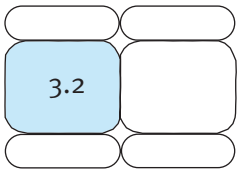
L'EVOLUZIONE DEI LINGUAGGI DI PROGRAMMAZIONE: ANALISI E PROSPETTIVE

Giancarlo Succi

È meglio usare FORTRAN o Basic? C o Pascal? C++ o ADA? Java o C#? Generazioni di informatici hanno combattuto battaglie all'"ultimo sangue" per difendere il "loro" linguaggio di programmazione, adducendo le più svariate motivazioni per giustificare la scelta. Nei 50 anni di storia dei linguaggi di programmazione, la scelta tra "cambiare linguaggio" e "cambiare il linguaggio" sembra sia ricaduta sulla seconda opzione. In questo articolo, si analizza lo sviluppo dei linguaggi con riferimento ai loro processi di sviluppo.



1. LINGUAGGI ... MA SOLO DI PROGRAMMAZIONE?

Nel gergo informatico comune, quando si parla di un "linguaggio", si fa di solito riferimento a un linguaggio di programmazione. Tutti sanno che esistono linguaggi di specifiche, linguaggi di analisi ecc., ma alla domanda: "Che linguaggio usate?" tipicamente la risposta è "Java" oppure "C" o ancora "C++" e così via. Spesso, questa è stata definita come un'anomalia, un limite per il sistema di sviluppo. Può darsi. Di fatto, l'utente finale vuole un sistema *software* che funzioni. Analisi, progetto e tutte le fasi precedenti la codifica sono certamente importantissime a questo scopo. Tuttavia, solo la programmazione fornisce un sistema funzionante, ovvero ciò che ha un valore diretto e tangibile.
Posto in altri termini, mentre esistono prodotti software costruiti solo sul codice, magari costruiti molto male, ma tant'è funzionanti e rispondenti ai requisiti del cliente, non esistono prodotti software costruiti solo con i documenti di analisi, di progetto ecc., e tramite i linguaggi di riferimento per tali fasi.
Pertanto, e senza nulla voler togliere ai linguaggi non di programmazione, in questa trattazione ci si concentrerà sui linguaggi di programmazione (Figura 1).

2. CARRELLATA STORICA

Come si legge in tutti i libri di introduzione alla programmazione, nei primi calcolatori, il programma era definito da circuiti elettrici: veri e propri collegamenti fisici. Si passò poi ai codici binari e ai linguaggi assemblativi a essi associati, ciascuno dei quali si riferiva in modo quasi completamente univoco a specifici elaboratori. Cambiando elaboratore, di solito, cambiava anche il linguaggio.
Il primo sostanziale passo in avanti nella programmazione fu a cavallo tra gli anni '50 e gli anni '60 quando si passò ai cosiddetti "linguaggi di alto livello", ovvero a linguaggi che esprimevano la computazione da fare in modo procedurale, per esempio, il FORTRAN (*FORmula TRANslation*) [1] e il COBOL (*Common Business-Oriented Language*) [25], o in modo funzionale, per esem-

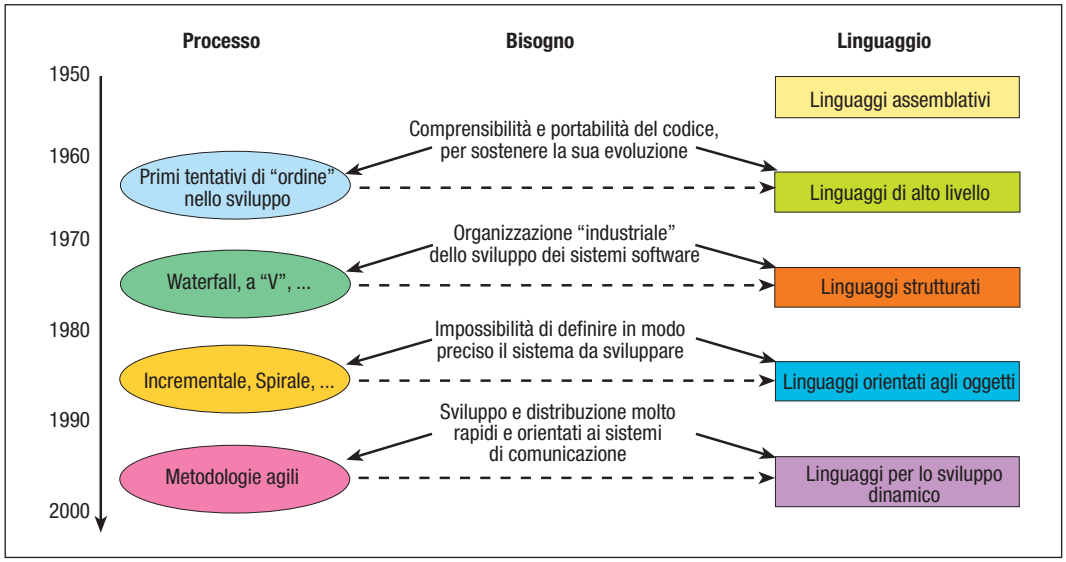


FIGURA 1
Evoluzione dei linguaggi

pio, il LISP (*LISt Processing language*) [24].
A quel tempo, i problemi significativi non riguardavano tanto l'organizzazione del processo di produzione, di solito lasciato nelle mani di pochi esperti, quanto la comprensione e la memoria di ciò che veniva sviluppato all'interno della stessa comunità degli sviluppatori. Di qui, la necessità di trascrivere la computazione in un formalismo familiare ai programmatori dell'epoca; era il tempo delle astrazioni: formule matematiche (FORTRAN), astrazioni logiche (LISP) o transazioni economiche (COBOL)[1]. Inoltre, la diffusione e la diversificazione del mercato dei calcolatori elettronici rese pressante il bisogno di scrivere programmi che funzionassero su più piattaforme di calcolo.
Presto ci si accorse che questo non bastava (fine anni '60 e primi anni '70). I programmi diventavano più grandi e richiedevano la collaborazione di più persone, anche più gruppi di persone. Occorreva strutturare il processo di sviluppo per renderlo più organico. Prendendo a prestito concetti e terminologie dalle teorie di *management* in voga all'epoca, e nei decenni precedenti per essere precisi, i*n primis* una versione molto statica del Fordismo [3], ci si concentrò sull'idea di organizzare il processo di sviluppo in moduli indipendenti con interfacce chiare e componibili. Si diffusero concetti quali la programmazione strutturata, il *data hiding* e l'incapsulamento. Nacquero l'ALGOL [10], il C [21], il Pascal [20], il Modula 2 [35] e l'Ada [15]: anche il FORTRAN fu fatto evolvere in questo senso.
Il processo di produzione modulare nella sua traduzione informatica, il modello di sviluppo "a cascata", dominò gli anni '70 e i primi anni '80. I limiti di questo modello, soprattutto la sua incapacità di gestire la flessibilità richiesta dalla produzione del software, cominciarono a essere evidenti verso la metà degli anni '80. Ci furono piccole variazioni di rotta, come i modelli di sviluppo "a V", ma sostanzialmente il modello modulare e gli associati linguaggi strutturati rimasero predominanti. L'idea guida era che la mancanza di precisione, e l'incertezza che la causava, poteva e doveva essere risolta a priori, tramite specifiche più accurate e circostanziate.
Negli anni '80 la comunità scientifica acquisì la consapevolezza che i problemi non erano solo legati alla carenza umana nelle attività di definizione del sistema, ma anche all'esistenza di una zona di ombra intrinseca allo sviluppo di un qualunque sistema software, che non permetteva di definire in modo completo e corretto fin dall'inizio le caratteristiche che il sistema software avrebbe avuto alla fine.
Il punto di partenza per lo sviluppo di un sistema software sono, in effetti, bisogni veri o percepiti come tali.
Ma tanti di questi bisogni si esprimono non in

