Carlo Muzzì
Maurizio Caramellino

Programmazione Configurativa

Abstract
La "Programmazione Configurativa" o "Configurative Programming" è la proposta di un nuovo paradigma per lo sviluppo di software basato sull'idea che si possano realizzare applicazioni senza utilizzare i formalismi e gli strumenti tipici dei tradizionali linguaggi di programmazione, in generale dominio degli sviluppatori professionali, preferendo invece un approccio che prevede di configurare un ambiente software potenzialmente utilizzabile da chiunque sappia utilizzare un computer. Questo paradigma facilita le attività di sviluppo e di manutenzione del software perché favorisce la partecipazione diretta dei committenti alla produzione del software e semplifica la distribuzione e l'uso del codice realizzato.

www.configurativeprogramming.org

Copyright © Carlo Muzzì, Maurizio Caramellino 2003-2009. Tutti i diritti riservati.
Ultima revisione significativa Agosto 2007. Stampato digitalmente in Italia.
Questo documento è protetto dalle normative internazionali sul diritto d'autore. La riproduzione, parziale o totale, è permessa solo specificando il titolo e gli autori. Tutti i marchi i marchi o copyrights o trademarks citati nel testo sono di proprietà dei loro rispettivi detentori.

Le informazioni riportate o espresse in questo documento, così come le teorie proposte, sono state oggetto di verifiche accurate; gli autori rimangono comunque disponibili a correggere eventuali imprecisioni. Tutto quanto contenuto in questo documento è fornito come è e nello stato in cui è senza alcun tipo di garanzia. Utilizzando in qualunque modo le informazioni ed i materiali riportati si accettano i rischi e le responsabilità per perdite, danneggiamenti, costi e altre conseguenze che, direttamente o indirettamente, dovessero derivare. Gli autori declinano ogni responsabilità (anche derivanti da negligenza) per tutte le conseguenze di qualsiasi persona che agisca, o che si sia astenuta dal compiere qualunque azione, facendo affidamento su quanto contenuto in questo documento.

URL: http://www.configurativeprogramming.org
E-mail: authors@configurativeprogramming.org

Introduzione

Nella prima sezione di questa trattazione percorriamo rapidamente l'evoluzione dei linguaggi di programmazione, evidenziando i miglioramenti significativi ottenuti ad ogni singolo passo del processo evolutivo e mettendo a confronto i linguaggi con le differenti metodologie proposte per il governo del ciclo di vita del software. Sulla base di quest'analisi esponiamo, nella seconda sezione, le motivazioni che ci hanno condotto a definire questo nuovo paradigma di programmazione.
Nella terza sezione presentiamo il modello assiomatico proposto come base teorica di questo nuovo paradigma, discutendone i principi ed analizzando i miglioramenti introdotti nella produzione del software. Nella quarta sezione trattiamo di come tale paradigma potenzialmente consente di definire diversi linguaggi di programmazione, permettendo anche di estendere la comunità degli sviluppatori, in particolare ragioneremo sul perchè questi linguaggi renderanno più semplici e rapide le attività di sviluppo e la manutenzione del software.
La quinta sezione conclude la trattazione analizzando il primo linguaggio di programmazione definito in base a questo nuovo modello, illustrando, con particolare enfasi, le scelte che hanno condotto alla definizione degli elementi più significativi.

I - Il processo evolutivo dei linguaggi di programmazione

La programmazione costituisce da sempre una delle principali aree di studio della computer science; negli ultimi ottanta anni i differenti approcci teorici utilizzati hanno condotto alla definizione di paradigmi e linguaggi di programmazione differenti.

Inizialmente, quando ancora non esistevano i computer così come ci sono noti oggi, la comunità scientifica era prevalentemente orientata alla determinazione dei principi di base della scienza dell'informazione. Siccome in quel periodo gli aspetti filosofici della materia iniziavano ad essere inquadrati all'interno di modelli di riferimento matematici, era naturale che anche gli approcci alla programmazione fossero anch'essi orientati a modelli di tipo prevalentemente matematico.

Le *macchine di Turing*, proposte[1] da A.M. Turing, costituivano proprio il classico esempio di macchina (o computer) ideale in grado di eseguire diversi algoritmi ma, tipicamente, all'interno di un ambito di applicazione prevalentemente matematico[i]; infatti, una macchina di Turing utilizzava dei sistemi numerici per rappresentare le informazioni trattate, lo stato assunto dopo ogni singola attività svolta e le istruzioni che poteva eseguire.
Per lo scopo di questa trattazione non è necessario approfondire le basi della teoria della computazione, ma è invece rilevante considerare una importante conseguenza di queste macchine ideali: Turing indicò che era possibile inventare una singola macchina più generale, che chiamò *universal computing machine*, in grado di simulare le attività di una qualunque altra macchina di Turing.

L'idea che si poteva programmare un calcolatore pensando di lavorare su un altro calcolatore introduceva quel concetto di astrazione che permise di evitare l'utilizzo dei linguaggi macchina, detti anche di basso livello, per sostituirli con linguaggi formali di livello più alto. È noto che i primi calcolatori erano programmati producendo codice direttamente in binario (il codice era scritto con i cosiddetti *linguaggi di prima generazione* o *1GL*) gli sviluppi successivi portavano ad utilizzare linguaggi assemblativi di tipo simbolico (definiti *linguaggi di seconda generazione* o *2GL*) in cui le stringhe binarie venivano sostituite da corrispondenti simboli testuali: come esempi di linguaggi simbolici di tipo assembly possiamo citare *IBM BAL* e *VAX Macro*.

L'utilizzo dei linguaggi simbolici induceva però alcuni sostanziali problemi: i programmi erano sempre scritti per la famiglia di calcolatori che doveva eseguirli, quindi erano dipendenti da uno specifico hardware; inoltre il programmatore lavorava con processi mentali ostici che erano difformi da quelli tipicamente usati dagli esseri umani.

Nel periodo che intercorse tra gli anni '50 e gli anni '60 venne profuso parecchio impegno nel cercare di individuare delle soluzioni a tali problemi; questi sforzi si concretizzarono nella definizione di diversi linguaggi di programmazione, ognuno maggiormente rivolto a specifiche aree applicative del mondo reale, che avevano in comune un lessico ed una sintassi più vicina al livello umano: per questa ragione vennero definiti come *linguaggi di alto livello*[ii].

Questi linguaggi (noti anche come 3*GL* o *linguaggi di terza generazione*) utilizzavano tipicamente parole chiave e codici derivati dalla lingua inglese; in tale modo i codici sorgente erano più facilmente leggibili e comprensibili da un programmatore umano ma naturalmente non potevano essere compresi da un computer senza una traduzione nel suo

---

i Secondo una congettura di A. Church, nota come Tesi di Church, la classe dei problemi che potevano essere modellizzati e risolti con una macchina di Turing corrispondeva alla classe delle funzioni computabili; in effetti lo stesso Turing dimostrò come non fosse risolubile il problema dell'arresto: una macchina di Toring potrebbe non essere in grado di rispondere se un'altra macchina si arresterà o meno.

ii Nel seguito di questa trattazione vedremo varie classificazioni di alcuni linguaggi di programmazione; siccome anche i linguaggi, come gli altri software, subiscono periodiche revisioni ed aggiornamenti, potrebbe accadere che versioni diverse di un medesimo linguaggio appartengano a famiglie diverse.

specifico linguaggio macchina che doveva avvenire preventivamente o in fase di esecuzione: inizialmente la traduzione veniva realizzata ogni volta che si eseguiva un programma tramite uno specifico componente, detto *interprete*, che in fase di esecuzione traduceva le istruzioni di alto livello in codice macchina; successivamente, la necessità di velocizzare la fase di esecuzione e migliorare il processo di astrazione, conduceva alla definizione del concetto di *compilatore*: veniva introdotto uno strumento dedicato che permetteva una traduzione che veniva eseguita, in modo ottimizzato, una sola volta prima dell'esecuzione.

I linguaggi di alto livello sono anche noti come orientati al compito (*task-oriented*) in quanto definiti con riferimento all'ambiente in cui i programmatori avrebbero dovuto operare; da questo approccio derivò una classificazione dei linguaggi in tre grandi famiglie ancora utilizzata oggi:

- i *linguaggi imperativi* in cui scrivere del codice per istruire il computer su "cosa fare" e "cosa ottenere"; il programma è costituito da una sequenza di istruzioni da eseguire in ordine prestabilito, mediante istruzioni di flusso che mutano il contenuto della memoria del computer (assegnamento di valori a variabili, passaggio di parametri). Appartengono a questa classe i linguaggi storici più comuni, quali l'ALGOL ed il FORTAN dedicati al calcolo scientifico e particolarmente utilizzati da matematici, fisici, astronomi; il COBOL destinato ad applicazioni di tipo gestionale; il Pascal, il BASIC e il Modula-2 utilizzati in ambito education; il C per i sistemi operativi ed i giochi; l'Ada utilizzato per applicazioni militari, mission critical, sistemi embedded, comunicazione, CAD e finance;
- i *linguaggi funzionali* usati per il calcolo funzionale puro; basati sull'uso di funzioni che chiamano se stesse (ricorsione) o altre funzioni. Non utilizzano il tradizionale concetto di assegnazione di valori a variabili perché la struttura dati prevalente è la lista. Il linguaggio più noto appartenente a questa classe è il LISP, particolarmente utilizzato nell'intelligenza artificiale, nelle applicazioni militari e nel mondo education, proprio per la sua caratteristica peculiare: quella di permettere la diretta implementazione in un programma del modello computazionale del λ-calcolo di Church[2].
- i *linguaggi dichiarativi* detti anche *logici* perchè al programma è richiesta la dimostrazione della verità di una asserzione; il sorgente è costituito da una serie di asserzioni di fatti e regole senza che sia necessario specificare a priori il flusso di esecuzione, sarà il programma a cercarlo in base all'obiettivo che gli viene indicato. Questi linguaggi non hanno un ambito specifico di applicazione, ma sono particolarmente validi per risolvere problemi che riguardano entità e relazioni; il PROLOG è il più noto linguaggio dichiarativo.

