GitHub è un servizio pubblico di gestione del codice sorgente basato sul sistema Git, inventato da Linus Torvalds anima e per anni programmatore di Linux.

I sistemi di controllo del codice sorgente sono una vecchia conoscenza del mondo della programmazione. Dato che spesso per risolvere taluni problemi si fanno esperimenti che convivono col codice in produzione, che va manutenuto, i sistemi di controllo del codice sorgente (SCCS) offrono vari servizi per effettuare queste operazioni. Principalmente la gestione dei rami di sviluppo e la storicizzazione delle versioni. In pratica con un sistema di questo tipo ciascun programmatore è in grado di provare modifiche senza correre il rischio che queste inquinino il codice testato, in correzione e in produzione ma anche di gestire le versioni, cioè di ricostruire esattamente il contenuto di tutti i file sorgenti che in un dato giorno erano stati inseriti nel sistema.

A differenza di altri sistemi, GIT offre la possibilità di gestire sia un sistema locale privato sia un sistema globale condiviso.

Non entro nei dettagli di GIT, che peraltro sono molto complessi, ma GitHub è un servizio globale che fornisce un server che copre la parte Git condivisa.

Il servizio è complesso e vi sono varie forme di uso, ma per la programmazione open source il servizio è gratuito, quindi è possibile crare una propria pagina (la mia è questa fjovine (Francesco Iovine)) in cui memorizzare propri contributi liberi e open source che derivano o da sviluppi completamente indipendenti (io ho solo progetti originati da me e contenenti solo codice mio) o da fork, cioè che partono da un progetto già esistente e supportato magari su GitHub e lo estendono in altre direzioni, senza ovviamente contaminare il sorgente da cui partono.

Dunque questo è un primo uso per uno sviluppatore: una vetrina che permetta a eventuali datori di lavoro/committenti di misurare direttamente le tue competenze guardando il tuo modo di scrivere codice dai sorgenti.

Fresco di acquisizione da parte di Microsoft, GitHub è il cuore di un servizio di hosting di repository Git, ovvero una gestione del codice sorgente basata su cloud… ma questo è solo l’inizio. GitHub implementa infatti anche funzionalità per la revisione del codice (richieste di pull, diff e richieste di revisione), gestione del progetto (incluso monitoraggio e assegnazione dei problemi), integrazioni con altri strumenti di sviluppo, gestione del team, documentazione e “social coding”.

Molto simile a una sorta di social network per sviluppatori, GitHub è un ambiente aperto in cui i programmatori possono condividere e collaborare liberamente (anche ad hoc) su codice open source. GitHub rende facile trovare codice utile, copiare repository per uso personale e inviare modifiche ai progetti di altri, diventando sede di praticamente ogni progetto open source di qualsiasi importanza.

Controllo della versione Git

Prima di poter capire cosa fa GitHub e come funziona, dobbiamo capire cos’è Git. Git è un sistema di controllo della versione distribuito originariamente scritto da Linus Torvalds nel 2005 per e con l’aiuto della comunità del kernel di Linux. Git è nato inizialmente come uno strumento da riga di comando, adattandosi alla sua origine nella comunità del kernel Linux. Potete ancora usare la riga di comando di Git se volete, ma non è più obbligatorio.

Al posto (o in aggiunta) della riga di comando, è infatti possibile utilizzare il client GitHub gratuito per Windows o Mac o qualsiasi altra GUI per Git o un editor di codice che si integri con Git. Tutte queste opzioni sono inizialmente più facili da usare rispetto alla riga di comando. La riga di comando Git è preinstallata sulla maggior parte dei sistemi Mac e Linux e supporta tutte le operazioni; le GUI in genere supportano un sottoinsieme di operazioni Git frequentemente utilizzato.