1 Si tralascia di menzionare altri linguaggi che, pur molto popolari all'epoca, direbbero molto poco al lettore di oggi.

forma completamente razionale e comunicabile univocamente, ma semplicemente come un "senso di bisogno". Così lo sviluppo parte da requisiti generici e solo gradualmente evolve verso modelli maggiormente definiti.
La comprensione di questa problematica permise un salto nelle modalità di sviluppo del software. La suddivisione modulare lasciò il posto a sviluppi del software in maniera incrementale, per così dire, "a piccoli pezzi", in modo che si potesse usare ognuno di questi pezzi per calibrare lo sviluppo successivo. Facendo un'analogia con la costruzione di una casa, è come se ci si fosse concentrati a costruire una casa facendo prima la costruzione generale e poi una camera alla volta dalle pareti fino all'arredamento e, decidendo sulla base di tale camera, come proseguire.
Lo sviluppo a piccoli pezzi si concretò in molteplici modelli di produzione, quale quello *prototipale*, quello *incrementale*, quello a *spirale* ecc.. In termini di linguaggi di programmazione viene dato l'avvio, nel suo complesso, all'epoca dei linguaggi orientati agli oggetti.
Nei linguaggi orientati agli oggetti il sistema da costruire è rappresentato come un insieme di entità che interagiscono tra loro e che, interagendo, producono il risultato finale; pertanto, lo sviluppo si concentra nella identificazione di tali entità e nella successiva scrittura del codice relativo ad esse e alle loro mutue relazioni. I più famosi tra i linguaggi di programmazione orientati agli oggetti sono senza dubbio Smalltalk [16] e C++ [31], anche se il primo tra tali linguaggi è molto probabilmente SIMULA [9].
Non esistono solo linguaggi di programmazione orientati agli oggetti. Ci sono anche linguaggi di analisi orientati agli oggetti, linguaggi di progetto orientati agli oggetti e così via. A questo proposito, si vuole menzionare l'*Unified Modeling Language* (UML) [29], ora particolarmente in voga.
Si parla, pertanto, spesso di sviluppo orientato agli oggetti come una metodologia a se stante, in cui ci si concentra su analisi orientata agli oggetti, progetto orientato agli oggetti, codifica orientata agli oggetti ecc.
In realtà, è importante distinguere tra strumenti e fini: probabilmente è più adeguato parlare di metodi che usano i vari tipi di linguaggi orientati agli oggetti e che, qualche volta, per esempio nel caso del famoso *Rational Unified Process* (RUP) [22], usano *solo* linguaggi orientati agli oggetti.
Lo sviluppo delle telecomunicazioni, ivi compreso il mondo del web, portò alla necessità di definire processi di sviluppo che tenessero conto di questa realtà molto dinamica e di linguaggi che supportassero gli associati processi di sviluppo. È questo il periodo dell'avvento delle metodologie agili di programmazione, che danno supporto esplicito sia alla mutevolezza di requisiti che alla dinamicità del sistema da sviluppare.
Gli utenti considerano molto importante poter eseguire gli stessi, identici segmenti di codice su piattaforme diverse, nonché trasferirli e combinarli dinamicamente, secondo schemi di calcolo sia sequenziali che paralleli, per produrre rapidamente nuovi *sistemi di elaborazione*, se così ancora si possono chiamare.
I modelli orientati agli oggetti fornirono un importante punto di partenza in quanto permisero l'identificazione di entità base di computazione. Ma occorse dare un maggior impeto sia alla dinamicità dello sviluppo e della fornitura al cliente, che all'interoperabilità tra sistemi e con risorse disponibili in rete, quali archivi, liste di utenti, e banche dati in genere.
Divennero allora popolari i cosiddetti *linguaggi per lo sviluppo dinamico*, in quanto sono in grado di sostenere lo sviluppo di sistemi componibili dinamicamente e interoperabili tra loro.
Una parte di questi linguaggi era formata da semplici evoluzioni di linguaggi di *scripting*, ovvero linguaggi interpretati, finalizzati a guidare il sistema in ciò che esso doveva fare.
Si diffusero, però, anche linguaggi di programmazione completi che ottenevano gli stessi scopi, incorporando anche quei costrutti che via via, nell'evoluzione dei linguaggi, si erano affermati come essenziali. I più famosi esempi di questi linguaggi sono Java e C#.
Questa è la situazione in cui ci si trova oggi. È importante sottolineare che in questa carrellata si sono messi in relazione processi di sviluppo software con specifiche istanze di linguaggi di programmazione. Aver associa-

to un linguaggio a un processo non deve però in alcun modo far pensare che quel linguaggio sia nato come una risposta al bisogno definito dal processo. Anzi, molto spesso il linguaggio in sé anticipa il processo cui fa riferimento. L'ALGOL, uno dei primi linguaggi strutturati, fu concepito alla fine degli anni '50 [10]; SIMULA, l'archetipo dei linguaggi orientati agli oggetti, venne sviluppato negli anni '60 [9]; i primi i linguaggi per lo sviluppo dinamico sono degli anni '70: il *Concurrent Pascal* [19] e tutta la varietà di linguaggi prototipali prodotti dal gruppo di Hewitt [18].

3. LA PROGRAMMAZIONE DI ALTO LIVELLO

Vediamo ora il nesso tra processo di sviluppo e linguaggio ad esso associato, al fine di discuterne il legame.
Come precedentemente osservato, la programmazione di alto livello fu motivata dal bisogno di avere programmi scritti in formalismi comprensibili a un largo numero di sviluppatori e non dipendenti da architetture specifiche.
Tre particolari categorie di sviluppatori ebbero, in questo senso, un peso preponderante:

- gli scienziati e i tecnici - fisici, astronomi, chimici, ingegneri ecc., - che usavano i linguaggi per sviluppare applicazioni legate alle proprie attività di ricerca e sviluppo;
- i matematici, che stavano definendo le teorie alla base dell'informatica e dell'intelligenza artificiale;
- i responsabili dei sistemi informativi aziendali, che volevano automatizzare vari tipi di transazioni economiche e finanziarie sia per ragioni di efficienza che per evitare errori umani.

Di conseguenza, tre linguaggi acquisirono un'importanza tale che ancora oggi, nelle loro successive reincarnazioni, continuano ad essere usati: FORTRAN, LISP e COBOL.

❑ *FORTRAN*

Gli scienziati avevano bisogno di un linguaggio che permettesse la scrittura e la successiva elaborazione di formule complesse. Nacque così il FORTRAN [1, 2], che iniziò la propria lunga storia nel 1954. Così lunga da rendere reale la profezia degli anni '70: "Non si sa come sarà il linguaggio di programmazione dell'anno 2000, ma si chiamerà FORTRAN![2]".
Il FORTRAN funzionò bene per le applicazioni di calcolo scientifico e, di conseguenza, ebbe un notevolissimo successo in questo dominio, come evidenziato dalla profezia precedentemente citata.