Ai fini della nostra trattazione è importante considerare il retroterra culturale nel quale si sviluppò la programmazione del tempo; in quel periodo la necessità prevalente era disporre di formalismi che permettessero di condividere facilmente tra programmatori diversi quanto realizzato da un singolo componente; era quindi necessario semplificare l'attività di manutenzione del codice e garantire la conservazione del know how: l'esigenza di rendere meno soggettiva l'attività di progettazione e sviluppo del software condusse alla definizione di specifiche metodologie di lavoro.

Siccome sul finire degli anni '60 lo sviluppo del software iniziò ad essere considerato un'attività quasi industriale, la necessità di far cooperare diversi programmatori sul medesimo progetto venne affrontata proprio con metodologie mutuate da quelle al tempo in voga nell'ambito industriale. In particolare le nuove metodologie di sviluppo introdotte erano basate sul concetto di *modulo*; il processo di sviluppo del software partiva dall'approccio di scomporre il problema iniziale in più parti o moduli che, pur venendo sviluppati indipendentemente l'uno dall'altro, potevano alla fine ricomporsi in un solo prodotto finale. Questo modello di sviluppo del software venne definito "a cascata" (*waterfall model*)[3] e, nell'ambito della teoria dei linguaggi, introdusse i concetti del data hiding, dell'incapsulamento e della programmazione strutturata.

L'obiettivo della programmazione strutturata era semplificare la scrittura del codice obbligando il programmatore ad utilizzare poche strutture di controllo che garantivano un solo punto d'ingresso ed uscita; questi obiettivi venivano raggiunti fornendo al programmatore linguaggi che implementando i principi del teorema di Böhm-Jacopini[4] permettevano di evitare l'uso deleterio del salto incondizionato (*goto*)[5] da cui derivava il dannoso fenomeno dello *spaghetti code*. ALGOL, Pascal, Modula-32, C, Ada sono alcuni esempi di linguaggi strutturati.

Un aspetto saliente di questo nuovo approccio alla programmazione si riscontrava nella minore importanza che veniva assegnata al linguaggio di programmazione usato[6] a vantaggio di una maggiore attenzione rivolta alle strutture dei dati ed agli algoritmi utilizzati nell'automatizzare[7] un problema; i dati e gli algoritmi potevano essere considerati in maniera indipendente dai formalismi che ogni specifico linguaggio utilizzava per implementarle e quindi i programmatori che avessero rivolto particolare attenzione ad esse avrebbero prodotto un codice migliore.

La programmazione strutturata forniva dunque un modello teorico che doveva condurre alla realizzazione di codice robusto; ma l'esperienza reale dimostrò che il software sviluppato non soddisfaceva mai le aspettative degli utenti. Fino alla metà degli anni '80 l'opinione comune era che la causa di tale imprecisione derivasse da errori commessi in una

delle varie fasi[iii] di realizzazione del software; quindi maggiore enfasi venne rivolta alla metodologia di sviluppo adottata.
Siccome il modello a cascata utilizzava un approccio in più fasi e prevedeva di passare alla fase successiva dopo che era stata conclusa e validata la precedente, era ovvio che un errore nella fase *i* causava errori in tutte le fasi dalla *i+1* in poi. Si pensò quindi di utilizzare il *modello a V*[8] (*V model*) in cui le macro-attività di analisi, progetto e sviluppo venivano prima affrontate dal generale al particolare (in senso discendente) e poi testate dal particolare al generale (in senso ascendente): in questo modo si sperava di ridurre gli errori perché le attività vicine agli utenti, ritenute più a rischio di errore, venivano testate prima.

Similmente ai processi metodologici anche i linguaggi di programmazione subirono una nuova fase evolutiva: l'esigenza di permettere l'utilizzo di istruzioni con formalismi più simili al linguaggio umano portò alla creazione dei *linguaggi di quarta generazione* o *4GL*: particolarmente utili negli scripting e nei linguaggi di interazione con i database.

Nonostante gli interventi correttivi su linguaggi e metodologie, l'esperienza reale continuò a dimostrare la persistenza del fenomeno della non conformità del software alle attese degli utilizzatori; ciò spinse la comunità scientifica ad accettare l'idea che era impossibile realizzare codice che soddisfacesse pienamente le attese degli utenti: per molti divenne chiaro che il processo di sviluppo era un'attività intrinsecamente incerta. Per far fronte a questa nuova consapevolezza si pensò di far evolvere nuovamente le metodologie, proponendo nuovi modelli di riferimento, ognuno in grado di affrontare l'incertezza in base alle specifiche necessità del dominio applicativo che richiedeva il software.

Il *modello incrementale* (*incremental delivery*) con le sue evoluzioni[9] costituì un valido approccio per quelle realtà che richiedevano di ricevere prima possibile una parte delle soluzioni attese: il software veniva realizzato e consegnato a singoli pezzi. Invece, per le realtà in cui l'incertezza tecnologica conviveva con l'incertezza dei requisiti, venne proposto il *modello a prototipi*[10] (*prototyping model*) basato sulla creazione preventiva di un prototipo della soluzione finale su cui approfondire discussioni e analisi.

I modelli proposti dalla letteratura, da quello a cascata a quello a prototipo, pur articolando in modo diverso le differenti fasi del processo di sviluppo utilizzavano tutti il medesimo approccio sequenziale; esistevano però degli ambiti applicativi (multimedialità, sistemi operativi, modelli di simulazione di fenomeni naturali, controllo di processi di produzione, ecc.) in cui non era possibile seguire un approccio sequenziale. Per questi casi venne proposto il *modello a spirale*[11] (*spiral model*) che partiva dal presupposto che tutti gli elementi rilevanti del progetto venissero portati avanti in parallelo fino ad un specifico momento (definito *milestone*) in cui tutti gli attori coinvolti nel progetto dovevano analizzare quanto realizzato per attuare delle correzioni; il processo di sviluppo era da intendersi ciclico perché dopo il primo milestone tutte le attività dovevano nuovamente ripartire, sempre in parallelo, fino al nuovo punto di milestone. Questo modello venne detto a spirale perché ad ogni ciclo il margine di errore tendeva a restringersi fino a convergere verso una soluzione ottimale.

Dal punto di vista della teoria dei linguaggi questi modelli portarono alla definizione del paradigma di *programmazione orientato agli oggetti*. L'approccio adottato da questo paradigma[12] consisteva nel modellizzare la realtà con delle entità (chiamate classi) contenenti sia le strutture dati sia le procedure (chiamate *metodi*) che vi operavano sopra; quando un programma creava un elemento di una classe si diceva che istanziava un oggetto ed oggetti diversi potevano comunicare tra essi tramite messaggi. Le basi di questa teoria derivavano dagli studi di Dahl e Nygaard sul linguaggio SIMULA[13] e condussero alla diffusione di diversi linguaggi object oriented quali: Smalltalk, C++, SQLWindows.

La programmazione ad oggetti è stata in grado di produrre un sostanziale miglioramento nell'attività di sviluppo del codice, ma tale miglioramento è avvenuto con un aumento del livello di complessità del codice stesso; siccome ogni singola parte del problema andava definita in modo preciso, era necessario padroneggiare linguaggi con molte opportunità. Trovare un numero sufficiente di programmatori in grado di utilizzare proficuamente questi linguaggi ha costituito da sempre il maggior ostacolo alla diffusione del modello ad oggetti. Con il nuovo millennio la complessità dei problemi da affrontare aumentò a seguito dei nuovi requisiti determinati dal consolidamento del paradigma del web: sostanzialmente al codice era ora richiesto di poter operare muovendosi su una rete mondiale, che doveva integrare ambienti e hardware operativi eterogenei e che non poteva garantire la velocità e l'affidabilità di connessione tipiche degli ambienti LAN.
Per risolvere queste problematiche venne proposta una nuova classe di linguaggi definiti *linguaggi per lo sviluppo dinamico* di software cui vennero affiancate delle nuove metodologie di sviluppo note come *metodologie agili*.

L'introduzione di Java costituì un tipico esempio di linguaggio in grado di fornire supporto a queste esigenze. In particolare, la necessità di produrre software che potesse operare su piattaforme hardware eterogenee veniva raggiunta

[iii] L'approccio tipico prevedeva che le macro fasi di analisi, progetto e sviluppo fossero ulteriormente suddivise in: studio di fattibilità, raccolta dei requisiti, analisi, progettazione, sviluppo, testing e messa in esercizio.

enfatizzando il principio di separazione tra il codice e l'hardware che lo eseguiva mediante l'introduzione del concetto di macchina virtuale (*virtual machine*): su ogni tipo di hardware era ovviamente necessaria una particolare versione della macchina virtuale ma, una volta presente, hardware diversi potevano eseguire un medesimo codice.
La tecnica utilizzata richiedeva che ogni classe di un programma Java fosse compilata singolarmente producendo un bytecode; in fase di esecuzione, quindi a run-time, la macchina virtuale avrebbe caricato i *bytecode* necessari e, se richiesto, li avrebbe eseguiti anche su computer diversi: questa tecnica risultò estremamente utile sia nelle architetture a più livelli tipiche degli ambienti corporate sia nelle soluzioni che utilizzano il web[iv] come modello di riferimento.