Git è diverso dai sistemi di controllo delle versioni precedenti come Subversion in quanto è distribuito piuttosto che centralizzato. È anche abbastanza veloce, soprattutto dal momento che la maggior parte delle operazioni avviene sul vostro repository locale. Tuttavia, l’utilizzo di Git aggiunge un livello di complessità. L’invio di codice al repository locale e l’invio di commit a un repository remoto sono passaggi separati. Quando i team dimenticano questo (o non gli è stato insegnato), si può arrivare a situazioni in cui diversi sviluppatori lavorano con basi di codice divergenti.

Un repository Git remoto può trovarsi su un server o su un altro computer dello sviluppatore e Ciò consente molti possibili flussi di lavoro per i team. Un flusso di lavoro comune prevede l’utilizzo di un repository server come repository “blessed”, al quale viene inviato solo codice verificato e ben testato, spesso tramite una richiesta pull emessa dal repository di uno sviluppatore.

Articolo VBOX

Il termine virtualizzazione fa riferimento alla possibilità di astrarre le componenti hardware del PC o di un qualunque dispositivo in modo da renderle disponibili sotto forma di risorse virtuali.

Su una macchina ospitante (detta host), indipendentemente dal sistema operativo installato, è possibile caricare ed eseguire più sistemi operativi, anche molto diversi l'uno dall'altro.

Se sulla macchina "host" fosse installata, ad esempio, una certa versione di Windows, all'interno della macchina virtuale, grazie alla virtualizzazione, si possono installare ed eseguire altre versioni di Windows, una distribuzione Linux oppure altri sistemi operativi.

Alla macchina virtuale, all'interno della quale si installerà il sistema operativo, vengono destinate parte delle risorse fisiche della macchina "host" in uso.

Eventuali contese per l'utilizzo di una stessa risorsa, vengono gestite dai software che si occupano della virtualizzazione (Virtualbox è uno di essi) e in particolare dal componente detto hypervisor.

Sebbene le macchine virtuali facciano già tanto, le sole non basterebbero a rendere il nostro viaggio liscio come la seta. Uno degli scenari più noiosi, infatti, è proprio la preparazione dell’ambiente di una macchina virtuale: dobbiamo avviare la macchina, entrarci, fare tutti i nostri setup, rigenerare un pacchetto di diversi gigabyte e condividerlo con il gruppo.

Se un giorno ci rendessimo conto di dover fare un upgrade di una componente: stesso giro e stesso scambio di gigabyte. Decisamente poco pratico…

Inoltre, la cosa peggiore, è che tutto questo processo di configurazione della macchina virtuale, anche noto come provisioning, non è tracciabile (a meno di qualche passaggio manuale) e quindi non facilmente ripetibile.

Vagrant invece, consente di avere delle macchine virtuali completamente “scriptabili” sia in fase di configurazione (RAM, networking, spazio disco…) che di provisioning (installazione di MongoDB, RabbitMQ…). Ognuno di questi script potrà poi essere aggiunto al nostro source control preferito e arrivare a versionare i vari scenari realizzando quella pratica nota come Infrastructure as code.

I vantaggi di questo approccio sono evidenti visto che non avremo più necessità di dover trovare noi un modo per condividere le virtual machine. Per comprendere la semplicità di utilizzo, basti pensare che una macchina virtuale Linux è possibile avviarla con due semplici comandi inseriti in un prompt di Windows.

PuTTY, un software open source che ora come ora è utilizzato per lo più dai web master in merito alla gestione (in remoto) dei propri server, usando SSH. Il software, per Windows, rende possibile l’interazione con i sistemi Unix remoti grazie all’utilizzo dei protocolli SSH, Rlogin e Telnet.

Il suo funzionamento, tutto sommato, non è poi così difficile come sembra al primo approccio. Per iniziare ad usarlo, si fa partire il programma attraverso Microsoft Windows, di poi è importante impostare tutti i parametri base necessari per la connessione al server remoto (ad esempio l’IP, l’user name e la password).