❑ *LISP*

Il LISP fu ideato da John McCarthy nel 1958 con lo scopo di riportare in forma di linguaggio di programmazione il modello computazionale del λ-calcolo.
Il λ-calcolo fu definito dai padri dell'informatica, tra cui Church [8], e si diffuse presto, diventando uno dei modelli computazionali prevalenti. Avere un linguaggio di programmazione basato integralmente su di esso rivestiva, pertanto, un'importanza unica per gli scienziati di allora, che pensavano, così, di risolvere gran parte dei loro problemi di sviluppo software tramite la trascrizione, quasi pedestre, di modelli in programmi.
Per esperti di λ-calcolo l'uso del LISP fu, ed è tuttora, particolarmente indolore. Questa è una delle fortune maggiori di tale linguaggio, ancora oggi, dopo quasi 50 anni dalla sua prima concezione, particolarmente popolare nel settore dell'intelligenza artificiale.

❑ *COBOL*

Il linguaggio COBOL, originato all'inizio degli anni '60 [25], ambiva a trasporre in linguaggio di programmazione le transazioni economiche e finanziarie. Il punto di riferimento di questo linguaggio erano gli operatori del mondo degli affari che avevano esperienza di bilanci, scritture contabili, tabulati, regole precise e spesso un poco ridondanti.
Già la procedura seguita per la specifica del linguaggio evidenziava questa origine. Esso fu definito da un comitato formato dai *leader* statunitensi dell'epoca. Tra gli altri, si ricorda, per quanto concerne il settore privato, la partecipazione di Burroughs Corporation, IBM, Honeywell, RCA, Sperry Rand e Sylvania Electric Products. Anche tre enti pubblici particolarmente rilevanti presero parte ai lavori: US Air Force, David Taylor Model Basin e il National Bureau of Standards.

2 Questa profezia è stata attribuita a vari studiosi tra cui J. Backus, S. Cray e D. McCracken.

Il comitato si suddivise in tre sottocomitati: il primo per la pianificazione a breve termine, il secondo per quella a medio termine, e il terzo per il lungo termine; quest'ultimo in realtà non incominciò mai a lavorare.
È chiaro che, con queste premesse il linguaggio che sarebbe risultato, sarebbe stato preciso, ma anche un po' ampolloso e pesante da gestire. Questo fu, in effetti, ed è tuttora, il COBOL.
Tirando le somme, non c'è dubbio che anche il COBOL assolva molto bene i propri compiti di chiarire le azioni che l'elaboratore svolge in un linguaggio comprensibile dai propri utenti principali, ovvero da persone che si occupano di amministrazione, contabilità e finanza.

4. LA PROGRAMMAZIONE STRUTTURATA E DI SISTEMA

Come prima menzionato, la diffusione degli strumenti informatici nonché l'ingrandirsi dei sistemi da sviluppare, impose la definizione di un processo di sviluppo.
L'idea che parve più ovvia fu quella di basarsi su modelli modulari, concentrati sulla suddivisione del lavoro in parti e la successiva specializzazione dei compiti. Il modello di sviluppo di riferimento fu chiamato *waterfall*, ovvero, *a cascata* proprio da questo.
Si assistette, allora, a una particolare focalizzazione sullo sviluppo di linguaggi che permisero la decomposizione flessibile del sistema in sottosistemi.
Come si è detto, anche il COBOL permetteva una suddivisione; tale suddivisione, però, era rigidamente predefinita dal linguaggio: le divisioni erano "quelle tre".
Si desiderava, invece, poter dividere in modo che i moduli fossero definibili in modo chiaro e univoco dallo sviluppatore, che doveva anche stabilire come un modulo interagisse con gli altri moduli.
Come già menzionato, il punto di partenza per questo approccio fu l'ALGOL [10], seguito poi dal C [21] e dal Pascal [20]. È, però, con il Modula 2 [35] e con l'Ada [15] che la programmazione strutturata acquisisce la sua completa espressione. Nel seguito, di questa sezione si analizzano il Modula 2 e l'Ada. Si presenta, poi, una breve riflessione sulla nuove versioni del FORTRAN, il FORTRAN 77 [12] e il FORTRAN 90 [13].

4.1. Modula 2

L'intento del Modula 2 è duplice:
- da un lato, vuole favorire la suddivisione del lavoro in moduli indipendenti e comunicanti solamente tramite chiare interfacce;
- d'altro lato, vuole semplificare la scrittura del modulo nel suo complesso, separando la parte dichiarativa, in cui si specifica quello che un modulo fa, dalla parte implementativa, in cui si implementano funzioni finalizzate a ottenere quanto precedentemente dichiarato.

In questo senso, il Modula 2 sembrò essere il passo fondamentale per risolvere i problemi della crisi del software tramite l'applicazione dei modelli fordisti.
L'organizzazione in moduli garantiva la parcellizzazione del lavoro, assicurando la possibilità di scrivere sistemi sempre più grossi. La separazione dell'interfaccia dall'implementazione assicurava sia la specializzazione delle competenze (l'implementazione era fatta da esperti dei diversi domini applicativi, ma l'uso era aperto a chiunque potesse leggere le specifiche descritte nell'interfaccia), sia che eventuali modifiche implementative operate a un modulo, dovute, per esempio, alla già allora veloce evoluzione degli strumenti informatici, non provocassero effetti catastrofici per altri moduli che lo usavano.
Il successo di Modula 2 non è tanto testimoniato dalla diffusione del linguaggio, che rimase piuttosto limitata, quanto all'influenza che ebbe su linguaggi sia esistenti che di nuova concezione. Modula 2 provò, infatti, che le idee dei modelli "a cascata" potevano avere riflessi essenziali sui linguaggi di programmazione. Sia C che FORTRAN si aggiornarono per acquisire il maggior numero possibile di caratteristiche di Modula 2 come una migliore separazione tra interfaccia e implementazione, la tipizzazione stretta ecc. Ada ricalcò tutta la struttura a moduli nei propri *package*.

4.2. Ada

Ada nacque sotto l'egida del Dipartimento della Difesa statunitense con l'ambizione di voler diventare il linguaggio universale per i

sistemi *embedded, in primis*, e poi per tutta l'informatica.
Con tale sponsorizzazione, il successo sembrava garantito. Il linguaggio rassomigliava molto a Modula 2, a parte la nomenclatura: il modulo si chiamava package.
Una peculiarità di Ada che vale la pena rimarcare è la presenza di una ricca scelta di modalità di "passaggio parametri": per valore, per risultato, per valore-risultato e poi, con una forzatura, per riferimento via puntatori. Si può intravedere in questo il desiderio di formalizzare, non solo al livello alto del modulo ma anche al livello più basso delle singole funzioni, i protocolli di interazioni tra le diverse parti di programma.
In ogni caso, essendoci poco da aggiungere a Modula 2, in chiave di chiarezza operativa gli ideatori di Ada si concentrarono sull'estensione del linguaggio per gestire ambiti di programmazione utili in sistemi *embedded* ma non considerati esplicitamente da Modula 2, come la programmazione concorrente e la gestione delle eccezioni, sull'ampliamento dei tipi astratti tramite l'uso dei generici e, infine, sulla definizione di una semantica precisa.
Queste estensioni, però, resero difficile la definizione formale dello stesso linguaggio e causarono dei ritardi evidenti nello sviluppo di compilatori che ne permettessero il largo utilizzo e la diffusione. Fu, pertanto, solo alla fine degli anni '80 che Ada cominciò a prendere piede su larga scala, quando ormai i modelli "a cascata" si avviavano al tramonto.