Più recentemente l'obiettivo di ridurre ulteriormente il livello di complessità dei linguaggi ha condotto all'introduzione del concetto di *Domain Specific Languages*[14] (*DSL*): per questo nuovo concetto sono state proposte differenti definizioni, generalmente accomunate dall'idea che i DSL siano linguaggi espressamente dedicati alla risoluzione di particolari problemi di specifici domini applicativi.
Lo scopo primario dei DSL mira a ridurre la distanza esistente tra il problema ed il codice e ciò è avvenuto semplificando la struttura del linguaggio che tende a modellarsi su una specifica classe di problemi. In talune circostanze la specializzazione del linguaggio ha permesso di demandarne la gestione direttamente agli esperti del dominio applicativo; per questa ragione questi linguaggi sono divenuti noti come linguaggi per non-programmatori o *end-user languages*: il linguaggio macro dei fogli elettronici, l'HTML, le specifiche sintattiche tramite BNF, l'SQL ne costituiscono esempi.

L'obiettivo di una maggiore dinamicità è stato perseguito anche per il processo di sviluppo del software, che è stato reso più semplice e leggero, introducendo nuovi modelli metodologici definiti *agili*[15] (*agile methodologies* o *lightweight methodologies*) il cui scopo era fornire modelli di riferimento che costituivano un ragionevole compromesso tra sviluppare codice senza usare metodologie di sviluppo (con il rischio del caos e dell'assenza di risultati) e sviluppare codice utilizzando metodologie estremamente rigide e pesanti (con il rischio di allontanare il software prodotto dalle attese effettive dei clienti).

Queste metodologie[16] hanno accettato il fatto che la realtà è in continua evoluzione pertanto ogni problema deve essere affrontato con un approccio adattativo e non predittivo: è necessario adattarsi dinamicamente alla realtà mentre essa muta e non cercare di pianificare tutto quello che potrebbe accadere. Inoltre la metodologia agile è orientata verso il team di sviluppo e non verso il processo; se il focus delle metodologie tradizionali mirava a definire processi senza tener conto del team di sviluppo che li avrebbe automatizzati, con la metodologia agile è espressamente richiesto di supportare l'attività del team di sviluppo. In particolare la capacità di sviluppare progressivamente il codice, testandolo continuamente ed eventualmente ripensando eventuali scelte fatte, è strettamente connessa alla capacità adattative del team di sviluppo che dovrà contenere non solo programmatori e gli esperti di information technologies, ma anche i committenti, ossia i clienti del software richiesto.

Tra le diverse metodologie agili proposte, la più nota è stata la *Estreme Programming*[17](*XP*) di particolare interesse per la grande enfasi posta verso il concetto del test. Come noto l'attività di test è sempre stata una fase prevista da tutti i processi di sviluppo del software, ma nella XP il test assume un ruolo cruciale: il programmatore dovrà scrivere i test che verificheranno il codice prima ancora di scrivere il codice stesso. Questo approccio, fortemente orientato al test, è divenuto generalmente noto come *test-first*.

Il tentativo di semplificare l'attività di sviluppo del software è stato anche l'obiettivo perseguito dai linguaggi generalisti (*general purpose languages* o *GPL)*, prevalentemente tramite l'adozione di strumenti di tipo visuale o grafico. Al programmatore questi linguaggi hanno fornito un ambiente integrato di sviluppo (IDE) di tipo visuale che permetteva di configurare in modo rapido le interfacce grafiche degli utenti (form, griglie, text box, button, ecc.) integrandole con il codice scritto dal programmatore. Questi linguaggi hanno semplificato il problema della gestione delle gerarchie delle classi orientate agli oggetti fornendo degli appositi strumenti visuali; hanno permesso di sviluppare codice assemblando ed adattando delle componenti software semi-lavorate acquistabili sul mercato o reperibili nel mondo open-source; hanno facilitato l'accesso alla documentazione ed all'help fornendo meccanismi che facilitano la creazione e l'utilizzo di risorse disponibili sul web. Le successive evoluzioni di Visual Basic, SQL Windows, Delphi, sono stati esempi di questi linguaggi.

Una più ampia diffusione di queste soluzioni si è avuta con l'introduzione del *Microsoft .NET Framework*; un componente aggiuntivo al sistema operativo che ha permesso di disporre di una piattaforma integrata per lo sviluppo di applicazioni orientate ai servizi in grado di operare anche in ambienti operativi in cui il web[18] riveste un ruolo primario. Di particolare interesse è stata l'introduzione, anche in questo contesto, del concetto della macchina virtuale già discusso per Java, implementato mediante l'utilizzo del *Common Language Runtime* (CLR).
Il .NET Framework ha permesso anche di astrarre il linguaggio utilizzato per lo sviluppo: un programmatore ha potuto utilizzare un linguaggio a lui già noto, scelto tra quelli aderenti alle specifiche del .NET Framework, per scrivere codice

iv L'applet è un tipico esempio di una parte di codice che un server web invia a dei client per un'esecuzione locale.

e produrre la versione compilata per il CLR. C#, Visual Basic .NET, J#, ASP.NET, Delphi 8, DotLisp, sono tutti esempi di tali linguaggi.

Anche se il termine[v] è utilizzato in modo non uniforme, i linguaggi di ultima generazione vengono talvolta indicati come 5*GL* o *linguaggi di quinta generazione*.

II - La nostra opinione intorno all'opportunità di un nuovo paradigma di programmazione

Nella sezione precedente abbiamo brevemente riassunto i passi evolutivi percorsi dal software; quanto descritto evidenzia come il processo di sviluppo sia stato costantemente influenzato dall'esigenza di *semplificare* l'attività di realizzazione e mantenimento del codice.

Questa esigenza è stata perseguita attraverso la definizione di vari paradigmi da cui sono scaturite differenti metodologie di sviluppo e molteplici linguaggi di programmazione. La mutua relazione esistente tra metodologie e linguaggi è da tempo oggetto di attenzione; particolarmente se consideriamo la metodologia come l'implementazione di uno specifico modello tra quelli proposti per migliorare i processi di sviluppo del software.
Anche se vi sono casi in cui i linguaggi hanno anticipato una metodologia, usualmente i tempi di evoluzione delle metodologie sono più rapidi dei linguaggi semplicemente perché studiate da più persone; infatti se i linguaggi di programmazione sono interesse di relativamente pochi individui (un sottoinsieme di quanti interessati alla computer science), le metodologie che migliorano il processo di sviluppo sono oggetto di ricerche più ampie perché derivano dall'applicazione al software delle più generali teorie del management sul miglioramento del processo di produzione.

Questa relazione tra processo e linguaggio si può esprimere anche in una misura del grado di complessità; la figura 1[19] riporta l'evoluzione della complessità in rapporto al linguaggio e alla metodologia.

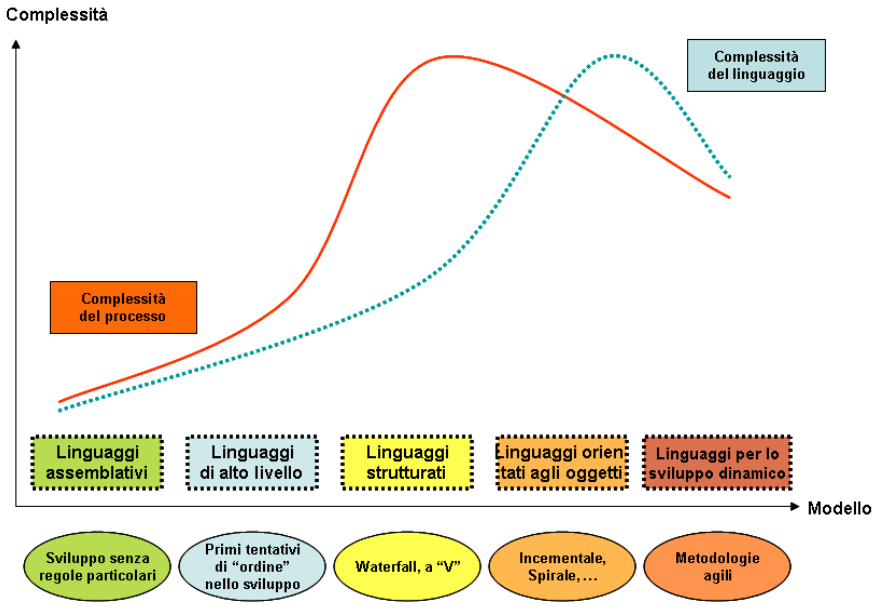

Fig. 1 – evoluzione della complessità in rapporto al linguaggio e alla metodologia

L'andamento del grafico evidenzia quanto già discusso; la semplificazione non solo costituisce una esigenza comunemente avvertita ma il percorso su tale via è già stato intrapreso; in effetti gli ultimi modelli proposti hanno già portato alla definizione di metodologie e linguaggi tendenti a ridurre il problema della complessità.

