Università degli Studi di Parma

Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Corrado Lombardi

Giulio Destri

I PROCESSI DI SVILUPPO SOFTWARE: L'EVOLUZIONE AGILE E IL DEVOPS

Collana "I quaderni"

URL originale: http://www.giuliodestri.it/documenti/D01_GuidaAlWeb2.pdf

Indice

Agile Programming

"Se ti andasse a fuoco la casa, spiegheresti ai pompieri cosa devono fare e come? No, perché li reputi esperti e capaci di auto-organizzarsi. La stessa cosa vale per un team Agile"
(Coach Agile, settembre 2013)

Cosa significa Agile? Perché è vantaggioso?

Con il termine "Metodologie Agili" ci si riferisce ad una serie di metodologie di sviluppo software ispirate dal "Manifesto Agile", impiegate per superare i limiti emersi dal modello tradizionale "a cascata" (waterfall). Tramite queste metodologie i team di sviluppo software sono in grado di ridurre sensibilmente il rischio di non rispetto dei tempi e/o di commettere errori di interpretazione dei requisiti, organizzando il proprio lavoro in piccole iterazioni, chiamate *sprint,* della durata di poche settimane (tipicamente 2-3) che permettono il rilascio del software in modo incrementale. In questo modo i progetti software di entità medio-grande possono essere suddivisi in tanti progetti di dimensioni più piccole, aventi ciclo di vita proprio, rilascio in produzione compreso, lavorabili nell'ambito di un singolo sprint.

Agile non è, in se, un processo caratterizzato da regole. E' più una filosofia, un "modo di fare". Detta linee guida e raccomandazioni che vengono poi implementate da diverse metodologie. Tra le più comuni citiamo:

Extreme Programming (XP) - Per le attività ingegneristiche di sviluppo software.

Scrum - Per le attività di gestione del processo nel suo insieme.

Tra i principali vantaggi offerti dall'Agile Programming, vi sono una maggiore qualità del software rilasciato, in quanto ogni parte incrementale di prodotto viene testata nell'ambito di uno sprint, ed una maggiore capacità di adeguamento del team di sviluppo ad eventuali variazioni in corso d'opera dei requisiti da parte del committente, cosa molto attuale nei progetti software odierni.

Uno dei requisiti chiave per il buon funzionamento dell'impiego delle metodologie agili è la continua interazione tra sviluppatori (in toto o tramite rappresentanti) ed il committente (cliente o Product Owner) al fine di aver sempre una visione chiara ed aggiornata di esigenze, requisiti e feedback del lavoro svolto da una parte e dello stato dell'arte del prodotto dall'altra.

Come riportato sul sito web www.agileprogramming.org [8], l'Agile Programming offre ai team ripetute possibilità di valutare un progetto all'interno del suo ciclo di vita, inteso come insieme di sviluppi incrementali, e di aggiornare quindi le valutazioni precedenti in base a cambiamenti del contesto. Queste possibilità sono diretta conseguenza del sistema di lavoro a sprint. Al termine di ogni sprint il team presenta un pezzo di software (quindi un componente software) funzionante al Product Owner per approvazione. Questa enfasi su un pezzo di software garantisce che il team ha correttamente compreso i requisiti. Dal momento che gli sprint si ripetono ed il prodotto continua a guadagnare funzionalità (incrementi di

prodotto), l'agile programming è descritto come "iterativo" e "incrementale". Nel modello tradizionale o a cascata, il team ha una sola occasione per svolgere correttamente ogni parte di un progetto. Non è così nell'agile programming, dove ogni aspetto relativo allo sviluppo è revisionato durante l'intero ciclo di vita di un progetto. Virtualmente, non c'è possibilità che un progetto segua a lungo una direzione errata. Dal momento che il team verifica con il committente lo stato del progetto (ed i rilasci incrementali) cadenze regolari, c'è sempre il tempo per cambiare in corsa (strategia "inspect-and-adapt").

Gli effetti di questa strategia "inspect-and-adapt" sono evidenti. Riducono significativamente i costi di sviluppo ed il time to market. I team Agili proseguono e raffinano la raccolta ed analisi requisiti durante lo sviluppo del prodotto, e questo non può quindi ostacolare il progresso del lavoro svolto dal team. Inoltre dal momento che il team sviluppa in cicli di lavoro corti e ripetitivi, gli Stakeholders hanno l'opportunità di assicurarsi che il prodotto in sviluppo incontri la vision del committente.

In sostanza, si può affermare che l'Agile Programming aiuta le aziende a costruire il prodotto che i clienti desiderano. Anziché consegnare software "fumoso" e sub-ottimizzato, l'Agile Programming mette in condizione i team di costruire il miglior software possibile. L'Agile Programming protegge quindi l'adeguatezza del prodotto rispetto al mercato e previene il rischio che il lavoro svolto finisca in un cassetto senza mai venir rilasciato e questo "accontenta" sia sviluppatori che Stakeholders.

Ruoli Agili

Introduciamo alcuni ruoli coinvolti in un’organizzazione Agile. Essendo Agile una filosofia, le caratteristiche specifiche di dettaglio di ogni ruolo dipendono dal processo e dalla metodologia adottata.

Product Owner: E’ il responsabile dell’attività del team. Accetta o respinge le richieste del committente e, assieme a lui, le prioritizza. Una volta raccolte le richieste del cliente le propone al team sotto forma di attività da fare. In funzione della priorità e della capacità di lavoro del team, queste attività vengono opportunamente inserite negli sprint ai quali il team lavorerà.

Sviluppatori: Cuore pulsante del team. Analizzano tecnicamente le richieste pervenute dal committente tramite il Product Owner e ne stimano la portata. Si occupano poi di design, coding e test del software lavorato durante ogni sprint.

Stakeholders: Tutti coloro che sono coinvolti in un prodotto. Non fanno parte del team. Oltre agli utenti finali, potrebbero essere stakeholders anche altri membri dell’organizzazione che devono essere coinvolti. Ad esempio l’ufficio legale nel caso il software abbia alcuni aspetti legali da gestire (es. trattamento dati personali, oppure accettazione condizioni di utilizzo) oppure gli amministratori di sistema nel caso il software debba essere installato su macchine che richiedono particolari accorgimenti hardware e software.

Business Owners: Uno o più stakeholders. Hanno un ruolo attivo durante il ciclo di sviluppo. Supportano il team qualora il Product Owner e gli sviluppatori dovessero richiedere chiarimenti e vengono coinvolti in test del prodotto in lavorazione. Agli occhi dell'organizzazione sono considerati, al pari del team di sviluppo, responsabili della buona riuscita del progetto.

Agile Coach: E' un esperto delle metodologie Agili. Il suo compito è quello di supportare uno o più team nell'adozione e nell'utilizzo di queste metodologie e quindi contribuire ad una crescita professionale costante dei membri del team.

Agile manifesto e Agile programming

Pubblicato nel 2001, il "Manifesto Agile" (agilemanifesto.org [9]) è il documento in cui vengono elencati i concetti che ispirano la metodologia Agile. I firmatari, in ordine alfabetico, sono Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas.
Di seguito vengono riportate la versione originale tratta da [9] e la traduzione italiana del manifesto (© 2001 gli autori sopra elencati)

"We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more."

"Stiamo scoprendo modi migliori di creare software,
sviluppandolo e aiutando gli altri a fare lo stesso.
Grazie a questa attività siamo arrivati a considerare importanti:

Gli individui e le interazioni più che i processi e gli strumenti
Il software funzionante più che la documentazione esaustiva
La collaborazione col cliente più che la negoziazione dei contratti
Rispondere al cambiamento più che seguire un piano

Ovvero, fermo restando il valore delle voci a destra,
consideriamo più importanti le voci a sinistra."

Il manifesto suggerisce linee guida da adottare in un ambito di Agile programming, ed è stato ispirato da 12 principi così sintetizzati:

- La nostra massima priorità è soddisfare il cliente rilasciando software di valore, fin da subito e in maniera continua.
- Accogliamo i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente.
- Consegniamo frequentemente software funzionante, con cadenza variabile da un paio di settimane a un paio di mesi, preferendo i periodi brevi.
- Committenti e sviluppatori devono lavorare insieme quotidianamente per tutta la durata del progetto.
- Fondiamo i progetti su individui motivati. Diamo loro l'ambiente e il supporto di cui hanno bisogno e confidiamo nella loro capacità di portare il lavoro a termine.
- Una conversazione faccia a faccia è il modo più efficiente e più efficace per comunicare con il team ed all'interno del team.
- Il software funzionante è il principale metro di misura di progresso.
- I processi agili promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitamente un ritmo costante.
- La continua attenzione all'eccellenza tecnica e alla buona progettazione esaltano l'agilità.
- La semplicità - l'arte di massimizzare la quantità di lavoro non svolto - è essenziale.
- Le architetture, i requisiti e la progettazione migliori emergono da team che si auto-organizzano.
- A intervalli regolari il team riflette su come diventare più efficace, dopodiché regola e adatta il proprio comportamento di conseguenza.

Superare i limiti dello sviluppo tradizionale

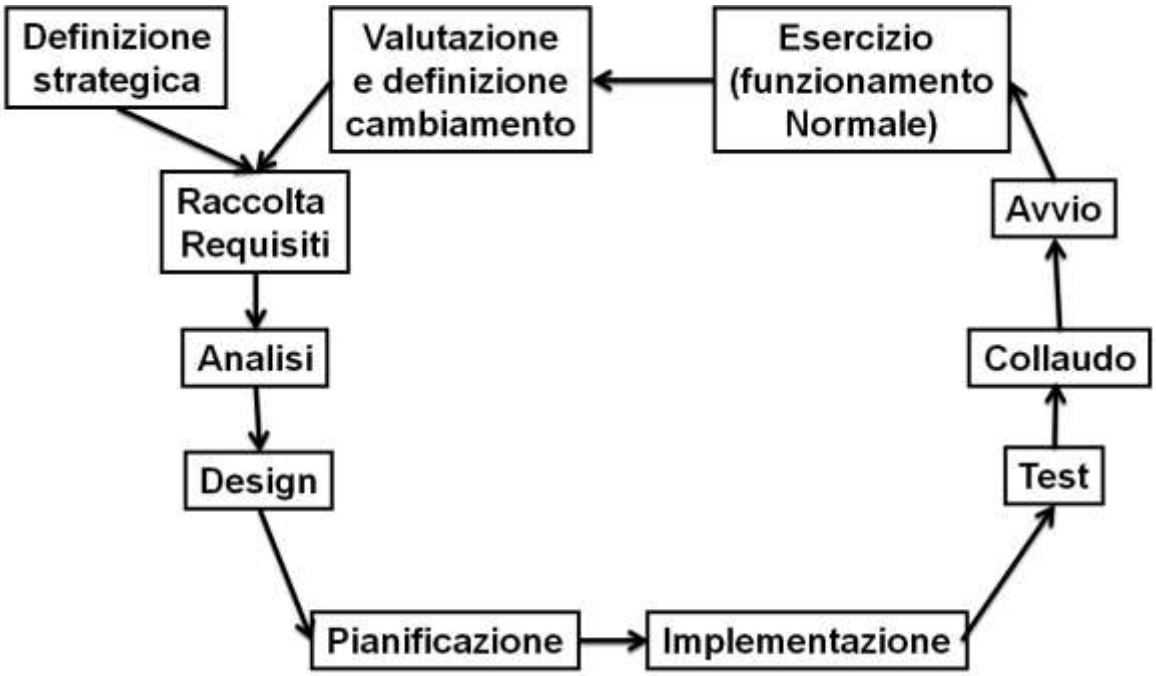


Figura 1 Evoluzione a spirale del modello a cascata

Un articolo pubblicato in [10] evidenzia come nel metodo tradizionale a cascata le fasi dello sviluppo software sono consecutive e sempre nello stesso ordine, ovvero Analisi, Design (Progettazione tecnica), Coding (Implementazione), Testing. Questo modello è ideale per progetti:

Contract Based: Il cliente richiede software che rispetti una serie di requisiti messi per iscritto a priori (esistenza di un contratto). Essendo il modello a cascata "document driven", porta a contratti basati pesantemente sui requisiti. Questo aiuta a garantire che tutto ciò che è specificato nel contratto sia rispettato.

Focalizzati sull'analisi: Alcuni sviluppi software richiedono che l'analisi sia totalmente completata a priori. Questo è il caso di sistemi complessi e/o critici che richiedono diversi step di validazione ed approvazione. Essendo un processo sequenziale, il modello a cascata è adatto a questo tipo di progetti

Non richiedono solo il software: ma anche, ad esempio, il manuale dell'utente, lo schema architetturale, etc. etc. Nello sviluppo tradizionale molti documenti ed artefatti vengono creati in aggiunta al software stesso e, in alcuni casi, questi prodotti sono considerati importanti tanto quanto il software.