5. MODELLI A V

All'inizio degli anni '80 il modello di sviluppo "a cascata" cominciò a evidenziare i propri limiti, soprattutto per quanto concerne l'incapacità di gestire la variabilità dei requisiti.
Come precedentemente menzionato, la soluzione fu quella di "cercare di essere più precisi" nello sviluppo del software, attribuendo alla scarsa precisione degli analisti, dei progettisti e degli implementatori, il mancato successo del linguaggio.
Si passò allora al modello di sviluppo "a V", in cui si associava a ogni fase di sviluppo (ovvero analisi, progetto e codifica), una batteria di test per verificare la sua correttezza. I test andavano eseguiti quando la batteria di test sottostante era stata completata. In pratica, il flusso delle attività era: analisi, progetto, codifica e poi test della codifica, test del progetto, test dell'analisi. Dato il movimento discendente (dal generale al particolare) nella fase di sviluppo e ascendente (dal particolare al generale) nella fase di test, si chiamò questo modello "a V".
Nello sviluppo del codice si ebbe un riflesso diretto del modello a V tramite le cosiddette "asserzioni".
In pratica, si definivano per ogni funzione pre-condizioni e post-condizioni che dovevano essere soddisfatte prima e dopo l'esecuzione di ogni funzione. Inoltre, si potevano inserire delle espressioni booleane in punti critici di una funzione, le asserzioni, che dovevano essere sempre valutate vere. Il problema passava allora alle definizioni di queste condizioni e asserzioni e alla loro valutazione durante l'esecuzione del programma [4]. Di fatto, nessuna versione di un linguaggio di programmazione effettivamente usato inglobò l'idea delle asserzioni, a parte le librerie di asserzioni del compilatore *gnu* del C. Essa rimase, quindi, nell'aria e ora si è praticamente tradotta nel concetto di *test first*, tanto caro agli sviluppatori utilizzanti le metodologie agili.

5.1. FORTRAN 77 e FORTRAN 90

L'evoluzione del FORTRAN verso la programmazione strutturata e di sistema avvenne in due passi: il FORTRAN 77 e il FORTRAN 90.
Il FORTRAN 77 venne definito con l'intento di eliminare gli aspetti chiaramente obsoleti delle versioni di FORTRAN ancora largamente in uso nei primi anni '80: il FORTRAN 4 e il FORTRAN 66.
In particolare, venne definito un costrutto per i cicli e si obbligò il programmatore a definire esplicitamente tutte le variabili, in modo da limitare le possibili sviste.
Le novità portate dal FORTRAN 90 furono molteplici: eliminò i vincoli di formattazione precedentemente descritti, permise la definizione di tipi strutturati e di moduli, aggiunse un insieme di costrutti di controllo equivalenti a quelli di Modula 2 e di Ada, tra cui

le tecniche di *passaggio parametro* tipiche di Ada, e incluse anche alcune tecniche primordiali per la definizioni di oggetti. Inoltre, aggiunse strumenti per la programmazione parallela, dato che gli sviluppatori in FORTRAN erano spesso scienziati interessati a calcoli su strutture di grandi dimensioni.

6. LA PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Verso la metà degli anni '80, l'informatica comprese che i problemi dei modelli di sviluppo *a cascata* nascevano dall'intrinseca incertezza che pervade il modo stesso di sviluppare software.
Questo cambiava radicalmente la possibile soluzione. Occorreva un modello, non tanto focalizzato a raggiungere una chimerica esattezza, quanto capace di gestire una congenita variabilità.
I modelli di divisione del lavoro diventavano, di colpo, inadeguati: se lo scopo di un sistema da sviluppare non era chiaro, una qualunque sua suddivisione rischiava di rendere ancora meno chiaro lo scopo del sistema.
Si pensò allora a uno sviluppo *a piccoli pezzi*. L'idea fu di prendere come punto di partenza l'ambito applicativo dei sistemi da sviluppare e di studiare a fondo tale ambito con lo scopo di capire quali elementi andavano sicuramente considerati nel sistema da sviluppare. Per esempio, se si trattava di sviluppare un sistema per la gestione di un aeroporto, occorreva considerare l'aspetto dei voli, dei passeggeri, dei biglietti e così via. Si poteva allora partire nello sviluppo dal modello di comportamento di tali entità.
Tali entità furono chiamate oggetti e, continuando la metafora, trasmissioni di *messaggi* tra diversi oggetti si sostituirono a *chiamate a funzioni*. Ogni oggetto aveva una serie di *metodi* per la risposta ai messaggi che rimpiazzarono il *corpo delle funzioni*[3].
L'uso degli oggetti facilitò anche le comunicazioni con il cliente, in quanto gli oggetti erano presi dal linguaggio del dominio applicativo noto al cliente stesso e, quindi, il flusso della computazione diventava spesso di più semplice comprensione.
Lo sviluppo a piccoli pezzi si concretizzò in diversi modelli di produzione, tra cui si ricordano i modelli incrementali, quelli prototipali e quelli a spirali. Tutti facevano esplicito riferimento alla programmazione orientata agli oggetti.

6.1. C++

Infatti, tra gli scopi del C++ ci fu quello di rendere facile la transizione alla programmazione orientata agli oggetti per la comunità degli sviluppatori di sistema in ambiente Unix, solita a programmare in C.
Il C++ è un linguaggio in questo senso controverso: i puristi lo considerarono, fin dall'inizio, non un vero linguaggio orientato agli oggetti, ma un ibrido e gli preferirono sistemi più integri, quali Smalltalk.
In questo articolo, non si vuole entrare nel dibattito su quale sia il migliore linguaggio orientato agli oggetti. Di fatto, però, C++ è il più diffuso e, quindi, rappresenta un punto di riferimento importante se si vuole studiare l'evoluzione dei linguaggi di programmazione.
In primis, si ha la presenza del codice dell'implementazione insieme a quello della dichiarazione, seppure con una separazione tra *zona pubblica* – accessibile a tutti, e *zona privata* – accessibile solo agli implementatori. Questo sembra contraddire l'idea della divisione e della specializzazione del lavoro. Ma se la dimensione del codice rimane contenuta, secondo l'idea dello sviluppo incrementale, essa serve a fornire una specifica operazionale al metodo. Chiaramente, se il codice è lungo e complesso, la sua presenza è dannosa.
L'idea di documentare il codice con il codice stesso ebbe molta fortuna e ora è largamente usata in Java, anche tramite una specifica strutturazione dei commenti, con un particolare strumento chiamato JavaDoc, che analizza il codice ed estrae parti di commenti formattati secondo una semplice e chiara sintassi per fornire in modo automatico una documentazione sintetica ma efficace del sistema.
Le definizioni di zona pubblica e zona privata

[3] Peraltro, i metodi sono anche chiamati *funzioni membro*.

diventano, pertanto, quasi un suggerimento all'utente della classe su dove concentrare, in modo prioritario, la propria attenzione, piuttosto che una separazione contrattuale di compiti[4].

6.2. FORTRAN 95 e FORTRAN 2000 e altri linguaggi

Il FORTRAN proseguì la propria evoluzione verso l'universo della programmazione a oggetti con due versioni: il FORTRAN 95 e il FORTRAN 2000. Esse vengono qui nel seguito brevemente tratteggiate.

Il FORTRAN 95 arricchì il linguaggio con una serie di costrutti di controllo più raffinati e con ulteriori specializzazioni del concetto di funzione: le funzioni pure e le funzioni di elementi. Le funzioni pure non hanno effetti collaterali, non agiscono cioè su variabili globali o di ambiente. Le funzioni di elementi hanno come argomenti e restituiscono solo valori scalari. Si definirono, inoltre, tecniche per una gestione più pulita dei puntatori, inclusa la loro inizializzazione e l'allocazione e deallocazione dinamica della memoria. Infine, si rese più robusta la gestione dei tipi derivati.

Il FORTRAN 2000 è attualmente in corso di definizione e sta compiendo un passo sostanziale nel rendere FORTRAN completamente orientato agli oggetti [28]. In particolare, si sta rafforzando la gestione dei generici estendendoli ai tipi derivati, l'uso dei puntatori a funzione e la gestione di oggetti polimorfi. Inoltre, sta migliorando l'interfacciamento con altri linguaggi orientati agli oggetti e non, per rendere più agevole l'interoperabilità tra sistemi, secondo i cardini dello sviluppo a piccoli pezzi.