Riteniamo però che un ulteriore passo verso la riduzione della complessità possa avvenire solo introducendo nuovi modelli di riferimento che permettano di definire metodologie e linguaggi effettivamente più semplici; ma quali possono essere le leve su cui agire per compiere questo nuovo balzo evolutivo verso l'aumento della semplicità?

Crediamo che una risposta a questa domanda possa essere trovata ripercorrendo il processo di evoluzione dei linguaggi e dei modelli discusso nella prima parte di questa trattazione; in particolare rivolgendo maggiore attenzione ai linguaggi piuttosto che alle metodologie di sviluppo. Difatti l'esperienza ci induce ad affermare che un valido supporto metodologico riesca ad esaltare l'efficacia di un linguaggio, ma nessuna metodologia, per quanto valida, potrà far evolvere significativamente le potenzialità di uno specifico linguaggio di programmazione verso un livello superiore.

v Alcuni autori considerano queste caratteristiche già soddisfatte dai 4GL, intendendo i 5GL come l'evoluzione dei 4GL verso l'utilizzo delle basi di conoscenza.

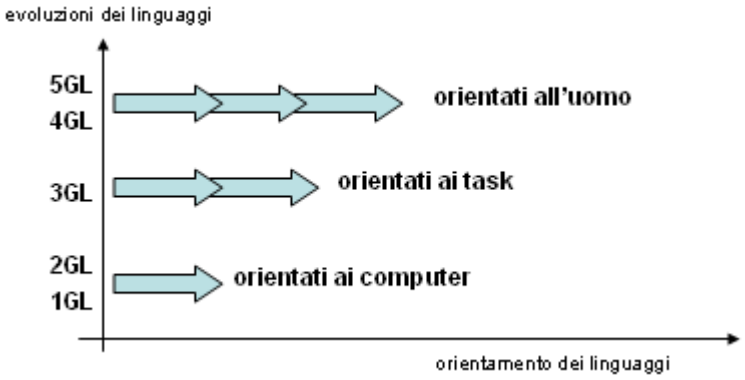

Fig. 2 – spostamento del focus dei linguaggi durante il loro processo evolutivo

Lo figura 2 rammenta brevemente come alla base dell'evoluzione dei linguaggi vi sia stato lo spostamento del loro orientamento; se i primi linguaggi erano prevalentemente orientati alla macchina fisica, le generazioni successive hanno cercato di astrarsi dai calcolatori per avvicinarsi sempre più alle esigenze umane. Se però analizziamo con più dettaglio quanto discusso in precedenza sui cosiddetti linguaggi human-oriented, constatiamo come essi non abbiano cercato di avvicinarsi alla generalità degli esseri umani bensì alle esigenze di quel ristretto gruppo di individui che opera nel tradizionale ambito della computer science.

In effetti, pur condividendo gli obiettivi che alcuni DSL hanno cercato di perseguire nel tentativo di avvicinare gli utenti finali alle problematiche della programmazione, rileviamo che questo sia avvenuto a scapito di una eccessiva riduzione delle potenzialità generali dei linguaggi che sono stati confinati in specifici domini applicativi: tanto che il numero dei potenziali utilizzatori più che crescere si è ridotto ulteriormente confinando i linguaggi al mondo degli esperti di un singolo dominio.

Queste considerazioni ci permettono di affermare che *l'obiettivo della semplificazione del linguaggio si può raggiungere solo definendo dei formalismi che permettono di aumentare il numero di individui in grado di creare nuove applicazioni software.*

È ovvio che si raggiunge questo obiettivo se si è in grado di rispondere alla seguente domanda: "Quali caratteristiche deve avere un linguaggio di programmazione per essere definito semplice?". Per trovare una risposta a questa domanda ci è di nuovo utile esaminare le caratteristiche salienti delle diverse generazioni di linguaggi di programmazione succedutesi finora.

Innanzi tutto non crediamo che la semplificazione può essere raggiunta utilizzando formalismi che forniscono meccanismi evoluti di rappresentazione della realtà congiunta ad un ricco insieme di strumenti e comandi per manipolarla: la storia dei linguaggi object-oriented ha chiaramente dimostrato la validità di questa via ma purtroppo solo dei programmatori molto esperti riescono a percorrerla.
D'altro canto neanche l'utilizzo di formalismi logici astratti permette di raggiungere l'obiettivo: la stessa macchina di Turing, come altri approcci[20] alla teoria della computabilità, hanno dimostrato che un insieme estremamente ridotto[vi] di strutture di dati e istruzioni potrebbe essere diffuso verso un gruppo più vasto di individui, ma il loro utilizzo difficilmente permetterebbe di realizzare i software usualmente presenti sul mercato.
Ugualmente poco fruttuosa, ai nostri fini, è anche stata l'esperienza di far partecipare gli esperti dei domini applicativi al processo di definizione dei linguaggi. Nel 1959 la prima versione del linguaggio COBOL[21] venne proprio definita da una comitato - "The Short Range Committee" – sorto su iniziativa del governo degli Stati Uniti e composto dai leader dei primari organismi pubblici e privati dell'epoca; il risultato è stato un linguaggio di enorme successo perché supporta in modo estremamente valido le esigenze finanziarie, amministrative e del business in genere (del resto COBOL è l'acronimo di "COmmon Business Oriented Language"), ma il cui utilizzo è sempre stato dominio dei programmatori.

Dunque l'esperienza passata dimostra come il tentativo della semplificazione dei linguaggi possa essere perseguito solo discostandosi dagli approcci tradizionali; la nostra opinione è che la *programmazione configurativa* costituisca un nuovo e significativo approccio per raggiungere questo obiettivo.

L'informatica utilizza il verbo *configurare* in diverse circostanze, il *Cambridge Advanced Learner's Dictonary* lo definisce come "to adjust something or change the controls on a computer or other device so that it can be used in a particular way"; nella nostra visione esso permette di attribuire un significato più ampio al termine "programmare". Riteniamo che l'espressione "programmazione configurativa" possa esprimere un nuovo modello di sviluppo del software basato su questa idea: *è possibile automatizzare una certa attività programmando un calcolatore, in maniera indiretta, tramite un ambiente di esecuzione presente sul calcolatore stesso che viene configurato utilizzando opportunamente un insieme predefinito di strumenti elementari e di regole che ne determinano il corretto utilizzo.*

---

vi In particolare ci riferiamo a quella macchina ideale, definita Unlimited Register Machine (URM), derivata dagli studi di Stepherdson e Sturgis; questa macchina è estremamente semplice perché utilizza solo quattro istruzioni ed un solo tipo di dato costituito da una successione infinita di registri.

Nella sezione successiva definiremo meglio questo nuovo modello di programmazione esponendo un insieme di assiomi che permettono di definirlo; come poi questo nuovo approccio possa essere utilizzato per estendere la comunità degli sviluppatori sarà oggetto di discussione della quarta sezione di questa trattazione.

III – La Programmazione Configurativa

La "*Programmazione Configurativa*" o "*Configurative Programming*" è un nuovo paradigma per lo sviluppo di software che non prevede l'utilizzo dei formalismi e degli strumenti tipici dei tradizionali linguaggi di programmazione, ma adotta un nuovo modello di sviluppo basato sul principio della configurativitá.

Gli assiomi della programmazione configurativa sono i seguenti:

1) definiamo "*Applicazione Configurata*" o "*Configured Application*" un qualunque software realizzato attraverso il modello della programmazione configurativa

2) definiamo "*Ambiente di Esecuzione*" o "*Execution Environment*" l'infrastruttura dedicata all'esecuzione di una generica applicazione configurata

3) l'ambiente di esecuzione deve permettere lo sviluppo, la distribuzione, l'esecuzione e l'interazione di applicazioni configurate sia su computer stand-alone e sia su architetture a più livelli composte da calcolatori in vario modo connessi in reti pubbliche o private

4) un'applicazione configurata è sviluppata mediante l'allocazione e la configurazione degli elementi posti a disposizione dall'ambiente di esecuzione

5) gli elementi forniti dall'ambiente di esecuzione possono essere aggregati, anche ripetutamente, per costituire elementi più complessi

6) gli elementi forniti dall'ambiente di esecuzione possono essere configurati nelle proprie caratteristiche, nelle azioni che possono svolgere e negli eventi cui saranno soggetti durante il proprio ciclo di vita

7) definiamo "*Fonte Dati*" o "*Data Source*" la struttura dati (es. file, database, basi di conoscenza) contenente le informazioni che un'applicazione configurata può manipolare

8) l'insieme delle configurazioni che costituiscono un'applicazione configurata sono conservate in una fonte dati specifica che definiamo "*Libreria delle Configurazioni*" o "*Configurations Library*"

9) un'applicazione configurata può accedere a fonti dati diverse e le manipola tramite gli elementi posti a disposizione dall'ambiente di esecuzione

10) gli strumenti messi a disposizione dall'ambiente di esecuzione per manipolare le fonti dati devono essere indipendenti da una particolare implementazione della fonte dati

11) l'ambiente di esecuzione permette di manipolare la libreria di configurazione allo stesso modo in cui manipola le altre fonti dati

12) l'ambiente di esecuzione deve esporre elementi dedicati all'interazione tra una applicazione configurata ed un'altra e tra un'applicazione configurata e software realizzati mediante linguaggi di programmazione tradizionali

Dall'enfasi che i vari assiomi pongono sul concetto di configurazione è derivata l'idea di definire questo nuovo modello di programmazione come "Programmazione Configurativa"; procediamo quindi, col discutere alcuni rilevanti concetti che derivano dall'analisi degli assiomi stessi.

