Università degli Studi di Parma

Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Giulio Destri

Corrado Lombardi

I PROCESSI DI SVILUPPO SOFTWARE: LA STORIA

Collana "I quaderni"

URL originale: http://www.giuliodestri.it/documenti/D06_StoriaSviluppoSoftware.pdf

Indice

Il software nella nostra vita di ogni giorno

Il software ha ormai decenni di vita. I primi calcolatori programmabili entrarono nel mercato negli anni'50. E oggi il software è pervasivo, non solo nei sistemi informativi di aziende ed organizzazioni, ma anche entro le automobili (centraline), gli elettrodomestici, i sistemi industriali… in generale nella maggior parte degli strumenti che determinano la nostra vita di ogni giorno.

Ciò nonostante ancora oggi esistono tante problematiche legate allo sviluppo di software, che troppo spesso richiede tempi più ampi o costi maggiori di quanto preventivato. Inoltre il software non è un prodotto "statico" che, una volta completato rimane costante ma, nella maggior parte dei casi, si deve evolvere nel tempo perché cambiano i presupposti che ne hanno richiesto la creazione.

I modelli tradizionali dello sviluppo software

Nei modelli tradizionali, lo sviluppo software avviene attraverso una successione di fasi differenti, l'output di ogni fase costituisce l'input della successiva.

Si tratta di modelli iterativi che, solitamente, prevedono che una fase non possa iniziare se prima non è terminata quella che la precede.

Il modello tradizionale per eccellenza è il **modello a cascata (Waterfall)** la cui prima formale descrizione è riportata in un articolo del 1970 di Winston W. Royce ([2]). La prima definizione del modello a cascata appare invece in uno scritto del 1976 di Thomas E. Bell e T.A. Thayer ([3])

Questo modello ha origine in industrie manifatturiere ed edili, settori altamente strutturati nei quali modifiche in corsa al prodotto (after-the-fact) sono fattibili, ma a costi molto elevati. Ovvio pertanto che venga posto il massimo rigore sulle fasi di analisi e progettazione.

Dal momento che, agli albori dello sviluppo software, non erano stati "pensati" modelli atti allo scopo, non si fece altro che mutuare questo modello iterativo in uso nelle imprese manifatturiere ed implementarlo nello sviluppo software.

Il modello prevede fasi successive: Raccolta ed analisi requisiti, Design, Implementazione, Integrazione, Test, Installazione e Manutenzione.

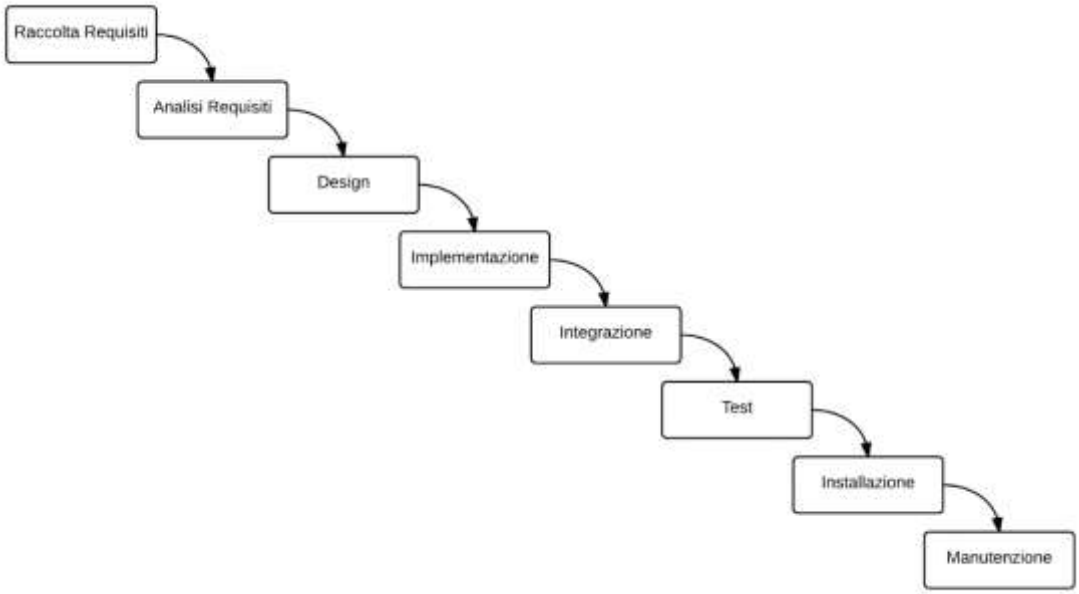

Figura 1 Grafico del modello a cascata

Le regole di implementazione del modello a cascata stabiliscono che una buona percentuale del tempo dedicato al progetto deve essere investito nelle fasi di raccolta ed analisi dei requisiti. Tipicamente viene investito il 20-40% del tempo a disposizione nelle fasi di Raccolta ed analisi dei requisiti ed in quella di design. Il 30-40% viene dedicato alla fase di implementazione ed integrazione, il resto del tempo è dedicato a test ed installazione.

L'idea centrale è che il tempo investito precocemente per assicurare che tutti i requisiti vengano correttamente rispettati comporta notevoli benefici in seguito. In termini monetari, prevenire un bug durante una delle fasi iniziali (Requisiti o Design) comporta uno sforzo economico enormemente minore rispetto al risolverlo durante una fase avanzata come ad esempio Implementazione o Test (dalle 50 alle 200 volte più oneroso).

Pertanto la metodologia rigorosa di sviluppo a cascata stabilisce che una fase non possa iniziare se quella che la precede non è completa al 100%.