L'approccio generale è più ridondante del C++. In particolare, c'è una marcata separazione tra interfaccia e implementazione che ricorda i linguaggi strutturati; questa interfaccia addirittura disaccoppia il nome del metodo dal corpo del metodo, richiedendo la presenza di un esplicito operatore per la connessione: l'operatore ⇒. La filosofia generale, però, è chiaramente quella ibrida a oggetti del C++.

7. LA PROGRAMMAZIONE DI SISTEMI COMPONIBILI DINAMICAMENTE

La programmazione orientata agli oggetti causò un miglioramento sostanziale dello sviluppo del codice. Di contro, però, richiese una competenza decisamente superiore da parte degli sviluppatori.

Si potrebbe quasi asserire che mentre i modelli *a cascata* volevano risolvere i problemi con la formalizzazione delle attività da svolgere e la specializzazione del lavoro, i modelli *a piccoli pezzi* si concentravano sulla definizione di un linguaggio ricco che permettesse di catturare nel modo più completo possibile i requisiti dei clienti, comunque organizzati in insiemi di dimensioni ridotte.

7.1. Limiti dell'approccio orientato agli oggetti

Due fattori tra loro connessi emersero, però, con effetti dirompenti. Il primo fu che ci si rese conto che l'uso del C++ o di un altro linguaggio di simili connotazioni richiedeva un'adeguata formazione e che la formazione del personale in un linguaggio complesso era ovviamente non semplice, soprattutto se tale personale proveniva da anni di uso di linguaggi di programmazione meno evoluti.

Il secondo fattore fu la domanda del mercato di prodotti in tempi brevi e il contestuale comparire sulla scena del web, che rendeva ancora più impellente tale domanda e aggiungeva la necessità di avere un linguaggio che si adattasse a combinare pezzi di codice che si muovessero sulla rete.

La combinazione di questi due fattori contribuì alla fortuna del linguaggio Java.

Le metodologie agili, poi, definirono un processo di sviluppo che sosteneva gli stessi obiettivi: limitare la complessità di sviluppo, favorire la consegna del prodotto all'utente in tempi brevi e certi anche se con funzionalità ridotte, garantire la qualità e eliminare gli sprechi dovuti a sviluppi di moduli inutili o ad attività sì connesse con lo sviluppo ma non direttamente produttive.

[4] Si noti che in Smalltalk, il linguaggio di programmazione agli oggetti considerato forse più *puro*, questa suddivisione non esiste.

7.2. Java

Nacque con l'intento di definire un linguaggio ad oggetti quanto più *completo e puro* possibile, utilizzando come base la sintassi del C e senza compiere gli errori dell'Objective C [5], un precedente linguaggio nato con i medesimi scopi ma naufragato per l'ambiguità di fondo di non avere né la pulizia di un linguaggio ad oggetti né la flessibilità del C.
Per semplificare l'apprendimento e la gestione dei programmi scritti nel linguaggio, e ridurre, quindi, lo sforzo per l'apprendimento di costrutti inutili, si eliminarono alcune caratteristiche particolarmente onerose del C++: la gestione della memoria dinamica lasciata all'utente, i generici, l'ereditarietà multipla delle classi e il passaggio parametri per riferimento.
Per facilitare l'uso del sistema in modo dinamico sulla rete si cambiò il meccanismo di generazione dell'oggetto e dell'eseguibile, si resero tutte le funzioni virtuali (con *binding* ritardato) e si definirono costrutti specifici per la gestione della concorrenza.
Dall'analisi del processo di compilazione ed esecuzione (Figura 2) si può capire l'aderenza delle specifiche del linguaggio agli scopi di Java precedentemente delineati e ai modelli di sviluppo agili.

Nel caso del C++, come nel caso dei linguaggi di alto livello[5] e dei linguaggi strutturati finora considerati, fu scelto il seguente processo per la generazione di un programma eseguibile:

- dapprima un modulo, chiamato "compilatore", provvede ad analizzare il *file sorgente* e a generare un file "oggetto", o "OBJ", che è indipendente dal linguaggio sorgente originario, ma dipendente dall'hardware e dal sistema operativo;
- quindi, un secondo modulo, chiamato *linker*, dipendente dallo specifico hardware e sistema operativo, ma spesso indipendente dal linguaggio sorgente, si occupa di cercare e unire i diversi moduli oggetto, sia quelli sviluppati dal singolo programmatore che quelli provenienti dal sistema operativo, per costruire il modulo eseguibile, quello che può essere caricato e fatto funzionare sul calcolatore da solo e in un unico blocco.

In alcuni testi e in alcuni ambienti di sviluppo le due fasi sono trattate in modo così consequenziale da essere chiamate genericamente entrambe *compilazione*.
Va da sé che in questo contesto non è assolutamente scontato che un modulo oggetto o un modulo eseguibile sviluppati per una data configurazione di hardware e sistema operativo funzionino su un'altra configurazione.

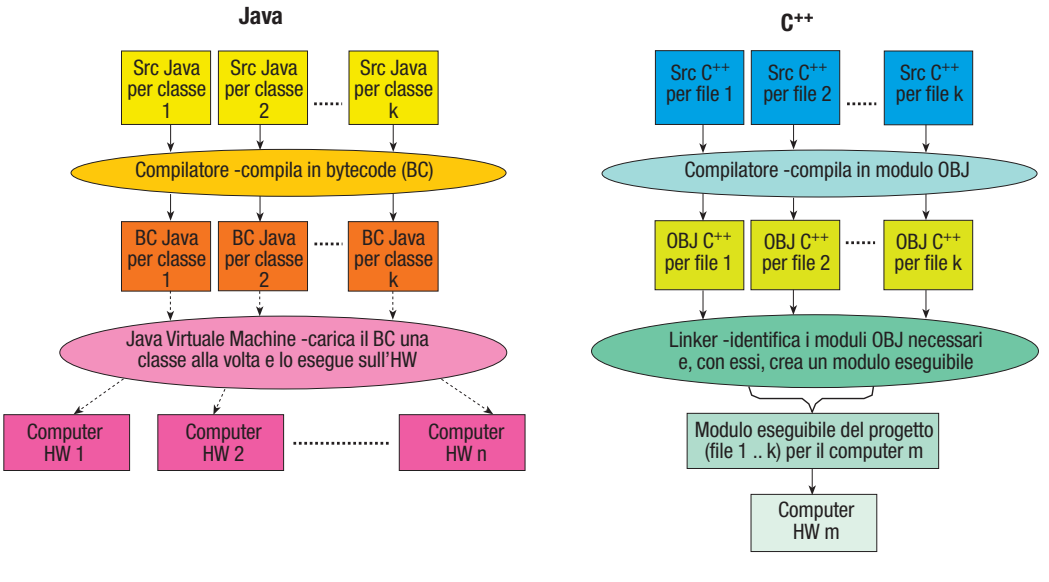


FIGURA 2
Analisi del processo di compilazione ed esecuzione in Java e in C++

[5] In realtà, il LISP ha un processo di interpretazione più simile a quello di Java che agli altri. Di fatto, però, il *programmatore medio* del LISP di solito non percepisce questa differenza, e quindi non sfrutta le potenzialità ad essa collegate.

