilizzatore.

Un secondo tipo di criteri si riferisce al costo:

?? *Modalità di commercializzazione del programma (se in acquisto o con canone d'uso, o gratuito)*. Non è affatto detto che l'acquisto sia la soluzione migliore, data la rapida obsolescenza dei programmi. La soluzione canone d'uso può essere competitiva. I programmi gratuiti possono essere eccellenti programmi, ma possono anche essere solo l'epifenomeno della personalità del loro Autore, e del suo legittimo desiderio di fare.
?? *Costo degli aggiornamenti* (se annuali o sulla base di versione). Si tenga presente che un software non aggiornato è soggetto ad un rapido decadimento a causa del fatto che è impossibile farvi assistenza, ed a causa del fatto che resta rapidamente tagliato fuori dal resto del mondo, dati, almeno, gli attuali frenetici ritmi di sviluppo dei sistemi operativi e delle caratteristiche hardware, tacendo quelle delle normative e delle consuetudini.

?? *Costo della assistenza, sia per e-mail che telefonica*. L'assistenza è spesso indispensabile per venire a capo di problemi o trarsi fuori da situazioni di dubbio. Essa dovrebbe sempre essere disponibile a chi usa un programma.

Un terzo tipo di criteri si riferisce alla affidabilità:

?? *Anno di uscita del prodotto e numero della versione di rilascio.* Programmi con maggiore vita – naturalmente a parità di ogni altro fattore - hanno più probabilità di essere privi di errori. Le versioni ".0" sono molto più esposte a bug delle versioni ".1", ".2", ecc..
?? *Esistenza di test pubblici effettuati dal produttore o da terze parti.* La presenza di casi di studio disponibili è sicuramente una buona cosa, anche se di per sé non prova la affidabilità di un prodotto.
?? *Struttura e linguaggio del programma, stile di scrittura dei sorgenti.* Un programma con varie DLL (Dynamic Link Libraries) o componenti, scritto con linguaggi ad oggetti è intrinsecamente più sicuro di un gigantesco programma procedurale. Sorgenti scritti con ordine e strutturati sono assai più facilmente gestibili (e mediamente affidabili) di sorgenti scritti senza alcuna struttura ed in modo caotico.
?? *Presenza di porting recenti.* Il recente trasferimento di un programma da un sistema operativo ad un altro (o da una struttura procedurale ad una struttura ad oggetti, o da un linguaggio ad un altro linguaggio) è una possibile causa di problemi.
?? *Lavori realizzati usando il programma.* Se molti utenti professionali hanno usato il programma e con esso hanno realizzato lavori significativi, e sono disposti a dichiararlo, questo è un indice indiretto della affidabilità del programma.
?? *Presenza della bug list e sua eventuale pubblicità.* La scoperta di bug, e la relativa lista dovrebbe essere pubblica o almeno essere informazione a disposizione degli utenti.
?? *Disponibilità del produttore ad eseguire test a richiesta.* Se il potenziale Cliente – anche non acquisito - chiede che venga eseguito un particolare test a sua scelta, eventualmente pagando un ragionevole costo per il lavoro svolto, il produttore dovrebbe accettare la richiesta.
?? *Tempo medio di correzione dei malfunzionamenti ed invio patch.* Il produttore dovrebbe dichiarare quanto impiega in media a correggere un bug segnalato dagli utenti. Se la risposta è che non ci sono mai bug sarebbe meglio diffidare.

Accanto ai criteri da seguire, occorre forse elencare alcuni criteri da non seguire, o, almeno, da non considerare esaustivi:

?? *Fama o notorietà del produttore* (non proprio sempre è indice di bontà del prodotto).
?? *Pubblicità del prodotto*.
?? *Numero di copie vendute.*
?? *Dimostrazioni del prodotto che non siano il proprio uso quotidiano*.
?? *Prezzo*. Un prezzo elevato può semplicemente essere un prezzo esoso, e non è *necessariamente* indice di qualità sopraffina. Un prezzo particolarmente basso può non tradursi in un risparmio, bensì in un mare di guai successivi.

Infine va detto che chi acquista un programma dovrebbe prima avere le idee chiare su quello che vorrebbe avere dal programma, avendo bene in mente lo stato dell'arte e quali sono le reali possibilità oggi accessibili, nei vari settori. In caso contrario si è di fatto in balìa di affermazioni promozionali non necessariamente, e non sempre, fondate su fatti.