Sulla base del modello a cascata sono nati altri modelli che ne condividono le linee guida, seppur con variazioni più o meno leggere, quali, ad esempio, la possibilità di ritornare alla fase precedente o addirittura alla fase di design, qualora emergano criticità durante la fase in corso, tali da ostacolarne il progresso.

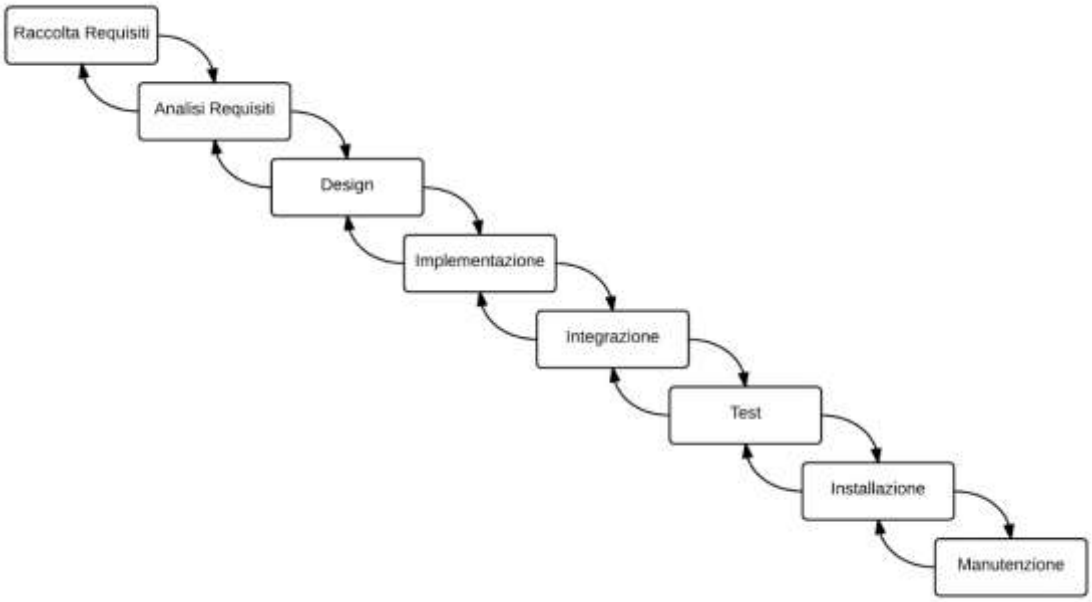

Figura 2 Modello modificato con possibilità ritorno a fase precedente

Nel suo *waterfall final model*, W. Royce illustrò che il feedback raccolto durante la fase di Test, potrebbe evidenziare problemi tali da raccomandare un ritorno alla fase di Design ed, eventualmente, da questa alla raccolta o all'analisi dei requisiti, qualora sia necessaria una rivisitazione dei requisiti per risolvere questi problemi.

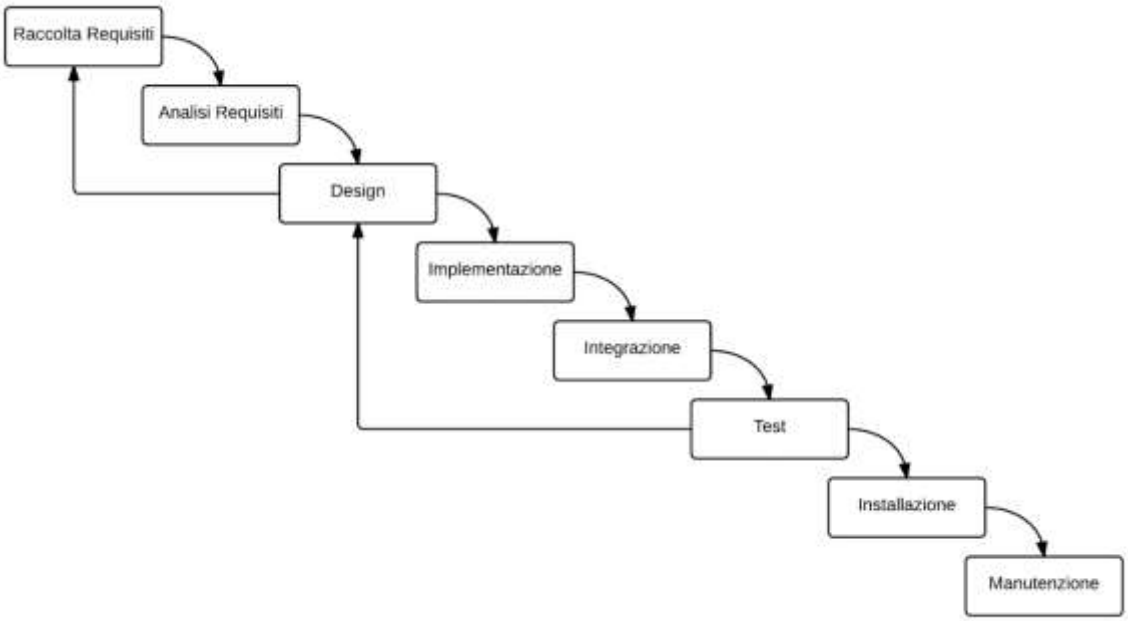

Figura 3 Modello modificato, con possibile ritorno alle fasi di Design ed Analisi dei requisiti

Tre varianti del modello a cascata vengono presentate in [4]

La prima è il **Waterfall with overlapping phases (Sashimi)**. Questo modello preserva la suddivisione in fasi, ma rende meno netto il confine tra di esse, introducendo una sorta di esecuzione parallela, almeno in parte, tra le fasi. Modello ideale per progetti nei quali le idee circa requisiti e lavoro da svolgere non sono chiari all'inizio, ma lo divengono a lavorazione in corso.