È di conseguenza esclusa la possibilità che lo stesso codice eseguibile possa migrare di macchina in macchina continuando a funzionare senza nulla toccare.
Inoltre, occorre che tutto il sistema sia costruito in un solo colpo e che, al momento del link, si abbia a disposizione la versione finale di tutti i moduli oggetto necessari; tale versione non è più cambiabile[6].
In Java, fu preferito un approccio completamente differente.
Il codice è compilato, una classe alla volta, da un compilatore Java che lo trasforma in un modulo *simil-oggetto* chiamato di solito "bytecode". Per ogni classe pubblica (ovvero, ogni classe utilizzabile al di fuori del package di definizione) esiste uno e un solo file bytecode. Il bytecode è lo stesso per ogni compilatore Java, indipendentemente dalla configurazione di sistema operativo e hardware in uso.
Un altro modulo chiamato *virtual machine* o *macchina virtuale*[7] provvede, quindi, a caricare ed eseguire una ad una le classi effettuando la ricerca del bytecode delle classi necessarie dinamicamente nel corso dell'esecuzione del programma stesso.
La macchina virtuale disaccoppia il bytecode dal sistema operativo e dall'hardware sottostante, fornendo tutti quegli strumenti necessari per l'esecuzione del programma e occupandosi anche della gestione automatica della memoria dinamica e della concorrenza.
Il funzionamento delle macchine virtuali è, in pratica, lo stesso[8] per ogni sistema operativo e per ogni hardware, anche se, ovviamente, il codice eseguibile della macchina virtuale non è necessariamente lo stesso.
Ne risulta che il programma può tranquillamente migrare da un computer all'altro, purché ovviamente il calcolatore sia dotato della macchina virtuale adeguata allo scopo.
Il caricamento in memoria fatto per classi permette, inoltre, di limitare la quantità di codice da trasmettere sulla rete, in quanto si possono utilizzare le parti di classi di sistema presenti sul calcolatore destinazione. Inoltre, il caricamento parziale di codice permette di rendere maggiormente dinamici lo sviluppo e la successiva evoluzione del sistema.
Se si definiscono in modo chiaro le dipendenze mutue tra classi, cosa difficile ma non impossibile, soprattutto se si mantengono gli assunti della programmazione orientata agli oggetti, di rendere gli oggetti costantemente riferiti alle entità del dominio applicativo, è allora possibile che gli oggetti di versioni diverse della stessa classe rispondano agli stessi messaggi con risposte sicuramente non identiche, ma equipollenti dal punto di vista dello sviluppo del sistema, in modo che il sistema possa evolvere dinamicamente.
Un uso tipico di questo approccio dinamico sia per quanto concerne la migrazione del codice che la definizione di versioni multiple della stessa entità, si ha nella possibilità di caricare nelle pagine web frammenti di codice Java chiamati *applet* che vengono trasmessi dal *server* web al calcolatore che sta visualizzando la pagina web e vengono poi eseguiti sulla macchina virtuale residente sul calcolatore destinazione.
Java riconobbe l'importanza di *usare* la sintassi di un linguaggio affermato, il C++, in modo da invogliare i programmatori in tale linguaggio a considerarlo come una potenziale alternativa.

7.2. Autodocumentazione e test-first in Java

Per completare la panoramica su Java e approfondire il legame che lo lega con i modelli di sviluppo agile, è importante menzionare che Java fu concepito anche per fornire un supporto all'autodocumentazione del codice. Insieme al linguaggio fu costruito un *tool* per documentare la struttura in classi

[6] Si semplifica la realtà ignorando, volutamente, le librerie di sistema caricabili dinamicamente o quelle condivise: esse non sono state pensate per cambiare il comportamento a *run time* di un sistema quanto per poter ridurre le dimensioni dell'eseguibile e rendere più veloce il suo caricamento ed esecuzione, ovvero (come nel caso delle DLL sui vecchi sistemi operativi DOS) per renderlo eseguibile *in toto*.

[7] Spesso si usa anche in italiano il termine inglese.

[8] Si trascurano i problemi legati alle diverse versioni di macchina virtuale, poiché si tratta di un problema di gestione delle configurazioni e non di struttura del linguaggio di programmazione.

anche tramite l'identificazione e l'estrazione di commenti particolari che il programmatore può scrivere all'uopo.
I cultori delle metodologie agili estesero subito a Java l'approccio *test-first*, originariamente pensato per Smalltalk.
Test-first è un processo di scrittura del codice in cui dapprima si definisce per ogni metodo di una classe, non appena si sa di doverlo sviluppare e si conoscono i suoi parametri, uno o più test che ne verifichino il corretto comportamento e, quindi, si usano i test come guida per lo sviluppo della classe. La similitudine tra l'approccio test-first e le asserzioni precedentemente citate è evidente.
Test-first è particolarmente efficace in Java nello sviluppo di sistemi componibili dinamicamente, in quanto le moltitudini di test che vengono così via via create diventano un controllo molto efficiente contro possibili disallineamenti che si vengono a creare quando si costruisce il codice a pezzetti e a intervalli successivi, magari anche in luoghi distribuiti geograficamente.
L'approccio test-first fu consolidato in Java con jUnit, il *porting* di sUnit, lo strumento precedentemente usato in Smalltalk per poter eseguire automaticamente tutti i test in un colpo solo. In un contesto di sviluppo agile, jUnit rappresentò un passo essenziale per assicurarsi che i test fossero effettivamente sviluppati prima del codice e costantemente eseguiti.
In ultima analisi, si può dire che l'autodocumentazione e il test-first rappresentino un'ulteriore forte evidenza che lega processo di sviluppo agile ai linguaggi per i sistemi componibili dinamicamente.
A questo punto si potrebbe obiettare che Smalltalk fu proprio il precursore dei linguaggi per sistemi componibili dinamicamente e non tanto dei linguaggi a oggetti. Questa obiezione è, infatti, molto interessante: delinea, in modo significativo, l'importanza di tale linguaggio ed evidenzia l'ispirazione di chi lo pensò. Del resto, Smalltalk è legato anche al filone del LISP e dei linguaggi funzionali [30] e, come già menzionato, ci sono legami non ovvi ma profondi tra i modelli funzionali proposti dal LISP e i linguaggi per sistemi componibili dinamicamente.

8. LINGUAGGI, LINGUAGGI, LINGUAGGI, ...

In questa veloce carrellata sono stati evidenziati solo una minima parte dei linguaggi di programmazione usati.
In particolare, non è stata spesa neanche una parola sulla serie dei linguaggi Microsoft a partire dai primi BASIC adattati per il DOS fino a C# ecc.. Questo non vuole essere in alcun modo una censura nei confronti della più grande azienda software del mondo, che molti criticano ma che indubbiamente ha finanziato ricerche di altissima qualità che hanno contribuito alla crescita del mondo dell'informatica.
Bisogna riconoscere però che, quando ha operato direttamente e non tramite i finanziamenti alla ricerca erogati all'Università, Microsoft si è focalizzata nella direzione del perfezionamento di strutture linguistiche già esistenti e in quella degli ambienti di sviluppo per rendere quanto più efficiente possibile alcune tecniche di sviluppo considerate strategiche, piuttosto che nella creazione di nuove strutture linguistiche di interesse per questo lavoro.
Ci sono poi i linguaggi di *script* che hanno costituito la base della programmazione di sistema. Anche in questo caso, l'averli ignorati non significa averli sottovalutati. Il linguaggio della *C-shell* ha formato generazioni di sistemisti Unix, così come PERL successivamente ha permesso la creazione dei prototipi dei siti web dinamici, e quindi PHP[9] e così via.
Bisogna però ripetere quanto detto per Microsoft: sono linguaggi molto efficaci ma che sublimano alcune caratteristiche esistenti in un dato linguaggio per un dominio applicativo ben preciso: per esempio, la programmazione di pagine web, la creazione di portali, la programmazione *in the small* di sistema. Come tali, sono fuori dall'ottica di questo articolo.
Una menzione va poi ai linguaggi funzionali, al di là del LISP, e a quelli logici. Per gli altri,