A prescindere dalla filosofia adottata, le principali attività da svolgere non cambiano, quel che cambia è quando e come vengono svolte. Vediamo uno schema, sempre pubblicato nell'articolo sopracitato:

	Waterfall	**Agile**
Analisi	Fatta all'inizio. Tutti i requisiti vengono identificati a priori	Tutte le attività sono svolte in parallelo durante l'intero ciclo di vita del progetto
Architettura e design	Successivi all'analisi	
Pianificazione	Fatta all'inizio del progetto. Spesso contrattualizzata	
Coding	Ha una fase a se. Spesso è pilotato dai documenti prodotti nelle fasi precedenti	
Testing	Di solito svolto da processi batch di dimensioni considerevoli	
Approvazione/Valutazione	Ogni fase (Analisi, Design, Coding, Testing) richiede approvazione	

Lo stesso autore in [11], fa un interessante confronto tra la metodologia classica waterfall e le metodologie agili, lo riportiamo di seguito.

Nello sviluppo tradizionale, le fasi di Architettura e Design sono considerate le più critiche. Il team è spinto a creare in anticipo l'architettura che rispetti tutti i requisiti. Questo richiede molte risorse ed energie all'inizio del progetto.

I principali argomenti sostenuti dai tradizionalisti circa l'importanza di avere l'architettura definita in toto dall'inizio sono:

- Il design dell'applicazione non cambia durante lo sviluppo
- Aiuta ad evitare di scoprire in una fase successiva che l'architettura non è adatta allo scopo.

L'Agile, dal canto suo, è basata sull'assunzione che l'incertezza è un fatto comune ed attuale e così diffusa che nessuna architettura definita rigidamente in principio può, o almeno potrebbe, essere completamente corretta durante tutto il processo di sviluppo software. Quindi la filosofia Agile adotta un approccio il più semplice possibile ed incline ai cambiamenti nei confronti dell'architettura e del design di un progetto.

Dal punto di vista dell'integrazione dei componenti dell'applicazione, l'Agile programming pone il focus sul fatto che questa avvenga al più presto (incoraggia al rilascio di funzionalità incrementali al termine di ogni sprint) e che vengano rilasciate funzionalità anziché moduli. Il focus sull'integrazione è ritenuto molto importante in quanto spesso rappresenta una delle parti più complesse durante la costruzione di un'applicazione software.

Dall'altro lato, l'approccio waterfall è focalizzato sul completamento di moduli tecnici evidenziati dalla fase iniziale di Architettura e Design. Questo spesso causa parecchi problemi e deviazioni sulla tabella di marcia, a causa dei problemi che tipicamente sorgono durante l'integrazione tra loro dei componenti di un'applicazione.

Filosoficamente, l'Agile promuove molti principi volti ad evitare gli sprechi, ad esempio incoraggiando a tenere la minima documentazione indispensabile, favorendo invece l'adozione di potenti pratiche ingegneristiche (TDD, pair programming, refactoring, …), open door management e team auto-organizzati. Questo deriva dal principio che il punto di forza dello sviluppo sotware sono le persone.

Il metodo tradizionale invece è più orientato ai processi ed alla documentazione. Non pone nelle persone la stessa importanza/fiducia delle filosofie agili, suggerisce invece misure atte a misurarne il rendimento e controllarne l'operato.

In definitiva, queste sono le principali differenze tra sviluppo Tradizionale e Agile Programming

Agile	**Waterfall**
Architettura infomale ed incrementale	Architettura molto documentata e completata prima dell'inizio del coding
La ownership del codice è condivisa tra gli sviluppatori	Ogni sviluppatore è responsabile di una determinata area
Integrazione continua	Integrazione fatta alla fine o a tappe predeterminate
Focalizzato sul completamento delle storie (funzionalità) in piccole iterazioni (sprint)	Focalizzato sul completamento di moduli (parti dell'architettura) a scadenze prefissate
Ben ingegnerizzato (TDD, XP, design patterns,...)	Non necessariamente ingegnerizzato
Pochi processi e documentazione	Molti processi e documentazione
Sviluppatori cross-competenti, ben informati su tutte le tecnologie	Pochi architetti/sviluppatori hanno la visione di insieme, le altre figure

impiegate.	professionali sono molto specializzate
Ruolo principale: sviluppatori	Ruoli principali: Architetti e sviluppatori
Open door policy: gli sviluppatori sono incoraggiati a rivolgersi al business ed al management in qualsiasi momento. Il punto di vista di tutti deve essere considerato.	Solo pochi sviluppatori ed alcuni architetti possono rivolgersi al business. E principalmente prima dell'inizio dello sviluppo e/o alle scadenze prefissate (milestones)

L'Extreme Programming

L'Extreme Programming (XP) è una metodologia di sviluppo adottata per mettere in pratica le filosofie Agili durante lo sviluppo di uno o più processi software. Tramite questa metodologia i team di sviluppo sono in grado di rilasciare il software richiesto in tempi rapidi, così come è stato richiesto e mette in condizione gli sviluppatori di rispondere in maniera tempestiva alle mutate esigenze del committente o del Product Owner, aspetto molto importante.

L'extreme programming enfatizza il lavoro di team: manager, clienti e sviluppatori sono tutti parte di un team collaborativo, in grado di auto-organizzarsi in modo che il problema (lo sviluppo di un progetto software in questo caso) venga risolto nel metodo più efficiente possibile.

Un aspetto sorprendente dell'XP sono le sue semplici regole, dal significato trascurabile, se prese singolarmente, di grande efficacia se applicate, anche solo in parte, assieme.

Di seguito una breve descrizione, tratta da extremeprogramming.org [12] delle regole e delle pratiche adottate, ed una serie di accorgimenti utili, suddivisi per fase di progetto:

Pianificazione (Planning Game)

Il **Planning Game** è una riunione tenuta tipicamente una volta a settimana, ed è composta da due fasi.

La fase di **Release Planning** ha lo scopo di determinare, coinvolgendo il cliente, quali requisiti devono essere lavorati nelle le prossime tre/quattro iterazioni e quando il software che li soddisfa dovrebbe essere rilasciato. I requisiti vengono tracciati sotto forma di **User Stories** per le quali gli sviluppatori forniranno, una volta appresi i requisiti, una stima, espressa in **story-points** della mole di lavoro necessaria per poterla rilasciare. Una volta che la storia è stata stimata, sviluppatori e cliente definiscono in quale sprint sarà lavorata e, di conseguenza, quando il lavoro verrà consegnato. In questa fase gli sviluppatori devono tenere conto della capacità di lavoro a loro disposizione ed il cliente sarà quindi chiamato a fare delle scelte assegnando una priorità alle sue richieste.

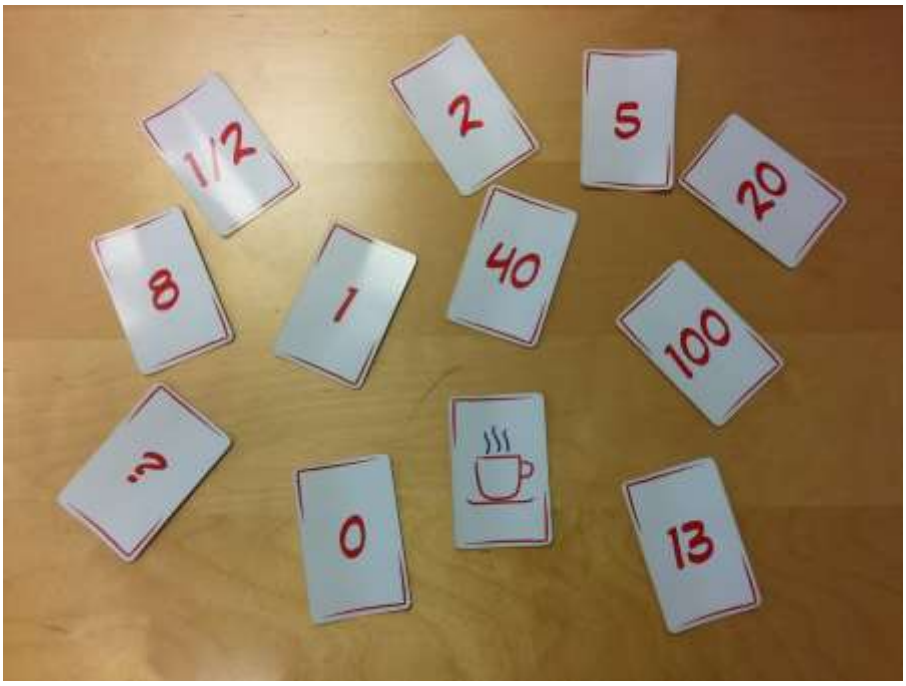


Figura 2 Le "Planning Poker Cards". Il Planning Game è, appunto, un "gioco". Ogni membro del team può usare una carta per esprimere la sua stima.

La seconda fase invece è quella nominata **Iteration Planning**, durante la quale il cliente non è coinvolto. In questa fase gli sviluppatori pianificano la propria attività da svolgere sulle storie definite durante la fase precedente. Durante questa fase le User Stories possono venire suddivise in task tecnici. Gli sviluppatori (o le coppie di sviluppatori) prendono in carico i task ai quali intendono lavorare nell'ambito dell'iterazione.

Gestione

Accorgimenti gestionali utili, volti a mettere il team in condizione di lavorare al meglio:

- Sistemare il team in un unico open space.
- **Impostare un ritmo sostenibile**. La capacità di lavoro del team è limitata. E' bene che sia chiaro cosa il team è in grado di fare durante uno sprint e quali attività invece devono essere suddivise su più sprint. Non ha senso, se non in casi di emergenza, fare affidamento su straordinari ed eroismi individuali per terminare in tempo quanto pianificato.
- **Stand up meeting.** 10', in piedi, all'inizio di ogni giorno lavorativo per condividere le attività svolte e quel che si prevede di fare in giornata.
- Misurare la **Project Velocity.** La Velocity indica quanto lavoro stà venendo effettivamente fatto nell'ambito del progetto. Un metodo per misurarla consiste nel confrontare quanti story-points sono stati evasi (storie completate) con quanti ne sarebbero stati evasi con un andamento

regolare. Esempio: in uno sprint da 40 story points della durata di 20 giorni lavorativi ci si aspetterebbe che al 14° giorno il numero di story-points evasi non si discosti troppo da 28. Nel caso vi sia uno scostamento è necessaria una analisi delle cause (stime errate? Urgenze?)

- **Spostare la gente.** Tutti devono essere in grado di lavorare ad ogni aspetto del progetto, in modo da evitare colli di bottiglia o squilibri del carico di lavoro all'interno del team.
- **Cambiare quel che non funziona.** Nel puro spirito auto-organizzativo del team, ogni cosa che "non va" deve essere affrontata e risolta assieme.

Design

- **Semplicità.** Non andare a ricercare sempre soluzioni sofisticate, e non reinventare sempre la ruota.
- Impiegare **Metafore di sistema.** Ovvero assegnare nomi consistenti e facilmente intuitivi a tutti gli oggetti coinvolti, in modo che siano facilmente spiegabili e comprensibili.
- **CRC (Class Responsibility Collaboration) cards game** durante le sessioni di design. Si impiega un foglietto di carta per ogni oggetto (classe) ipotizzato. Il foglio riporta nome, responsabilità e collaboratori della classe. In questo modo il team definisce "su carta" (è proprio il caso di dirlo) l'architettura di un insieme di classi deputate alla soluzione di un problema. Tutto il team è coinvolto e tutti hanno una chiara visione dell'architettura prodotta.
- Creare **soluzioni spike** per ridurre i rischi. In caso di problemi a soluzione incerta o fumosa, il team deputa una o due persone allo studio del problema e di una o più possibili soluzioni, per un periodo predefinito di tempo, in modo che il problema possa essere affrontato meglio in un prossimo sprint.
- Non introdurre **funzionalità in anticipo.** Le funzionalità rilasciate devono essere quelle richieste, non quelle che "potrebbero servire in futuro". Se serviranno, verranno richieste.
- **Refactor** ovunque, quando possibile. Riscrivere parti di codice di difficile comprensione ("marcite") in modo da renderne più semplice la leggibilità e la manutenibilità.