Figura 4 Modello Sashimi

Sempre in tema lavorazione parallela, viene presentato il modello **Waterfall with subprojects** la cui idea chiave è la suddivisione di un progetto in N sottoprogetti a lavorazione parallela dopo una prima fase di Design Architetturale e vengono uniti assieme prima della fase di Test generale del sistema.

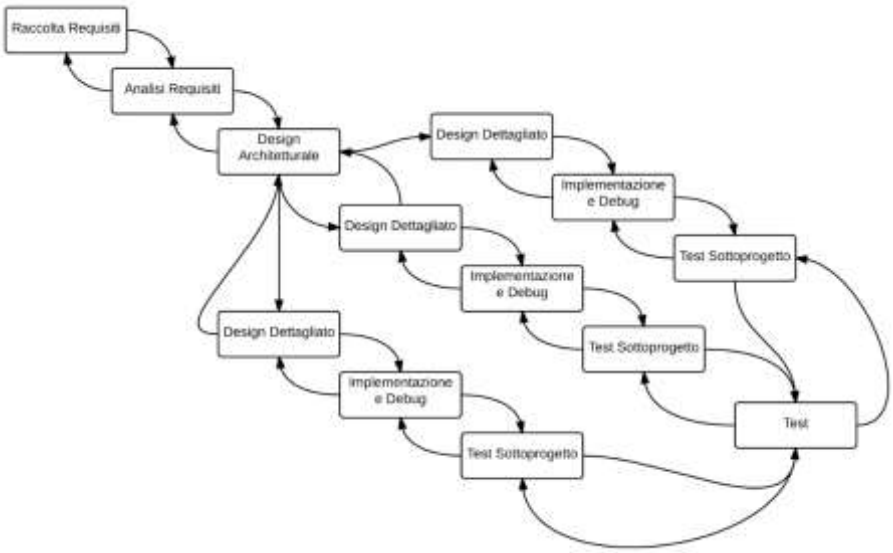

Figura 5 Waterfall with subprojects

L'ultimo esempio presentato è invece focalizzato sul tema della riduzione dei rischi ed è il **Waterfall with risk reduction.** In questo modello alcune fasi,

tipicamente analisi requisiti e design, vengono svolti secondo un approccio a spirale (risk reduction spiral). L'approccio a spirale prevede una evoluzione incrementale di un progetto o di una o più fasi di lavorazione. Ad esempio, in uno sviluppo a spirale, un progetto viene creato tramite lavorazioni a complessità incrementale.

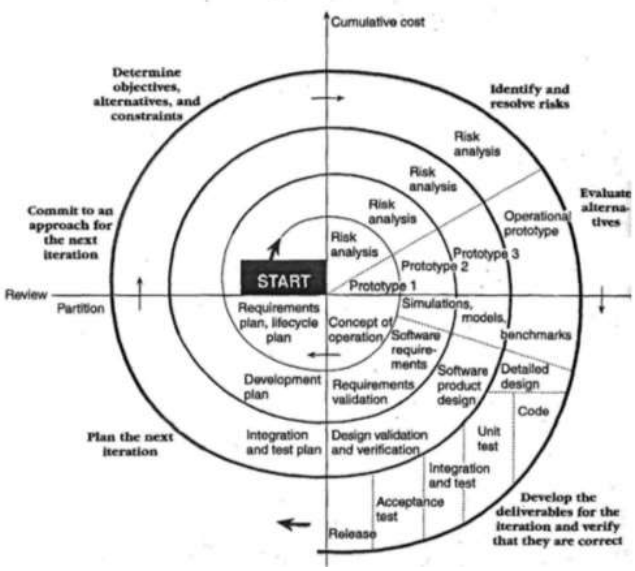

Figura 6 Risk reduction spiral (da [4])

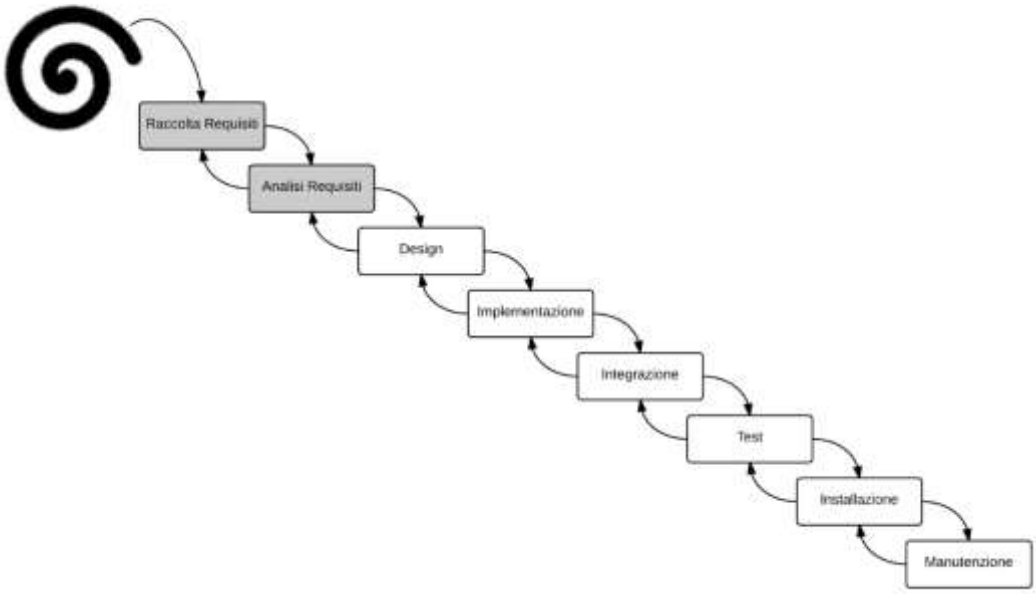

Figura 7 Modello risk reduction: le aree colorate indicano fasi che tipicamente sono svolte con approccio a spirale

