

Estetica degli artefatti digitali

© Stefano Penge - Creative Commons BY/SA/NC

Sommario

Estetica degli artefatti digitali.....	1
1. Programmare poeticamente.....	1
2. Leggere poeticamente.....	3
3. Codice sorgente come opera d'arte.....	4
4. Estetica del codice sorgente.....	6
6. Risultati.....	8

Questo articolo fa parte di una serie di interventi che si propongono il difficile compito di indagare la possibilità di applicare anche ai codici sorgenti dei programmi per calcolatore alcuni concetti, tecniche e teorie che sono state sviluppate in questi anni a proposito dei testi linguistici più tradizionali. Si tratta di un compito che si è dato come programmatico il Laboratorio di Linguistica degli Artefatti Digitali della facoltà di Scienze della Comunicazione dell'Università La Sapienza.

All'interno di questa sfida generale, ci occuperemo qui di una questione specifica, che però finisce per andare a toccare tutte le altre e in qualche modo le rappresenta. O meglio, tenteremo di toccare i diversi punti del problema generale dall'angolazione degli studi estetici.

La questione da cui partiamo è sia possibile un'estetica della programmazione. Non un'estetica degli artefatti digitali intesi come prodotti finali realizzati tramite computer e fruibili con i nostri sensi (“computer graphics”, “computer music” [...]), ma proprio degli artefatti digitali primari, in se stessi, allo stadio originario di codice sorgente. Più precisamente:

- è possibile produrre (scrivere) codice sorgente con intenti (metodi, contesti...) estetici?
- è possibile considerare (leggere) esteticamente del codice sorgente?

Le due questioni sono solo apparentemente simili, e fanno riferimento a due modi diversi di intendere l'espressione “digital poetry”, che possiamo indicare con “poesia digitale” e con “programmare poeticamente”. In un caso dunque il centro è l'oggetto, nell'altro il processo produttivo.

1. Programmare poeticamente

In un primo senso, si può semplicemente scrivere un programma che produca un testo poetico (in cui “poetico” sta per “vincolato formalmente”, adeguato a questo o quel canone), o meglio un programma che produca infiniti testi poetici. Si tratta di un esempio di *arte generativa* [...]: viene polemicamente rotto il legame tra poesia e poeta, tra estetica e creatività individuale, per sottolineare in maniera forzata gli aspetti formali, universali, dunque ripetibili, per disfarsene (Dada) o per assumerli come spazio di ricerca. Questa è la strada più vicina all'intuizione originale dell'OuLiPo, l'officina di letteratura potenziale che ha formalizzato l'estetica del vincolo; da questo punto di vista, “Cent Mille Millions de Poèmes” di Raymond Queneau (1960) è una versione “carta e inchiostro” di un programma che faccia esattamente la stessa cosa, cioè selezioni casualmente dieci versi da un archivio di 10 sonetti di 14 versi. Il numero totale di sonetti costruibili in questo modo è di 10^{14} (100,000,000,000,000). Il libro in realtà è una macchina: o meglio, lo è l'insieme

costituito dal libro più le istruzioni per l'uso, più l'umano che lo utilizza sfogliandolo selettivamente.

Diverse versioni digitali sono state realizzate in seguito di questo *concept*;¹ la cosa interessante della versione software è naturalmente proprio il fatto che in questi casi sia un testo (un codice sorgente) a produrre altri testi.

Negli stessi anni dell'OuLiPo, altri passi in questa direzione erano stati mossi un po' ovunque, dalla Germania alla Francia agli Stati Uniti [Lutz, Sutcliffe, Gysin ...], fino alla nascita dell'ALAMO (Atelier de Littérature Assistée par la Mathématique et l'Ordinateur) agli inizi degli anni '80 [Balpe].

In questo tipo di operazione, viene cancellato l'autore individuale [Lyotard] e viene portato in primo piano il sistema produttivo, inteso come insieme di algoritmi più dizionario. Chiaramente, la poetica di riferimento non è indifferente: Florian Cramer, uno degli studiosi che più si è avvicinato allo studio del codice sorgente da un punto di vista estetico, mostra come scrivere un programma (in PERL) che produca poesia secondo il modello dadaista [Tzara ...]².