In informatica, con il termine PuTTY, ci si riferisce ad un client (il cui sviluppo risale ai primi mesi del 1999 da parte del programmatore Simon Tatham) con i protocolli SSH, Telnet, rlogin e TCP che permette di stabilire una sessione remota cifrata ad un altro computer emulando un terminale, effettuando il login con l’IP del computer a cui ci si vuole connettere.

Ant è una libreria JAVA sviluppata dalla Apache che permette di automatizzare il processo di sviluppo di applicazioni Java. Con Ant, infatti, è possibile creare un progetto che compila, genera la documentazione, realizza file jar, war ed ear ed effettua il deploy di un’applicazione web su un application server, tutto con il semplice lancio di un comando.

Ant è un programma che si lancia da riga di comando. I comandi che Ant esegue sono letti da un file XML, di solito chiamato build.xml. In questo file bisogna definire le operazioni (target) disponibili e, per ciascuna di esse, i comandi da eseguire (task).

Il file build.xml è composto da un tag <project> che contiene a sua volta tutti i comandi disponibili, denominati target.

Ciascun target può avere target dipendenti, ciò vuol dire che, se viene richiesta l’esecuzione di un target, Ant eseguirà prima i suoi target dipendenti.

Analizziamo i comandi principali (task) che Ant mette a disposizione:

<echo>: permette di scrivere un messaggio sulla console; esempio: <echo message="Questo è in nostro primo progetto Ant"/>;

<property>: permette di definire una o più property. È possibile anche definire le property in un file esterno. Le Properties sono case sensitive ed immutabili: chiunque setti per primo il valore per la property la congela per la restante esecuzione dello script. Le property saranno poi visibili all’interno del file build.xml utilizzando la seguente sintassi: ${nomeVar};

<jar>: permette di creare un file jar. È necessario definire il nome del file jar da creare e la directory contenente i file class. È possibile definire eventualmente sia necessario anche le directory o i file da escludere;

<war>: permette di creare un file war. È possibile definire i file che andranno nella root (fileset), le librerie e i file classes che andranno nella directory WEB-INF.

<javac>: permette di compilare i file java. Ant scandisce ricorsivamente la directory specificata nel parametro srcdir alla ricerca di file java per i quali il relativo file .class non è presente o è meno recente del sorgente. Con questo controllo viene evitato di compilare i file già compilati che non hanno subito modifiche rispetto all’ultima compilazione.

Pagina: 12

Tale suite risulta essere uno dei principali prodotti open source di test automation per web application: sostanzialmente è caratterizzato da due tool, Selenium IDE e Selenium Web Driver (ve ne sono poi altri due, Selenium Grid e Selenium Remote Control che non ho analizzato).

Selenium IDE è un plugin per Firefox (non esiste omologo per altri browser), che permette di catturare una sessione utente di una web application (comprese le azioni di imputazione di dati ….), produrre automaticamente uno script che può poi essere rieseguito come tale o modificato.

Il tool è adatto per test semplici (sostanzialmente adatto per l’entry level …), non richiede alcuna conoscenza tecnica pregressa (sebbene un profilo tecnico, tipo programmatore, probabilmente riesce ad ottenere il meglio dallo strumento ….), ed è molto facile da utilizzare: ha tuttavia dei limiti che, anche con le personalizzazioni degli script ottenuti automaticamente, difficilmente possono essere superati.

La maggior parte della comunità che utilizza la suite Selenium utilizza per i propri scopi Selenium Web Driver.

Selenium Web Driver permette di creare delle vere e proprie suite di test programmando i test cases nei principali linguaggi di programmazione (Java, Ruby, Python, ecc …): nel mio caso ho utilizzato Java ma questa non è la scelta esclusiva anche se è quella per cui forse si trovano maggiori esempi, documentazione e supporto.

Sostanzialmente Web Driver “emula” una sessione browser ma lo fà server-side.