Coding

- Cliente **sempre disponibile.** In XP il cliente è inteso come un rappresentante dell'utenza finale. E' fondamentale che ogni membro del team possa chiarire al più presto direttamente con lui eventuali dubbi che

dovessero insorgere. Chi sia questo interlocutore deve essere chiaro a tutti i membri del team.

- Il codice deve essere scritto seguendo **standard concordati (code-conventions).**
- **Test Driven Development (TDD).** Il codice viene scritto a partire dai test. Vista l'importanza e soprattutto l'efficacia della pratica, verrà trattata in un paragrafo apposta.
- **Pair Programming.** Il codice viene scritto da una coppia di sviluppatori usufruendo di una singola workstation. Uno sviluppatore utilizza tastiera e mouse ed è focalizzato sul codice in scrittura. L'altro segue più la visione di insieme ed opera una continua review del codice scritto dal collega. I ruoli vengono invertiti dopo non più di un'ora, l'ideale è introdurre una pausa durante lo scambio di ruoli.
- **Commit (integrazioni) frequenti.** Ideale 2 volte al giorno.
- **Code Review.** Il codice committato nel sistema di controllo versione (VCS) è frutto di revisione da parte di altri membri del team, che possono fornire utili suggerimenti ed indicazioni circa quanto scritto tramite commenti nel codice che verranno recepiti o meno dall'autore del pezzo interessato.
- Impiegare **un computer dedicato all'integrazione.** Esistono numerosi tools (Jenkins, ad esempio) che permettono di avere sempre sotto controllo la compilazione ed il processo di build e packaging della code base integrata.
- **Collective ownership. "**Il codice è di tutti", tutti devono essere liberi di modificarlo allo scopo di refattorizzarlo, fixare bug, etc. senza chiedere il permesso all'autore originale. Questo contribuisce alla diffusione della conoscenza e quindi alla riduzione dei colli di bottiglia.

Testing

- Tutto il codice deve essere coperto da classi di **Test Unitari (Unit Test).** I test unitari coprono singole classi o gruppi ristretti di classi a forte collaborazione e vengono eseguiti durante la build del codice.
- Il codice deve superare i test unitari prima che possa essere rilasciato. Il team deve assicurare sempre il corretto funzionamento dei test unitari. La manutenzione dei test deve essere importante tanto quanto quella del codice dell'applicazione.
- Quando viene rilevato un bug, vanno creati i test che assicurino che non si ripresenti in futuro.
- I **test di accettazione (Acceptance tests)** vanno eseguiti spesso ed il loro risultato viene pubblicato. Questa tipologia di test sono test a scatola

chiusa, scritti sulla base delle User Stories e verificano che il sistema dia il risultato atteso.

User Stories

I requisiti del cliente o del Product Owner vengono convertiti, durante il Planning, in User Stories. Una User Story rappresenta un prodotto (anche parziale) che verrà consegnato al termine dello sprint. Può essere una storia a se stante, nel caso la richiesta sia semplice, come ad esempio la raccolta di una informazione supplementare tramite web service.

Nell'ottica dei rilasci incrementali che stà alla base dell'XP, una storia può però anche essere parte di un progetto più ampio che è stato quindi suddiviso in più storie che non necessariamente verranno completate tutte nell'arco di un solo sprint. Ad esempio un progetto che prevede una nuova integrazione web services con servizi esterni all'organizzazione aziendale potrebbe esser diviso in storie del tipo: stabilire la connessione col servizio esterno, generare gli stub per l'interazione col web service, implementare la chiamata ad un determinato servizio, e così via.

Le storie vengono scritte in linguaggio semplice e comprensibile, spesso riccorrendo al paradigma: "In quanto *[tipologia di utente]* voglio *[requisito della storia]* in modo *da [scopo del requisito]*". Esempi di User Stories potrebbero essere: *"In quanto responsabile della user-experience voglio che il pulsante "contattaci" presente nella home page del nostro sito venga posizionato in alto a destra, in modo da renderlo più visibile all'utente finale"* oppure *"In quanto applicazione di comparazione prezzi, voglio che venga chiamato il servizio "catalogo" dei web services esposti dal fornitore XXX, in modo da poter avere sulla mia applicazione i loro articoli ed i relativi prezzi"*.

Le User Stories vengono accompagnate da criteri di accettazione (acceptance criteria). Una serie di test black box, definiti appunto **acceptance tests,** che verificano l'effettivo rispetto dei requisiti da parte della storia. Questi test vengono eseguiti solitamente a mano da persone diverse rispetto allo sviluppatore o agli sviluppatori che hanno lavorato alla storia (potrebbero essere anche il cliente o il Product Owner) prima che questa venga rilasciata in produzione, allo scopo di "certificare" il lavoro fatto dagli sviluppatori.

L'entità di una User Story deve essere sempre stimabile in fase di planning. Nel caso questo non sia possibile, sono necessarie ulteriori iterazioni con Product Owner e/o cliente al fine di raccogliere gli elementi necessari alla stima. Nel caso questo non sia possibile, si ricorre a **spike.** Per spike si intende un task che non può esser considerato una User Story in quanto non ne sono chiari requisiti ed entità, ma è comunque necessario che uno o più sviluppatori investano tempo a definire meglio l'attività richiesta affinché possa diventare una User Story.

Test Driven Development (TDD)

Spesso la fase di testing è vista come un fastidio da parte degli sviluppatori software, ed ancor di più lo è la necessità di dover scrivere delle classi di test (codice che testa il codice). La tecnica TDD consiste nella stesura del codice a partire dalle classi di test che ne assicurano il buon funzionamento, test unitari (Unit Test) in questo caso. Questo potente approccio garantisce la copertura di test quasi totale (test coverage) del codice scritto e riduce significativamente il rischio che nel codice possano esservi bug.

Una volta individuati uno o più scenari riguardanti il funzionamento ed il comportamento del software che si va a scrivere, questi vengono convertiti in test unitari. Utile in questi casi è il paradigma "Given-When-Then". Ad esempio "Dato che (Given) 4 diviso 2 risulta 2, Quando (When) chiamo il metodo dividi, Allora (Then) il metodo restituisce 2" ma anche "Dato che 0 non è un divisore valido, quando chiamo il metodo dividi, allora il metodo mi restituisce un errore".

Una volta individuati tutti i casi di test si procede secondo questo ciclo, introducendo un caso di test alla volta:

1. Si scrive il codice relativo al caso di test
2. Lo si lancia e lo si lascia fallire
3. Si scrive codice con il solo scopo di far avere esito positivo a tutti i test scritti (compreso quello fallito al punto 2)
4. Si procede al refactoring del codice scritto al fine di eliminare gli *smell.* Gli *smell* sono porzioni di codice che potrebbero, col tempo, portarlo a degenerare (esempio metodi lunghi o ripetuti, ricorso a variabili inutili, …).

Un ulteriore beneficio dato da questa procedura è che il codice scritto, essendo scritto in modo incrementale e soggetto a continui refactoring rimane semplice e facilmente comprensibile e manutenibile.

Valori ispiratori dell'XP

L'Extreme Programming è basato su valori. Le regole sopraelencate altro non sono che la naturale estensione e la conseguenza della massimizzazione di questi valori. Infatti, l'XP non è in realtà un insieme di regole ferree, quanto un modo di lavorare in armonia con i valori personali ed aziendali. I valori ispiratori dell'XP, così come sono definiti in [13] sono di seguito descritti.

"Semplicità: *Faremo quanto necessario e richiesto, nulla di più. Questo massimizzerà il valore creato a fronte dell'investimento fatto. Faremo piccoli e semplici passi verso l'obiettivo e miticheremo i fallimenti quando accadono. Creeremo qualcosa di cui saremo orgogliosi e lo manterremo a lungo a costi ragionevoli*

Comunicazione: *Siamo tutti parte di un team e comunichiamo faccia a faccia quotidianamente. Lavoreremo assieme su tutto, dai requisiti al codice. Creeremo assieme la miglior soluzione possibile al problema.*

Feedback: *Prenderemo seriamente ogni commitment ad ogni iterazione, rilasciando software funzionante. Mostriamo spesso il lavoro fatto (demo), quindi prestateci attenzione e richiedete ogni modifica ritenuta opportuna. Parleremo del progetto ed adatteremo ad esso il nostro processo e non il contrario.*

Rispetto: *Ognuno dà e riceve il rispetto che si merita in quanto membro del team. Ognuno apporta valore anche quando questo è semplicemente entusiasmo. Gli sviluppatori rispettano l'esperienza dei clienti, e vice versa. Il Management rispetta il nostro diritto di accettare responsabilità e ricevere autorevolezza circa il nostro stesso lavoro.*

Coraggio: *Diremo la verità circa progressi e stime. Non documentiamo scuse per il fallimento perché ci aspettiamo di riuscire nel nostro scopo. Non abbiamo paura di nulla perché nessuno lavora da solo. Ci adatteremo ai cambiamenti ogni volta che dovessero accadere."*

Un caso di successo: SCRUM

Lo Scrum è una metodologia Agile iterativa ed incrementale per la gestione dello sviluppo di prodotti software. E' definita come "una strategia flessibile ed olistica di sviluppo, dove il team lavora unito al raggiungimento di un obiettivo comune". Il termine Scrum è mutuato dal rugby, infatti, indica il pacchetto di mischia. Come metodologia Agile, mette in discussione i principi dell'approccio tradizionale ed incoraggia la auto-organizzazione dei team e la comunicazione continua (fisica o virtuale) tra membri del team.

Un principio chiave dello Scrum è il riconoscimento che, durante un progetto, il committente può cambiare idea su requisiti e necessità, e che questi cambi di idea non possono essere gestiti facilmente tramite un progetto gestito con approccio tradizionale.

Scrum pone l'accento sull'abilità del team a rilasciare in maniera rapida e rispondere altrettanto rapidamente ai cambiamenti emergenti anche eventualmente a discapito di una comprensione e/o definizione solo parziali del problema nella fase iniziale (fonte: [14])

Figura 3 Un team Scrum davanti alla propria "board"

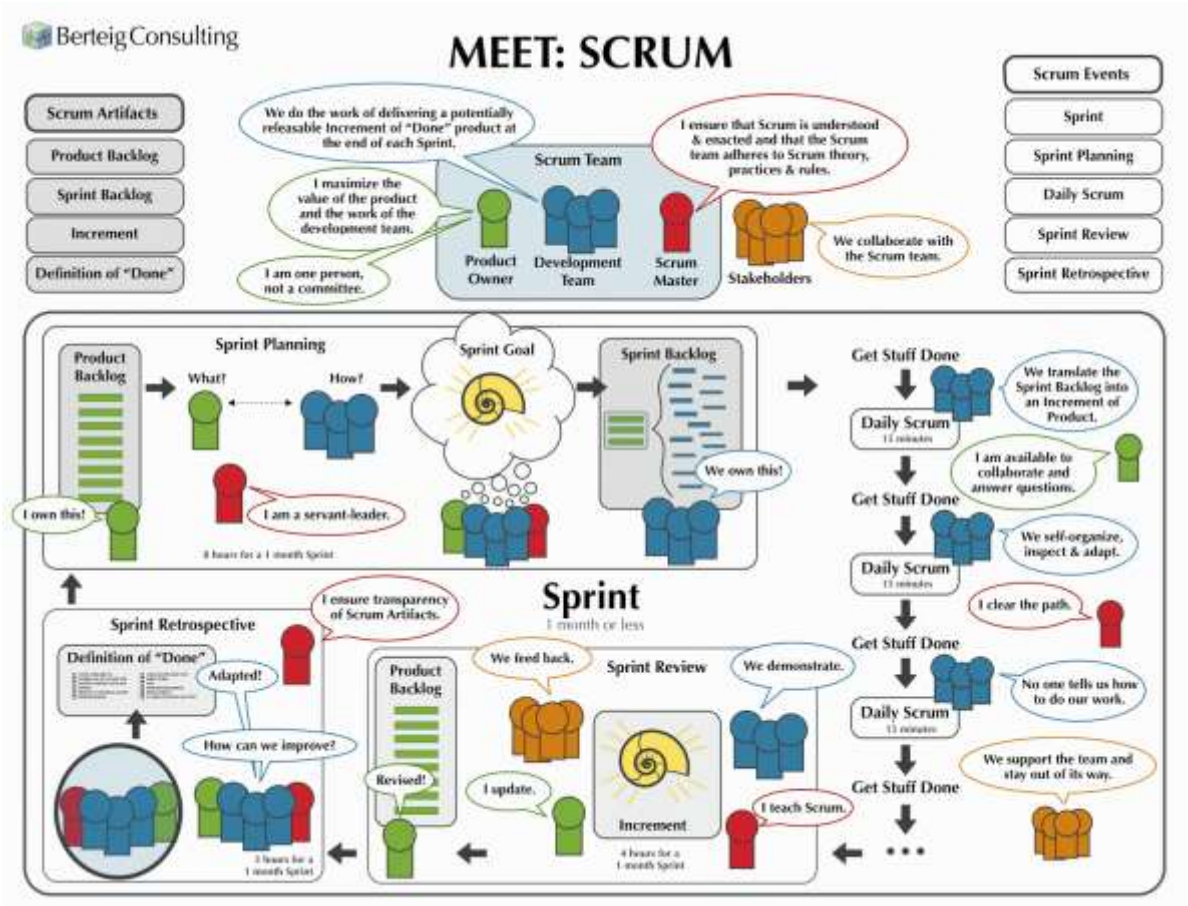
Nata come metodologia di sviluppo software, è stata adottata da diverse multinazionali (es. Toyota) per la gestione dei processi in altre aree aziendali.