Il passaggio da un sistema fisico, finito, ad un sistema virtuale, potenzialmente infinito, rappresenta anche un punto di svolta per tutta la letteratura combinatoria [...]. Se la letteratura tradizionale ha prodotto solo esemplificazioni abbreviate di testi infiniti prodotti dall'incrocio di elementi finiti, o ha semplicemente rappresentato al suo interno questa produzione infinita (come la macchina dell'Accademia di Lagado e la Biblioteca di Babele), la macchina computer sembra invece essere lo strumento ideale per la realizzazione di una vera opera d'arte combinatoria:

- non produce un testo singolo, ma un flusso testuale che (nel tempo) può essere infinito;
- non acquisisce i suoi elementi di partenza da un repertorio finito, ma da una sorgente continuamente rinnovabile (ad esempio, siti web, lo spam, etc):
- può utilizzare i dati elaborati per modificare le stesse regole con le quali procede.

Da questo punto di vista, un software di questo tipo è anche una metafora generale dell'azione poetica, che è sempre intertestuale [Kristeva...] e usa continuamente testi esistenti per produrre altri testi.³

L'altra maniera di intendere l'espressione “programmare poeticamente” è più rilevante per il nostro discorso: si tratta di scrivere programmi che siano *essi stessi* testi poetici (e che, eventualmente, con la loro esecuzione producano altri testi). Naturalmente non i programmi nella loro versione finita, pronta per l'esecuzione o addirittura già in esecuzione, ma la versione originaria, primitiva, sorgente. In qualche modo, il testo fuori dal tempo.

Qui la questione del carattere estetico del codice sorgente diventa più rilevante.

Alla prima domanda, se sia possibile produrre (scrivere) codice sorgente con intenti (metodi, contesti...) estetici, si può da questo punto di vista rispondere con l'esibizione di una serie di esempi. Il valore dell'esempio naturalmente è solo quello di mostrare un intento, non un risultato; se i prodotti siano oggetti estetici a tutti gli effetti è questione diversa, più generale, nella quale qui però non abbiamo bisogno di entrare.

Nel 1962 l'OuLiPo dichiara tra i propri obiettivi quello di scrivere poesie con il linguaggio Algol. Dieci anni dopo Noel Arnaud scrive effettivamente un testo poetico usando il lessico di quel linguaggio di programmazione [...].

Agli inizi degli anni '90, all'interno della comunità di programmatori PERL, probabilmente del

1 <http://www.parole.tv/cento.asp>; http://www.bevrowe.info/Queneau/QueneauRandom_v4.html ;
<http://www.smullyan.org/smulloni/queneau/index.html>

2 http://www.netzliteratur.net/cramer/concepts_notations_software_art.html

3 Una variante interessante della macchina che riproduce se stessa è ovviamente il virus; e infatti sul virus come oggetto estetico è stato scritto più volte [Jaromil, Tozzi...].

tutto all'oscuro dei rischiosi tentativi del gruppo degli ouilipiani e di quanto era seguito nei venti anni trascorsi [Larry Wall ...], nasce come sfida, come divertimento, l'idea di scrivere programmi privi di scopo ma che avessero chiare somiglianze formali e contenutistiche con testi poetici, in particolare con la forma minimalista dell'haiku. Due idee paradossali: la prima, quella di scrivere programmi senza obiettivo, nega l'essenza stessa della programmazione così come è insegnata ovunque. Programmare è “tradurre in un linguaggio un algoritmo che risolve un problema dato”; qui non abbiamo il problema, e l'algoritmo sembra perdere importanza nei confronti del linguaggio in cui è scritto. La seconda idea è quella di condurre il lettore colto a cogliere delle somiglianze tra alcune forme d'arte poetica (non a caso, una forma che si confa alla limitatezza del dizionario di un linguaggio come il PERL, circa 250 termini) e delle “forme di codice”, che soddisfino tutti i requisiti formali di un codice sorgente, e in particolare l'essere effettivamente eseguibile su qualche macchina.⁴

La Perl Poetry nasce come gioco, come dimostrazione di abilità, e si apparenta in questo alle gare di “codice offuscato” [...], in cui il programmatore ha come obiettivo quello di scrivere codice incomprensibile ma funzionante. Tuttavia alcuni prodotti della PERL poetry sembrano andare al di là dei limiti del gioco formale, soprattutto in quanto sfruttano le caratteristiche non solo del lessico ma anche della sintassi e del modello di funzionamento di un linguaggio [Alan Sondheim, Sharon Hopkins ...].

Ciò che differenzia questi tentativi da quelli “originali” è probabilmente l'idea di comunità di lettori/autori che essi presuppongono. Se gli exploit degli anni '70/'80 erano opere senza pubblico, la Perl Poetry invece acquista senso solo perché, nel frattempo, è nata una generazione di lettori sufficientemente esperti che sono in grado di apprezzare le sfumature nell'uso del linguaggio. Questa diversa situazione storica rende possibile la transizione dalla produzione automatica di testi poetici alla scrittura di programmi poetici.

