Capitolo 1

Introduzione

Questo corso si occupa di presentare alcune tecniche avanzate dei moderni linguaggi di programmazione, con il fine di far imparare a comprendere, ragionare e valutare.

I linguaggi di programmazione sono tantissimi (più di 703), alcuni parlano tra di loro, altri invece formano una torre di babele.
Non esiste un linguaggio migliore in assoluto, c'è invece il linguaggio più adatto a una tale situazione. Ecco che è importante saper valutare. Per poter scegliere un linguaggio è importante valutare alcuni fattori quali: efficienza, retrocompatibilità, interoperabilità (cooperare e scambiare informazioni), portabilità, correttezza, produttività (velocità di scrivere un primo prototipo rapidamente) e espressitività (capacità di scrivere in modo chiaro le linee di codice).
Inoltre è importante, quando si sceglie un linguaggio, capire cosa si deve scrivere, un piccolo *script* o un enorme base di dati per centinaia di programmatori e il tipo di programmatore che deve scrivere il linguaggio, esperto o principiante. Infatti se si costringe un programmatore a scrivere in un linguaggio a lui non gradito, lo stile del codice diventa innature, dove è più facile inseririvi dei *bug*.

Se si riesce a individuare un insieme di misurazione, si riesce a tradurre un linguaggio in numeri; ma attenzione misurare ha un significato diverso di valutare.
Si pò difatti cercare di misurare quale è il linguaggio di programmazione maggiormente in uso. Come si fa? Possono essere impiegate, per esempio, le ricerche sul *Web* svolte dagli utenti per i linguaggi, oppure il numero di progetti presenti su GitHub. Tuttavia c'è da considerare che questi possono rappresentare anche altri indicatori, magari il fatto che si discuta di un linguaggio è sinonimo di incertezze, inoltre il mondo *open-source* può avere le sue preferenze di programmazione.
Vediamo ora due indici che misurano i linguaggi di programmazione:

- **TIOBE Index**: è un indice di serietà che dichiara come sono stati calcolati gli indici. Su questa lista troviamo in ordine di apparizione:
  - Python, linguaggio con popolarità in crescita, perchè ha librerie già pronte come nel caso dell'AI, dei microcontrollori o dei sistemi *embedded* (*special purpose*);

  - Objective-C, nato assieme al C. Apple ha costretto per una sua scelta interna aziendale i programmatori a utilizzare tale linguaggio. Dal 2014 ha, tuttavia, rilasciato un nuovo linguaggio Swift (Objective-C + *feature*) che ha proclamato la morte di Objective-C.
  - Scratch, linguaggio a blocchi per l'apprendimento dell'informatica ai bambini delle elementari. Sta crescendo a supporto della didattica nelle scuole, tuttavia è riservato a un dominio specifico.
  - COBOL, rimane in buona posizione perchè usato dalle banche;
  - Rust, linguaggio nuovo, che da un supporto efficace alla programmazione di sistema.

- **IEEE Spectrum**, genera un buon *ranking* e permette all'utente di modificare i pesi delle misurazioni. È specializzato su applicativo. Sulla lista troviamo in ordine di apparizione:

  - Swift, che riespetto al caso precedente si trova molto più in alto rispetto a Objective-C;
  - Fortran, che è uno dei primo linguaggi sviluppati. Si presenta *codebase legancy* (codice che non è più supportato dall'applicazione) e mantenuto per le applicazioni scientifiche.

Però non sempre per far ordine tra i linguaggi si può solo utilizzare una lista, difatti possono venire anche classificati: livello basso, medio e alto o in base ai linguaggi di programmazione. Si può anche produrre una tavola periodica dei linguaggi o qualcosa di più divertente, come basandosi sui personaggi del Signore degli Anelli.

Esistono diversi paradigmi di programmazione:

- **Imperativi**: il programma è un insieme di istruzioni eseguite che cambiano lo stato del programma stesso;
- **Funzionali**: il programma è un insieme di funzioni matematiche da valutare, si evita il concetto di stato e dati modificabili;
- **Orientati agli Oggetti**: il programma è un insieme di oggetti e azioni su questi oggetti;
- ***Event-Driven***: il programma è un insieme di eventi scatenati dall'interazione con l'utente;
- **Dichiarativi**: il programma è un insieme di regole (SQL, Prolog ..);
- **Concorrenti**: il programma è un insieme di flussi concorrenti in esecuzione (segnali e messaggi inviati e ricevuti dai *thread*) (Java, Ada, Go ..);
- **Multiparadigma**: *mix* dei paradigmi sopra enunciati. Alcuni esempi sono Ada e C++ (imperativi e OO), Python (funzionale, OO, imperativo).

Le tecniche per osservare questa giungla di linguaggio non sono finite. Si può difatti creare uno storico con:

- Data di nascita dei vari linguaggi;
- Con una *Timeline*, in cui per ogni linguaggio vengono indicate anche le *release*. Con presenza di intersezioni quando un linguaggio ne influenza un altro.

Oppure con il punto di vista dell'utilizzatore, il programmatore, del linguaggio. In questo caso si fanno uso dei punti d'entrata:

- Programmazione con BASIC;
- Programmazione a basso livello, si parte da asm;
- Programmazione numerica/scientifica, si parte da R;
- *Scripting* di un programma e mondo *Web*, si parte da sh.