Ruoli

Un team Scrum è composto dai già descritti **Product Owner** e **Developers**, dallo **Scrum Master,** il cui compito è quello di agevolare il lavoro del team, collaborando col Product Owner alla definizione del progetto affinchè i requisiti siano chiari, assicurandosi che il team sia in grado di auto-organizzarsi secondo i principi Scrum (in questo caso agisce com **Agile Coach**) e facendosi carico di tutte le incombenze che potrebbero distrarre gli sviluppatori dalla loro attività primaria. Lo Scrum Master, ad esempio, si rivolge ai sysadmins in caso di necessità tecniche (acquisto e manutenzione workstations, richiesta server di sviluppo, collaudo e/o produzione) oppure alle funzioni preposte in caso di necessità logistiche (scrivanie, stanze, materiale di cancelleria, …).

Metodologia di lavoro

Di seguito si riporta un interessante schema, tratto da [15] che riassume la metodologia di lavoro Scrum.



Backlog

Il backlog è un elenco di User Stories modellate dal Product Owner assieme al committente ed eventualmente uno o più sviluppatori, sulla base di quelli che sono i requisiti del cliente. Le User Stories possono anche venir modellate in maniera “approssimativa” e raffinate successivamente.

Esistono tre tipi di backlog, **di prodotto**, **di team** e **di sprint.** Il backlog di prodotto raccoglie i requisiti divisi per tipologia di prodotto, e quindi per i relativi Stakeholders e Business Owners. Queste storie vengono poi smistate dai Product Owner dei vari team all’interno del **backlog di team** del team che se le prenderà in carico. Una volta che sono in questo backlog, il team le può analizzare e stimare (grooming) ed includerle nel **backlog di sprint** relativo allo sprint nel quale il team le lavorerà.

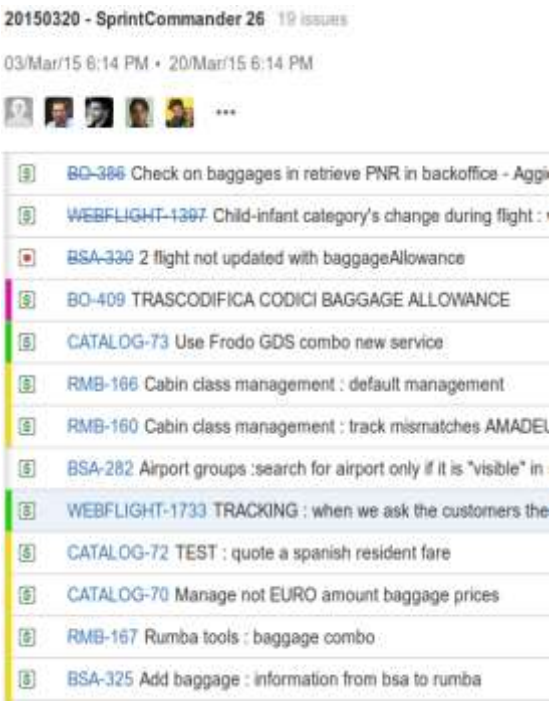


Figura 4 Esempio di backlog di sprint, sono indicati gli estremi dello sprint e (alcune) storie in lavorazione.

Grooming

Dal momento che la metodologia Scrum incentiva la definizione “parziale” delle User Stories nella loro fase iniziale, il team è chiamato a fare attività di Backlog Refinement una volta che queste sono state prese in carico dal Product Owner ed inserite nel backlog di team.

Questa attività viene svolta tramite riunioni periodiche di durata variabile definite sessioni di **grooming** (letteralmente “toelettatura”). Durante queste sessioni il team analizza (“spulcia”) i requisiti del cliente e, se questi sono abbastanza chiari,

procede alla definizione dei dettagli tecnico/implementativi della storia ed alla eventuale sua suddivisione in più task tecnici da svolgere, viceversa vengono contattati gli stakeholders della storia per ulteriori chiarimenti.

Dopo aver definito i dettagli della storia, il team procede a stimarne l'entità. Unità di misura tipica dell'entità di una storia sono gli **story points.** La sequenza più comune adottata per definire gli story points è la serie di Fibonacci. Nello spirito di auto-organizzazione del team, la metodologia Scrum sconsiglia di paragonare il concetto di story point ad altre misure (giorni/uomo in primis). L'unico modo per misurare uno story point è paragonare tra loro le storie e le relative stime fornite dal team. Secondo questo principio, il concetto di story point può variare da team a team.

Planning e Scrum Board

Prima dell'inizio di ogni sprint il team si riunisce in una sessione di **planning.** Durante questa riunione, lo Scrum Master calcola il numero di story points disponibili durante lo sprint in partenza, in funzione di durata dello sprint, presenze degli sviluppatori ed andamento degli sprint precedenti (rapporto tra story points pianificati e story points effettivamente evasi).

E' buona norma tenere conto anche del tempo investito dal team su interferenze esterne a cui può essere soggetto (imprevisti, urgenze, ...). In funzione degli story points disponibili, il Product Owner propone al team un elenco di storie da mandare in lavorazione, prese dal backlog di team tra quelle già analizzate e stimate durante un grooming.

Gli sviluppatori accettano le proposte del Product Owner, oppure propongono modifiche circa le storie da lavorare. Il Product Owner, accetta o respinge le proposte degli sviluppatori in funzione delle priorità delle storie per i relativi Business Owner. Le storie selezionate per lo sprint, entrano quindi a far parte del backlog di sprint.

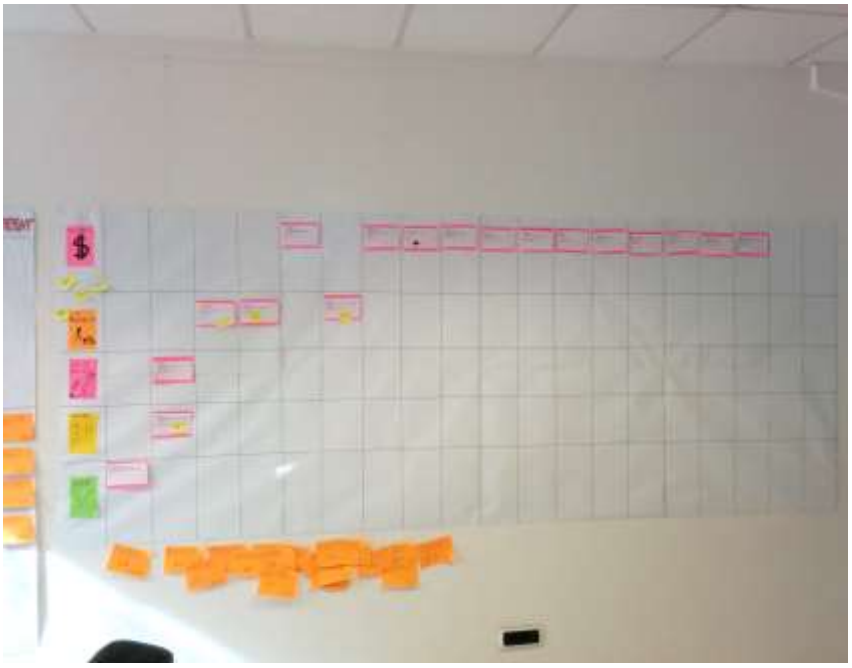


Figura 5 Una Scrum Board alla quale sono appese le storie appartenenti allo sprint (colonne) e il loro stato (righe).

Uno dei punti della metodologia Scrum, afferma che tutto ciò che è in lavorazione all'interno di un team deve essere "visualizzato". Nell'ambiente di lavoro di un team Scrum "i muri parlano". Sono presenti infatti grafici, poster, post-it e qualunque strumento possa contribuire alla "visualizzazione" del lavoro da parte del team.

Uno strumento sempre presente è la "Scrum Board", detta anche "Scrum Wall". Si tratta di una tabella a doppia entrata nella quale da un lato vengono inserite le card relative alle storie in lavorazione durante lo sprint, ed i relativi task e dall'altro lo stato di lavorazione del task o della storia (ad esempio: "Da fare", "In lavorazione", "Da revisionare", "In test", "Fatto").

L'organizzazione degli stati da inserire nella Scrum Board, così come le regole per il passaggio da uno stato all'altro sono lasciate all'auto-organizzazione del team. In particolare il team definisce un set di regole ("Definition of Done", DoD) che disciplinano la messa in "Fatto" di un task o di una storia.

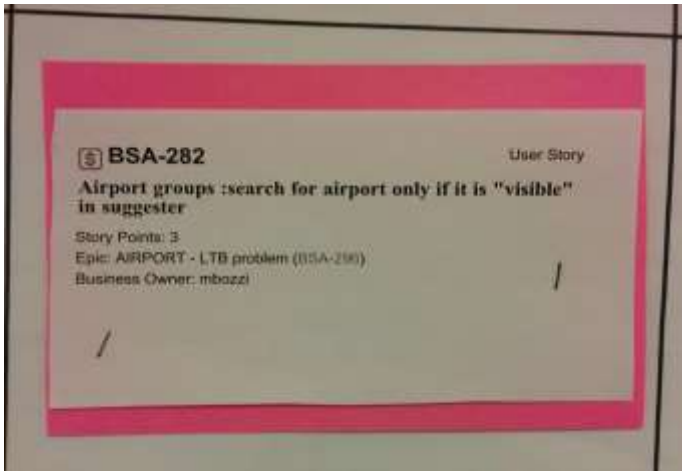


Figura 6 Esempio di User Story così come appare sulla Scrum Board

Figura 7 Descrizione e criteri di accettazione della storia di cui sopra

Sprint

Il lavoro di uno Scrum team viene scandito dagli sprint. Durante gli sprint gli sviluppatori, soli o in pair lavorano alle User Stories pianificate durante la sessione di planning. Quotidianamente il team si riunisce in **stand-up meetings,** incontri tenuti davanti alla Scrum-Board della durata massima di 10’ - 15’ (e proprio per questo svolti in piedi, stand-up). Durante questi incontri gli sviluppatori condividono tra loro l’attività svolta il giorno precedente ed i programmi per la giornata in corso. Eventuali spunti che richiedono discussioni più approfondite (es. dubbi su dettagli tecnici da adottare) che dovessero emergere durante lo stand-up vengono rimandati ad incontri successivi senza che sia necessaria la partecipazione dell’intero team.

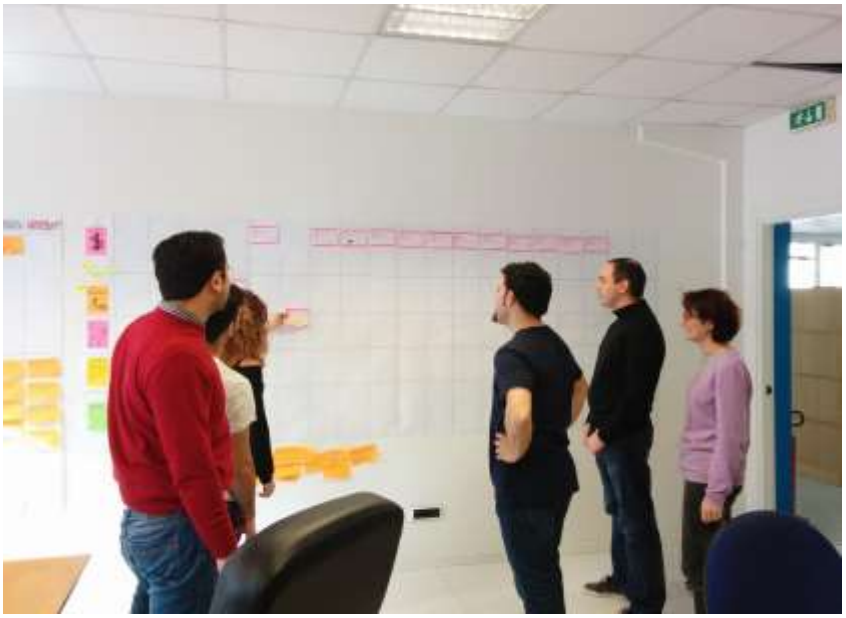


Figura 8 Stand up meeting in corso

Al termine dello stand-up, viene aggiornata la **Burndown Chart,** un grafico giorni/story-points, indicando il numero di story points ancora da evadere alla data dello stand-up. Sulla Burndown Chart viene anche riportato l'andamento ideale di uno sprint (stesso numero di story points evasi tutti i giorni).