2. Leggere poeticamente

La seconda domanda, se sia possibile considerare (leggere) esteticamente del codice sorgente, è più insidiosa perché implica l'estensione dei criteri di giudizio di un'opera d'arte (prodotta manualmente e destinata ad un fruitore umano) a oggetti speciali che si differenziano dai testi poetici almeno per quattro ragioni:

- non sono scritti in nessuna lingua naturale conosciuta;
- non sono declamabili;
- la loro qualità estetica (ammesso che ci sia) è un effetto collaterale del loro utilizzo normale;
- non sono destinati ad un lettore umano.

Insidiosa, perché costringe a definire i termini del discorso in maniera precisa e a immaginare immediatamente controesempi, alcuni ovvi e sotto gli occhi di tutti, altri meno.

Esistono poesie *non* scritte in lingue naturali? Viene immediatamente da pensare alle lingue artificiali come quella in cui Tolkien scrive i poemi elfici, o linguaggio transmentale di Chlebnikov; o, da un altro punto di vista e secondo un'altra accezione di “lingua artificiale”, alle poesie in esperanto⁵ o in volapuk.

Esistono poesie non leggibili, nel senso di non *declamabili* ad alta voce? Non so se attualmente esistano esempi di poesia in lingua dei segni, ma so per certo che esistono barzellette che i sordi si raccontano e che nascono, come ogni forma di umorismo, dalle caratteristiche specifiche di quella lingua.

4 Sharon Hopkins "Camels and Needles: Computer Poetry Meets the Perl Programming Language"
<http://www.wall.org/~sharon/plpaper.ps>

5 http://it.wikipedia.org/wiki/Pre%C4%9Do_sub_la_Verda_Standardo

Esistono testi poetici che hanno uno scopo altro dal puro godimento estetico? Qui siamo nel regno ben noto dello slogan e del testo pubblicitario, per non entrare nemmeno nel dominio della retorica e del discorso che utilizza strumenti linguistici per muovere gli animi.

L'ultimo punto (il destinatario non umano) merita un esame attento. Dobbiamo distinguere tra lettura ed esecuzione, che nel caso della poesia sembrano coincidere, ma non in altri campi delle arti del linguaggio, come il teatro o la musica.

La nostra esperienza di lettori adulti ci porta ad identificare lettura ed esecuzione in un senso ancora più profondo. Apparentemente, la lettura sembrerebbe l'esecuzione di un programma, e ogni testo non sarebbe altro che un programma di cui noi, lettori, siamo l'hardware. In realtà ogni testo può essere eseguito in molti modi diversi in base alle infinite variabili di contesto: letto mentalmente, letto a voce alta, declamato, citato, sillabato, E' il lettore, almeno nel senso di Eco, che crea l'opera.

Anche il codice sorgente di un programma viene di norma scritto per essere letto (e non eseguito) da molti soggetti diversi:

- l'autore stesso in un secondo momento
- coautori
- revisori successivi
- studenti
- studiosi di linguaggi
- storici della programmazione

oltre che da un software, un interprete o da un compilatore in funzione della esecuzione successiva.

Allo stesso modo uno spartito e un copione vengono letti (in quanto questa operazione non coincide con l'esecuzione) dalla stessa persona in momenti diversi, e da persone diverse nello stesso momento. E in alcuni casi la partitura viene scritta per una doppia lettura, visiva e musicale in senso stretto.

In realtà non è così unica questa natura doppia, di oggetto/programma, che sembra caratterizzare il codice sorgente. Anche senza pensare ai testi performativi [...], che sono scritti per produrre effetti reali, o a quelli generativi per antonomasia, come i manuali di scrittura, qualunque testo può essere scritto e pensato come fine a se stesso, unico, o come motore di altri testi, quindi universale. La parola orale è unica, contestualizzata, legata all'evento del suo essere pronunciata. Per renderla universale, cioè forma, occorre ricorrere agli artifici della grammatica (il futuro, l'imperativo, i quantificatori universali). La parola scritta invece è già forma, pronta per essere riapplicata in contesti diversi indipendentemente dalla sua struttura. Il testo, propriamente, nasce appunto come universale, e con lui la *langue* si differenzia dalla *parole*. Con l'invenzione della stampa questa suddivisione si ripete all'interno del testo stesso: per la prima volta viene separato il testo come "programma" (il cliché) dal testo come oggetto diretto di fruizione (la pagina stampata). Due supporti diversi, due contesti di fruizione/produzione diversi.