Il ciclo di vita del software

Con il termine “ciclo di vita” del software ci si riferisce alle fasi attraversate da un progetto software durante la sua evoluzione, dall’idea iniziale alla manutenzione. Il risultato finale è il prodotto software stesso, accompagnato da tutta la documentazione ad esso associata.
In [5] vengono elencate e descritte le fasi comprese nella struttura base di un progetto software, ne riportiamo un estratto.

Le fasi, che rispecchiano quanto definito dal modello a cascata, sono:

- Raccolta dei requisiti
- Analisi
- Progettazione
- Implementazione
- Test
- Installazione in produzione
- Manutenzione (ordinaria ed evolutiva)

e vengono così descritte.

Raccolta dei requisiti ed analisi

Il progetto inizia con la fase di raccolta dei requisiti.

In questa fondamentale fase vengono adottate e combinate tra loro diverse tecniche volte ad “estrarre dalla mente del cliente o del committente tutto ciò che il progetto software dovrà fare”.

Il “Manuale dell’Analista” [6] definisce questa fase come *requirement elicitation,* che può essere svolta con una combinazione di 12 metodi diversi quali ad esempio: Interviste, Osservazioni sul campo, Gruppi di Lavoro, ecc.

Il termine elicitazione è mutuato dalla psicologia e significa “tirare fuori” informazioni, in questo caso, mediante domande o altri comportamenti stimolanti.

Alla fase di raccolta requisiti segue quella di analisi. Fase estremamente importante in quanto, un errore commesso a questo punto, può avere ripercussioni molto serie sul risultato finale.

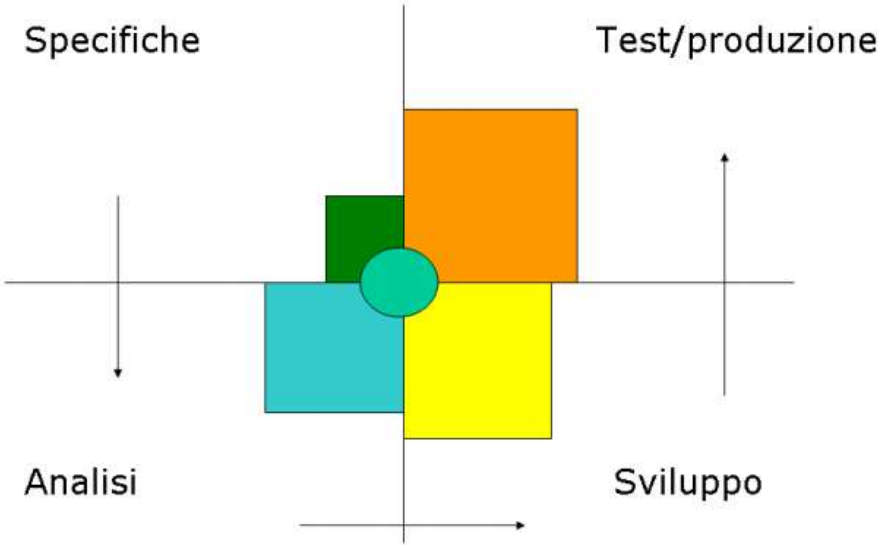

Figura 8 Impatto degli errori fatti in fase di raccolta delle specifiche sulle fasi successive.

A seconda delle dimensioni del progetto l’analisi può essere svolta in toto all’inizio, oppure iterativamente durante il processo. A seconda del modello di sviluppo che viene adottato, l’analisi può essere ripetuta o meno durante il ciclo di vita del software.

L’obiettivo della fase di analisi è la descrizione completa e formalizzata, con un livello di dettaglio adeguato di tutto ciò che il sistema deve fare (requisiti funzionali), dell’ambiente in cui dovrà operare (requisiti non funzionali) e dei vincoli che dovrà rispettare. Notare che la descrizione specifica cosa il sistema dovrà fare, non il come (modello a scatola nera). Queste descrizioni vengono raccolte in documenti chiamati documenti di specifica, brevemente specifiche.

La fase di progettazione (Design)

Partendo dall'output della fase di analisi, che deve essere preciso e privo di ambiguità, la fase di progettazione definisce le istruzioni operative per la realizzazione del progetto (dettagli implementativi). Le istruzioni devono avere il giusto livello di dettaglio ed essere raccolte in documenti opportunamente strutturati. Pertanto, la progettazione di un'applicazione è composta dalle attività per individuare la soluzione implementativa migliore rispetto agli obiettivi funzionali, a quelli non funzionali ed ai vincoli. Queste attività possono essere di varia natura, possono essere svolte in tempi e modi diversi in base all'approccio seguito, ma in generale aiutano progettisti e team di sviluppo a prendere decisioni importanti, spesso di natura strutturale.