Ma le possibilità non sono terminate. Si può misurare un linguaggio anche guardando la sua evoluzione, con la regola *Darwiniana*. La regola *Darwiniana* dice che le speci non si sviluppano in modo lineare, ma a grafo. I linguaggi hanno fatto e fanno lo stesso. Si può per questo riutilizzare l'idea della linea del tempo, non con la data di nascita, ma i più popolari in ogni dato periodo. Tale tecnica permette di linearizzare i salti evolutivi.
Vediamo di seguito alcuni dei salti evolutivi che hanno visto i linguaggi responsabili.
Il primo salto evolutivo è degli anni '80 dove ogni casa ha il proprio PC. Questi sono anche gli anni dove nascono i paradigmi di programmazione (C, PASCAL, ..). Successivamente, negli anni '90, nascono i linguaggi a oggetti, prima solo in ambito accademico per poi traslare sulle industrie; queste necessitavano di un *software* di grandi dimensioni e richiedevono un linguaggio in grado di organizzare il codice in modo affidabile, riusabile ed economico. È in questo periodo che nascono gli IDE e arriva Intenet, e il concetto di popolarità e sicurezza (grazie a Java). Con l'entrata in scena di Java assume importanza anche il *typing* statico, infatti rispetto al C++ offre una maggiore sicurezza. Con il proseguire degli anni i tipi iniziano a diventare verbosi, si cerca una maggiore produttività, nascono linguaggi come Ruby e Javascript, e inizia la guerra ai tipi. Dal momento in cui i *computer* diventano *multicore*, dunque l'*hardware* assume caratteristiche di concorrenza, si inizia a usare la programmazione concorrente, linguaggio esistente già da tempo.

Perchè la programmazione concorrente è difficile?

1. La concorrenza porta con se non determinismo, difatti ogni esecuzione porta un risultato diverso;
2. I linguaggi non sono pensati per essere concorrenti.

Quando si pensa a un algoritmo in pseudocodice poi c'è sempre l'onere di tradurlo in un linguaggio. Esiste dunque la necessità di ridurre questo *gap*, per limitare i possibili *bug* che si potrebbero creare. Esistono per farlo, alcuni modelli di concorrenza: *multithread*, *message passing*, *reactive programming* per la programmazine distribuita e GP-GPU che effettua *parser* di dati.

Ai giorni d'oggi c'è la rinascita della programmazione funzionale (*lambda*).
La programmazione imperativa è sequenziale temporale percui è difficile ottenerne parallelismo. Invece in quella funzionale esistono dei costrutti per aggreggare le funzioni, inteso come il comportamento. Va a mattoncini, riuscendo in questo modo a lavorare in parallelo. Un vantaggio del funzionale puro rispetto all'imperativo è che con causa *side-effect*, dunque può venire usato prima e dopo nel tempo senza effetti collaterali.
Va reso noto come la OOP non sia però esclusivamente imperativa, infatti esistono che la implementano linguaggi a oggetti funzionali.

Importante ai nostri giorni è anche la programmazione distribuita, dove tutto viene messo su *cloud*, in remoto. In questo modo i linguaggi riescono a lavorare asincronamente, sono scalabili, elastici e resistenti al fallimento.

Dove stiamo andando? Verso la *Data-Science*, strutture dati enormi, dove vi è la necessità di algoritmi in grado di lavorarci.
Un esempio JAVA8 che e quasi un nuovo linguaggio rispetto a JAVA7. In questa versione si sono introdutte strutture come passaggi di funzione con i *parallel stream*, che permetto di lavorare su strutture dati enormi in parallelo. Tale cambiamento è stato indispensabile altrimenti JAVA sarebbe uscito dai giochi.

In conclusione quando un programmatore sceglie un linguaggio porta uno stile. Cosa si dice è influenzato dal come lo si dice.

Alcune incoerenze presenti all'interno degli articoli:

- Goodbye, Object-Oriented Programming. C. Scalfani, Medium, July 2016.
  - "Si perde tempo a creare gerarchie, sta bene solo sui libri."
    Tuttavia quando si parla di *inhertiance* non si tratta di classificare, difatti è solo un metodo che permette di pensare a contratti "si comporta come" (voglio ereditare certi comportamenti ed estenderli).
  - "Una classe viene riusata in un altro progetto."
    Ma nella OOP la riusabilità è intesa nel medesimo progetto che negli anni deve essere mantenuto.
  - "Le interfaccie sono sufficienti a fare polimorifsmo".
    Tuttavia va considerato che esistono diversi tipi di polimorfismo. In questo caso specifico si interpreta come *subtyping* dunque le interfacce sono si sufficienti, ma esiste anche il *parametric polymorfism* con *template/generics*.
- Object-Oriented Programming - The Trillion Dollar Disaster. Ilya Suzdalnitski, Medium, July 2019.
  - "Complessità del codice che ne esce".
    Questo perchè è stato fatto uso dell'OOP con accessi concorrenti, ma esiste anche la OOP sequenziale. Invece quando vengono trattati i *design pattern* esiste effettivamente un *gap* tra quello che il linguaggio permette di scrivere e ciò che i *pattern* offrono.

– "La OOP nasce dall'accademia".
Non è vero è un linguaggio uscito proprio dall'accademia.