L'analisi degli scostamenti tra andamento ideale e quello reale è un indicatore sia delle prestazioni del team che, soprattutto, della correttezza delle stime fatte durante le sessioni di grooming. Un andamento sotto la media, più veloce di quello ideale potrebbe indicare storie sovrastimate. Parimenti, un andamento troppo sopra la media, più lento rispetto all'ideale, indica uno sprint problematico, nel corso del quale le storie sono state sottostimate, oppure la presenza di situazioni particolari (es. urgenze) che hanno inciso sul tempo a disposizione per la lavorazione delle storie.

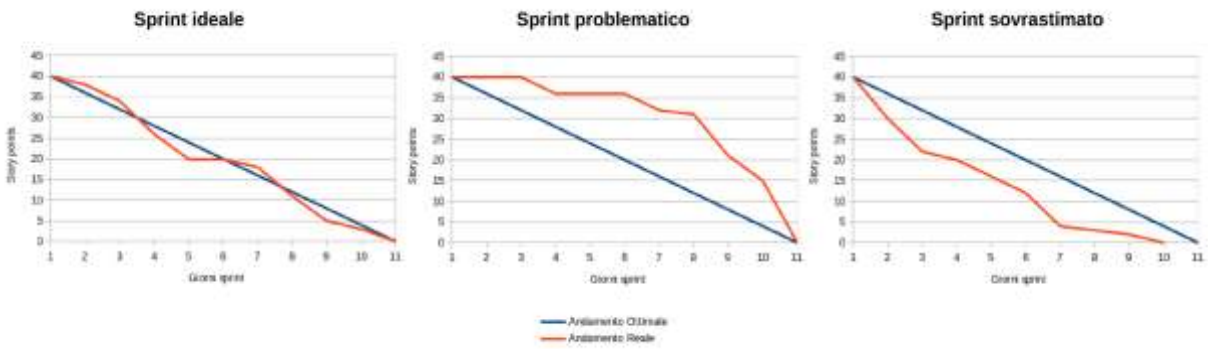


Figura 9 Esempi di burndown chart

Demo

Nei giorni immediatamente precedenti o successivi il termine dello sprint, il team presenta al Product Owner ed agli Stakeholders il lavoro fatto e prossimo al rilascio in produzione, organizzando una sessione di **Demo.** In questo modo gli Stakeholders hanno, tra le altre cose, evidenza del fatto che gli investimenti fatti nelle ultime settimane (lavoro degli sviluppatori e costi accessori) saranno a brevissimo ripagati con il valore del prodotto creato dal team (ROI immediato o quasi).

Retrospettiva

Terminato lo sprint, il team si riunisce in **retrospettiva.** Un meeting dedicato esclusivamente al team, senza interferenze esterne (se non concordate e motivate). Durante questo incontro, viene discusso l'andamento dello sprint, le cose andate bene, quelle andate male e quindi migliorabili. Vengono decise azioni da intraprendere negli sprint successivi in modo che il team possa lavorare in condizioni migliori rispetto a quello precedente e tutti siano in grado di rendere al meglio.

E il giro ricomincia..

Continuous Development

You can't just ask customers what they want and then try to give that to them.
By the time you get it built, they'll want something new.
[Steve Jobs]

Il processo di Continuous Development

Con il termine Continuous Development ci si riferisce ad una serie di tecniche che permettono lo sviluppo iterativo di applicazioni software. Le funzionalità sviluppate, così come le modifiche, diventano immediatamente parte di un prodotto software. Senza bisogno di attendere l'esecuzione di processi di integrazione. A seconda della tecnica adottata, vedremo che il concetto di "immediatamente disponibile" può variare.

Continuous Integration

La Continuous Integration è una pratica tramite la quale gli sviluppatori committano sul sistema SCM il proprio lavoro in modo frequente, almeno una volta al giorno. Ogni commit scatena un processo di compilazione e build della codebase presente nel sistema SCM (esempio GitHub, Svn, …) che la verifica, eseguendo anche una suite di test automatici predisposti dagli sviluppatori. Nel caso qualcosa non vada per il verso giusto durante questo processo di build e test (condizione detta di "build rotta") gli sviluppatori ricevono notifica immediata del problema e possono porvi rimedio immediatamente, eventualmente facendo rollback del commit che ha causato problemi. Uno dei requisiti della pratica CI è che il codice presente sotto SCM sia sempre compilabile, buildabile e superi i test automatici a cui è sottoposto.

Questo approccio porta ad una significativa riduzione dei problemi di integrazione e permette ai team di sviluppare software coeso in maniera rapida.

Si noti come questa pratica sia in netta contrapposizione ai metodi tradizionali, che prevedono integrazione differita, che avviene solamente a fine sviluppo, dopo un periodo che può durare settimane (ma anche mesi o addirittura anni!) e si pensi anche a quanto problematico possa essere gestire problemi di integrazione che dovessero emergere in questa fase.

Continuous Delivery

La Continuous Delivery è la naturale estensione della Continuous Integration.

Tramite questo approccio, gli sviluppatori garantiscono che ogni modifica apportata al codice, committata sull'SCM ed integrata (Continuous Integration) è potenzialmente rilasciabile in produzione tramite un processo automatico avviato con un click (push button deploy).

Per arrivare ad avere del software rilasciabile è necessario che la codebase presente sull'SCM sia sempre buildabile. Ricordiamo che la build prevede l'esecuzione sul codice compilato di una serie di test automatici volti a garantire il rispetto dei requisiti da parte del software. Le pratiche di Continuous Delivery non escludono completamente l'esecuzione di test manuali che, possono venire eseguiti, per particolari funzionalità critiche non testabili in modo automatico, oppure per funzionalità nelle quali è necessario un feedback utente, prima di lanciare il processo di pubblicazione del software in produzione con un semplice click.

Questa pratica tende a semplificare, in termini di operazioni da compiere, un processo da sempre critico come il rilascio in produzione del software.

Spesso il rilascio viene eseguito raramente a causa sia della complessità delle operazioni da fare, sia per la "paura" che qualcosa vada storto durante queste operazioni unita a quella di rilasciare in produzione software malfunzionante. A questo si aggiunge, talvolta, il fatto che il rilascio ha una durata importante e viene fatto in orari in cui il sistema è sottoposto a carichi minori, che spesso coincidono con orari festivi e/o notturni.

Le pratiche di Continuous Integration e Continuous Delivery permettono di superare agilmente questi limiti introducendo test a copertura del codice prodotto, ed una sequenza automatizzata delle operazioni di rilascio. Tra le conseguenze dei rilasci frequenti una delle più importanti è la riduzione della differenza (delta) tra due versioni consecutive e quindi un rollback non troppo complicato, e al tempo stesso non troppo penalizzante per gli utenti, che dovranno temporaneamente rinunciare alle nuove funzionalità rilasciate.

Continuous Deployment

Lo step successivo è il Continuous Deployment. Tramite questo processo, viene rimosso l'intervento umano minimo previsto dalla Continuous Delivery che avvia il rilascio in produzione del software. Potenzialmente, ogni commit delle modifiche al codice innesca un processo di compilazione, build e testing automatico del software. Qualora la build del codice ed i test automatici abbiano esito positivo, si procede alla crezione di eseguibili (war, exe, jar,...) e, sempre automaticamente, alla pubblicazione in produzione dell'eseguibile generato.

Nei progetti per i quali vengono adottate la Continuous Delivery ed il Continuous Deployment, di norma, non si applicano regole di stop allo sviluppo (code freeze). Trattandosi comunque di metodologie di ispirazione Agile, questo aspetto viene lasciato all'auto-organizzazione degli sviluppatori.

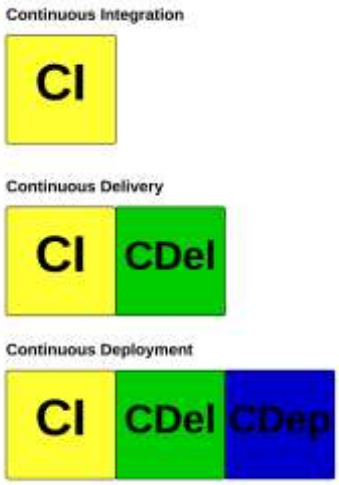


Figura 10 Schema logico che illustra come Continuous Delivery e Continuous Deployment siano l'estensione della Continuous Integration.

Un concetto fondamentale nel Continuous Deployment è quello di **pipeline,** un pattern che copre tutte le fasi della produzione software, dallo sviluppo al rilascio. La pipeline modella una sequenza di operazioni che vengono svolte automaticamente, dal momento in cui una modifica software viene committata sull'SCM, a quando questa diviene parte del software rilasciabile. L'esecuzione della pipeline può partire automaticamente ogni volta che vengono effettuati commit sulla codebase del progetto, può essere pianificata per girare ad orari o intervalli predefiniti, oppure può essere lanciata manualmente da personale abilitato a farlo.

Figura 11 Schema della pipeline.

Cosa serve per impostare il processo di Continuous Deploy

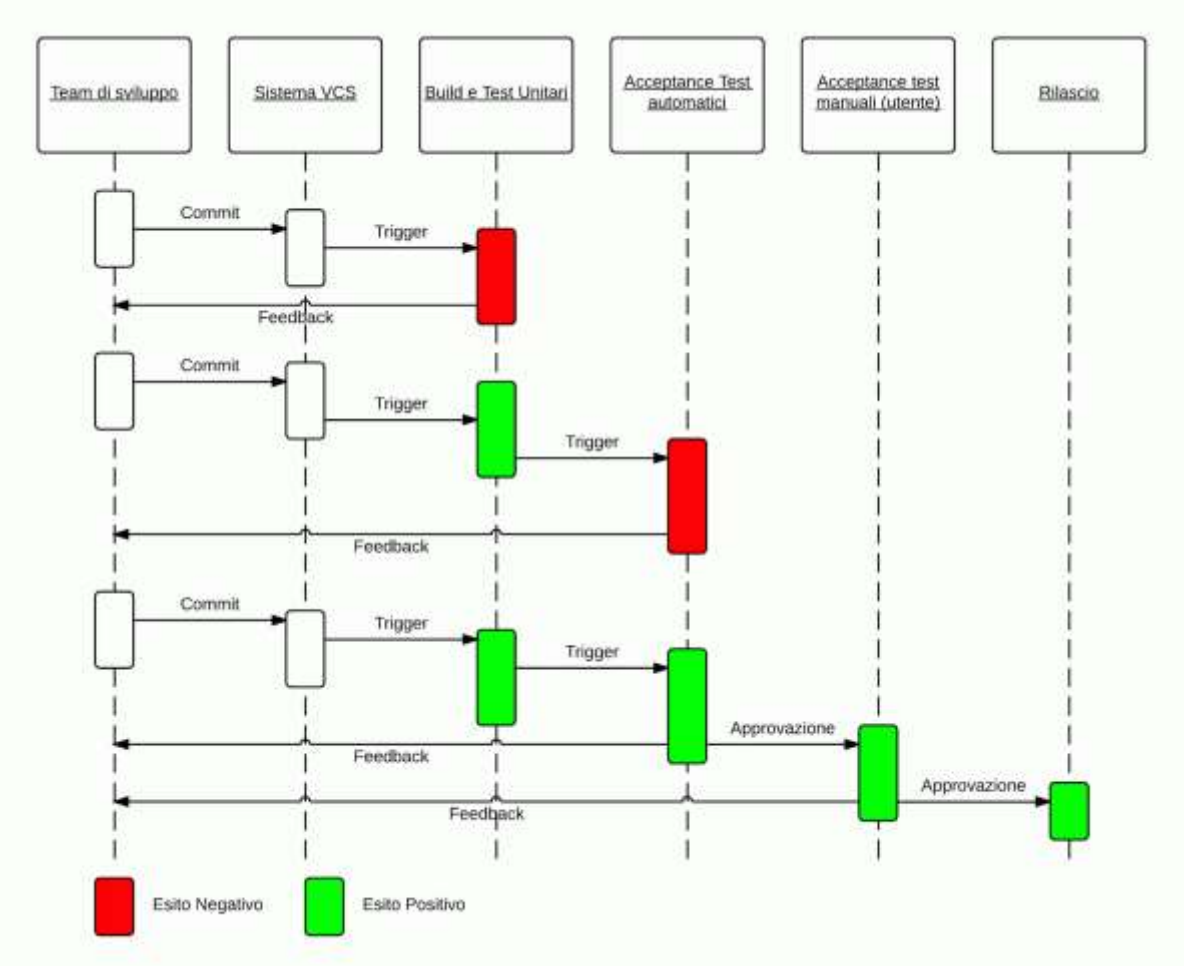


Figura 12 Esempio di sequence diagram di una deployment pipeline eseguita automaticamente ad ogni commit su sistema VCS. Notare l'importanza del feedback ad ogni stato.