Il risultato della progettazione è la definizione dell'architettura del sistema, intendendo con questo termine l'organizzazione strutturale del sistema stesso, che comprende i suoi componenti software, le proprietà visibili esternamente di ciascuno di essi (l'interfaccia dei componenti) e le relazioni fra le parti.

In analisi, requisiti e struttura del sistema sono rappresentati in forma astratta e (teoricamente) indipendente dalla tecnologia. La progettazione tiene conto anche di tutti i fattori relativi all'utilizzo di una tecnologia concreta e quindi, a differenza dell'analisi, la progettazione non può essere svolta indipendentemente dalla tecnologia utilizzata.

Durante la progettazione devono anche essere definiti tutti gli aspetti necessari per una implementazione non ambigua.

Uno strumento molto usato nella progettazione è il diagramma di flusso o flow-chart, oppure la sua evoluzione UML activity diagram o diagramma delle attività. Ambedue gli strumenti servono a realizzare la scomposizione delle attività da compiere in elementi sempre più piccoli che possano essere facilmente implementati con opportuni insiemi di istruzioni del linguaggio di programmazione scelto.

La fase di implementazione (Coding)

La scrittura del codice sorgente può essere svolta con molti strumenti, il più semplice dei quali è l'editor di file ASCII di base (notepad, gedit, vi, …). Per la complessità che i moderni linguaggi ad oggetti richiedono però tale approccio è troppo poco produttivo. Dovendo infatti garantire il rispetto di tempi stretti, è necessario disporre di tutte le funzioni di facilitazione integrate entro un unico strumento per la scrittura del codice sorgente.

Un Integrated Development Environment (IDE), è un software che aiuta i programmatori nello sviluppo del codice. Normalmente consiste in un editor di codice sorgente, un compilatore e/o un interprete, un tool di building automatico, e (solitamente) un debugger. Ormai sempre più spesso è integrato con sistemi di controllo di versione (SVN o GitHub, ad esempio), con sistemi di gestione delle

dipendenze da librerie esterne (Maven) e con uno o più tool per semplificare la costruzione di una GUI.

Alcuni IDE, rivolti allo sviluppo di software orientato agli oggetti, comprendono anche un navigatore di classi, un analizzatore di oggetti e un diagramma della gerarchia delle classi.

Sebbene siano in uso alcuni IDE multi-linguaggio, come Eclipse, NetBeans, IntelliJ Idea e Visual Studio, generalmente gli IDE sono rivolti ad uno specifico linguaggio di programmazione.

La scrittura del codice sorgente, per quanto spesso poco considerata, rimane comunque la fase fondamentale di ogni progetto informatico. L'analisi e la progettazione possono essere state svolte al meglio, ma, se il codice viene scritto male, l'applicazione risultante avrà problemi e funzionerà male. Facendo un'analogia nell'ambito dell'ingegneria civile, è chiaro che anche il progetto più bello, se i pilastri non sono fabbricati bene, se i mattoni non sono posati con accuratezza o se i materiali impiegati sono di scarso pregio, darà origine ad un edificio di pessima qualità.

Anche per questo, metodologie di programmazione più moderne tendono a fare diventare la fase di scrittura del codice quella principale in tutto il progetto informatico.

Durante la stesura del codice ha luogo anche il primo debugging, ossia la rimozione degli errori di sintassi e degli errori più evidenti, come la mancata inizializzazione di variabili, che possono compromettere la compilazione o il funzionamento del programma. Gli IDE più moderni hanno ottimi sistemi di live debug. Tendono ad evidenziare in tempo reale, oltre al codice che inevitabilmente porterà ad errori di compilazione, anche frammenti di codice inutili, blocchi ripetuti, inizializzazioni di variabili non necessarie, etc. etc.

La fase di Test

Una volta che il programma è stato completato o comunque può iniziare a funzionare, occorre verificare che il suo funzionamento sia conforme a tutte le specifiche che erano state stabilite nella fase di analisi. Questo è lo scopo fondamentale della fase di test.

Gli errori che portano al non rispetto delle specifiche sono di solito molto più insidiosi da scoprire rispetto a quelli che vengono trovati durante il debugging. Non sono errori di sintassi, ma errori logici e concettuali, come, per esempio, lo scrivere il segno '+' invece del segno '-' entro un algoritmo che richieda la sottrazione e non la somma.

La fase di test prevede quindi di verificare il comportamento effettivo del programma rispetto a quello previsto e di segnalare le differenze di comportamento ai programmatori che dovranno procedere alla ricerca e alla eliminazione delle cause di tali differenze.

I modelli di sviluppo software più moderni pongono l'accento sulla fase di test, anteponendola in alcuni casi alla fase di sviluppo (ad esempio il Test Driven Development, TDD) e prevedendo l'esistenza di suite di test eseguite automaticamente durante la build ed il packaging del progetto, allo scopo di ridurre il più possibile il rischio di regressione.

La fase di avvio ed entrata in produzione

Dopo il test, raggiunto un livello sufficiente di qualità, il programma può entrare in produzione. Questo termine ha un significato diverso secondo il tipo di programmi in realizzazione.

Per i programmi destinati alla vendita presso il pubblico, o alla distribuzione se gratuiti, questa fase rappresenta il rilascio sul mercato, fisico (negozio), virtuale (e-commerce, download) o mobile (PlayStore, AppStore) che sia. Per i programmi realizzati specificatamente per un cliente (i cosiddetti "programmi custom"), questa fase rappresenta l'installazione ed il collaudo presso la sede del cliente che li ha richiesti. In ultimo, per le applicazioni web (siti di e-commerce, portali, gaming-on-line, …) rappresenta l'installazione ed il collaudo su uno o più server web (Apache, Tomcat, IIS, ...) ed il tuning di questi ultimi.