Il codice sorgente di un programma assomiglia molto al cliché del tipografo. Come quello, è invisibile per il fruitore finale (o almeno lo è nel contesto economico dominante: il movimento del software libero e dell'opensource puntano forse in altra direzione), è fluido, modificabile, unico. Con alcune differenze: la principale è che il codice sorgente è fatto della stessa materia della sua versione compilata, e il suo contesto d'uso è spesso lo stesso di quello del suo prodotto (tastiera e monitor di un computer). Il codice sorgente nasce innanzitutto come configurazione in una memoria volatile, e quasi simultaneamente come immagine su uno schermo; solo in seguito assume esistenza separata come informazione depositata su un supporto (memoria non volatile). Ma è chiarissimo

che il codice sorgente mantiene una sua esistenza indipendente dal supporto (a differenza della pagina di un testo tradizionale), tant'è che si può ricopiare infinite volte senza produrre differenze percepibili. E' questa indistinguibilità dell'originale dalla copia che fonda, in positivo e in negativo, tutta l'economia del digitale.

3. Codice sorgente come opera d'arte

Dunque considerare il codice sorgente, al limite, come opera d'arte. Quindi artisti, esposizioni, critica, mercato. E opera d'arte implica stile, scuola, moda. Dove sono i corrispettivi di questi concetti?

Prima di tutto possiamo prendere in considerazione (e velocemente mettere da parte) l'*arte digitale concettuale*, cioè l'uso di lettere e segni come elementi puramente visivi privi di significato. La cosiddetta ASCII art [...], che nasce con una tecnologia semplice come quella della telescrivente, e rinasce più volte con i fax e le e-mail, è interessante soprattutto perché poggia su una diversa operazione di fruizione che non è una lettura. E' noto che un opera di ASCII art per essere fruita deve essere guardata ad una certa distanza, in modo da rendere visibili le forme e nascosti i simboli che compongono le forme stesse. D'altra parte, il godimento estetico di queste opere nasce esattamente nel limite in cui si passa da un'operazione all'altra, dalla lettura allo sguardo d'insieme e viceversa; è anzi, potremmo dire, nell'oscillazione tra le due operazioni. Sarebbe sicuramente interessante fare un parallelo con le opere in Braille-art [...].

Prosegue in qualche modo il discorso dell'ASCII art il filone di arte contemporanea che utilizza come suoi materiali non semplici simboli ma frammenti di comunicazione che passano sulla rete, la cosiddetta *net-art* [...].

Qui ci troviamo però in un settore che in realtà ha poco a che fare con la questione del codice sorgente. Tuttavia il passaggio per la disposizione bidimensionale di un testo sequenziale ci porta ad occuparci della impaginazione tipografica, che sovrappone un codice visivo ad un codice linguistico. L'impaginazione è la maniera più semplice di tradurre indicazioni prosodiche, relative alla performance del testo, in maniera visiva. L'a-capo alla fine del verso utilizzato nella trascrizione di una prosa in poesia corrisponde ad una pausa musicale; la riga di separazione tra due quartine corrisponde ad una pausa più lunga. Il diverso allineamento dei versi serve a sottolineare la struttura metrica e a facilitare la gestione degli accenti.

Se teniamo come riferimento sullo sfondo le opere di Hrabanus Maurus o i calligrammi di Apollinaire e le poesie futuriste, che dell'impaginazione fanno un uso molto più intenso, possiamo forse considerare anche in questo caso la disposizione delle parole e dei segni sulla pagina non solo un artificio al servizio della lettura, ma un elemento estetico che si appresenta in qualche modo allo stile.

Nella scrittura del codice sorgente di un programma, gli elementi dell'impaginazione (a capo, righe vuote, indentazione e tabulazioni) nascono come indicazioni visive e facilitazioni per il lettore/correttore. Possono però diventare essi stessi elementi che richiamando l'esperienza di lettura di poesie del lettore sottolineano e dichiarano gli intenti estetici dell'autore. Significativo è anche il fatto che esistano delle piccole librerie di funzioni dette, non a caso, "beautifiers" che permettono di visualizzare diversamente lo stesso codice sorgente (in termini di font, colori, dimensioni) a seconda del linguaggio a cui appartiene, ma anche dello stile generale dell'interfaccia utilizzata dall'utente/lettore sul suo computer, e in ultima analisi dei suoi gusti. Significativo perché spostano sul *lettore* l'onere dell'estetizzazione del testo.