La sua installazione (è necessario usare Eclipse …), è molto semplice e ben documentata (nel senso adatta anche a chi non ha grosse conoscenza tecniche, si tratta di un tutorial veramente descritto “passo – passo” ….), sul sito di Selenium o in vari altri blog.

A differenza di Selenium IDE è quindi rivolto ad una utenza decisamente più tecnica, diciamo figure professionali con profili da programmatore (e la cosa si evidenzia andando a vedere il supporto che si ottiene on-line sui vari StackExchange, StackOverflow, ecc …, nel senso che chi risponde sono tutte figure di questo genere, lo si intende dal livello di risposte fornite).

Jenkins è una evoluzione (fork) di un progetto chiamato Hudson nato in seno ad Oracle, che dal 2011 viene sviluppato dalla comunità Open Source. Jenkins è sviluppato in Java e viene utilizzato principalmente per la compilazione di progetti Maven o Gradle, ma la vasta platea di plugin ne permette un uso molto variegato. Inoltre le esecuzioni dei job possono essere parametrizzate e anche schedulate.

Ci sono diverse piattaforme software commerciali che aiutano a creare ambienti efficaci di DevOps e CI/CD, ma c'è anche una soluzione open source molto diffusa che, tra l'altro, muove proprio alcune delle soluzioni commerciali. Si tratta di Jenkins, un progetto che viene citato spesso nel mondo dello sviluppo e che vale la pena conoscere anche solo sommariamente.

Jenkins nasce oltre dieci anni fa in Sun Microsystems con un nome diverso (Hudson). Allora è in sintesi un modulo server che il suo creatore Kohsuke Kawaguch ha creato per automatizzare i processi di test e rilascio del codice Java. Qualche anno dopo da Hudson nasce Jenkins, che ne è sostanzialmente un fork e che attualmente ha una diffusione molto superiore a quella del predecessore.

Il principio di base di Jenkins non è cambiato ma si è ampliato molto il suo raggio d'azione. Non copre solo il codice Java ma praticamente tutte le piattaforme principali, grazie a circa 1.400 plugin che gli permettono di dialogare con qualsiasi ambiente di sviluppo. Esiste ancora sotto forma di modulo eseguibile da un application server ma è anche distribuito come immagine Docker, il che ne semplifica l'uso in ambienti virtualizzati.

Jenkins dialoga tra l'altro con i principali repository di codice, in primis ovviamente GitHub, e la gestione del processo di sviluppo, test e distribuzione si basa su un elemento chiave: le pipeline. Una pipeline è, in sintesi e semplificando, una sorta di script che definisce i passi (step anche nel linguaggio di scripting) da eseguire in sequenza nel processo, raggruppati in più sezioni definite stage.

Uno sviluppatore può creare una pipeline molto dettagliata che indica gli step da fare (ad esempio richiamare un particolare comando di shell o lanciare uno specifico servizio) per compilare il codice, testarne il buon funzionamento e poi distribuirlo in un ambiente limitato o proprio in produzione. Idealmente queste fasi di compilazione, test e distribuzione a più stadi corrispondono ciascuna a un diverso stage. Per ogni stage si possono definire le operazioni da compiere alla fine, come minimo in funzione del fatto se lo stage si è completato con successo o meno.

Jenkins fornisce dei servizi di integrazione continua per lo sviluppo del software. Viene eseguito lato server all'interno di un server web che supporta la tecnologia Servlet e quindi può essere utilizzato da remoto all'interno di un Web browser. Può essere usato con i principali strumenti di gestione del codice sorgente, come AccuRev, Concurrent Versions System, Subversion, Git, Mercurial, Perforce, Clearcase e Rational Team Concert. Può eseguire progetti scritti in Ant o Maven, ma anche script bash o comandi batch di Windows. Lo sviluppatore principale è Kohsuke Kawaguchi. Il progetto è disponibile sotto la licenza MIT, quindi è open source.