Iniziamo con l'evidenziare che questo modello permette di sviluppare il software mediante la configurazione degli elementi posti a disposizione dell'ambiente di esecuzione; siccome è l'ambiente di esecuzione che permette al programmatore di comporre delle istruzioni che automatizzano una certa attività, è possibile assimilare il ruolo di questo modello a quello di un linguaggio di programmazione. In realtà l'esatto ruolo di questo modello è più ampio; per comprenderlo osserviamo la figura 3 richiamando alcuni concetti già discussi in precedenza.

Innanzitutto notiamo che questo ambiente di esecuzione ha la capacità di eseguire a run-time dei comandi ricevuti da chi lo programma: è quindi possibile considerarlo un tipo particolare di interprete.
Il fatto che la configurazione avvenga su elementi forniti da uno strato software presente sul calcolatore, una sorta di middleware che accanto ai meccanismi di accesso alle fonti dati (database, file system, ecc.) espone anche dei comandi per manipolarli, ci permette di pensare che abbia anche una finalità analoga al CRL di Microsoft.

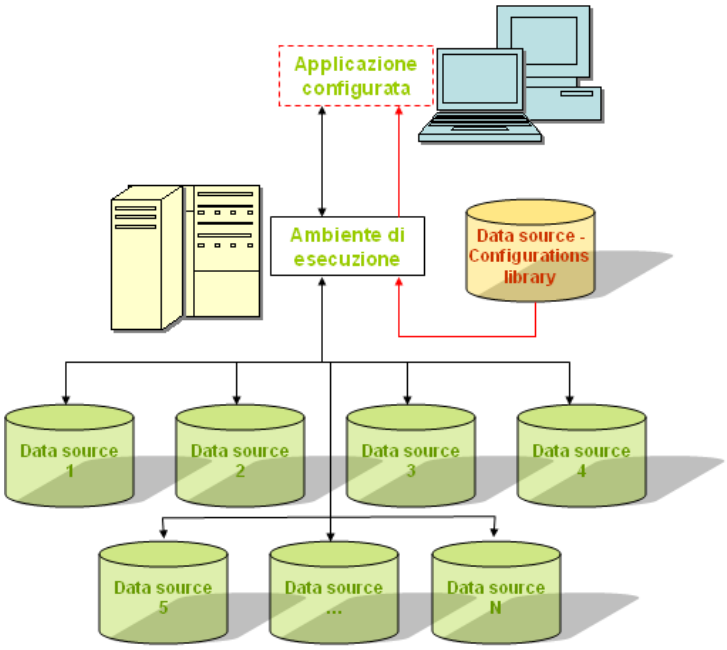

Fig. 3 – descrizione del modello configurativo

Inoltre l'ambiente di esecuzione può essere visto da coloro che lo programmano come una macchina ideale con finalità simili alla virtual machine di Java; infatti è possibile che un'applicazione realizzata secondo questo modello di sviluppo funzioni su calcolatori di famiglie diverse se ogni calcolatore disporrà di versioni dedicate dell'ambiente di esecuzione che però offrono al programmatore lo stesso ambiente da configurare.

Possiamo dunque ritenere come queste considerazioni siano tutte valide per un sistema conforme ai principi della programmazione configurativa, quindi è corretto considerarlo un *modello chiuso* che racchiude:

1. gli elementi da configurare e le regole per manipolarli
2. i meccanismi che permettono di sviluppare il software
3. l'ambiente che esegue il software

Possiamo quindi considerare questo sistema come una macchina ideale in grado di eseguire dei programmi che vengono sviluppati dai programmatori e conservati nella libreria delle configurazioni. Alla richiesta di esecuzione di un programma questa macchina prima reperirà la necessaria configurazione dalla libreria, poi la utilizzerà per configurare una sua particolare istanza costituente una nuova macchina dedicata specificatamente all'esecuzione di questo programma, infine manderà in esecuzione il programma attivando questa macchina dedicata; in fase di esecuzione la macchina dedicata potrà interagire con i suoi utilizzatori ricevendo input, elaborando dati e generando output.
Notiamo immediatamente che questa macchina soddisfa i requisiti dell'architettura[22] di Von Neumann: è dunque possibile, da un certo punto di vista, considerarla una implementazione concreta della macchina di Turing universale e quindi affermare che dispone di *potenza computazionale* analoga a quella della stessa macchina di Turing universale.

La caratteristica tipica dell'architettura di Von Neumann di trattare in modo analogo dati e programmi, caricando entrambi in memoria in fase di esecuzione, trova uno scenario applicativo tutto nuovo all'interno della programmazione configurativa. Mentre in un linguaggio di programmazione tradizionale il software viene scritto con formalismi e strumenti diversi da quelli comunemente usati per i dati, nella programmazione configurativa il software può essere editato con gli stessi meccanismi utilizzati per i dati e conservato sulle stesse memorie di massa (database, file systems, ecc.). Questa possibilità rende estremamente fluido il processo di generazione del codice che può essere sviluppato in modo più rapido rispetto ai linguaggi dinamici e senza dover necessariamente abbracciare un solo dominio applicativo come avviene coi DSL.

In sostanza la potenza espressiva di questi linguaggi, intesa come capacità di affrontare ampie classi di problemi, dipende da com'è stata affrontata l'implementazione del principio della configurabilità. Pensiamo che un valido linguaggio configurato sia quello che espone dei meccanismi sufficientemente ampi a coprire le esigenze standard di un singolo dominio applicativo o, alternativamente, di domini applicativi diversi accomunati da un livello standard di utilizzatori; la definizione corretta di tali meccanismi diventa quindi il punto focale di sviluppo della configurabilità.
Come la tecnica delle costruzioni ci insegna che partendo da un insieme ridotto di materiali di base (mattoni, legno, acciaio, acqua, ecc.) si possa, utilizzando tecniche diverse, aggregarne quantità differenti per realizzare piccole costruzioni o grandi edifici, allo stesso modo riteniamo sia possibile definire un insieme di elementi di base (i materiali) e di regole (le tecniche di costruzione) per manipolarli che permettono di affrontare la sfida dei processi di automatizzazione.
Crediamo inoltre che questa facilità nella creazione del codice apra un nuovo suggestivo scenario nello sviluppo del software: il modello di programmazione configurativa può divenire un modello estremamente adatto per lo sviluppo di codice a piccoli pezzi in quanto favorisce un approccio di tipo sia *adattativo* e sia *incrementale*.
Se infatti un programmatore professionale è in grado di affrontare e sviluppare un'applicazione mediante un approccio a singoli passi, è anche vero che il consolidamento di ogni passo richiede sforzo, conoscenza e capacità di visione che non sono riscontrabili nelle generalità degli utilizzatori di un computer. Invece la programmazione configurativa permette ad un programmatore non professionale di sviluppare veramente per singole parti, permettendogli di focalizzare l'attenzione su uno specifico aspetto e di provarne subito il funzionamento e l'efficacia senza dover realizzare in modo completo un singolo modulo software; questa possibilità deriva proprio dal meccanismo di realizzazione del codice che

evolve da un processo basato sulla scrittura ad uno basato sulla configurazione degli elementi[vii] semi-lavorati già posti a disposizione dall'ambiente di esecuzione

Da quanto esposto risulta evidente come la programmazione configurativa sia particolarmente applicabile con le più recenti metodologie di sviluppo del software: infatti la capacità di sostenere un approccio adattativo la rende coerente ai principi delle metodologie agili. Tuttavia questo approccio è anche applicabile con successo con le metodologie più tradizionali, perchè:

- Supporta significativamente il processo di sviluppo basato sulla metodologia a spirale; in quanto permette, dopo ogni milestone, di modificare agevolmente gran quantità di codice semplicemente modificando la configurazione già eseguita.
- È utile nel modello a prototipi non solo perchè velocizza la fase di realizzazione del prototipo stesso ma soprattutto perché, modificando poche configurazioni del prototipo di base, permette di ottenere facilmente molteplici prototipi ognuno utilizzabile in contesti sperimentali differenti.
- Soddisfa il modello incrementale perché l'esigenza di produrre il software per singoli pezzi da fornire sequenzialmente al committente è facilmente applicabile sia nella fase di produzione dei singoli pezzi che in quella di distribuzione (sarà sufficiente inserire una nuova configurazione nella libreria delle configurazioni).
- Favorisce la partecipazione ordinata al processo di scrittura delle configurazioni da parte di diversi programmatori impegnati sul medesimo progetto, pertanto migliora i risultati ottenuti dall'applicazione delle metodologie basate sui modelli a V o a cascata.

La capacità di affrontare con successo il processo di sviluppo e mantenimento del software, utilizzando differenti modelli di sviluppo, ci consente di affermare che la programmazione configurativa è valida *indipendentemente* dalla metodologia usata.