Sia Jez Humble e David Farley, nel loro libro [16], che Martin Fowler in un interessante articolo ([17]), elencano e descrivono una serie di pratiche ottimali da adottare per implementare processi in ambito Continuous Development.

Le pratiche vengono descritte da un punto di vista astratto, senza scendere nel dettaglio tecnico sul come metterle in pratica. Esistono numerosi tool, sia a pagamento che open-source, che permettono, singolarmente o in uso combinato tra loro, di raggiungere gli scopi dettati dalle pratiche descritte. Nella sezione relativa all'attività pratica svolta in Volagratis, vedremo un esempio di come queste pratiche sono state implementate dal punto di vista tecnico e quali strumenti sono stati utilizzati. Riportiamo le pratiche, estratte dalle pubblicazioni sopracitate.

Unico repository per i sorgenti

I progetti software (programmi) coinvolgono numerosi file che devono essere opportunamente combinati tra loro per costruire un prodotto (una vera e propria orchestrazione). Tenerne traccia è un effort oneroso, in particolare quando sono coinvolte più persone. Non c'è da sorprendersi quindi che, negli anni, i team di sviluppo software abbiano costruito tools per gestire tutto questo. Questi tools, chiamati Source Code Management, Configuration Management, Version Control Systems, repositories o anche in altri modi, devono essere *parte integrante del progetto*: bisogna evitare soluzioni "artigianali" quali, ad esempio, il ricorso a dischi e directory condivisi.

La maggior parte di questi tools sono open source, quindi non è necessario fare investimenti, se non il tempo necessario ad installare il sistema e comprenderne in maniera almeno basilare il funzionamento.

Una volta messo in piedi il sistema, assicurarsi che il posto da cui prelevare i sorgenti sia noto a tutti, nessuno dovrebbe chiedere "Dove è il tale file?", tutto dovrebbe essere nel repository e rintracciabile.

Tutto il necessario ai fini del funzionamento di un programma eseguibile, deve essere nel repository, compresi: script di test, file properties, database schema, script di installazione e librerie esterne.

Nel caso delle librerie esterne, può essere utile l'impiego di tool quali, ad esempio, Maven, che permettono la gestione centralizzata delle dipendenze tra progetti sollevando, almeno in parte, gli sviluppatori da tale onere.

E' sconsigliato invece, mantenere nei repositories i prodotti delle build (packages).

L’insieme dei file sorgenti posti sotto controllo di versione, non ancora inclusi in una release software, viene indicato con vari termini quali, ad esempio *trunk, baseline, master*. Nel seguito ci si riferirà a tali file con il termine ***Mainline.***

Automatizzare la Build

Partire dai sorgenti per ottenere un sistema funzionante può spesso essere un processo complicato che coinvolge compilazione, trasferimento file, caricamento di schemi nei database, ed altro ancora.

Questa parte può, e deve, essere automatizzata. Chiedere alle persone di eseguire "strani" comandi o cliccare attraverso finestre di dialogo è una perdita di tempo e può favorire errori.

Gli ambienti automatizzati per le build sono una caratteristica comune dei sistemi (esempio Ant, Ruby, Nant, MSBuild, ...)

Un errore comune è il non includere tutto nella build automatica. La build dovrebbe comprendere, ad esempio, il prelievo degli script relativi alle modifiche al database e la loro esecuzione nell'ambiente in cui il programma verrà mandato in esecuzione.

L'idea di base è che, una volta installata una macchina vergine ex novo, dopo un checkout del repository questa macchina dovrebbe essere in grado di buildare ed eseguire il programma senza problemi.

Nel caso la build di un grosso progetto richieda parecchio tempo, particolari accorgimenti possono essere adottati in modo da non dover costruire da zero il prodotto tutte le volte che vengono apportate modifiche, anche elementari.

Rendere la Build Auto-Testante

Build significa compilazione, link e tutto ciò necessario ad un programma per essere eseguito partendo dai sorgenti. Un programma può girare, ma ciò non esclude la presenza di eventuali bug nel codice. Un buon modo per intercettare bug in modo veloce ed efficiente è includere test automatici nel processo di build. La crescita dell'Extreme Programming (XP) e del Test Driven Development ha contribuito a diffondere il concetto di codice auto-testante. Ovviamente non è strettamente necessario il ricorso al TDD per avere del codice auto-testante, sicuramente questa tecnica è di grande aiuto nella predisposizione dei test.

E' però necessario avere una suite di test automatici che coprano la maggior parte della mainline, allo scopo di scoprire eventuali bug. I test vengono poi lanciati tramite un comando ed il loro risultato deve riportare eventuali test falliti. In una build auto-testante il fallimento di un test comporta il fallimento della build stessa.

Negli ultimi anni ha acquisito popolarità la famiglia di tool open-source XUnit, ideali per questo tipo di test e per costruire un ambiente completamente auto-testante.

Ovviamente non si può contare sui test per trovare tutto. Come spesso si dice: i test non dimostrano l'assenza di bug. Tuttavia, la perfezione non è l'unico punto per cui si ottiene valore dalla build auto-testante. Test imperfetti, lanciati spesso, sono molto meglio dei test perfetti che non sono mai stati scritti.

Tutti committano sulla mainline ogni giorno

Integrazione è soprattutto comunicazione. L'integrazione mette gli sviluppatori in condizione di parlare con gli altri sviluppatori delle modifiche che hanno fatto. Il primo prerequisito per uno sviluppatore, affinché possa committare il proprio lavoro sulla mainline, è che questa continui a buildare correttamente. Questo include, ovviamente, il passaggio dei test automatici. Come avviene per ogni commit, lo sviluppatore prima aggiorna la propria codebase locale con la mainline su repository, risolve eventuali conflitti, builda in locale. Se la build va a buon fine, committa le modifiche fatte.

Facendo questo frequentemente, gli sviluppatori scoprono velocemente eventuali conflitti tra il proprio codice locale e la mainline. La chiave per risolvere i problemi velocemente è individuarli velocemente. Con commit frequenti, eventuali problemi emergono altrettanto frequentemente e, a questo punto, non essendovi molte differenze tra due commit adiacenti, sono relativamente facili da risolvere. I

conflitti che rimangono "sommersi" per settimane, possono essere invece estremamente difficili da risolvere.

Se ci sono solo poche ore di modifiche tra un commit e l'altro, i punti del programma in cui il problema potrebbe nascondersi rimangono circoscritti.

In generale, più frequentemente si committa, meno punti andranno verificati in caso di conflitti e, quindi, più rapidamente i conflitti verranno sistemati.

I commit frequenti incoraggiano inoltre gli sviluppatori a spezzare il proprio lavoro in piccoli intervalli da qualche ora ciascuno. Questo aiuta ad avere consapevolezza dei progressi fatti.

Eseguire tutti i Commit Tests localmente prima di committare

Prima di committare le proprie modifiche sulla mainline è necessario scongiurare il rischio che queste possano far fallire la build della mainline. Oltre ad aggiornare la propria codebase locale all'ultima versione della mainline ed assicurarsi che il codice venga correttamente compilato, è necessario eseguire localmente anche i test che verificano il corretto funzionamento delle modifiche introdotte (self-testing, vedi sopra). Sono gli stessi test che verranno eseguiti quando verrà lanciato il processo di build della mainline presente sotto SCM.

Alcuni moderni server di Continuous Integration sono in grado di gestire automaticamente questo passaggio. Intercettano le modifiche committate e svolgono automaticamente i test. Solamente in caso di esito positivo di questi test, le modifiche vengono effettivamente committate sulla mainline, viceversa viene notificato il problema allo sviluppatore o agli sviluppatori che l'hanno creato in modo che possa essere risolto.

Ogni Commit lancia una build della mainline su una Integration Machine

Attraverso commit giornaliere, un team ottiene build testate frequenti. Questo significa che la mainline, o meglio il prodotto risultante dalla build dei file che la costituiscono, rimane in uno stato funzionante. Nella pratica, le cose potrebbero andare diversamente. Un motivo è la disciplina, ovvero persone che non aggiornano la propria codebase locale. Un altro motivo sono le differenze ambientali tra le macchine dei vari sviluppatori.

Bisogna quindi prevedere build regolari su una macchina dedicata (integration machine) conseguenti ad ogni commit. E ogni commit viene considerato completato solamente se la build da lui scatenata va a buon fine. La responsabilità di monitorare che questo accada ricade sullo sviluppatore che committa, in modo che lui stesso possa sistemare eventuali inconvenienti. Come corollario, non si esce dall'ufficio se gli ultimi commit di giornata hanno rotto la build sulla integration machine.

Per assicurare questo, oltre a lanciare build manuali sulla integration machine dopo ogni build, si può ricorrere ad un continuous integration server.

Un continuous integration server agisce da monitor sul repository. Ogni volta che viene eseguita una commit sul repository, il server esegue il check out della codebase sulla integration machine, lancia la build e notifica a chi ha committato l'esito della build. Il commit si considera concluso solo a ricezione avvenuta della notifica. Onde evitare spam di notifiche, si può restringere l'invio di una notifica solamente in caso di fallimento della build. E' comunque responsabilità dello sviluppatore, verificare l'esito della build avvenuta sulla integration machine in seguito al suo commit.

Alcune organizzazioni ricorrono a build schedulate, questo però è leggermente diverso rispetto al concetto di continuous build esposto sopra e non è sufficiente come pratica di Continuous Integration, in quanto uno degli scopi della continuous integration è scovare problemi in tempo brevissimo. L'emergere di un bug durante una build notturna, ad esempio, significa che, potenzialmente, questo bug è rimasto nascosto per un intero giorno lavorativo, e potrebbe essere quindi di non facile risoluzione.

Non iniziare altre attività fino a quando non vengono superati i Commit Tests

Il sistema di Continuous Integration è una risorsa condivisa per il team. Se viene effettivamente impiegato come descritto poc’anzi, con commit frequenti, ogni rottura della build implica un temporaneo blocco per il team e, di conseguenza, per il progetto. Questi inconvenienti, tuttavia, sono normali e da mettere in conto. L'importante è quindi che gli errori che li hanno causati vengano individuati e riparati nel minor tempo possibile.

Al momento del commit gli sviluppatori che lo hanno eseguito sono responsabili del monitoraggio della build della mainline. Fino a quando la compilazione della mainline non è terminata ed i commit tests sono stati eseguiti e superati sulle macchine dedicate alla Continuous Integration, gli sviluppatori non dovrebbero iniziare nuove attività, riunioni e pause pranzo incluse.

Se il commit va a buon fine, sono allora liberi di dedicarsi a nuove attività. Se fallisce sono già pronti per determinare la natura del problema e per risolverlo, con un nuovo commit o con un revert alla situazione precedente al commit che ha rotto la build. In questo secondo caso, le modifiche vengono quindi rimosse dalla mainline fino a quando non verranno ricommittate funzionanti.

Non committare se la build è rotta

Uno dei "peccati capitali" della Continuous Integration è il commit sulla build rotta. Se la build si rompe, gli sviluppatori che hanno causato la rottura stanno lavorando, o almeno dovrebbero, per risolvere il problema al più presto.

Se dei colleghi committano una o più modifiche che rompono la build (non facendola compilare o facendo fallire dei test automatici, per esempio), devono poter risalire al problema senza ostacoli, per risolverlo nel migliore dei modi. Ulteriori commit di modifiche, scateneranno nuove build che falliranno ed i cui risultati potrebbero mescolarsi con quelli della prima build che si è rotta, creando confusione.