S.5.5.-Normativa e software

Un aspetto di estrema importanza, totalmente sottovalutato al momento nella discussione tecnico-scientifica, è quello che riguarda la creazione di programmi di calcolo che automatizzino regole di verifica prescritte dalle normative. Anche qui la credenza che tale attività debba seguire binari di rigida uniformità è purtroppo assai diffusa: nulla di più inesatto.

Se, con Nicola Zingarelli, definiamo "enunciato" qualsiasi sequenza finita di parole appartenenti a una lingua, possiamo senz'altro affermare che le norme sono fatte di enunciati. Non è tuttavia detto che un enunciato possa essere programmabile, o, se lo è, che possa esserlo in modo univoco.

Consideriamo il seguente enunciato (di fantasia, ma assai realistico come tipo di struttura): il numero di foglie verdi V diviso per il numero completo di foglie F si definisce indice di buona salute I = V/F.

Apparentemente questo enunciato è programmabile, in realtà non lo è affatto. Infatti, supponendo di saper spiegare ad un programma cosa è una foglia, cosa è una "foglia verde"? Quando una foglia è "verde"? Meglio ancora, quando un colore è "verde"?

Lo stesso enunciato per essere davvero programmabile dovrebbe avere per esempio le seguenti aggiunte: si dice "foglia verde" una foglia in cui almeno l'85% della superficie sia verde. Si dice "verde" un colore che, nel modello di colore RGB (red, green, blue) abbia R <= 15, G >= 230, B <= 15.

L'esempio rende ragione del differente punto di vista che l'esperto di programmazione ha rispetto al Normatore. Il Normatore a volte dà per scontati termini o definizioni che in realtà – dal punto di vista di chi programma – non lo sono affatto. Il programmatore è costretto a trasformare ogni decisione o informazione in rigorose ed univoche definizioni che siano formalmente inattaccabili e non può in alcun modo far conto su alcun senso comune. Il senso comune nella programmazione non esiste, e concetti assolutamente ovvi per un bambino (la foglia "verde"), diventano problemi anche spinosi per chi programma.

Un primo ordine di problemi riguarda la reticenza delle Norme ad affermare la propria incompletezza, autolimitando il proprio campo di applicazione a quello che consegue dall'analisi rigorosa degli enunciati. Così, nel non detto, è possibile dire e fare qualsiasi cosa. Il legittimo desiderio di generalizzazione del Normatore si scontra con il fatto che le campagne sperimentali sono spesso limitate a casi specifici, magari frequentissimi ed importantissimi, ma certo non esaustivi. Un classico esempio sono i coefficienti di distribuzione necessari per valutare il momento equivalente per la verifica a stabilità di colonne in acciaio presso-inflesse. Alcune norme non forniscono formule chiuse valide nel caso di una distribuzione di momento generica, ma solo alcune tabelle che si riferiscono ai casi più frequenti (EC3, AISC). Tali norme, nel contesto di software di calcolo generali, sarebbero a rigore non programmabili.

Un secondo ordine di problemi riguarda la ambiguità di certe definizioni, che possono essere interpretate in mille modi diversi e la norma tace. Ne sia un esempio la seguente affermazione presa dalle CNR-10022: "elemento compresso irrigidito collegato ad un'anima in corrispondenza di un bordo ed irrigidito in corrispondenza dell'altro bordo da una semplice piegatura, $b_0/t <= 60$". Peccato che non sia rigorosamente definito cosa sia un'anima. Il normatore aveva qui in mente le sezioni a C ed a I, o quelle rettangolari cave, per le quali cosa sia un'anima è evidente. Ma un programma di calcolo sui formati a freddo deve funzionare anche per profili di forma qualsiasi, ed in quel contesto il concetto di "anima" deve essere definito. Cosa distingue un'anima da un'ala

irrigidita da un punto di vista topologico? Niente. Ma forse la norma non era fatta per profili formati a freddo generici, bensì per profili a C, ad L, ad I, a Z.

Un terzo ordine di problemi riguarda il fatto di prospettare metodologie di calcolo estremamente complesse senza curarsi di tutte le conseguenze logiche che da queste derivano. Due esempi: 1) il caso dei profili sottili che si parzializzano, dando luogo ad un procedimento iterativo in cui – a rigore – ogni sezione efficace di ogni elemento, in ogni combinazione di carico cambia da iterazione ad iterazione, con rotazione degli assi principali della sezione efficace rispetto a quelli della sezione lorda, senza che venga chiarito come trattare rigorosamente il problema in termini ad esempio di verifiche a stabilità. 2) la classificazione delle sezioni nell'EC3, nel caso di flessione deviata o profilo privo di assi di simmetria. L'asse neutro plastico deve a rigore essere trovato con un procedimento iterativo ma di questo non c'è traccia nella norma, né sembra che la norma lo preveda.