Le esecuzioni possono essere azionate in vari modi, tra cui un commit oppure ad ogni intervallo di tempo oppure attraverso l'interfaccia web.

Sebbene nasca all’interno della comunità Java e sia stato pensato principalmente per la CI in tale ambiente, finora non abbiamo mai parlato di linguaggi supportati.

Una delle caratteristiche di Jenkins è la sua estrema modularità e, soprattutto, l’esistenza di un numero incredibile di plugin che gli permettono di:

interfacciarsi a qualsiasi CVS,

effettuare una build per quasi tutti i linguaggi moderni,

utilizzare anche altri build automation tool (Maven, Gralde, Ivy, etc…).

Al fine di comprendere l’incredibile modularità ed estensibilità di Jenkins, è interessante dare uno sguardo all’interfaccia ExtensionPoint per comprendere l’enorme numero di implementazioni esistenti.

Facciamo un salto indietro nel tempo allora, torniamo agli inizi del '900 e alla rivoluzione industriale che ha caratterizzato questo periodo storico importante. Fu proprio Henry Ford che ebbe una nuova visione del mercato, una visione di massa che antepose l'offerta alla domanda.

"Avranno l'auto che vorranno, del colore che vorranno, purché sia nera." Henry Ford

Una frase rimasta nella storia, una frase che ha segnato l'era della Produzione di Massa. Ciò che Ford voleva era realizzare delle auto con caratteristiche tecnologiche e commerciali in grado di poter essere riprodotte e vendute in grandi quantità.

Questo semplice concetto sarà poi ripreso e adattato a tutte le realtà industriali dell'epoca. Ciò che contraddistingue la produzione di massa è l'industrialità, ossia l'ottimizzazione dei processi produttivi mediante il raggiungimento della massima efficenza caratterizzata dalla standardizzazione delle attività.

Fu a partire dalla seconda metà del '900 che si iniziarono a vedere i primi cambiamenti di pensiero in ambito industriale. Tutto partì dal Giappone, in modo particolare dalla Toyota Motor Corporation, nata nel 1937. Dopo la seconda guerra mondiale la profonda crisi di quegli anni spinse verso una nuova ottica industriale che permettesse la riduzione dei costi a fronte di un grande aumento di produttività.

Sotto la guida di Taichii Ohno venne definito un nuovo modello di produzione industriale, che permettesse di rispondere alla necessità di flessibilità della produzione mantenendo comunque elevata la produttività aziendale. L'ascesa di Toyota fu possibile grazie al Toyota Production System (TPS), una metodologia innovativa incentrata sulla continua caccia agli sprechi e sull'importanza del coinvolgimento di tutti all'interno dell'azienda.

Nacque così il concetto di Lean Production o produzione snella, una filosofia che cambia completamente il punto di vista aziendale e sposta di gran lunga l'attenzione verso il cliente piuttosto che sulla produttività.

Cambia la logica che da Push diventa Pull.

La produzione si adegua al mercato eliminando gli sprechi e semplificando i processi per creare un valore aggiunto superiore agli occhi del cliente finale.

Il concetto di Lean Production ha continuato ad evolversi e svilupparsi nel corso del tempo e ancora oggi costituisce il punto di partenza per le migliori aziende, in quanto costituisce una delle più efficienti soluzioni industriali.

Il metodo TPS costituisce un modello organizzativo di eccellenza, soprattutto in un mercato sempre più competitivo ed esigente, influenzato da grandi incertezze sul futuro.

Semplificare i processi, concentrarsi sull'evitare ed eliminare gli sprechi, mantenere sempre una flessibilità tale da riuscire ad adeguarsi al mercato in evoluzione, è importante per garantire all'azienda una certa continuità temporale.