Al termine di questa fase i programmi iniziano la propria vita operativa, durante la quale svolgono il compito previsto nel contesto per cui sono stati progettati, che può proseguire anche per molti anni.

La fase di Manutenzione

Durante la vita operativa possono verificarsi necessità di interventi correttivi o di aggiornamento sui programmi, che prevedono nuove fasi di progettazione, implementazione e test. Tali interventi correttivi sono raggruppabili in due distinte famiglie:

1. **Manutenzione ordinaria**, l'insieme di interventi correttivi necessari per via di errori sfuggiti ai test o dovuti al funzionamento del programma in condizioni non previste durante la sua progettazione, come ad esempio il funzionamento su Windows8 di un programma nato per WindowsXP;
2. **Manutenzione evolutiva**, l'insieme di interventi di variazione od arricchimento delle funzioni del programma per via di nuove necessità operative del programma stesso; un esempio è l'aggiornamento continuo dei programmi gestionali per stare aggiornati rispetto alle normative fiscali.

In generale, comunque, ogni programma durante la sua vita è soggetto a interventi evolutivi e correttivi. Solo una piccola percentuale di programmi non viene più toccata dopo il rilascio.

Le esigenze dei sistemi odierni

A parte alcune eccezioni presenti in determinati settori (“di nicchia”) ancora basati su sistemi legacy, la stragrande maggioranza dei sistemi informatici, prodotti software inclusi, è in evoluzione rapida e continua. Il time-to-market, ovvero il lasso di tempo che trascorre tra l’inizio di un progetto, o l’idea stessa di progetto, ed il suo rilascio sul mercato, è di importanza strategica per i committenti e per gli sponsor. Che vedono così ripagati in un tempo ragionevole gli investimenti sostenuti per lo sviluppo del progetto e, qualora riescano ad essere tra i primi a proporre un prodotto software sul mercato, guadagnano un discreto vantaggio sui propri concorrenti. Senza dimenticare che la rapidità con cui un prodotto è disponibile agli utenti finali, soprattutto per i prodotti b2c disponibili on line immediatamente, e la possibilità di rilasciare on line frequentemente aggiornamenti e correzioni di eventuali malfunzionamenti, costituiscono un ulteriore incentivo a tenere un time-to-market il più corto possibile.

Nell’era del web è inoltre possibile avere feedback sul prodotto a stretto giro, questo permette a chi gestisce il progetto di decidere se procedere con sviluppi ulteriori sullo stesso ed eventualmente quante risorse reinvestire in esso.

Inoltre, vista l’elevata dinamicità del sistema, il processo di sviluppo deve essere reattivo rapidamente ai cambi di idea (mind-change) da parte del committente, in corso d’opera.

Alla luce di tutto questo, il ciclo di vita del software deve essere stretto e condurre ad un risultato, nel più breve tempo possibile. Risultato che può anche essere composto da un prodotto minimo che verrà poi ampliato con ulteriori sviluppi. In questi casi si parla di Minimum Viable Product.

I limiti dei processi di sviluppo tradizionali

“Se l’industria dei trasporti avesse avuto la stessa evoluzione della tecnologia informatica, oggi sarebbe possibile volare in 5 secondi da Roma a Stoccolma, con 50 centesimi” [7]

E’ un paragone che descrive perfettamente l’evoluzione dei sistemi informatici hardware e software.

Risulta perlomeno difficile pensare che modelli sviluppati negli anni 70, siano i più efficaci da adottare, a distanza di oltre 40 anni, su sistemi e tecnologie completamente diversi nati durante un progresso tecnologico senza precedenti. Rispetto alle esigenze dei moderni sistemi, questi modelli presentano pertanto dei limiti, che andiamo brevemente ad analizzare.

Fase di test tardiva

I test sono una fase cruciale all'interno del ciclo di vita del software. La rapidità con cui è possibile rilasciare correzioni di bug o malfunzionamenti, non deve in nessun modo essere un incentivo a rilasciare software con difetti più o meno evidenti, nonostante questi siano purtroppo inevitabili.

Introdurre i test solamente all'ultimo momento, appena prima della messa in produzione, dopo che il progetto è stato di fatto completato e migliaia di righe di codice sono state scritte, può essere rischioso. Devono essere individuati con precisione tutti gli scenari da testare, con tutti i possibili casi d'uso e, in questo, un'analisi completa e ben fatta è sicuramente di aiuto. I problemi possono però insorgere nel momento in cui dovessero emergere malfunzionamenti. Questi vanno riportati agli sviluppatori i quali potrebbero avere difficoltà ad individuarne le cause in mezzo a tutto il codice già scritto che costituisce il progetto finendo inevitabilmente a ritardare il completamento del progetto.

Burocrazia

La specifica dei requisiti produce un documento scritto che vincola il prodotto da sviluppare e ciò non sempre soddisfa le esigenze del cliente. Si tratta pur sempre di specifiche basate su un documento inanimato che non necessariamente aiuta nel definire le esigenze, che, invece, appaiono subito chiare dopo il primo rilascio del software, inoltre tale documento deve essere completo e chiaro prima di procedere allo sviluppo, ma non sempre ciò è possibile.

L'utente spesso non conosce tutti i requisiti dell'applicazione perché non può conoscerli, motivo per cui non sempre il documento dei requisiti è completo e, quindi, si ha un passaggio alla fase successiva con documentazione incompleta o poco chiara.