Un quarto ordine di problemi riguarda l'assenza di riferimenti alle ricerche ed ai lavori che giustificano certi assunti della normativa: essi sono certo scontati per gli addetti ai lavori ma non lo sono per chi usa la norma. Anche l'assenza di esempi certificati da parte del Normatore consente ogni tipo di possibile interpretazione.

Sfortunatamente non sembra che questo genere di considerazioni faccia parte delle preoccupazioni tipiche di chi scrive le normative. Invero molte norme sono al momento attuale in questa situazione: per un verso esse sono assolutamente troppo complicate per poter essere soddisfatte per mezzo di calcoli "a mano"; per un altro verso esse sono scritte in modo tale da essere in varie zone (spesso peraltro fondamentali) totalmente non programmabili o programmabili solo a patto di affiancare ad esse – le normative –, un certo numero di enunciati aggiuntivi che di fatto crea chi scrive il programma, sulla base della propria esperienza e della propria cultura. Ne consegue che il programma che implementa le norme è sempre anche, contemporaneamente, un programma che rappresenta il *know how* e la cultura tecnica di chi lo ha realizzato, e che per di più rappresenta una delle possibili interpretazioni del dettato normativo, una specie di sovrainsieme dello stesso che, a rigore, chiunque potrebbe opinare. Inutile dire che l'arbitrio, che con tanta cura si era cercato di eliminare per mezzo di norme più stringenti e complesse, modelli sofisticati, ricerca teorica e sperimentale, nonostante ogni buona volontà da parte di chi sviluppa il software, è pronto a rientrare sotto forma di "programma automatico di verifica secondo la normativa XXX".

S.5.6.-Prassi e prospettive

A conclusione di quanto presentato si può dunque dire che in questi anni si è in una fase di transizione: da una parte non si può più fare a meno del software, dall'altra non ci si può ancora fidare di esso in modo completo, non soltanto per i problemi relativi ai possibili bug, ma anche perché spesso la semplicità d'uso dei programmi dà accesso al loro uso anche a persone che sono parzialmente (talvolta totalmente) prive delle conoscenze necessarie a dominarli. Il monito a conoscere le discipline oggetto di un certo programma si scontra con la necessità economica di disporre di elaborazioni sempre più complete e formalmente sofisticate da parte degli utenti, i quali sono spesso presi essenzialmente dalla necessità di fare le cose in fretta e senza alcuna fatica. Oggi si lavora verso il mito della automazione totale, e già qualcuno pretenderebbe di fornire procedure automatiche per le quali bastino pochi click per avere gli elaborati di progetto.

A ciò si aggiunge il fatto che le software house sono per lo più isolate nel loro prezioso lavoro di sviluppo, in quanto non si reputa comunemente che il lavoro da esse compiute contenga alcuno spunto di novità o di reale interesse. Abbiamo così una situazione frammentata e incoerente,

che vede il mondo professionale impiegare a dosi massicce i programmi per ottenere cospicue economie di scala; le software house lavorare a pieno ritmo per star dietro alle richieste e rendere praticabili le norme (di tutti i tipi: dalle norme di calcolo a quelle sulla sicurezza); le Università che in genere non reputano i problemi legati allo sviluppo software come degni di alcun interesse. A riprova basti considerare, ad esempio, che nessuna Università italiana si è fatta promotrice, negli ultimi venti anni, di alcun codice di calcolo di tipo generale in grado di competere con quelli fatti all'Estero.

In realtà le software house sono e saranno sempre più dei centri di ricerca e sviluppo di fondamentale importanza perché è essenzialmente grazie a loro che sono efficacemente impiegabili nella realtà economica i risultati della ricerca teorica e sperimentale. Occorrerà comprendere, anche in Italia, che i tre attori fondamentali, Ricercatori, Sviluppatori, Utilizzatori, devono trovare una qualche forma di collaborazione al fine di assicurare l'indispensabile sviluppo culturale, economico e tecnologico del nostro Paese.

S.5.7.-Bibliografia minima

[1] B. Beizer, *Software Testing Techniques*, International Thomson Computer Press, Second Edition, 1990.

[2] J. Holdsworth, *Software Process Design: Out of the Tar Pit*, Mc Graw Hill International Software Quality Assurance Series, 1994

[3] B. Stroustrup, *The C++ Programming Language*, Addison Wesley Publishing Company, Second Edition 1991.