L'aspetto che richiama più fortemente il concetto di stile è forse quello degli standard dei linguaggi di programmazione. "Standard" in questo contesto significa che al di sopra delle regole sintattiche del linguaggio (la cui violazione conduce al syntax error) esistono delle regole altrettanto

formali e specifiche che però sono facoltative; riguardano la formattazione del testo, ovvero il numero di spazi o di tabulazioni prima di una linea di codice, dove andare a capo, quando usare le maiuscole, se usare spazi prima o dopo la punteggiatura e le parentesi, come scrivere i commenti, etc. Si tratta di regolare l'occupazione mediante forme dello spazio della differenza non significativa per l'interprete, il quale semplicemente ignora spazi, tabulazioni e commenti. Scrivere buon codice significa quasi sempre aderire ad uno standard, soprattutto se il codice deve far parte di un progetto più grande in cui partecipando diversi autori. Non rispettare uno standard significa essere dei *parvenus*, dei rozzi macinatori di codice, che non appartengono a nessuna comunità. A differenza delle regole sintattiche, gli standard hanno una vita indipendente dai linguaggi, trasversale ad essi; più standard possono essere in concorrenza tra loro e forse questo è l'elemento che li rende più simili a stili estetici e a mode [...].

L'ultima grande famiglia di fenomeni candidata per sostenere il ruolo di stile è quella dei cosiddetti “paradigmi di programmazione”. È noto che i primi linguaggi di alto livello utilizzavano una metafora del computer come robot che deve essere istruito e controllato attraverso appunto “istruzioni” e “controlli”. Successivamente si sono però affermate metafore differenti: il computer viene visto come macchina calcolatrice universale (paradigma della programmazione funzionale), come risolutore di problemi (paradigma della programmazione logica), come insieme di attori programmabili e interagenti (paradigma della programmazione orientata agli oggetti). Lo stesso problema può essere risolto diversamente, costruendo diverse strutture dati e segmentando diversamente i compiti, a seconda del paradigma scelto. Lo stesso algoritmo può essere scritto in modi diversi (è anzi un topos classico dei manuali di programmazione mostrare come si calcola l'ennesimo numero di fibonacci nei diversi paradigmi/linguaggi).

Il paradigma è legato molto strettamente alla sintassi del linguaggio; il linguaggio Java, ad esempio, è fortemente legato al paradigma degli oggetti e dei metodi. Scegliere un linguaggio per un progetto significa quindi implicitamente scegliere un paradigma. Tuttavia spesso lo stesso linguaggio – soprattutto quelli più recenti - permette di far convivere uno stile funzionale e uno object-oriented, anche all'interno dello stesso programma. Questo significa che può essere il programmatore a scegliere, sulla base di sue esperienze precedenti, preferenze, ideologie, il paradigma che preferisce. Per quanto sia stato ampiamente dimostrato, ad esempio, che il modello ad oggetti – e i linguaggi che lo adottano - dal punto di vista dell'efficienza non è ideale, sempre più programmatori lo apprendono e lo preferiscono, in nome di una “qualità estetica” superiore.

4. Estetica del codice sorgente

Questo senso di “qualità estetica” del codice sorgente è stato portato alla luce da autori come D. Knuth e S. Levy, ma che è molto sentito nelle comunità di programmatori.

Viene fatto riferimento al carattere “estetico” di un codice sorgente nello stesso modo in cui si parla di “bellezza di un teorema matematico”. Qui il modello estetico che si tiene presente è quello dell'arte come rappresentazione del Bello. Parlare di “bel codice” è un modo per sottolinearne gli aspetti che per convenzione escono dalle considerazioni di efficienza, ma che innegabilmente per la maggioranza dei programmatori sono importanti nella definizione del valore di un codice sorgente:

- chiarezza, comprensibilità
- equilibrio tra le parti
- uso di standard
- modularizzazione e riuso
- ...

Queste qualità possono essere considerate positive in se stesse (Levy), oppure solo in quanto a

loro volta implicano e significano aspetti quantitativamente misurabili, come la diminuzione del tempo necessario per scrittura o per la manutenzione del codice, soprattutto se da parte di molti programmatori (Knuth).

Si tratta di un'operazione che se dal punto di vista culturale è piuttosto rozza (viene preso un modello storicamente determinato di arte e di bello e viene utilizzato come modello universale trasportandolo in un campo nuovo), dal punto di vista che stiamo accogliendo qui ha invece un'enorme importanza perché sottrae tutta l'opera del programmatore (non solo quella dell'artista digitale) al dominio dell'efficienza e dell'efficacia e la ricolloca all'interno del più vasto campo delle produzioni linguistiche professionali e non, dalla scrittura di una lettera alla creazione di uno slogan.