Ma non si tratta soltanto di applicazioni a livello esecutivo, ma anche teorico. Il modello si allarga al pensiero stesso di ogni componente dell'organizzazione, il cosiddetto Lean Thinking che deve coinvolgere ogni componente della squadra. Il pensiero snello si basa su cinque pilastri fondamentali:

1. identificazione del valore aziendale per il cliente

2. identificazione del flusso di valore per ogni prodotto

3. far scorrere il flusso del processo produttivo in modo continuo

4. utilizzo di logiche di mercato Pull, di risposta effettiva al cliente

5. continua ricerca della perfezione.

La produzione di massa (detta anche: "a flusso", "ripetitiva", "in serie" o "seriale", "continua") è la realizzazione di grandi quantità di prodotti standardizzati, spesso compiuta con catene di montaggio o linee a trasferimento.

Basata sui principi formulati da Frederick Taylor, sull'utilizzo di parti intercambiabili prodotte con macchine utensili rispettando tolleranze di lavorazione, fu grandemente diffusa da Henry Ford nel XX secolo, specie con la sua proverbiale Ford Modello T.

La prima proposta più importante di organizzazione del lavoro fu l’organizzazione scientifica del lavoro proposta da Taylor. Egli partì dall’idea che per acquistare efficienza, era necessario progettare un organizzazione centralizzata, nella quale fossero divisi rigidamente i compiti di decisione e pianificazione da quelli di esecuzione.

Il processo di lavoro doveva essere smontato in una serie di operazioni (limitate) ognuna delle quali definisse un posto di lavoro. Queste operazioni dovevano poi essere standardizzate fissandone tempi e metodi, in modo da renderle esattamente prevedibili.

Taylor Costituisce la base dalla quale riparte Ford, che riesce , dove in qualche modo taylor aveva fallito. Ovvero il successo di Ford sta nell’aver adattato al lavoro operaio grandi masse dequalificate. E trova nella catena di montaggio ( formata da macchine veloci e non flessibili) , il mezzo della sua realizzazione. Dunque i tempi e ritmi di lavoro sono dettati dalla macchina rendendo l’operaio come un ingranaggio facente parte della stessa..

Celebre è la produzione del modello T che attuando la politica vendere tanto a poco prezzo, Ford riesce a vendere una miriade di questo modello, ed ogni americano avrà accesso a questo mezzo, grazie alle vendite e quantità prodotte spropositata scende anche il prezzo dell’auto stessa in poco tempo , e ford aumenta i salari..

I progressi tecnologici si accompagnarono al progressivo ingrandimento dei complessi industriali. Mentre in pieno '800 erano relativamente rare le fabbriche che superavano il migliaio di dipendenti, questo ordine di grandezza cominciò a farsi più frequente verso la fine del secolo. Espansione produttiva e fusione tra imprese facevano imboccare la strada che avrebbe condotto al gigantismo industriale, esploso negli anni '20 del 20° secolo. La concentrazione di manodopera in grandi stabilimenti, se da un lato era destinata ad apparire come la risposta più logica alla esigenza di una produzione su larga scala, dall'altro lato cominciava a porre problemi organizzativi nuovi per i quali non potevano più valere le vecchie soluzioni di origine artigianale.

Offerta di forza lavoro non qualificata e alta mobilità[3]

L'espansione dell'industria richiedeva un reclutamento di manodopera sempre più largo. I figli del proletariato industriale formatosi nei decenni precedenti non erano più sufficienti a soddisfare il crescente bisogno di forza lavoro, e si ricorse quindi al reclutamento di masse di estrazione contadina. Il fenomeno diffuso in tutti i paesi industrializzati, assunse dimensioni imponenti negli Stati Uniti dove a cavallo dei due secoli affluirono milioni di immigrati appartenenti in gran parte agli strati più poveri dei paesi del vecchio mondo. Masse di ex contadini polacchi, irlandesi, italiani, si aggiunsero a negri, portoricani e messicani dando luogo ad una imponente offerta di lavoro, in larga misura dequalificato. Va anche osservato che la manodopera era estremamente mobile sia perché le imprese non garantivano alcuna sicurezza di impiego, sia perché i lavoratori erano continuamente alla ricerca di un lavoro migliore. Il tasso di avvicendamento nelle fabbriche era quindi estremamente alto - spesso non ci si fermava che pochi mesi o addirittura poche settimane – e il continuo ricambio di manodopera acuiva il problema del rapido apprendimento di elementari procedure di lavoro.