Un'ulteriore interessante riflessione deriva dall'analisi dell'ultimo assioma: il requisito di disporre di linguaggi configurativi che permettono di interagire con altri linguaggi tradizionali è legato all'opportunità di non considerare il modello configurativo come una panacea per tutte le esigenze di automatizzazione. La sezione seguente dimostrerà che le possibilità offerte dal modello configurativo si estrinsecheranno al meglio nei contesti in cui verranno soddisfatte le esigenze di vaste comunità di programmatori; in particolare vedremo come spingendo questo modello al suo limite estremo otterremmo dei nuovi linguaggi il cui utilizzo sarebbe più complesso dei linguaggi tradizionali: il 12° assioma ci preserva da tale degenerazione perché permette l'integrazione di moduli software realizzati tramite il modello configurativo con altri moduli sviluppati con linguaggi tradizionali.

IV – Principio dell'orientamento della programmazione configurativa

Nelle sezioni precedenti abbiamo discusso e definito il modello della programmazione configurativa, ora esamineremo come tale modello contribuisca all'aumento della comunità degli sviluppatori; in particolare si analizzerà perché questo approccio permette di realizzare e manutenere software in modo più semplice e rapido.

L'analisi del modello assiomatico evidenzia come la base di questo nuovo approccio alla programmazione deriva dall'aver spostato il focus del programmatore da un processo di scrittura ad uno di configurazione; se tale spostamento di focus semplifichi effettivamente l'attività di sviluppo di software o invece la renda più complessa dipende da quanto si riesca a semplificare il processo di configurazione stessa.
È ovvio che un processo di configurazione estremamente articolato, basato su un insieme elevato di elementi configurabili ed aggregabili in modo estremamente minuzioso, difficilmente condurrà alla definizione di un linguaggio di programmazione semplice, quindi non ne permetterà la diffusione su una comunità più vasta di programmatori. Siccome l'esigenza di semplificare i meccanismi di sviluppo del software è legata all'esigenza di semplificare i meccanismi da configurare è necessario definire dei livelli di configurazione che possano essere ritenuti di facile comprensione e utilizzo da parte di categorie diverse di utilizzatori. Questa esigenza ci spinge ad affermare che la programmazione configurativa può essere considerata come *orientata all'individuo* e quindi basata su un approccio *più soggettivo che oggettivo*.

Questa considerazione rende estremamente ostico il compito di definire uno specifico linguaggio configurativo, semplicemente perché individui diversi hanno un'idea differente del concetto di "semplicità"; ad esempio: un matematico potrebbe considerare facile l'utilizzo di modelli numerici, un filosofo potrebbe privilegiare modelli logici, un manager probabilmente gradirebbe modelli basati su tabelle. Dal punto di vista dei linguaggi di programmazione queste differenti visioni si tradurrebbero nella creazione di ambienti di sviluppo differenti; nel nostro esempio probabilmente il matematico spingerebbe verso linguaggi funzionali, il filosofo verso linguaggi logici, il manager verso linguaggi macro dedicati alla manipolazioni di tabelle dati.

---

vii Sebbene questi meccanismi sono già presente in precedenti paradigmi di sviluppo (orientati agli oggetti, utilizzando componenti, ecc.), la programmazione configurativa tende ad affrontarli da diversi punti di vista.

Da ciò consegue che l'implementazione del paradigma della programmazione configurativa può avvenire in modalità diverse: ognuna orientata verso categorie differenti di utilizzatori. In questo senso possiamo definire che *la programmazione configurativa costituisce una "Programmazione Orientata all'Individuo" o "Individual Oriented Programming"*.

Per creare un linguaggio dedicato ad uno specifico dominio applicativo (come nel caso dei DSL) è sufficiente definire un linguaggio di programmazione configurativo dedicato che sia in grado di soddisfare le esigenze di un individuo generico individuo: un ideale rappresentante di questa categoria di utilizzatori. È ovvio che il successo di un tale linguaggio configurato dipenderà primariamente dall'aver definito correttamente l'individuo ideale, in particolare avendo chiare le sue esigenze e capacità operative, secondariamente, dall'aver definito meccanismi di configurazione che un tale individuo considererà effettivamente semplici.

Chiaramente il raggiungimento di questo obiettivo è frutto di un processo empirico basato sull'esperienza di quanti parteciperanno al processo di definizione dei linguaggi; il fatto che le attività umane siano molteplici e quindi molteplici possano essere i contesti applicativi, ognuno composto da una pluralità di singoli individui, permette di desumere che i processi di definizione eventualmente intrapresi potrebbero condurre alla produzione di una gran quantità di linguaggi diversi che potrebbe degenerare in una nuova babele linguistica.

Il tentativo di scongiurare una tale degenerazione nonché l'esigenza di definire linguaggi che, similmente ai general purpose language, possano essere utilizzati in domini applicativi eterogenei, quindi incontrare le esigenze ed il favore di una vasta comunità di utilizzatori, ci induce a proporre la seguente definizione: *la "programmazione orientata agli utenti" o "User Oriented Programming" è costituita da tutte le implementazioni del paradigma di programmazione configurativa che abbiano l'obiettivo di permettere ad un generico utente finale di sviluppare direttamente una propria applicazione configurata.*

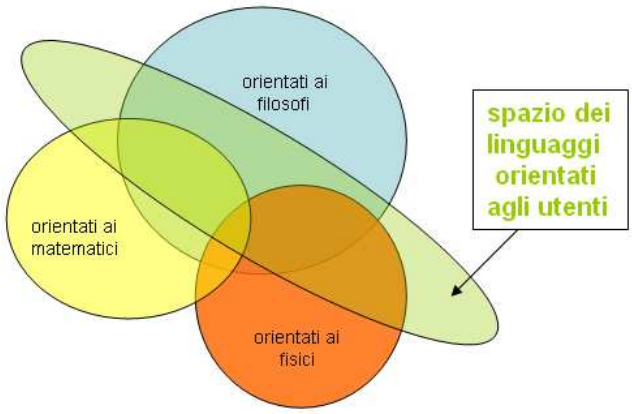

Fig. 4 – un'ideale classificazione di linguaggi orientati a differenti categorie di individui

Naturalmente anche questa definizione è estremamente ampia e contiene aspetti di soggettività; per tali ragioni discuteremo ora di alcuni principi generali di riferimento che possano guidare alla corretta definizione di un linguaggio di programmazione orientato agli utenti.

Un linguaggio di tipo configurativo deve esporre meccanismi da configurare; nella definizione di quali debbano essere i meccanismi che un programmatore potrà configurare è utile stabilire i confini del problema che si vuole automatizzare. Come già discusso sopra, il 12° assioma garantisce che un'applicazione configurata possa operare insieme ad altre applicazioni sviluppate con linguaggi tradizionali. Questa possibilità ci permette di evitare di considerare il modello configurativo come una panacea per l'automatizzazione di tutte le possibili attività; ha senso utilizzare il modello configurativo solo nei casi in cui esso produca degli effettivi benefici, sostanzialmente nei casi in cui:

- Sia richiesta la partecipazione di più individui, con competenze professionali eterogenee, all'attività di creazione e mantenimento di un'applicazione software; per esempio, nella realizzazione di sistemi di controllo di gestione di realtà corporate.
- Nelle situazioni in cui il modello di distribuzione del software realizzato in maniera configurata costituisca un vantaggio strategico per il produttore del software stesso. Per esempio, una software house potrebbe considerare vantaggioso realizzare col modello configurato anche solo l'infrastruttura di gestione di una applicazione (profili utente, MDI, menu, barre di comandi, semplici maschere di data-entry, query sui dati, videate di cambio password, supporto alla reportistica, funzionamento in reti pubbliche e private, ecc.) e realizzare con linguaggi tradizionali delle specifiche componenti che contengono un know-how complesso o per il quale è necessaria una tutela maggiore della proprietà individuale.
- Quando gli sviluppatori non dispongono delle esperienze e delle competenze richieste dai linguaggi di programmazione tradizionali; ad esempio, nello sviluppo di applicazioni per uso scolastico o domestico.
- Più in generale nei casi in cui l'utilizzo del modello configurativo permetta di sviluppare più semplicemente applicazioni complete o parti significative delle stesse

Negli altri casi l'utilizzo del modello configurativo deve essere valutato in modo specifico per evitare che la sua applicazione divenga controproducente; ad esempio nei casi in cui la gestione di un'applicazione configurata divenga più complessa rispetto ad un'analoga applicazione sviluppata con linguaggi di programmazione tradizionali.

Queste riflessioni evidenziano come l'insieme dei progetti informatici che possono affrontarsi con successo tramite la programmazione configurativa è comunque estremamente ampio. Tra i molti ambiti possibili di applicazione crediamo che un significativo interesse si avrà nella produzione di software per uso personale.
La generale diffusione delle tecnologie informatiche ha fatto in modo che le abitazioni di centinaia di milioni di individui contengano capacità di calcolo paragonabili a quelle delle aziende; gli ultimi anni hanno visto un incremento esponenziale delle soluzioni software espressamente realizzate e commercializzate per il mercato consumer; non è però significativamente aumentata la possibilità che un singolo consumatore possa costruirsi in proprio applicazioni che soddisfino le proprie personali esigenze. Riteniamo che il modello configurativo permetterà di raggiungere tale obiettivo.