Se, in altre parole, il campo dell'estetica non è solo quello delle opere d'arte, c'è materia per uno studio estetico dei codici sorgenti - di tutti i codici sorgenti, non solo di quelli creati con intento artistico.

La questione dell'estetica della programmazione è stata posta forse per la prima volta in maniera esplicita da Pierre Lévy.⁶

L'aspetto interessante dell'approccio di Lévy è dichiarato già nel titolo, che pone l'accento sul processo più che sul prodotto. Non è tanto il codice ad essere riconoscibile come opera d'arte ma la programmazione stessa ad essere iscritta nel novero delle belle arti.

La premessa fondamentale è la considerazione degli artefatti digitali come testi apparentati a tutti gli altri prodotti di scrittura umana.

La seconda premessa è che i gradi di libertà permessi da un qualsiasi linguaggio di programmazione sono molti di più di quelli comunemente immaginati dai profani o dai professionisti della "buona programmazione" e consentono di produrre infinite versioni diverse dello stesso testo.

Sulla prima premessa non dovrebbe esserci molta esitazione. La programmazione è stata a lungo, ed è ancora in grandissima parte, affare di scrittura diretta. Esistono, marginalmente, sistemi di produzione di programmi diversi dalla scrittura (programmazione visuale), ed esistono funzioni avanzate di riuso e citazione tramite scorciatoie di pezzi di un testo all'interno di un altro, ma la maniera normale di creare un programma è proprio battendo su una tastiera lettera per lettera, parola per parola, riga per riga. Quello che non è immediatamente evidente è che la programmazione è anche scrittura nel senso più esteso del termine. La scrittura di un testo (saggistico, poetico, commerciale) passa per fasi diverse, alcune delle quali non affidate ad operazioni meccaniche di aggiunta sequenziale di segni alfanumerici: l'ideazione, la progettazione, la revisione, lo spostamento, il riuso. Anche la produzione del codice di un programma passa per queste fasi, anche quando queste non siano codificate da protocolli e regole. Un programmatore "vede" il programma che sta scrivendo come un testo dotato di unitarietà e complessità, composto da parti con ruoli ben definiti. Un autore di programmi è alle prese con le stesse variabili di un autore di testi narrativi. Si trova di fronte scelte di stili, di suddivisione, di equilibrio, di introduzione degli attori, di contestualizzazione.

Questo spazio della differenza è l'oggetto di questa analisi.

La seconda premessa è apparentemente più difficile da accettare, sia per chi non si sia mai occupato di programmazione, sia per chi invece sia esperto di questa materia.

I primi considerano la fase di produzione dei programmi come un'attività meccanica, al massimo parente della scrittura della dimostrazione di un teorema.

Immaginano la scrittura dei programmi come una specie di trascrizione automatica di un ragionamento in un codice (non una vera lingua, ma appunto un "linguaggio"). Non riconoscono ai

6 De la programmation considérée comme un des beaux arts, 1992, Paris, La Découverte.

linguaggi di programmazione una semantica come quelle degli altri linguaggi verbali. Non apparentano la programmazione con la scrittura teatrale, ma al massimo con la matematica, sulla base della limitatezza del lessico, della assenza di ambiguità, della rigidità della sintassi, dell'opacità dei simboli. La parentela con la scrittura musicale (che usa un linguaggio ancora più chiuso) viene invece esclusa, se mai è presa in considerazione, sulla base della mancanza degli effetti emotivi del prodotto finale, con un cortocircuito tra scrittura ed esecuzione. Buona parte della responsabilità di questa visione parziale sta dall'altro lato del campo, dove poco si è fatto per comunicare, ad esempio, l'enorme varietà di linguaggi di programmazione esistenti (ne sono stati recensiti circa 2500), e la presenza di esempi sostanziosi di linguaggi inventati per puro piacere (linguaggi cosiddetti "esoterici", ...).

I secondi – gli esperti di programmazione - sanno bene che i diversi linguaggi di programmazione hanno potenza espressiva diversa, ma sono spesso convinti che sia possibile definire a priori "la" maniera migliore di codificare l'algoritmo che risolve il problema, e che questa maniera sia insegnabile. Rinunciare a questa convinzione significherebbe, a loro avviso, rinunciare alla possibilità di migliorare il codice sorgente. E' una posizione di difesa, di casta, che forse potrebbe guadagnare in universalità con il riconoscimento degli infiniti margini di miglioramento possibili che si aprono quando si guarda al codice sorgente come un testo in senso pieno.