Il modello a cascata, e le sue evoluzioni, obbligano a usare standard pesantemente basati sulla produzione di una data documentazione in determinati momenti per cui il lavoro rischia di essere burocratizzato.

Attività svolte controvoglia e/o sotto scadenza

A meno che in azienda non siano presenti figure preposte, cosa assai rara, le attività burocratiche, test e scrittura documentazione su tutte, vengono svolte dagli sviluppatori. Uno dei pilastri del modello a cascata è proprio la predisposizione della documentazione funzionale del prodotto.

Salvo rari casi, nessuno sviluppatore (*per lo meno nessuno tra le centinaia di colleghi incontrati in quasi 13 anni di carriera, ndA*) svolge volentieri queste attività, spesso viste come un fastidio ("Il mio lavoro è scrivere codice") e quindi portate avanti, più o meno inconsciamente, controvoglia. Nel caso test e documentazione siano da svolgere sotto scadenza a causa di ritardi nella fase di scrittura codice ed integrazione dei sorgenti, la situazione non può che peggiorare.

Le metodologie Agili (si veda il prossimo capitolo) mettono a disposizione una tecnica molto potente per oviare, almeno in parte, al "peso" dei test (TDD, Test

Driven Development) e suggeriscono di adottare uno stile di scrittura del codice pulito e lineare “che si documenti da solo”. Esistono inoltre tool e framework che facilitano il lavoro di scrittura della documentazione, anche di quella meno tecnica (BDD, ad esempio).

Regressione

I test vengono spesso svolti manualmente dagli sviluppatori o da personale addetto (Tester). Nel caso la funzionalità sviluppata sia completamente nuova, si può essere ragionevolmente certi che una volta superati i test previsti, il rischio di rilasciare bug o malfunzionamenti in produzione è relativamente contenuto, a patto che il documento di test (Test case) preveda adeguata copertura.

Nel caso invece il rilascio riguardi una modifica ad una funzionalità già rilasciata in precedenza, potrebbero presentarsi problemi. Vi è infatti il rischio concreto che queste modifiche vadano ad introdurre bug indesiderati in parti del prodotto interessate indirettamente dalla modifica, introducendo così il fenomeno della *regressione*. Per contenere questo rischio dovrebbe essere sempre pianificata una suite di test appositi (Regression test). Si tratta anche in questo caso di test manuali più difficili da pianificare e non sempre svolti completamente. Se lo sviluppo coinvolge il rilascio urgente di correzioni di bug emersi in produzione la fase di regression test potrebbe venire svolta in maniera più o meno incompleta, accentuando il rischio di introdurre regressione.

Linearità

Spesso si hanno cicli di feedback per la correzione degli errori. Tale feedback deve essere lineare e, quindi, non si possono effettuare salti a ritroso ma vanno ripercorse tutte le fasi in maniera lineare.

Rigidità

Nella metodologia a cascata, ogni fase viene congelata quando si passa alla fase successiva, per cui non è possibile un’interazione tra clienti e sviluppatori durante il ciclo di vita dopo la parte iniziale.

Tutta l’analisi è concentrata all’inizio del ciclo di vita. Tutti i possibili scenari d’uso, le funzionalità e le caratteristiche che il prodotto dovrà avere devono essere chiariti durante questa fase, prima che vengano definiti i dettagli implementativi e una singola linea di codice venga scritta. Eventuali dubbi o problematiche che dovessero emergere in una fase successiva come, ad esempio uno scenario di difficile implementazione oppure modifica delle specifiche da parte del committente, verrebbero gestite in modo articolato.

Tutto il modello è orientato alla singola data di rilascio (Monoliticità) che spesso si pone a mesi o anni dopo l’inizio della prima fase per cui se vengono commessi eventuali errori o cambiano i requisiti, questi verranno implementati dopo

parecchio tempo e comunque alla fase di consegna seguirà subito un altro adattamento perché il software sarà già obsoleto.

Come descritto in precedenza, i sistemi moderni richiedono flessibilità, massima reattività ai frequenti cambiamenti di specifiche in corsa nonché un processo snello che conduca in tempi rapidi ad un risultato, anche minimo e/o parziale.

Bibliografia

[1] Wikipedia, «Ingegneria del Software,» Wikimedia Fundation, [Online]. Available: http://it.wikipedia.org/wiki/Ingegneria_del_software. [Consultato il giorno 7 03 2015].

[2] W. W. Royce, «Managing the development of large software systems,» [Online]. Available: http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf. [Consultato il giorno 26 02 2015].

[3] T. Bell e T. A. Thayer, *Software requirements: Are they really a problem?,* IEEE Computer Society Press, 1976.

[4] S. McConnell, Rapid Development: Taming Wild Software Schedules, Redmond: Microsoft, 1996.

[5] G. Destri, Sistemi Informativi: Il Pilastro Digitale Di Servizi E Organizzazioni, Milano: Franco Angeli, 2013.

[6] K. Brennan, A Guide to the Business Analysis Body of Knowledge (BABOK Guide), Toronto: International Institute of Business Analysis, 2009.

[7] C. Valeria, 2004. [Online]. Available: http://www.ce.uniroma2.it/courses/ac05/lucidi/Intro_4pp.pdf. [Consultato il giorno 20 01 2015].

[8] «Methodology,» Agile Programming, [Online]. Available: http://agileprogramming.org/. [Consultato il giorno 16 11 2014].

[9] «Manifesto for Agile Software Development,» 2001. [Online]. Available: http://agilemanifesto.org/. [Consultato il giorno 16 11 2014].