Quando non viene rispettata questa regola, quindi, il fix della build richiede maggior tempo e, come conseguenza, le persone si "abituano" a vedere la build rotta e si arriva ad una situazione in cui la build rimane rotta per la maggior parte del tempo. Questo continua fino a che qualcuno nel team decide che "quando è troppo è troppo" e con un notevole sforzo ripara la build.

Mai andare a casa se la build è rotta

Lasciare la build rotta a fine giornata lavorativa, o peggio, a fine settimana non è una buona pratica. Al rientro in ufficio potrebbe esser necessario diverso tempo per rifocalizzare l'attenzione su quel che potrebbe aver rotto la build. Inoltre, il resto del team sarebbe costretto ad uno stop forzato ad inizio giornata o settimana lavorativa o peggio, a dover investire energie per aggirare il problema. Nel caso dei team distribuiti in punti con fusi orari diversi questo aspetto assume ancora maggiore criticità.

Per essere chiari, non è richiesto di rimanere in ufficio fino ad orari improponibili per sistemare una build, viceversa è raccomandato fare commit frequenti e lontani da orari-limite, in modo da aver tempo di fronteggiare eventuali anomalie. In alternativa, attendere il giorno successivo per committare. Molti sviluppatori esperti e molti team Agili definiscono un orario oltre il quale non si committa (ad esempio un'ora prima della fine dell'orario lavorativo) e tutti i commit non fatti oltre l’orario limite diventano la prima attività da svolgere nella prossima giornata lavorativa.

Sistemare immediatamente le build rotte

Questo è diretta conseguenza di quanto appena esposto.

Se la build della mainline fallisce, deve essere sistemata immediatamente. Non è un fatto negativo in se "rompere la build", anche se una perseveranza di questo potrebbe indicare scarsa attenzione da parte degli sviluppatori nella fase pre-commit (build in locale su tutto).

Quando questo avviene, il ripristino della build assume priorità massima. Non è necessario che tutto il team si dedichi a ciò, di norma bastano una o due persone.

Spesso il metodo più rapido per sistemare la build è ripristinare la situazione all'ultimo commit fatto prima che si manifestasse il problema, riportando di fatto il sistema indietro. A meno che la causa del problema sia evidente, è buona norma lasciare la mainline aggiornata all'ultimo commit funzionante e ricercare il problema che ha rotto la build eseguendo debug su una singola workstation.

Essere sempre pronti a tornare alla versione precedente

Come abbiamo visto, la rottura della build è una cosa normale. Nei progetti di dimensioni importanti, è lecito aspettarsi che ciò avvenga almeno una o due volte al giorno, nonostante i test svolti in locale prima di committare modifiche allevino questo rischio.

In queste circostanze, i fix normalmente consistono in commit di poche righe di codice che risolvono bug in tempo rapidissimo.

Tuttavia, a volte, il compito è più arduo sia per un errore un po' più "grave" del solito, e questo non significa colpevolizzare chi l'ha commesso, sia perché il bug non è di semplice individuazione e sia perché subito dopo il commit delle modifiche ed il successivo fail della build, si potrebbe realizzare di aver trascurato particolari importanti nell'implementare le modifiche appena committate. Indipendentemente dalla ragione, è importante ripristinare il corretto funzionamento della build alla svelta. Se il problema, per qualunque ragione, non può essere risolto rapidamente, occorre riportare la mainline alla situazione precedente al commit che ne ha rotto la build facendo revert della modifica.

L'approccio mentale suggerito è lo stesso, facendo un paragone, che hanno i piloti di aereo quando stanno per atterrare. Ovvero essere pronti a tornare indietro ("go around") e fare un ulteriore tentativo nel caso qualcosa vada storto.

Definire un Time-Box per il fix della build rotta

Quando la build si rompe, investire non più di X minuti per sistemarla. Se dopo X minuti non è stata sistemata, procedere col revert della mainline alla versione precedente alla rottura, recuperabile dal versioning control system. L'entità di X è lasciata all'auto-regolamentazione dei team di sviluppo. In alcuni casi, se trascorso questo limite ci si sente confidenti di essere prossimi alla soluzione, si può ricorrere ad alcuni minuti extra. Ad esempio, se dopo dieci minuti si stà procedendo con la build locale, è ovviamente possibile terminare la build locale, procedere col commit e, nel caso la build sia tornata in uno stato funzionante, il fix si può considerare finito. Viceversa, procedere col revert e, con calma, a tutti i controlli del caso.

Mantenere la build rapida

Uno degli scopi della Continuous Integration è il fornire un rapido feedback. Nulla è più "stressante" di una build che dura "a lungo". Sul concetto di "build lunga" si potrebbe discutere per giorni. Dipende dai punti di vista e dalle abitudini dei team. Alcuni potrebbero percepire come lunga una build della durata di un'ora, mentre altri sognano di avere una build che duri "solo" un'ora.

Le linee guida dell'Extreme Programming definiscono 10 minuti come durata ragionevole di una build e i progetti più moderni rispettano queste linee. Vale la

pena sforzarsi affinché ciò avvenga. Ogni minuto di riduzione della build, è un minuto "regalato" agli sviluppatori ogni volta che loro committano (e che devono attendere l'esito della build della mainline...).

Non sempre è possibile ridurre "magicamente" l'intero processo di build alla durata desiderata.

La pratica consigliata, è l'impostazione di una deployment pipeline. L'idea dietro alla deployment pipeline (nota anche come build pipeline o staged build) è che, di fatto, ci sono build multiple eseguite in sequenza. Il commit sulla mainline scatena la build primaria, detta anche commit build, che è quella che deve essere veloce.

Una volta che la commit build è a posto, le altre persone possono lavorare sul codice con confidenza. Tuttavia ci possono essere ulteriori test, più lenti, che devono essere svolti. Questi possono, ad esempio, essere demandati a macchine dedicate e/o eseguiti in tempi diversi.

Un semplice esempio è una deployment pipeline a due stadi. Il primo si occupa della commit build, ovvero della compilazione e dell'esecuzione dei test unitari che non dipendono dai dati presenti su database. Questi test sono molto veloci e mantengono tutto il processo sotto i 10 minuti stabiliti. Tuttavia alcuni bug potrebbero non essere scovati dai test unitari, in particolare quelli eventualmente presenti in parti che richiedono interazioni più di larga scala, come ad esempio interazioni col database.

La build svolta al secondo livello (build secondaria) lancia una suite differente di test, più lenti, che verificano il comportamento del codice su dati presi dal database reale, testando così il comportamento del codice nel suo insieme piuttosto che nelle sue singole parti.

In questo scenario, il primo livello viene usato come ciclo principale di Continuous Integration. La build secondaria invece, viene eseguita (o schedulata) quando possibile, prelevando il codice eseguibile preparato dall'ultima build di primo livello andata a buon fine e svolgendo test ulteriori. Se questa seconda build fallisce, l'impatto sul team non è importante quanto quello che avrebbe un fallimento della commit build, tuttavia deve essere sistemata in tempi ragionevolmente rapidi.

Un bug rilevato dalla build secondaria spesso riguarda un problema rilevabile anche da uno o più test unitari eseguiti dalla commit build. In generale, ogni problema emerso ad un livello successivo al primo, porta allo sviluppo di nuovi test unitari da eseguire durante la commit build in modo da renderla sempre più efficace ed abbassare il rischio che i livelli successivi rilevino bug o malfunzionamenti.

Il principio può essere esteso a più livelli successivi. Le build secondarie possono anche essere eseguite in parallelo. Ad esempio se i test secondari durano due ore, si possono migliorare le performances distribuendoli equamente su due macchine che lavorano in parallelo. Parallelizzando le build secondarie in questo modo si può introdurre ogni sorta di test automatici all'interno del processo di build.

Testare in un clone dell’ambiente di Produzione

Lo scopo dei test è (anche) quello di eliminare ogni problema che il sistema potrebbe avere in produzione. L'ambiente in cui il sistema si troverà a girare in produzione è molto importante. Se i test vengono svolti in un ambiente differente, ogni differenza è sinonimo di rischio che quello che succede in ambiente di test non succederà in produzione.

La pratica ideale sarebbe quindi quella di impostare l'ambiente di test il più possibile uguale a quello di produzione. Utilizzare lo stesso software, alla stessa versione per il database, la stessa versione del sistema operativo. Tutte le librerie presenti nell'ambiente di produzione devono essere riportate nell'ambiente di test, anche se il sistema non le usa. Utilizzare gli stessi indirizzi IP, le stesse porte e lo stesso hardware.

In realtà ci sono dei limiti. Se si scrive, ad esempio, software per macchine desktop è praticamente impossibile testare su un clone di tutte le potenziali macchine su cui il sistema verrà installato (sic!). Similmente, alcuni ambienti di produzione sono molto costosi, impensabile quindi la loro duplicazione.

Nonostante questi limiti, l'obiettivo rimane quello di replicare l'ambiente di produzione al meglio e comprendere i rischi legati ad ogni differenza tra l'ambiente di test e quello di produzione.

Negli ultimi tempi ha acquisito sempre maggior popolarità il ricorso alla virtualizzazione, vale a dire più ambienti simulati su una stessa macchina fisica. Risulta quindi relativamente facile impostare uno o più ambienti virtuali all'interno dei quali svolgere i test.

Non escludere i test che falliscono

In presenza di una build rotta a causa del fallimento di alcuni test, gli sviluppatori tendono a commentare il codice di questi test, bloccandone l’esecuzione, in modo da poter committare agevolmente le proprie modifiche ed avere la build correttamente funzionante. Questo è comprensibile, ma sbagliato. Quando uno o più test che hanno sempre funzionato falliscono, può essere complicato scoprirne il motivo. Siamo veramente in presenza di regressione? Magari una o più assunzioni fatte dai test non sono più valide, oppure sono state fatte modifiche alla funzionalità testata per un motivo valido. Scoprire quale di queste condizioni si è verificata può coinvolgere più persone per un discreto periodo di tempo, ma è essenziale scoprire cosa stà succedendo e quindi sistemare il codice se si è introdotta regressione, modificare i test se la funzionalità ha cambiato comportamento o addirittura eliminarli nel caso non servano più.

L’esclusione dei test che falliscono deve essere vista come una *extrema-ratio*, a cui si deve ricorrere, ad esempio, se è necessario un importante ed articolato sviluppo.

Assumersi le proprie responsabilità

Assumersi la responsabilità degli inconvenienti che i propri commit potrebbero causare alla build della mainline.

Se i test scritti per la funzionalità introdotta o modificata passano, ma altri no, significa che è stata introdotta regressione.

E’ responsabilità di chi ha sviluppato questa funzionalità sistemare anche gli altri test che in seguito alla modifica hanno smesso di avere esito positivo.

Questa pratica ha diverse implicazioni. In primis gli sviluppatori devono avere accesso a tutto il codice impattato dalla propria modifica, in modo da poter sistemare ciò che si dovesse rompere. Questo esclude la possibilità che gli sviluppatori abbiano accesso esclusivo su una parte ristretta della codebase. Nella pratica di Continuous Integration, tutto il team ha accesso alla intera codebase. Nel caso questo non sia proprio possibile per cause non facilmente risolvibili, si deve per forza ricorrere alla collaborazione da parte di chi può mettere mano a certe parti della codebase.

Rendere gli ultimi eseguibili creati accessibili facilmente

Una delle parti più difficili dello sviluppo software è assicurare che venga buildato il software corretto. Abbiamo visto che è difficile specificare i requisiti corretti in anticipo. Le persone trovano molto più semplice notare qualcosa fatto in modo non completamente corretto e indicare i cambiamenti necessari. I processi Agili sono basati anche su questo comportamento umano.

Per aiutare in questo, chiunque è coinvolto in un progetto software deve poter avere gli ultimi eseguibili creati ed essere in grado di mandarli in esecuzione: per dimostrazioni, test esplorativi o anche semplicemente curiosità.

Fare questo è abbastanza semplice: basta assicurarsi che ci sia un posto noto dove le persone possono trovare gli ultimi eseguibili. Può essere utile mettere diversi eseguibili in questo posto. Oltre all'ultimissima versione, si potrebbe mettere anche l'ultima che ha passato i commit test, qualora questi test non fossero vincolanti al processo di build e costruzione degli eseguibili.