Alla base del TPS si trova l'idea di 'fare di più con meno', cioè di utilizzare le (poche) risorse disponibili nel modo più produttivo possibile con l'obiettivo di incrementare drasticamente la produttività della fabbrica. Nell'immediato dopoguerra la Toyota si trovava in gravi condizioni di mancanza di risorse, come peraltro gran parte dell'industria del Giappone, uscito sconfitto e stremato da una guerra devastante.

Resta oggi il problema di produrre con tempi e costi prestabiliti dei sistemi software di formidabili dimensioni, enormemente cresciuti rispetto ai pacchetti software di alcune decine di anni fa queste situazioni la neonata scienza si trova spesso in difficoltà e si sente il bisogno di teorie più evolute. (PER COLLEGAMENTO AD AGILE)

Se l'approccio iniziale era basato sui concetti dell'industria meccanica dell'inizio del XX secolo (tempi e metodi), adesso si capisce che tale impostazione è insufficiente: nell'industria meccanica si parla ormai di fabbrica immateriale costituita dalle conoscenze dei dipendenti, dai rapporti tra di loro, dalle aspirazioni comuni; ancor di più ciò vale per la fabbrica software.

In aggiunta molti hanno capito le caratteristiche originali del prodotto software (prima fra tutte l'immaterialità del prodotto principale - il codice eseguibile) che portano alla necessità di tecnologie meno note in altri settori: la più importante di tali tecnologie è probabilmente il controllo di configurazione (come le aziende).

La comparsa in letteratura e nella pratica dello sviluppo del software del concetto di "ciclo di vita" si può far coincidere con la nascita dell'ingegneria del software, in quanto rappresenta un passaggio storico dallo sviluppo del software inteso come attività "artigianale" (ovvero affidata alla libera creatività dei singoli individui) a un approccio più industriale, in cui la creazione di programmi e sistemi software viene considerata come un processo complesso che richiede pianificazione, controllo, e documentazione appropriati (così come avviene tradizionalmente nei settori più maturi dell'ingegneria).

Questa transizione si può ricondurre, in ultima analisi, all'aumentata complessità dei sistemi, all'avvento di un vero e proprio mercato del software, nonché a nuovi e più stringenti requisiti di qualità, legati per esempio all'uso di sistemi informatici in contesti critici (centrali energetiche, sistemi aerospaziali, armamenti e così via). Questo mutamento di prospettiva iniziò a verificarsi, storicamente, fra la fine degli anni sessanta e l'inizio della decade successiva. Da allora, la ricerca su questi temi è stata estremamente prolifica sia in ambito industriale che accademico.

In informatica con il termine sviluppo software si indica la serie di passi o percorso da svolgere per ottenere risultati di alta qualità e in tempi prefissati nello sviluppo di un prodotto o sistema software. Lo sviluppo software è composto da alcune attività che rappresentano un insieme di compiti da svolgere per sviluppare un software:

In ingegneria del software, il modello a cascata (waterfall model in inglese) o ciclo di vita a cascata (waterfall lifecycle) è il più tradizionale modello di ciclo di vita del software. Secondo questo modello, il processo di realizzazione del software è strutturato in una sequenza lineare di fasi o passi[1], che comprende:

analisi dei requisiti

progetto

sviluppo

collaudo

manutenzione

Questo modello riprende la sequenza di passi tipica della produzione manifatturiera, e fu il primo a essere applicato quando lo sviluppo del software cominciò a essere concepito come attività industriale.[2] Il modello è stato progressivamente abbandonato dall'industria del software, ma rimane un importante riferimento storico.