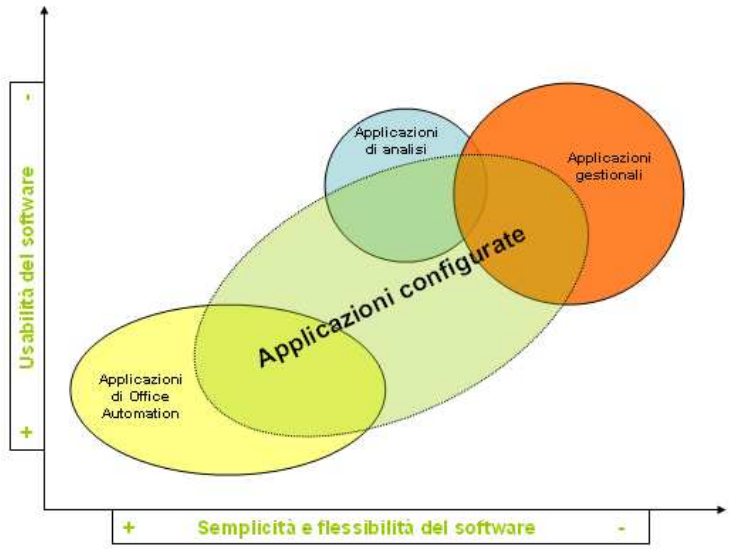

Fig. 5 – possibile utilizzo del modello configuraivo nelle organizzazioni

Crediamo inoltre che questo modello avrà una significativa diffusione anche negli ambiti delle organizzazioni che già dispongono di sistemi informativi. La figura 4 evidenzia come esiste uno spazio di diffusione considerevole costituito da tutte quelle esigenze per la cui automatizzazione risulti onerosa, in termini economici e temporali, l'adozione di soluzioni applicative tradizionali mentre l'utilizzo di strumenti tipici dell'office automation, quali i fogli elettronici, non offrono certezza nella corretta gestione del dato affrontato spesso in maniera destrutturata.

Riteniamo che i due esempi riportati, già da soli, espongano con chiarezza le possibilità offerte da questo modello. In generale pensiamo che l'approccio configurativo sia estremamente valido in tutte le realtà dove è necessario gestire in maniera rapida e strutturale il problema dell'incertezza del software.
Nella prima parte della trattazione abbiamo discusso dell'approccio tenuto dai differenti modelli metodologici nell'affrontare il problema dell'incertezza del software; gli studi di Rittel e Webber[23] hanno dimostrato come la realtà sia spesso costituita da *problemi maligni* o *wicked problems*. In effetti, esistono problemi reali che possono essere affrontati da molteplici punti di vista ed in base a livelli di conoscenza conflittuali, quindi lo sviluppo di software, anche se supportato da valide metodologie, difficilmente sarà pienamente conforme alle esigenze dei committenti.
In questo senso l'adozione della programmazione configurativa, grazie all'intrinseca possibilità di realizzare il software mediante un approccio adattativo, permette di correggere con rapidità le imperfezioni riscontrate e dunque può essere considerato un approccio vincente in tutte quelle applicazioni nate senza che in fase di analisi i vari attori coinvolti siano riusciti a definire, in modi condivisi ed immutabili, tutti i problemi[viii] dello specifico dominio applicativo.
Da ciò consegue che l'approccio configurativo è da ritenersi valido anche per risolvere quei criticismi che, dal modello a cascata[24] in poi, hanno spesso compromesso i processi di sviluppo e manutenzione del software.

V – Il primo linguaggio basato sul paradigma configurativo

Nelle sezioni precedenti abbiamo presentato e discusso il paradigma della programmazione configurativa, evidenziando come tale modello teorico innovi l'attività di sviluppo del codice. A conclusione di quanto finora discusso riteniamo utile presentare brevemente il primo linguaggio di programmazione configurato.
Questo linguaggio sperimentale, appartenente alla classe dei linguaggi di programmazione orientati agli utenti, è stato realizzato dagli autori di questa trattazione per accompagnare la proposta del nuovo paradigma di programmazione insieme con una concreta implementazione che dimostri l'applicabilità nel mondo reale delle asserzioni teoriche proposte. Siccome non è l'obiettivo di questa trattazione approfondire l'analisi di soluzioni software ci limiteremo ad evidenziare alcuni aspetti significativi.

Abbiamo visto che l'obiettivo della programmazione orientata agli utenti è definire dei meccanismi di programmazione che possano essere ritenuti semplici da un generico utente. Tale obiettivo è stato perseguito primariamente utilizzando interfacce visuali per l'interazione con gli utilizzatori e secondariamente basando il processo di impostazione delle configurazioni mediante l'editazione di tabelle.

viii I problemi noti e codificati, cioè in qualche modo governati dalle applicazioni, sono anche conosciuti come *problemi addomesticati* o *tame problems.*

La scelta delle interfacce visuali è stata motivata dall'ovvia constatazione che queste abbiano costituito la leva che ha permesso la diffusione dei calcolatori verso la generalità degli esseri umani. La rappresentazione tabellare è stata invece adottata per la capacità di coniugare la semplicità d'utilizzo alla possibilità di trattare analiticamente una gran quantità di dati. In sostanza l'idea di base dei fogli elettronici è stata ripresa e arricchita di ulteriori elementi a supporto di una gestione più strutturata dell'informazione. Un solo esempio, il programmatore non professionale potrà inserire nelle proprie applicazioni idee e suggerimenti trovati in altre applicazioni o fonti documentarie semplicemente utilizzando le tecniche del "copia e incolla": indubbiamente un modo di sviluppare codice molto semplice e suggestivo.

Dall'esigenza di utilizzare un ricco ambiente grafico, che fosse già utilizzato con semplicità dalla generalità degli individui, è discesa la scelta di utilizzare un'interfaccia Windows Forms[25] anziché un browser (l'utilizzo del browser avrebbe anche richiesto ulteriori competenze per la gestione delle infrastrutture di supporto).

Ulteriore elemento di interesse è la tecnica scelta per l'interazione con le fonti dati: per favorire il funzionamento anche tra calcolatori interconnessi su reti pubbliche a bassa velocità è stata privilegiata la logica di gestione dei dati in modalità disconnessa[26] (anche se utenti più evoluti potranno utilizzare meccanismi più tradizionali per l'interazione con i DB).

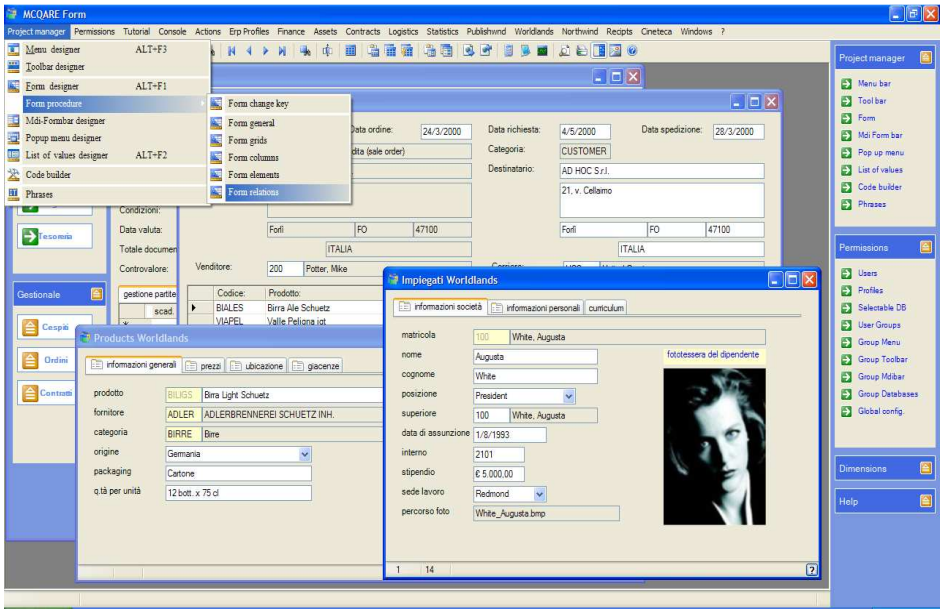

Fig. 6 – un esempio di moduli realizzati con questo nuovo linguaggio

L'esigenza di permettere l'interazione con altri linguaggi è stata soddisfatta arricchendo l'ambiente di un linguaggio di programmazione interno di tipo tradizionale; un linguaggio imperativo espressamente definito in modo da permetterne l'utilizzo anche a programmatori non professionali: i quali dovrebbero, con poco sforzo, riuscire almeno a costruire piccoli e semplici software (naturalmente i programmatori professionali potranno gestirne di più complessi).

La figura 6 è un esempio di moduli software realizzati mediante il linguaggio brevemente esaminato in questa sezione; riteniamo che questa immagine, da sola, testimoni a favore della validità dello specifico linguaggio, ma soprattutto evidenzi le possibilità offerte dal nuovo modello di programmazione configurativa ai programmatori professionali e più in generale a chiunque desideri poter sviluppare autonomamente una propria applicazione software operante in un dominio applicativo di interesse.

VI – Conclusioni

In conclusione di questa trattazione riteniamo si possa esprimere un giudizio conclusivo positivo sulla validità del paradigma di programmazione configurativa. Riteniamo che tale paradigma, nella sua interezza, costituisca una via del tutto nuova allo sviluppo del software: la programmazione orientata agli utenti permetterà di estendere la competenza dello sviluppo del codice verso un più vasto insieme di individui, mentre i tradizionali attori dello sviluppo del software riceveranno notevoli vantaggi perché non dovranno più occuparsi di tutti gli aspetti connessi al processo di sviluppo e manutenzione del software ma potranno dedicare le proprie competenze specialistiche agli aspetti ritenuti più strategici.