[10] A. Gutierrez, «Waterfall vs. Agile: Can They Be Friends?,» Agile Zone, 6 2 2010. [Online]. Available: http://agile.dzone.com/articles/combining-agile-waterfall. [Consultato il giorno 17 11 2014].

[11] A. Gutierrez, «Waterfall vs. Agile: Development and Business,» Agile Zone, [Online]. Available: http://agile.dzone.com/articles/waterfall-vs-agile-development-business. [Consultato il giorno 17 11 2014].

[12] «Extreme Programming: A Gentle Introduction,» [Online]. Available: http://www.extremeprogramming.org/. [Consultato il giorno 17 11 2014].

[13] «Extreme Programming Values,» [Online]. Available: http://www.extremeprogramming.org/values.html. [Consultato il giorno 19 11

2014].

[14] Wikipedia, «Scrum (software development),» Wikimedia Foundation, [Online]. Available: http://en.wikipedia.org/wiki/Scrum_(software_development). [Consultato il giorno 23 11 2014].

[15] T. Birch, «Agile Advice,» [Online]. Available: http://www.agileadvice.com/2014/03/20/referenceinformation/new-scrum-diagram-meet-scrum-by-travis-birch-csp/. [Consultato il giorno 01 03 2015].

[16] J. Humble e D. Farley, Continuous Delivery, Upper Saddle River, NJ: Addison-Wesley, 2011.

[17] M. Fowler, «Continuous Integration,» Martinfowler.com, [Online]. Available: http://www.martinfowler.com/articles/continuousIntegration.html. [Consultato il giorno 9 12 2014].

[18] «Extreme Programming Rules,» [Online]. Available: http://www.extremeprogramming.org/rules.html. [Consultato il giorno 19 11 2014].

[19] K. Beck, Extreme Programming EXplained: Embrace Change, Reading: Addison-Wesley, 2000.

[20] «I Principi Sottostanti Al Manifesto Agile,» [Online]. Available: http://agilemanifesto.org/iso/it/principles.html. [Consultato il giorno 16 11 2014].

[21] Wikipedia, «Scrum (software Development),» Wikimedia Foundation, [Online]. Available: http://en.wikipedia.org/wiki/Scrum_(software_development). [Consultato il giorno 23 11 2014].

[22] «10 Deploys Per Day: Dev and Ops Cooperation at Flickr,» [Online]. Available: http://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr. [Consultato il giorno 26 12 2014].

[23] «Continuous Delivery Agile Development and Experience Design,» [Online]. Available: http://www.thoughtworks.com/continuous-delivery. [Consultato il giorno 9 12 2014].

[24] «Continuous Delivery,» Continuous Delivery, [Online]. Available: http://continuousdelivery.com/. [Consultato il giorno 9 12 2014].

[25] Wikipedia, «DevOps,» Wikimedia Fundation, [Online]. Available: http://en.wikipedia.org/wiki/DevOps. [Consultato il giorno 26 12 2014].

[26] M. Finelli, «Sistemi Di Monitoring, Logging e Alerting Moderni,» [Online]. Available: http://www.slideshare.net/Codemotion/mla-moderni-finelli. [Consultato il giorno 26 12 2014].

[27] Bravofly, IT Dept., *Architettura sistema ricerca voli.*

[28] Bravofly, IT Dept., *Convenzioni in tema sviluppo software.*

[29] Bravofly, Press Office, *Descrizione Ufficiale Volagratis.*

[30] «Ansible Documentation,» [Online]. Available: http://docs.ansible.com/. [Consultato il giorno 13 02 2015].

[31] «Ansible Is Simple IT Automation,» [Online]. Available: http://ansible.com/. [Consultato il giorno 13 02 2015].

[32] «Apache Tomcat,» [Online]. Available: http://tomcat.apache.org/. [Consultato il giorno 06 03 2015].

[33] «Apache Subversion,» [Online]. Available: http://subversion.apache.org/. [Consultato il giorno 15 02 2015].

[34] Wikipedia, «Modello a Cascata,» Wikimedia Fundation, [Online]. Available: http://it.wikipedia.org/wiki/Modello_a_cascata. [Consultato il giorno 12 2014].

[35] Wikipedia, «Modified Waterfall Models,» Wikimedia Fundation, [Online]. Available: http://en.wikipedia.org/wiki/Modified_waterfall_models. [Consultato il giorno 11 2014].

[36] Wikipedia, «Waterfall Model,» Wikimedia Fundation, [Online]. Available: http://en.wikipedia.org/wiki/Waterfall_model. [Consultato il giorno 11 2014].

[37] Wikipedia, «Behavior-driven Development,» Wikimedia Fundation, [Online]. Available: http://en.wikipedia.org/wiki/Behavior-driven_development. [Consultato il giorno 05 02 2015].

[38] «Cucumber - Making BDD Fun,» [Online]. Available: http://cukes.info/. [Consultato il giorno 05 02 2015].

[39] «JUnit - About,» [Online]. Available: http://junit.org/. [Consultato il giorno 15 02 2015].

[40] «Maven – Welcome to Apache Maven,» [Online]. Available: http://maven.apache.org/. [Consultato il giorno 05 02 2015].

[41] «Welcome to Jenkins CI!,» [Online]. Available: http://jenkins-ci.org/. [Consultato il giorno 05 02 2015].

[42] C. Lombardi, "Tesi di Laurea: Impostazione di un processo di Contiunuos Development per il portale web Volagratis.IT" [Online] Available http://www.cs.unipr.it/Informatica/Tesi/Corrado_Lombardi_20150325.pdf