Se invece assumiamo questa differenza come significativa, ci si aprono nuovi orizzonti di indagine. Dal fatto bruto che un'istanza di codice sorgente sia *scritta*, possiamo derivare una serie di questioni-guida che aprono nuovi campi di lavoro:

- scritti in un certo momento → storia degli stili, scuole, mode
- scritti in un certo luogo → geografia culturale degli stili, ...
- scritti da qualcuno → stilistica, authorship analysis
- scritti per essere letti da qualcuno → retorica, estetica, leggibilità

Se il contesto (culturale/storico) influenza la produzione di testi musicali, poetici, etc, perché gli artefatti digitali dovrebbero sfuggire a questa regola?

Se esistono configurazioni di tratti minimi stilistici riconoscibili che sono comuni a gruppi di autori e di testi, potrebbero esistere configurazioni personali (idioletti)?

Nell'analisi di un codice sorgente dovrebbe essere possibile marcare un testo sulla base di alcuni indicatori. Per esempio:

- Età / classe sociale dell'autore
- Lingua madre dell'autore / del gruppo di lavoro
- Periodo storico / scuola di appartenenza
- Contesto di lavoro / tipologia professionale
- Sesso
- Stili, abitudini, altre competenze...

E più in profondità, si potrebbero utilizzare degli indicatori lessicali:

- aspetti nominali delle variabili, parametri, costanti
- aspetti verbali delle funzioni, dei programmi /moduli
- aspetti personali nei commenti

che potrebbero dire qualcosa sul grado di identificazione dell'autore con il programma, a partire

dall'uso della prima persona o della terza persona, del tempo presente o futuro, dell'uso di metafore.

Un livello di analisi più sofisticato è quello che fa uso di strumenti e concetti della linguistica attanziale [Greimas...]. Ad esempio, in un programma che usa il paradigma client/server (come ogni applicazione web) si potrebbero ricercare elementi che richiamano un dialogo tra autore e testo, o una vera e propria “avventura” in cui il modulo, ricevuti i parametri di avvio, cerca di raggiungere il suo completamento naturale evitando ostacoli (errori), utilizzando il supporto di aiutanti (funzioni collaterali), trasformando e consegnando alla fine un testimone (risultati) ai moduli che seguiranno.

Un programma verrebbe esplicitamente visto come una narrazione, come una storia.

[...]

6. Risultati

Quali risultati ci si aspetta di ottenere? A che serve un'indagine di questo tipo?

Per certi versi, è la domanda ad essere strana. Nessuno si domanda a cosa serve studiare la storia della composizione musicale, o la storia delle scuole pittoriche occidentali. Nessuno si domanda a cosa serve lo studio della nascita dello stile manzoniano condotta attraverso l'analisi dei brogli dei Promessi Sposi.

Se una domanda del genere nasce, significa che ancora teniamo la programmazione nel novero delle tecniche e non delle arti. Quanto scritto finora è appunto un tentativo di dimostrare il contrario, o almeno mostrare la sua possibilità teorica.

Ma si può tentare di rispondere più direttamente alla domanda, prospettando dei vantaggi concreti.

Rimettere la produzione di artefatti digitali al suo posto significa rendere ogni programmatore un autore consapevole del suo ruolo non di semplice operaio della catena di montaggio ma di scrittore. Il mestiere di programmatore è oggi confinato all'interno dei mestieri tecnici, per i quali si richiede una – minima - preparazione tecnico-scientifica. Tutti sono pronti a inneggiare alla genialità del programmatore solitario che inventa l'algoritmo di ricerca definitivo, ma poco viene fatto per creare le condizioni nelle quali questa singolarità diventi normalità. Si insegna a programmare bene, tenendo più bassi possibile i costi di correzione e manutenzione, ma non si insegna a scrivere codice sorgente di cui si possa andare fieri. Se la programmazione acquista valore soprattutto quando permette di risolvere problemi vecchi in nuove maniere, o addirittura problemi nuovi, a questo compito i futuri programmatori difficilmente vengono preparati adeguatamente.

Alcune modifiche (alcune proposte di curriculum didattico aumentato, se non alternativo) potrebbero essere utili a questo scopo, nei due versanti informatico e umanistico.