È ovvio che la capacità di introdurre semplificazione è direttamente proporzionale alla capacità di implementare il modello producendo dei linguaggi dal ragionevole livello di complessità. Nel definire questi linguaggi occorrerà evitare di partire dal presupposto di utilizzarli in tutte le circostanze, magari con l'obiettivo collaterale di estromettere i professionisti dello sviluppo del software dai processi di automatizzazione: chi tentasse di definire un siffatto linguaggio, anche nei casi in cui riuscisse nell'impresa, probabilmente otterrebbe un nuovo linguaggio *general purpose* ancora più complesso dei tradizionali linguaggi di programmazione.

È ragionevole invece attendersi una cospicua semplificazione della gestione di quelle sovrastrutture[ix] spesso ritenute necessarie per rispondere alle esigenze dei committenti anche nel caso di modeste esigenze di automazione: esigenze che di solito conducono ad applicazioni reali più complesse di quelle che lo specifico problema da automatizzare apparentemente richiederebbe.

In generale chiunque utilizzi software realizzato con il paradigma della programmazione configurativa godrà comunque di benefici anche nel caso non desiderasse partecipare attivamente alle fasi di sviluppo e mantenimento del codice. Primariamente la possibilità di agevolare l'attività di manutenzione del codice permette di ipotizzare vantaggi economici rilevanti: se le leggi di Leham e Belady[27] ci rammentano che ogni software, per poter soddisfare nel tempo i propri utilizzatori, deve essere sottoposto ad attività periodiche di aggiornamento, numerosi studi[28] hanno chiaramente dimostrato come il mantenimento del software assorba una parte cospicua dei costi complessivi per essi sostenuti, addirittura recenti[29] analisi hanno evidenziato come questa percentuale superi talvolta la soglia del 90%.

Ai significativi vantaggi economici devono poi aggiungersi i benefici di natura strategica derivati da una maggiore *indipendenza* del software realizzato col modello configurativo:

- La possibilità di memorizzare il codice editato allo stesso modo in cui si memorizzano i dati, rende più agevole la conservazione ed il trasferimento del know how contenuto in ogni singolo software. Inoltre, la possibilità tecnica di consultare e modificare in ogni momento il sorgente del codice permetterà al committente di contrattare accordi di licenza più estesi, che potranno giungere a garantire il mantenimento effettivo della proprietà intellettuale del codice sviluppato su commessa all'interno dell'organizzazione del committente, evitando eccessive dipendenze da singoli programmatori o software house.
- Semplificherà l'evoluzione delle applicazioni nel tempo: una nuova versione dell'ambiente di esecuzione, ad esempio necessaria per un aggiornamento di sistema operativo, potrà essere installata senza necessariamente modificare i programmi sviluppati che resteranno invariati nella libreria delle configurazioni; parallelamente una modifica delle applicazioni configurate, ad esempio per sostituire comandi obsoleti con altri più avanzati, potrà avvenire con semplici comandi di aggiornamento della libreria delle configurazioni.
- Un interessante beneficio si avrà in tutti quei contesti che necessitano di molte applicazioni diverse ma potenzialmente integrate (ERP, MRP, CRM, Human Resource, ecc.); la possibilità di costruirle utilizzando un medesimo ambiente da configurare, costituito dagli stessi elementi di base, permetterà di realizzare sistemi software normalizzati e standardizzati: quindi più facilmente usabili da utenti diversi.
- La programmazione configurativa permette anche di affrontare la questione[x] del *legacy dilemma*[30] con approccio del tutto nuovo: il mantenimento del codice nel tempo potrà avvenire in modo meno critico sia per la facilità di accedere al contenuto del codice stesso, di per sé nativamente organizzato in modo ordinato e documentato, e sia per la possibilità di adeguare, o anche riscrivere, le applicazioni configurate esistenti semplicemente modificando parti delle configurazioni già realizzate.

Naturalmente la programmazione configurativa non comporta solo benefici. Abbiamo già visto come essa non deve essere considerata come una panacea applicabile sempre e comunque; in alcune circostanze il suo utilizzo potrebbe risultare superfluo o addirittura dannoso. In particolare riteniamo che l'utilizzo del modello configurativo debba essere vagliato con attenzione nei software in cui l'elemento algoritmico prevale rispetto alla semplice interazione con i dati: ad esempio nei software contenenti *business rules* estremamente articolate e dedicate a singoli domini applicativi. In questi casi, l'utilizzo di linguaggi tradizionali, magari limitato allo sviluppo di specifici moduli da inserire in un più ampio sistema realizzato col modello configurativo, potrebbe risultare maggiormente conveniente.

Invece, è generalmente probabile che la programmazione configurativa favorirà la diffusione di nuovi prodotti, nuove tecniche, e nuove metodologie grazie all'effettiva possibilità di compartecipazione dei committenti finali al processo di sviluppo e mantenimento del software.

---

ix Esempio di queste esigenze, particolarmente sentite negli ambienti corporate, sono necessità di essere conformi alle diverse normative nazionali ed internazionali (i Sistemi di Qualità, la Riservatezza dei Dati, Sicurezza Informatica, ecc.) e l'esigenza di operare in Internet; esigenze che spesso richiedono complesse infrastrutture di supporto integrate con meccanismi flessibili per relazionare profili utente, ruoli aziendali e software utilizzati.

x Le organizzazioni che hanno l'esigenza di continuare a mantenere funzionanti nel tempo i propri sistemi legacy sono posti dinnanzi al seguente dilemma: ristrutturare completamente i sistemi legacy per continuare ad usarli o riscriverli cercando di riprodurre le medesime funzionalità.

Riferimenti

1 A.M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, in "Proceeding of the London Mathematical Society", XLII (1936). Leggere anche *A correction*, ibidem, XLIII, 1937.

2 A. Church, *The Calculi of Lambda-Conversion*, in "Annals of Mathematics Studies", n°6, Princeton, 1941.

3 W.W. Royce, *Managing the Development of Large Software Sytems: Concepts and Techiniques*, in "Proceeding of the WESCON", 1970.

4 C. Böhm e G. Jacopini, *Flow Diagrams, Turing Machines, and Language with Only Two Formation Rules*, in "Communications of the ACM", Vol. 9, n°5, 1966.

5 E.W. Dijkstra, *Go To Statement Considered Harmful*, in "Communications of the ACM", Vol. 11, n°3, 1968.

6 N. Wirth, *Sistematisches Programmieren*, Teubner Verlag, Stuttgart, 1972.

7 N. Wirth, *Algorithms + data structures = programs*, Prentice-Hall, 1976.

8 P.E. Rook, *Controlling software projects*, in "IEEE Software Engireering Journal", n°1, 1986.

9 T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.

10 J.E. Urban, *Software Prototyping and Requirements Engineering*, Rome Laboratory – DACS, 1992.

11 B. W. Boehm, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, Vol. 21, n°5, 1988.

12 B. Stroustrup, *What is "Object-Oriented Programming"? (1991 revised edition)*, AT&T Bell Laboratories, 1991.

13 O.-J. Dahl e K. Nygaard, *SIMULA- a language for programming and description of discrete event systems, introduction and user's manual*, Norvegian Computing Center, 1965.

14 L. Walton, *Domain-specific design languages*, 1996. URL: http://www.cse.ogi.edu/~walton/dsdls.html

15 Autori vari, *Manifesto for Agile Software Development*, 2001. URL: http://www.agilemanisfesto.org

16 M. Flower, *The New Methodology*, 2000. URL: http://www.martinfowler.com/articles/newMethodology.html

17 K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

18 J. Richter, *Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web*, MSDN Magazine The Microsoft Journal for Developer, Vol. 15, n°9, 2000.

19 G. Succi, *L'evoluzione dei linguaggi di programmazione: analisi e prospettive*, in "Mondo Digitale", n°4, 2003.

20 N.J. Cutland, *Computability: an introduction to recursive function theory*, Cambridge University Press, 1980.

21 U.S. Department of Defense, *COBOL, Initial Specifications for a Common Business Oriented Language*, Government Printing Office, 1960.

22 J. Von Neumann, *First Draft of a Report on the EDVAC*, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

23 H. Rittel e M. Webber, *Dilemmas in a general theory of planning*, in "Policy Sciences", n°4, Elsevier Scientific Publishing Company, 1973.

24 P. DeGrace e L. Hulet Stahl, *Wicked Problems, Righteous Solutions: A Catalog of Modern Engineering Paradigms*, Prentice-Hall, 1990.

25 J. Prosise, *Windows Forms: A Modern-Day Programming Model for Writing GUI Applications*, MSDN Magazine The Microsoft Journal for Developer, Vol. 16, n°2, 2001.

26 D. Sceppa, *Microsoft ADO.NET (Core Reference)*, Microsoft Press, 2002.

27 M. Lehman e L. Belady, *Program Evolution: Process of Software Change*, Academic Press, 1985.

28 J. Koskinen, *Software Maintenance Costs*, 2003. URL: http://www.cs.jyu.fi/~koskinen/smcosts.htm

29 R.C. Seacord, D. Plakosh e G.A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*, Addison-Wesley, 2003.

30 K. Bennett, *Legacy Systems: Coping with Success*, IEEE Software, Vol. 12, n°1, 1995.