[9] Il termine PHP è un acronimo ricorsivo che sta per *PHP Hypertext Processor*, ovvero, processore PHP di ipertesti. Questo acronimo ben evidenzia le caratteristiche di questo linguaggio, oggi diventato molto popolare.

anche se hanno creato strutture interessanti, non si può dire che si siano diffusi in modo significativo. Inoltre, e questa è una opinione molto personale dell'autore, hanno più usato e modellato teoricamente caratteristiche già precedentemente pensate piuttosto che gestito un sistema nuovo: i tipi generici sono nati prima di ML[10] (*Meta Language*); le gerarchie di classi sono nate molto prima di Haskell; l'analisi delle strutture "ad alberi" era presente in YACC prima del PROLOG; la programmazione *a clausole* era presente prima dei recenti linguaggi *a vincoli*.
Forse due aspetti sono originali a questi linguaggi: la valutazione *lazy* per la creazione di strutture infinite –propria di alcuni linguaggi funzionali quali LML e Haskell, e la possibilità di avere parametri che sono contestualmente di *input* o di *output*, come nel caso del PROLOG.
Ma, dopo l'entusiasmo iniziale, non pare che questi modelli abbiano originato sostanziali avanzamenti nella disciplina.
È opportuno sottolineare che questo non vuol dire affatto che tali linguaggi si siano rivelati inutili semplicemente, essi non si legarono mai ad alcun processo in quel binomio indivisibile di cui si discute. Con una metafora si può affermare che nessuno penserebbe di andare con un'auto di Formula 1 per i vicoli del centro storico di Genova, o con una bicicletta in autostrada, ma questo non significa che le macchine di Formula 1 o le biciclette siano inutili.

9. SI PUÒ TRARRE UNA CONCLUSIONE?

Si pensa che l'analisi effettuata confermi l'ipotesi di partenza sulla imprescindibile connessione tra linguaggio e processo.
È difficile ma interessante provare a predire il futuro dei linguaggi usando tale paradigma. Chiaramente non si possono che identificare tendenze passate e capire come queste tendenze possano evolversi.
Si può poi notare che i processi di produzione del software e poi i linguaggi hanno attraversato un'interessante parabola in termini di complessità (Figura 3).
Come già precedentemente ricordato, dapprima si cercò di rispondere ai bisogni dell'industria software con modelli di produzione sempre più complessi e in secondo momento, dato che i processi diventavano via via sempre più pesanti, i linguaggi li seguirono accrescendo le proprie funzionalità.
Finalmente, con i modelli incrementali, si arrivò a capire che rendere il processo ancora più complesso non avrebbe giovato: il sistema era diventato troppo complesso e occorreva trovare qualche altro approccio e semplificare il modello di sviluppo per sperare in ulteriori miglioramenti.
I linguaggi continuarono a crescere, in complessità, fino al proverbiale C++. Una divertente falsa intervista, con Stroustrup[11], l'inventore del C++, evidenzia come la comunità degli sviluppatori abbia percepito la complessità di questo linguaggio.
Alla fine, ci si rese conto che la complessità del linguaggio non era più sostenibile: semplicemente costava troppo formare sviluppatori per tali linguaggi o mantenere sistemi scritti in tali linguaggi. Con Java, iniziò la parabola discendente di complessità.
Quanto durerà la discesa? Chi scrive pensa che continuerà ancora. La diffusione dei processi agili è irreversibile in tutti i settori del mondo produttivo ed essa è solo agli inizi nell'industria informatica. Essa porta il bisogno di eliminare lo spreco nella programmazione.
L'eliminazione delle attività inutili è, quindi, un cardine delle future tendenze. Essa potrà realizzarsi come semplificazione dei linguaggi di programmazione, ovvero come l'uso di quei linguaggi che hanno sintassi ridotte al minimo, come Smalltalk.
Essa potrà anche apparire sottoforma di nuovi sistemi integrati per lo sviluppo, che automatizzino o semiautomatizzino le attività e i controlli del programmatore. Si pensi, per esempio, all'efficacia di JUnit in Java per il test e di Javadoc per la documentazione.
Si può, quindi, fare una semplice considera-

[10] ML fu il primo linguaggio funzionale ad avere una diffusione oltre la ristretta cerchia degli inventori.

[11] URL: http://www.nsbasic.com/newton/info/nsbasic/interview.shtml

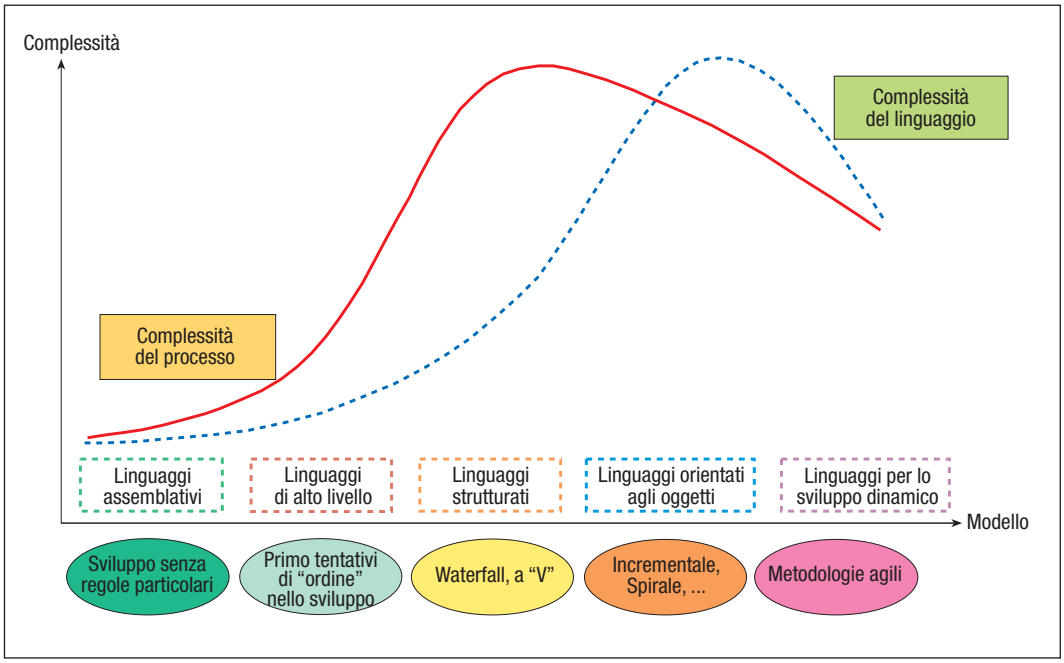


FIGURA 3
Evoluzione della complessità nel processo e nel linguaggio

zione, quasi scherzosa. Sicuramente, chiunque profetizzò le fortune del FORTRAN[12], o meglio, del suo nome e della struttura della sua sintassi, ebbe ragione. Questo è il mondo del FORTRAN e non si sarebbe affatto stupiti di trovarsi un FORTRAN 3000 "tra i piedi" tra qualche anno...
Di fatto, la storia dell'uso dei linguaggi di programmazione condiziona quelli che saranno i linguaggi in uso nel futuro. Gli economisti chiamano questo fenomeno *path dependency* [23].
Un esempio classico è il *layout* della tastiera della macchina da scrivere e del computer, la cui inefficienza è provata rispetto a soluzioni alternative, ma che risulta non conveniente cambiare in quanto il costo di transizione sarebbe maggiore del guadagno ottenuto cambiando.
La fortuna della famiglia dei linguaggi basati sul primo FORTRAN o di quelli basati sul C sta probabilmente in questo. Potrebbero esistere altri sistemi radicalmente diversi e superiori per scrivere codice. Per esempio, l'autore pensa che Smalltalk sia estremamente più semplice e compatto. Di fatto, la transizione a una sintassi diversa rende questo percorso di difficile praticabilità, forse impossibile.
Resta, dunque, sempre più attuale la strategia di *cambiare il linguaggio piuttosto che cambiare linguaggio*. C++ e Java si stanno muovendo in questa direzione.