Nella didattica della programmazione, da un lato, si potrebbero introdurre tutti gli elementi che restituiscono ai linguaggi e ai loro usi una complessità adeguata, come le loro storie e geografie, gli obiettivi, i contesti d'uso; ad esempio, prendendo a prestito i metodi della glottodidattica, si potrebbe cercare di migliorare la maniera in cui si impara il secondo o terzo linguaggio. Dall'altro si potrebbero mostrare le parentele con le branche della linguistica e della semiotica per introdurre competenze di scrittura (nel senso ampio di progettazione del codice sorgente) più avanzate, basate su metafore più potenti (testo, narrazione, attori). Stallman (l'inventore del concetto di copyleft e della licenza GPL) sostiene che l'unico modo di imparare a scrivere programmi è leggere altri programmi. Si potrebbe estendere l'apofrosma dicendo che l'unico modo di imparare a scrivere programmi è leggere, e sotto questa bandiera restituire alla figura del futuro programmatore uno spessore “umanistico”.

Non è ancora chiaro se questo tipo di formazione trasversale dovrebbe precedere quella specifica su un certo linguaggio, o se dovrebbe completare un curriculum specialistico. Ad oggi, gli studenti

di un corso di laurea in informatica sarebbero i primi ad essere stupiti da un approccio globale all'apprendimento della scrittura di codice sorgente come quello che stiamo delineando. Sviluppano molto presto, con tutta probabilità, una visione parziale dell'universo dei linguaggi di programmazione. Non ne conoscono la genesi, le linee di sviluppo, le varianti minori, ma solo il mainstream che è funzionale al loro impiego in “batterie di programmatori” efficienti e a basso costo. Non vengono apparentemente tenuti in considerazione fatti “linguistici” come la differenza tra tipologie di linguaggi, il processo storico che ha portato a creare un linguaggio e quello che ha portato a convergere su una certa versione standard, né sono spiegate le ragioni di questo processo.

Sicuramente, e reciprocamente, una formazione su questo universo sconosciuto della natura testuale dei codice sorgenti dovrebbe essere permessa e anzi raccomandato a quanti si occupano di semiotica, di stilistica, di scienze della comunicazione. Questi studenti oggi sono nativi digitali, ma ignorano completamente ciò che si nasconde al di sotto del livello della funzione e del programma, e pur essendo altamente informatizzati sono poco consapevoli delle possibilità di personalizzazione e modifica di ogni apparato digitale che pure usano con facilità. Il web 2.0 ha data ad ogni utente il ruolo di autore di informazioni, a patto di toglierli/le quello di autore tout court. L'attenzione con cui uno studente segue e smonta i meccanismi della televisione e dei social network impallidisce di fronte alla opacità più completa della più piccola delle funzioni che rendono possibile quella forma. Non è anomalo che uno studente di scienze della comunicazione sia ben consapevole dei codici retorici della pubblicità ma sia del tutto impreparato a capire il funzionamento di un motore di ricerca che filtra i risultati e mostra solo un sottinsieme dell'universo possibile. Questa mancanza non è solo carenza di competenze tecniche, ma di abitudine a considerare un programma come un risultato di un processo di scrittura, e come tale soggetto a possibile analisi e studio comparativo.

In mezzo, il codice sorgente sfugge tanto alla consapevolezza produttiva che a quella analitica.

In secondo luogo, nella costruzione e coordinamento dei gruppi di lavoro di programmatori, si potrebbe esplicitamente tener conto delle variabili “sociolinguistiche” in maniera più efficace di quanto si faccia normalmente imponendo standard di programmazione per ragioni di abbattimento dei costi di manutenzione e controllo. Un gruppo di lavoro che condivida lo stesso stile di programmazione lavora più piacevolmente, è in grado di condividere e integrare più facilmente la propria parte di produzione[...]. Questo risultato viene generalmente raggiunto nella grandi aziende e nei grandi progetti forzatamente, imponendo uno standard di programmazione al gruppo di lavoro. In contesti più piccoli, lo standard si afferma da solo come qualità emergente. Una consapevole gestione di queste variabili permetterebbe di costruire e gestire meglio i gruppi di lavoro [...]

Infine, uno studio etnografico dell'incrocio tra testi, modi di scrittura e cultura dei programmatori potrebbe forse darci degli indicatori di valutazione della qualità di un codice sorgente al di là degli indicatori quantitativi esistenti, che hanno a che fare sostanzialmente con la leggibilità, con la modularità, con l'abbondanza dei commenti; anche in questo senso, l'industria del software avrebbe da guadagnare nel trasformare strumenti di analisi in strumenti di predizione di successo.