Se si segue un processo di sviluppo per iterazioni ben definite (sprint), è utile mettere anche l'eseguibile prodotto a seguito degli sviluppi fatti durante le ultime iterazioni concluse. In particolare le dimostrazioni necessitano di software con caratteristiche conosciute, in queste circostanze vale la pena sacrificare l'ultimissima versione disponibile a discapito di una versione le cui caratteristiche siano note a chi la deve presentare.

Chiunque è informato sulla build

La Continuous Integration si basa sulla comunicazione, bisogna quindi assicurare che tutti siano informati circa lo stato del sistema e delle modifiche che vi sono state apportate.

Una delle cose più importanti da comunicare è lo stato della build della mainline. Diversi tool per la gestione della Continuous Integration espongono interfacce web che forniscono questa informazione.

I team, nell'ambito della propria auto-organizzazione, possono assicurare queste informazioni adottando gli accorgimenti che ritengono opportuni. Alcuni team, ad esempio, connettono alle macchine adibite alla continuous integration dispositivi luminosi o sonori che emettono opportuni segnali (esempio luce rossa vs. luce verde, oppure fischi vs. applausi, o anche peggio...) a seconda dello stato della build.

Questo aspetto assume importanza strategica laddove si adotta un approccio manuale verso la Continuous Integration. Deve essere chiaro a tutto il team chi è il collega che ha la temporanea responsabilità della build della mainline.

I server Continuous Integration possono fornire ulteriori informazioni rispetto al semplice stato della build. Ad esempio possono fornire informazioni circa le modifiche apportate alla mainline ed i loro autori.

Non dimentichiamo che le informazioni esposte via web possono essere utili per i team distribuiti.

Automatizzare il Deployment

Per fare Continuous Integration servono diversi ambienti, uno per i commit tests, uno o più per i test secondari. Dal momento che vengono trasferiti eseguibili tra questi ambienti, più volte al giorno, è ideale farlo automaticamente. E' quindi importante avere script che permettano di deployare facilmente l'applicazione in ogni ambiente utile ai fini di test.

Come conseguenza naturale di questo, si dovrebbero avere a disposizione script che permettono il deploy in produzione con la stessa facilità. Può non essere necessario deployare in produzione ogni giorno, ma avere a disposizione script per il deploy automatico aiuta ad accelerare il processo ed a ridurre gli errori.

Se si procede col deploy automatico in produzione, occorre considerare anche il rollback automatico. Bisogna essere in grado di poter riportare l'ambiente di produzione alla versione precedentemente installata e funzionante, nel caso emergano malfunzionamenti non rilevati in sede di test. La possibilità di poter svolgere automaticamente questo passo indietro riduce la tensione all'interno del team e del dipartimento, incoraggia le persone a deployare più frequentemente, e quindi a fornire agli utenti nuove funzionalità rapidamente.

Oltre al trasferimento degli eseguibili tra i server, gli script introdotti precedentemente hanno anche responsabilità di apportare eventuali modifiche all’ambiente in cui l’applicazione dovrà operare. Devono quindi essere incluse in questi script le modifiche al sistema operativo, ai server web ed alle macchine

virtuali, se previste, al database. Le stesse responsabilità, ovviamente al contrario, devono essere assunte dagli script di rollback.

Questi processi possono essere gestiti sia tramite “semplici” script bash o batch, che tramite l’impiego di tool atti allo scopo (Ansible, Puppet, ...) alcuni dei quali (Docker) orchestrano il rilascio completo dell’applicazione e di “tutto quel che le serve per funzionare” su di un server web.

Da dove cominciare?

Se in condizioni ottimali, quali ad esempio un progetto in fase di avvio, uno o più team di sviluppo software neo costituiti, e così via, implementare queste pratiche può essere una attività relativamente semplice. Nella maggioranza dei casi, però, il team di sviluppo e/o l’azienda si trovano a volere adottare metodologie di Continuous Development partendo da una situazione più o meno tradizionale. La domanda “Da dove cominciare?” sorge quindi spontanea. Le pratiche appena elencate portano tutti i benefici possibili, ma non per forza è necessario iniziare implementandole tutte.

Non c’è una ricetta fissa, molto dipende dal team di sviluppo e dalla natura del sistema su cui si va ad implementare il Continuous Development. Di seguito una sequenza di azioni da intraprendere che potrebbe essere impiegata nella maggior parte dei processi di adozione del Continuous Development.

Un primo passo potrebbe essere l’automatizzazione della build. Mettere tutto il necessario (sorgenti ma non solo) sotto versionamento in modo da poter lanciare la build con un singolo comando. Per molti progetti non è una impresa semplice ed è essenziale per il buon funzionamento di tutte le altre cose. Inizialmente si può partire con build occasionali lanciando manualmente il comando oppure con build automatiche notturne. Una build automatica notturna è un buon punto di partenza.

Successivamente, introdurre alcuni test automatici nel processo di build. Identificare le aree dove si verificano malfunzionamenti più frequenti e predisporre test automatici che siano in grado di evidenziarli. Sui progetti esistenti è difficile avere in tempi rapidi una buona suite di test, predisporli richiede tempo. Da qualche parte si dovrà pur cominciare (“*Roma non fu fatta in un giorno*”).

Successivamente, provare ad accelerare la commit build. Continuous Integration su una build della durata di qualche ora è meglio di niente, ma portare la durata della build ai dieci magici minuti è molto meglio. Questo di solito richiede parecchia *“chirurgia”* sulla code base al fine di eliminare le dipendenze dalle parti lente del sistema.

Se si stà invece iniziando un nuovo progetto, applicare metodologie Continuous Development dal principio. Tenere sotto controllo le durate delle build ed intervenire non appena queste superano la *regola dei dieci minuti.* Intervenendo rapidamente, si metteranno in essere tutte le necessarie ristrutturazioni prima che la code base diventi così grande da rendere dolorose le sue ristrutturazioni.

Ma soprattutto, cercare aiuto. Trovare qualcuno che abbia esperienza in tema Continuous Development, avendovi già lavorato in passato è di grande aiuto. Come ogni nuova tecnica, è difficile introdurla quando non si sa come dovrebbe apparire il risultato finale.

I vantaggi

Il vantaggio più grande e più ad ampio raggio del Continuous Development è la riduzione dei rischi.

Vicolo Cieco

Il problema dell'integrazione differita suggerita dalle metodologie classiche, è che è difficile predire quanto durerà e, peggio, è difficile percepire a che punto si è all'interno del processo. Il risultato è che ci si pone in un vicolo cieco durante una delle parti cruciali del processo di sviluppo di un prodotto software.

Il Continuous Development elimina questo problema. Non c'è processo di integrazione lungo, si elimina completamente il vicolo cieco. Ad ogni momento si sa a che punto si è, cosa funziona e cosa no, ovvero si conoscono i bug presenti nel sistema.

Bugs

I bug sono ciò che tipicamente distrugge confidenza e sicurezza e scompiglia pianificazioni e reputazioni. I bug nel software rilaciato rendono gli utenti "arrabbiati". I bug durante lo sviluppo ostacolano i programmatori, rendendo difficile il corretto funzionamento del resto del software.

Il Continuous Development non libera dai bugs, ma li rende molto più facili da individuare e rimuovere. A tal riguardo è quasi come avere codice auto testante (self-testing). Se si introduce un bug e lo si individua alla svelta, sarà semplice anche la sua rimozione. La facilità con cui il bug viene individuato deriva direttamente dal fatto che uno o comunque pochi sviluppatori hanno lavorato su una piccola porzione del sistema, apportando piccole modifiche (principio delle modifiche incrementali unito ai commit, ed ai relativi test, frequenti) e che la parte di sistema modificata sarà per forza di cose quella più "fresca" nella mente di chi ha sviluppato le modifiche.

I bug sono cumulativi. Più bugs si hanno, più difficile è rimuoverli. Questo è dovuto in parte al fatto che si possono avere interazioni tra bug, situazioni in cui gli errori sono il risultato di errori multipli, che rendono quindi difficile l'individuazione di ogni singolo errore, ed in parte a questioni psicologiche. Gli sviluppatori hanno meno energia per individuare e correggere bug quando ce n'è più di uno, un fenomeno che viene talvolta individuato come "The Broken Windows Syndrome".

Come risultato i progetti con Continuous Development tendono ad avere molti meno bug, sia in produzione che in sviluppo. Questo beneficio è sempre rapportato alla bontà della suite di test. Per raggiungere un livello di bug sufficientemente basso è necessario investire costantemente tempo sull'ampliamento e sul miglioramento della suite di tests.

Il deploy frequente è prezioso all'interno del ciclo di sviluppo, soprattutto perché permette agli utenti di avere nuove caratteristiche più rapidamente e di fornire un feedback circa queste nuove caratteristiche altrettanto rapidamente.

Questo aiuta ad abbattere una tra le più grandi barriere che ostacolano un processo di sviluppo software di successo, ovvero le barriere tra clienti (intesi come committenti, Business Owners, Stakeholders, etc…) e sviluppatori.

L'evoluzione ulteriore: DevOps

Nelle organizzazioni tradizionali le funzioni "Development", ovvero gli sviluppatori, e "Operations", altri professionisti IT (ad esempio i sistemisti) sono distinte. Semplificando, possiamo dire che le prime si occupano dello sviluppo software, le seconde del rilascio in produzione e del corretto funzionamento di quanto rilasciato.

Le funzioni hanno obiettivi diversi che, paradossalmente, rischiano di entrare in conflitto. Gli sviluppatori (Developers) mirano a rilasciare in fretta funzionalità nuove o migliorate, e quindi rilascerebbero software ogni giorno, mentre le Operations puntano ad avere il sistema sempre funzionante ed efficiente e tendono mantenere le cose allo stato attuale (funzionante) il più a lungo possibile. Questa differenza tende a rallentare i rilasci, e quindi il Business.

DevOps è la combinazione tra i due termini, "Development" e "Operations". Con questo termine ci si riferisce ad una metodologia di sviluppo software che enfatizza al massimo la collaborazione, la comunicazione e l'integrazione tra sviluppatori software e professionisti IT (sysadmins, DBA, etc.), come descritto su (da Wikipedia [18]). Lo scopo di questa metodologia è mettere un'organizzazione in grado di fornire prodotti e servizi software in tempi rapidi, evitando i "conflitti" descritti sopra.

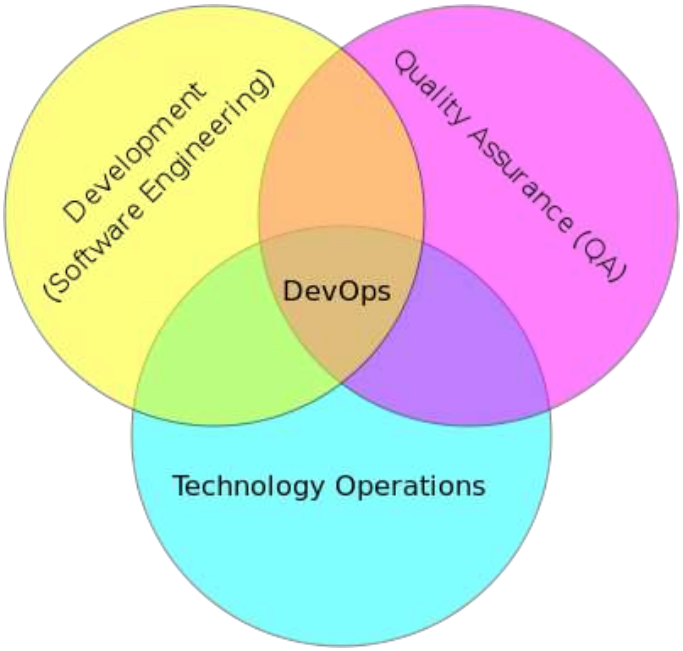


Figura 13 DevOps significa collaborazione tra reparti

Una volta adottate le metodologie Agili, permane comunque una separazione abbastanza netta tra i reparti Sviluppo, IT operations e servizio testing, controllo ed assicurazione qualità (QA). In

genere,la separazione è tra le attività di sviluppo e quelle connesse al rilascio. La metodologia DevOps promuove un set di processi ed accorgimenti atti a favorire comunicazione e collaborazione tra i reparti. Processi ed accorgimenti dipendono dall'organizzazione e dagli obiettivi che si intendono perseguire.

Nell'ambito di una organizzazione che eroga un servizio web, un esempio di DevOps potrebbe essere una collaborazione stretta tra Sviluppatori e Tester al fine di predisporre una suite di test automatici e manuali a garanzia e verifica di quanto sviluppato, e tra Sviluppatori e Sistemisti al fine di predisporre uno script automatico che permetta il rilascio in vari ambienti del software sviluppato.

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