Nell'articolo si è parlato in svariati punti del concetto di "*tipo di dato*".
I tipi di dati rispondono a due scopi.
Gli ingegneri del software hanno sempre nutrito il desiderio impossibile di poter controllare in modo statico la correttezza semantica di un programma. Di fatto, le uniche verifiche fattibili sono quelle sintattiche. In questo contesto si è cercato di raffinare sempre più ciò che è sintatticamente esprimibile, al fine di massimizzare quanto sia controllabile prima dell'esecuzione di un programma.
Il primo scopo dei "tipi di dato" è proprio quello di aumentare le parti controllabili tramite la definizione di vincoli sulla operazioni fattibili.
Così come, ad esempio, in matematica per ogni tipo di numero (naturale, intero, razionale ecc.) esistono un insieme di operazioni ammissibili, un dominio per tali operazioni e un co-dominio, con i tipi di dati si vuole fornire all'utente la possibilità di definire tipi come quelli matematici e operazioni ammissibili, in modo tale che, laddove si compia una operazione non ammissibile, un controllo sintattico possa tempestivamente informare lo sviluppatore.
I tipi di dati utilizzabili in un programma si sono evoluti da semplice combinazioni di più valori elementari, come nel primo FORTRAN, a complesse algebre di ordine superiore con relazioni funzionali e definizioni anche ricorsive, come in C++.
Anche nel caso dei tipi di dati, passando da C++ a Java si è assistito a una progressiva semplificazione dei tipi ammissibili.
Il secondo scopo dei tipi è stato quello di fornire un supporto alla modularità, tramite l'astrazione dei dati. Di questo si discute anche nella parte finale dell'articolo.

[12] Si fa riferimento alla citazione sulle fortune del nome FORTRAN, il cui autore, come detto in nota 2, non è determinato con certezza.

Bibliografia

[1] Backus J.W.: *Automatic Programming: Properties and Performances of FORTRAN Systems I and II*. Atti del Symposium of Mechanisation of Thought Processes, Teddington, UK, 1958.

[2] Backus J.W.: The History of FORTRAN I, II, and III. In: *IEEE Annals of the History of Computing*, Vol. 20, n. 4, 1998, p. 68-78.

[3] Banham R., Newman P.: *The Ford Century: Ford Motor Company and the Innovations That Shaped the World*. Artisan Sales, 2002.

[4] Broy M., Krieg-Brückner B.: Derivation of Invariant Assertions During Program Development by Transformation. *Transactions on Programming Languages and Systems*, Vol. 2, n. 3, 1980, p. 321-337.

[5] Budd T.: *An Introduction to Object-Oriented Programming*. Addison Wesley, 1991.

[6] Cox B.: *Object Oriented Programming – An Evolutionary Approach*. Addison Wesley, 1986.

[7] Chidamber S.R., Kemerer C.F.: A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, Vol. 20, n. 6, June 1994, p. 476-493.

[8] Church A.: *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[9] Dahl O.-J., Nygaard K.: SIMULA –An Algol Based Simulation Language. *Communications of the ACM*, Vol. 9, n. 9, 1966, p. 671-678.

[10] Dijkstra E.W.: *Primer of Algol 60 Programming*. Academic Press, 1962.

[11] Dodrill G.: *Coronado Enterprises Modula-2 Tutor*. 1987, URL: http://www.modula2.org/tutor/

[12] Ellis T., Miles R.: *A Structured Approach to FORTRAN 77 Programming*. Addison-Wesley Publishing, 1983.

[13] Ellis T., Miles R.: *FORTRAN 90 Programming*. Addison-Wesley Publishing, 1994.

[14] Feuer A.R., Gehani N.H.: Comparison of the Programming Languages C and Pascal. *ACM Computing Surveys*, January, 1982.

[15] Gehani N.H.: *Unix Ada Programming*. Prentice Hall, 1985.

[16] Goldberg A., Robson D.: *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1983.

[17] Graham P.: *History of FORTRAN*, URL: http://www.paulgraham.com/history.html

[18] Hewitt C., Bishop P., Steiger R.: *A Universal Modular Actor Formalism*. Atti della 3rd International Joint Conference on Artificial Intelligence, Stanford, CA, 1973.

[19] Hansen P.B.: The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, Vol. 1, n. 2, 1975, p; 199-207.

[20] Jensen K., Wirth N.: *PASCAL User Manual and Report*. Springer, 1975.

[21] Kernighan B.W., Ritchie D: *The C Programming Language*. Prentice Hall, 1978.

[22] Kruchten P.: *The Rational Unified Process: An Introduction*. Addison Wesley, 1998.

[23] Liebowitz S.J., StMargolis S.E.: Path Dependence, Lock-in, and History. *Journal of Law, Economics, and Organization*; 1995, anche disponibile on-line all'URL: http://wwwpub.utdallas.edu/~liebowit/paths.html

[24] McCarthy J.: *A Revised Version of MAPLIST*. MIT AI Lab., AI Memo n. 2, Cambridge, September 1958.

[25] McCracken D.D.: *A Guide to Cobol Programming*. Wiley, 1963.

[26] Meyer B.: *Eiffel: The Language*. Prentice Hall, 1992.

[27] Murtagh J.L., Hamilton J.A.: A comparison of Ada and Pascal in an introductory computer science course. *ACM SIGAda Ada Letters*, Vol. 18, n. 6, 1998.

[28] Reid J.: *The New Features of Fortran 2000*. URL: www.ukhec.ac.uk/publications/reports/N1507.pdf

[29] Rumbaugh J., Jacobson I., Booch G.: *The UML Reference Manual*. Addison Wesley, 1998.

[30] Sethi R.: *Programming Languages: Concepts and Constructs*. Addison Wesley, 1996.

[31] Stroustrup B.: *The C++ Programming Language*. Addison Wesley, 1996.

[32] Tsaptsinos D., Davies R., Rea A.: *Fortran 90 – An introduction to the language for beginners*. URL: http://www.pcc.qub.ac.uk/tec/courses/f90/ohp/header_ohMIF_1.html

[33] US Department of Defence: *Department of Defense Requirements for High Order Computer Programming Languages: Steelman*. 1978.

[34] Wheeler D.A.: *Ada, C, C++, and Java vs. The Steelman*. Ada Letters, July/August, 1997.

[35] Wirth N.: *Programming in Modula-2*. Springer Verlag, 1982.

GIANCARLO SUCCI è ordinario di Informatica e direttore del Center for Applied Software Engineering presso la Libera Università di Bolzano, dove si occupa di metodologie agili per lo sviluppo software, metriche software, ingegneria del software empirica, strumenti di e-learning nell'industria software, linee di prodotto software. È stato ricercatore presso l'Università di Trento, professore associato alla University of Calgary e quindi professore ordinario alla University of Alberta, dove ha co-diretto l'Alberta Software Engineering Research Consortium. È autore di più di 150 articoli scientifici e di 5 libri.
e-mail: giancarlo.succi@unibz.it