Il ciclo di vita a cascata fu il primo modello di ciclo di vita del software, sviluppato a seguito della software crisis. La sua teorizzazione rappresenta innanzitutto un importante mutamento di prospettiva nella pratica dello sviluppo del software, che viene per la prima volta concepita come processo industriale, cioè come una sequenza di sottoattività (tutte con relative documentazioni e controllo), anziché come attività "artigianale" (il cosiddetto approccio code and fix, che si potrebbe tradurre in italiano come programmazione per tentativi ed errori). Il ciclo di vita a cascata ebbe un enorme successo negli anni settanta ed è quello che ancora oggi viene più spesso associato alla programmazione procedurale e strutturata.

A partire almeno dagli anni ottanta il modello è stato soggetto a profonde critiche e revisioni, soprattutto dovute all'evoluzione del software stesso e dei linguaggi di programmazione. Benché gran parte delle critiche a questo modello siano oggi universalmente accettate, il ciclo di vita a cascata continua a rimanere un punto di riferimento importante, in sostanza un modello "canonico" rispetto al quale vengono spesso descritte le "variazioni" moderne; ed è spesso il primo modello di sviluppo software che si insegna agli studenti. Tra le nuove metodologie di sviluppo del software ci sono il modello a spirale e le metodologie agili. (PARTE FINALE SU CRITICHE)

Questo modello, non ha una reale e conclamata data di ideazione o un padre, poiché è l’evoluzione di un modello di sviluppo dei cicli di vita dei sistemi già in utilizzo nell’industria dell’ingegneria del software, ma possiamo affermare che Winston Royce, nel 1970, portò su carta questo processo, avendo la necessità di comprendere come gestire al meglio lo sviluppo di software di una certa complessità e dimensione.

el metodo waterfall ogni fase del ciclo della vita del prodotto si sussegue in sequenza, facendo progredire lo sviluppo come, per l’appunto, una cascata. Ogni fase prevede la produzione di un output che viene successivamente utilizzato come input della fase successiva in una concatenazione fino all’output finale da consegnare al cliente.

Il metodo però è stato accusato di avere con sé numerosi limiti che nel corso del tempo hanno fatto sì che questo metodo fosse abbandonato del tutto o in parte, adottando al suo posto altre tipologie di sviluppo:

Assenza di flessibilità;

Problematicità nella risoluzione al momento dell’insorgenza di errori;

L’implementazione di nuove funzionalità richiede tempi elevati;

Aumento considerevole dei costi.

Le modalità di consegna “a pezzi”, inoltre, permettono di avere riscontri frequenti e continui feedback, scongiurando scenari un tempo usuali, in cui i team di programmatori lavoravano per anni ad un progetto sequenziale, prima di rendersi conto della sua inefficacia soltanto a lavoro finito.

L'uso del termine agile[5] per riferirsi a metodi di sviluppo software fu introdotto dal Manifesto Agile pubblicato nel 2001.[6]

Ogni iterazione è un piccolo progetto a sé stante e deve contenere tutto ciò che è necessario per rilasciare un piccolo incremento nelle funzionalità del software: pianificazione (planning), analisi dei requisiti, progettazione, implementazione, test e documentazione.

Si tratta di un metodo innovativo basato sull’interazione continua con gli stakeholder

L’idea del Metodo Agile non si basa sull’approccio classico e lineare di progettazione, ma sulla possibilità di realizzare un progetto per fasi, chiamate “sprint”. Ad ogni sprint corrisponde una nuova funzionalità e viene verificata la soddisfazione del cliente, al quale viene mostrato il lavoro svolto fino a quel punto. Un sistema iterativo (ed interattivo) che consente di apportare agilmente modifiche al progetto, di abbattere i costi di produzione e, soprattutto, di evitare effort inutili ed un eventuale fallimento del progetto.